EE 471 Final Report

Design and Implementation of a Single Cycle CPU and a Pipelined CPU

Tyler Bowen, Jeremy Lawrence & Camilo Tejeiro

07/15/2013

EE 471

The following report details the development of a single cycle and Pipelined computer utilizing a MIPS architecture.

Table of Contents

ABSTRACT

This is the final lab project in the design of a pipelined single cycle computer with MIPS architecture CPU, building upon our previously designed subsystems of the Register File, ALU, and Data Flow Bus modules, completing the overall hardware design by adding hand compiled instructions and pipelining the processing of these instructions. New operations in the ALU have been added to facilitate the use of these new functions as well as new hardware for pipelining. The added instructions correspond to the machine code for each instruction and the requisite control signals have been added and implemented into the overall design.

INTRODUCTION

The objectives of this project are to use the previously designed and integrated subsystems from the subsequent projects, while still utilizing the Cyclone II FPGA chip on the Altera DE1 board with the Quartus IDE software environment, and implement a specific set of instructions. Five new operations: Load Word (LW), Store Word (SW), Jump (J), Jump Register (JR), and Branch when Greater Than (BGT), are to be added to the existing ALU subsystem so a series of instructions assigned to a new 128x32 instruction memory register array (IMEM) contained within the Control subsystem, are able to be executed. Also buffers are added between operational stages of instruction *fetching*, *decoding*, *execution*, *write-back*, and *next* instruction cycle for pipelining of the CPU operations. The CPU must also be able to handle control/instruction hazards via detection and data forwarding. The instructions set are originally designed in C language code, and are hand compiled into MIPS instructions and then converted into machine code instructions. The machine code will then be loaded into the control system for execution.

DISCUSSION OF THE LAB

Requirements Specification

- Identify the control steps and actions necessary to support the following instructions in a pipeline context: NOP, ADD, SUB, AND, OR, XOR, SLT, SLL, LW, SW, J, JR, and BGT.
- Develop the machine code to control the data-path elements to affect each of the control signals and actions to ensure proper execution of each instruction in a pipelining context.
- Develop hardware solutions for possible data hazards presented by jump and branch operations as well as alleviating bottlenecks due to memory write times.
- Hand compile a C code fragment into an assembly level program comprising instructions from the supported set.
- Hand compile the assembly level program into machine code.
- Load the machine code into memory then execute the program.

System Description (Summary)

The overall system operation will consist of: machine code instructions are loaded into the Instruction Memory (IMEM), these instructions are then fetched by the control unit, decoded and then executed. A Program Counter (PC) is used to track the progress of the instructions utilized and indicates the next instruction to be fetched. Once the control unit has decoded the instruction, it sends out the appropriate control signals to the requisite subsystems to implement the instructions. For example, the 16 bit word data that is stored within the SRAM (MEM) is accessed, sign extended and then loaded into separate registers in the Reg File (REG) for processing. From the REG, the data is passed through the ALU depending on what operation is being executed. Once the data is processes, the result is loaded back into the REG for further processing. This data is then written back into MEM so that it can be stored and not easily lost or overwritten. The following figure depicts an example of the basic architecture of the overall system:

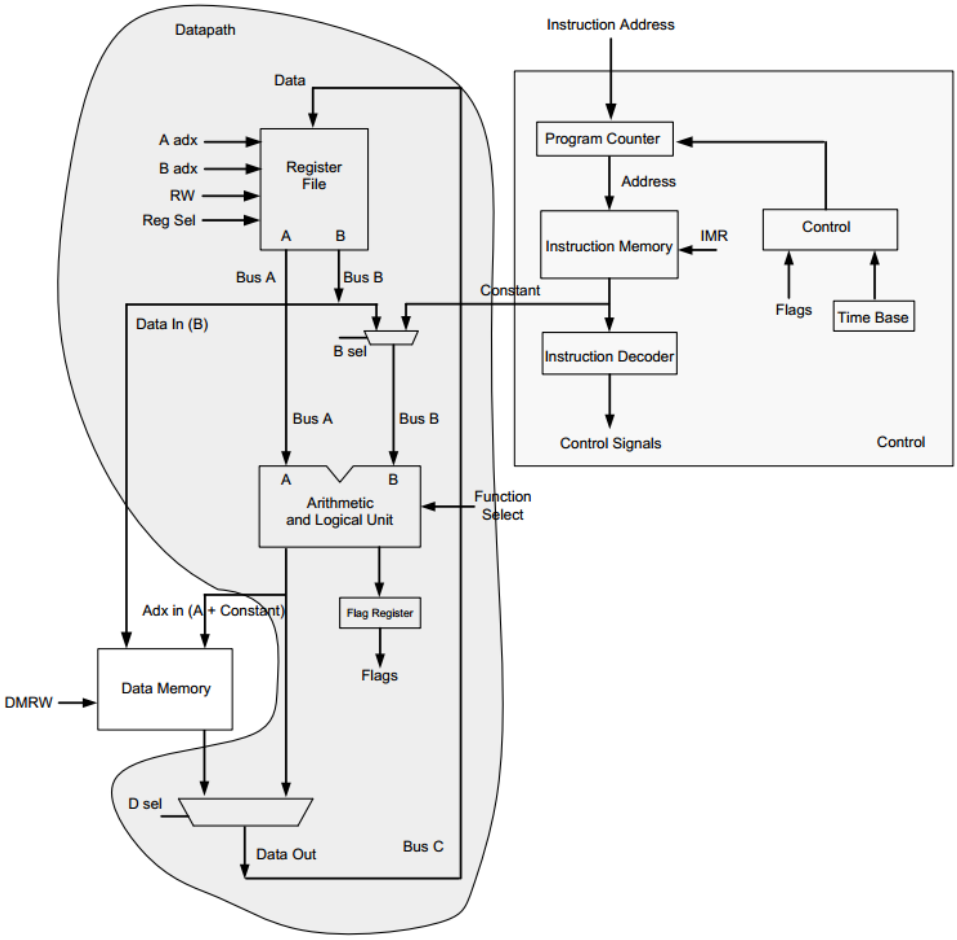


Figure 1: High Level System Architecture

Operating Specifications

For the testing and verification of the single cycle architecture prior to pipelining, the operation specifications implemented are from the following C code:

```
int A = 7;
int B = 5;
int C = 2;
int D = 4;
int* dPtr = &D;

if (A – B)  > 3
{
      C = 6;
      D = D << 2
}
else
{
      C = C << 5;
      *dPtr  = 7;
}
```

Figure 2: Sequential C code Instructions

With Pipelining implemented, the following code was utilized as the instruction set:

```
int A = 6;
int B = 4;
int C = 2;
int D = 4;
int* dPtr = &D;
unsigned int E = 0x7676;
unsigned int F = 0xA5A5;
unsigned int G = 0xFF;
unsigned int H = 0xC3;

if ((A – B)  > 3
{
      C = C + 4;
      D = C – 3;
      G = E | F;
}
else
{
      C = C << 3;
      *dPtr  = 7;
      G = E & F;
}
A = A + B;
G = (E ^ F) & H;
```

Figure 3: Pipeline C code Instructions

These codes were both compiled by hand into MIPS assembly code detailing the separate steps required to implement the C code instructions. The following figure show the basic flow of operations done with the code in Figure 2 and how it ties into the system. The Compiler and Assembler blocks were both done by hand without the use of a program software:

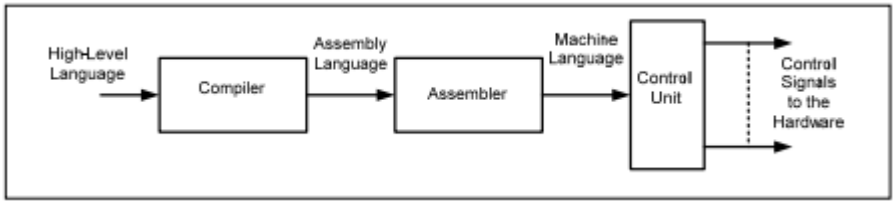


Figure 4: Instruction Code Compilation Flow Chart

Uploading of the instructions into the IMEM was done by our Program Loader module which populated IMEM and sent the appropriate variable data to SRAM. The following code is an example of assembly code for the instructions in Figure 2 (Figure 3's code can be viewed in the Appendix B: Pipelined CPU section of this report):

```
        .data
A:      .word   7                       #A = 7
B:      .word   5                       #B = 5
C:      .word   2                       #C = 2
D:      .word   4                       #D = 4
        .globl  main
        .text
main:
        lw      $s0, D                  #Sets a pointer (*dPtr = D)
        lw      $t0, A                  #Data A is loaded into the Reg File
        lw      $t1, B                  #Data B is loaded into the Reg File
        sub     $t3, $t0, $t1           #Loads the difference of A-B into t3
        bgt     $t3, 3, branch1         #Tests if t3 > const(3), otherwise
                                        # jumps to branch1.
        sw      C, 6                    #Stores the value 6 into C in MEM
        lw      $t0, D                  #Data D is loaded into the Reg File
        sll     $t0, $t0, 2             #Shifts the value of D left 2 bits
        sw      D, $t0                  #Stores t0 back into D in MEM
        j       branch2                 #Jumps to the next section of
                                        # instructions: branch2.
branch1:
        lw      $t0, C                  #Loads C into t0
        sll     $t0, $t0, 5             #Shifts t0 left five bits
        sw      C, $t0                  #Stores t0 into C (s2)
        sw      $s0, 7                  #Stores a const(7) into the *dPtr
branch2:
        noop                            #Instruction set completed
```

Design Procedure

Summary

To compile the MIPS instructions into machine code, a standard instruction architecture of the code and how it would be utilized was needed to be designed. Each instruction is 32bits long, comprised of various commands for the separate subsystems and what they are required to do.

For designing the pipeline, the CPU needed to be able to decode and execute an instruction and store the results in an assembly line fashion. One stage will perform its function on an instruction, send it on to the next stage, and immediately begin performing its function again on the next instruction coming down the pipeline. This poses potential hazards, such as when a subsequent instruction depends on variables from the preceding instruction. The handling of such hazards is discussed later in the Error Handling section.

High Level View

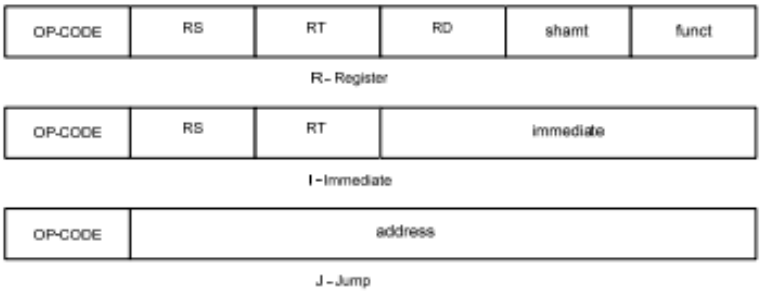


Figure 5: Machine Instruction Code Hierarchy

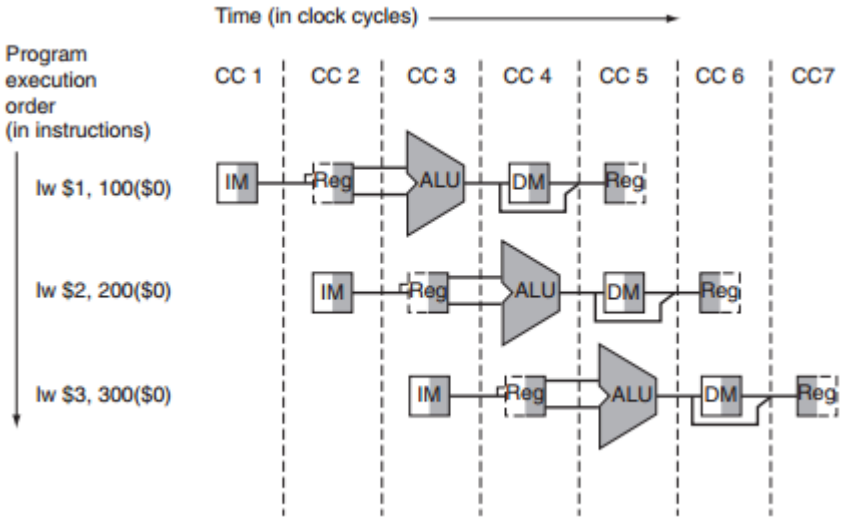


Figure 6: Pipelined Instruction Flow

Pseudo Description of Instructions

The 32bit instructions are divided into fields depending on the type of instruction. There are three types of instructions: R-type, I-type, and J-type. All three contain the OP-CODE field which specifies the operation of the instruction and also alludes to which type of instruction it is.

The R-type instructions are for Register-to-Register operations, where data in the REG is sent to the ALU for processing then loaded back into the REG.

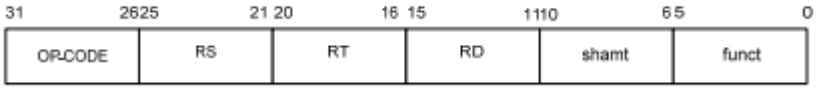


Figure 7: R-Type Format

The Rs field specifies the first register source in REG for the instruction. Rt is the second register source in REG for the instruction. A source register is the location of a set of data for use as an operand. Rd is the destination register in REG for the instruction. The destination register is the location that the processed data from the two operands is written into in REG.

The I-type instructions are used for Memory-to-Register, Register-to-Memory, or Branch type operations, where data is read from MEM to REG or written into MEM, and Branch is for If-Else and comparison operations.

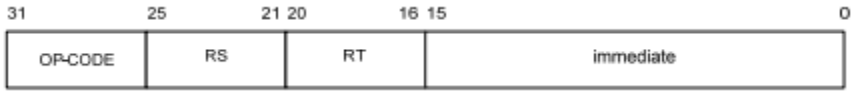


Figure 8: I-Type Format

The OP-Code, Rs, and Rt fields are the same as for the R-type instructions except now Rs and Rt can now represent locations in MEM as well. The Immediate field can be used to specify another operand, or constant variable depending of the I-type instruction used.

The J-type are for Jump operations, where the PC is instructed to either increment the PC (typically by 4 bytes) or “jump” the PC to another instruction contained within the IMEM.

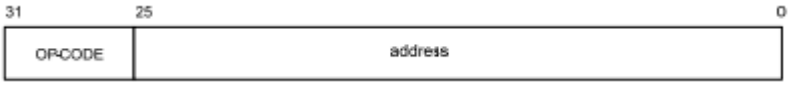


Figure 9: J-Type Format

Where the Address field specifies the location of the instruction to be jumped to.

Error Handling

Some of the hazards that are possible during pipelining are:

- Data flow bottlenecks between stages that have differing processing times.

- Jump or Branch hazards generated when instructions experience an If-else statement or when the PC is required to “jump” to another instruction that is out of sequence with the previous instruction.
- Subsequent instruction calls on resulting variables from preceding instruction set which has not finished propagating through the pipeline.

To handle these types of hazards, two modules were integrated into the Single Cycle CPU design. A Hazard Detector was added into the Control Module and a Data Forwarding Module was also tied into the system. The Hazard Detector would insert a stall into the pipeline when it detected an impending hazard due to a jump or branch instruction. This is done by flushing all of the current control signals and processing a NOOP instruction. This allows the pipeline to continue without severe interruptions in its operational flow.

The Data Forwarding unit is tied to every stage of the pipeline and accepts control signals that tell it if a piece of data that is being processed is needed in the subsequent instructions. If it is, the Forwarding unit then filters a copy of the required data into the requisite stage so that the pipeline flow is not interrupted or requires a stall to be inserted. The following figure depicts how both the Hazard Detector and the Forwarding Unit would affect the pipeline.

Also, the instructions were ordered into a “Lazy Write” format which loads all of the requisite data utilized in the code, preforms all of the operations within the code, and then stores all of the values into memory at the end. This eliminated most of the hazards caused by the extended time required in reading and writing from/to memory. All of the data required in the first set of R-type instructions was already written into the registers by the time the R-type instructions were fetched. Also by the time the first store word instruction was called, the requisite data to be written was already processed and waiting in the register file to be sent back into memory.

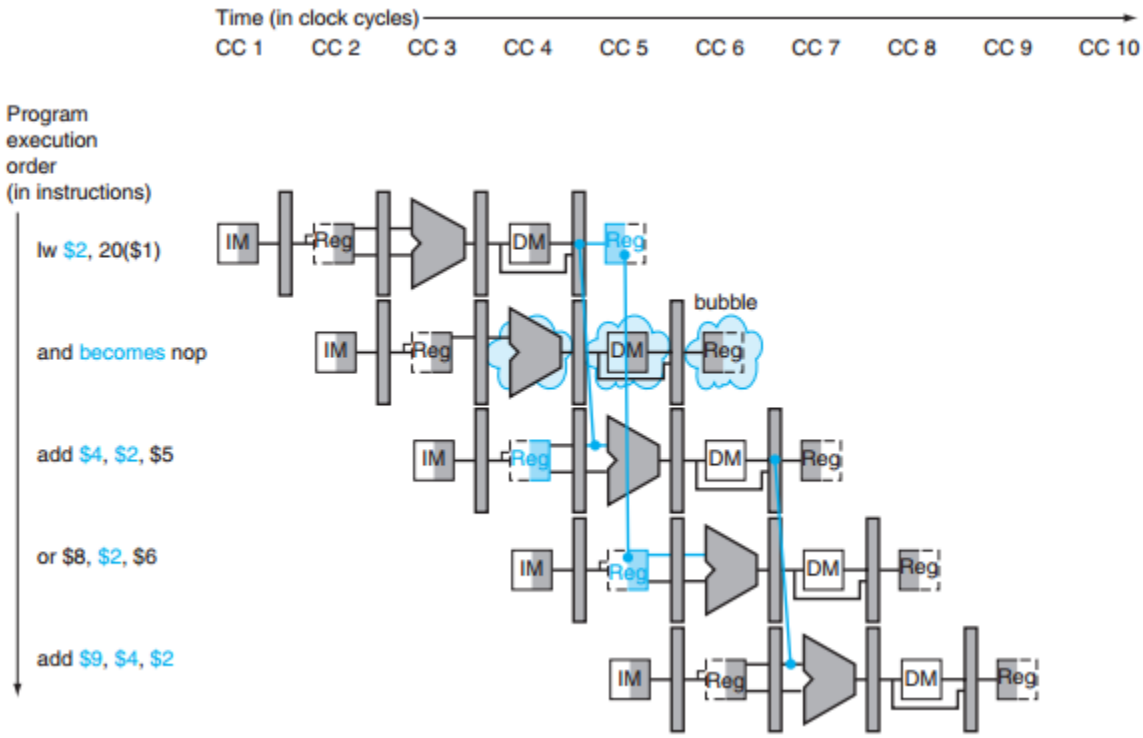


Figure 10: Hazard Handling via Detection and Forwarding

Error Handling

Software Implementation

Discussion of Specific Tasks

Single Cycle CPU: Please refer to the corresponding diagrams in the appendix A.10 for a visual description of the tasks detailed below. Also refer to the function blocks section for descriptions of the modules used to implement each task.

Loading programs:

The loading of programs for execution on the single cycle computer happens as a separate function from the rest of the functions of the CPU, which requires a disabling control signal to prevent the program counter from incrementing until the instructions and initial memory are loaded into the instruction memory and RAM respectively.

Instruction Interpretation (control):

Instructions are executed sequentially, incrementing by one from the program counter unless a jump or branch is taken. Each instruction is interpreted based on its op-code, and, if it is an R-type instruction, from its function code. These codes generate the controls for the ALU for operations, and controls for the RAM and register file read and/or write functions. The address for the ALU operands and RAM and register file read/write is also contained in each instruction.

The instructions also are decoded to set what data is put on the system buses. Data can be routed to the RAM and register file, and write control is for these modules is determined by the decoded instructions.

Execution (ALU):

The ALU for this lab must perform operations on 32-bit signed integers. There are 8 operations possible: no-operation, add, subtract, bitwise and, or, exclusive or, set on less than, and shift left logical. Inputs to the ALU are taken straight from the processor registers, and the ALU operation is determined by the function code from the current instruction. Flags from the ALU are used to determine Boolean checks for branch operations, and RAM addresses from fields in instructions. ALU results are put onto the system bus to be stored in the processor registers.

Memory Hierarchies:

There are three different memory modules used in the single cycle CPU: instruction memory, processor registers, and RAM. Instruction memory is initialized by the program loader and remains static through the execution of the instructions once the program loader is finished. Instructions can be accessed randomly on a branch or jump, but are usually executed sequentially. The processor registers behave in much the same way, storing data rather than instructions. However the processor registers are capable of being read from two locations at the same time. Simultaneous read and writes are possible for both the instruction memory and processor registers. The RAM is only able to store 16-bit words versus the 32-bit words for the instruction and processor registers. Access to the RAM is only read or write at one time.

Pipelined CPU:

Please refer to the corresponding diagrams in the appendix B.10 for a visual description of the tasks detailed below. Also refer to the function blocks section for descriptions of the modules used to implement each task.

Loading programs:

For the pipelined CPU, loading programs is implemented the same way as the single cycle CPU. A data loading flag tells the CPU to execute no-operation instructions into the pipeline until the program loader is finished.

Instruction Fetch:

Instructions are fetched from the instruction registers (same functionality as the single cycle CPU) in the first pipe of the pipelined CPU. The instruction is determined from the previous instruction, which is decoded (discussed below) and sends a control signal to determine

whether to simply increment to the next instruction in the registers, or jump or branch to another location in instruction memory.

Instruction Decode/ Data Retrieval:

The next pipe in the pipeline is the decoding of the instructions and reading from the processor registers (same functionality as the single cycle CPU). Instructions are decoded in much the same way as for the single cycle CPU, however the control signals generated are different. Data hazards are handled in the decoding of the instructions, and operands read from the processor registers are determined. Data read from processor registers are sometimes replaced with forwarded data in this pipe stage, depending on the data forwarding tasks discussed below. Addresses for the processor registers (and operands for the RAM addresses) and control signals for ALU functions are generated in this pipeline.

Execution:

The execution pipeline stage acts the same way as the execution of the operands from the single cycle CPU. Data is forwarded from this stage back to the previous Instruction Decode/Data Retrieval stage. This is discussed further in the data hazards/data forwarding below. Data from this stage of the pipeline is produced from the output of the ALU. Address for the RAM and register file writes, as well as control signals, are passed through this stage, essentially as a shift register.

Memory Hierarchy:

Like the single cycle CPU, the pipelined CPU has instruction registers, processor registers, and RAM. Unlike the single cycle CPU however the pipeline breaks up the reading and writing to the processor registers and RAM. In this final pipeline of the CPU, the RAM is accessed, either being written to, or being read from. If being read from, or if not being accessed at all, data from this stage is written directly to the processor registers. In other designs (see Patterson, Hennessy) an extra stage in the pipeline writes to the processor registers following a read from RAM. However our RAM is accessible in half of our system clock time, so we wrote directly into the processor registers from the RAM in the same cycle, cutting down on the number of stages by one, and thus saving much logic for data forwarding and hazards.

Data Hazards/Data Forwarding:

The data hazards include (as discussed in the previous design procedure section), R-type/RAM write data forwarding hazards, I-type RAM reading hazards, and jump/branch hazards.

Data forwarding is only from the execution pipe to the instruction decode/data retrieval pipe since the final memory pipeline writes directly into the processor registers and can be retrieved from them in one cycle. Forwarding is determined by saving the write address for the

processor registers from the previous instruction for one cycle, and checking this address against the current read addresses for the current instruction. If this previous write address matches one of the current read address then the data is forwarded to the instruction decode/data retrieval pipe.

The only data hazard that remained once the write back pipeline was removed from our design was the load word data hazard from needing an operand out of the RAM in the next instruction. In this case the control unit in the instruction decode/data retrieval pipe (which remembers the previous instruction and checks to see if the current instruction uses a word loaded from RAM in the previous instruction) then introduces an artificial no-operation into the pipe. This no-operation allows the RAM operand to be read into the processor registers in the next cycle and thus the next operation to be executed properly. This is a bubble that essentially stalls the execution sequence for one cycle. This could have been done at compile time, but we decided that implementing it in hardware was trivial.

Storing a word instruction data hazards were eliminated by the data forwarding unit, because data is forwarded from the execution unit like a regular R-type instruction. Since jump, jump register, and branch instructions are decoded, control signals generated, and next addresses calculated in the instruction decode/data retrieval pipe, then multiplexed in the previous instruction to determine the next address, bubbles in our instruction execution were not introduced from these operations in our design.

Functional Blocks

In the following sections we will provide an explanation of the software modules utilized to implement the single cycle processor and the pipelined processor, the top level block diagrams as well as the visual decomposition of each module can be found in appendices A.8-A.10 (single cycle) and B.8 – B.10 (pipelined).

Single Cycle CPU: (*Appendix A.8 – A.10)*

Program loader: The program loader module is composed of a finite state machine used to write the program instructions and the required data into the instruction memory and the SRAM respectively, it employs the use of a loadProgEn flag to indicate the processor to remain inactive (instruction adx constant at 0 and nop's executed continuously) while the instructions and data are being loaded.

Instruction memory: The instruction memory is a simple array of 128*32bit registers used to store the instructions required for the processor to execute a program, the implementation of the instruction memory allows for asynchronous reads but edge triggered writes.

Data Path Subsystem: The data path subsystem is instructed by control subsystem to route the data as required by the current instruction, it also computes the next instruction address based on sequential normal operation, jump instructions, jump register instructions or branch instructions.

Finally it receives commands from the program loader to halt the increment of the instruction address and keep the system idle under program loading operation.

Program Counter: The program counter is a 32 bit register used to store the next address of the instruction to be fetched, it executes synchronously with the positive edge of the clock.

Bus Driver: The bus driver is a resource managing module that allows for the implementation of a single system bus and a shared write enable signal for the processor registers, it is commanded by a control signal coming from the control subsystem that determines which module gets to write to the system bus, this control signal depends on either the current instruction or whether new instructions are being loaded into the instruction memory.

Memory Subsystem: The memory subsystem is most commonly known in computer architectures as the "main memory module", it provides for storing or reading capabilities from a 16 bit wide internal hardware SRAM memory. The implementation of this module was modified to be fully combinational to allow for single cycle access times for both reads and writes.

Processor Registers: This module is a 32 by 32 array of registers implemented in structural Verilog, 4 to 1 of 32 decoders were used at the input to provide for single register selections and two bus multiplexers (32*32 to 1*32) were used at the outputs to allow for dual registers read selection.

Control Subsystem: This module in simple terms describes the instruction decoder and the overall control of the processor. It provides the control flags needed for the data path, the bus driver, the ALU, the memory subsystem and the processor registers. Initially it looks at the instruction fields (op code fields and function field) and determines what kind of instruction is to be executed, from that point it sets the appropriate flags to finish the execute and write back states to complete the instruction execution, this module is also the one in charge of idling the system during instruction and data loading. A truth table was designed to implement the combinational logic needed to set the proper control flags based on the input instruction fields and program loader signals.

ALU Subsystem: The arithmetic unit from the ALU implements a carry look ahead adder for enhanced speed, it employs additional logic to compute the carry bits and feed them to the individual 4 bit adders instead of using the ripple carry adder design where the bits have to be carried from one stage to the next before the result can be produced, further it provides support for subtraction via the use of signed 2's complements addition. The logic unit from the ALU implements simple combinational logic for the bitwise OR, XOR and AND and makes use of a barrel shifter to implement the logic needed for the shift left operation.

Pipelined CPU: (*Appendix B.8 – B.10)*

***** Some software modules experienced very minor changes from the single cycle implementation; this is indicated in the module description

Program Loader: refer to single cycle design

Instruction Memory: refer to single cycle design

IFDPath: This module is the instruction fetch stage data path, it provides multiplexing capability to select between the next instruction address, the branch instruction address, the jump instruction address and the jump register instruction address via a control signal that comes from the control subsystem. It encapsulates the program counter and implements the adder to compute the next sequential address to execute under normal operation. Finally it is also commanded by the program loader to halt the system when instructions and data are being loaded into the system

IF/ID register: This module implements a simple register that buffers data from the instruction fetch stage to the instruction decode stage on every clock cycle.

Control Subsystem: This module implements the instruction decoder to be able to command other modules via control signals. It controls the operation of the ALU, the memory subsystem and the processor registers; its decoding operation was devised via the use of a truth table. It also implements the logic to detect potential hazards that cannot apply forwarding and acts promptly to recover gracefully by inserting a single stall and processing a nop instruction in the meanwhile.

Processor Registers: Refer to single cycle design

IDDPath: The instruction decode stage data path implements the logic to route either the data read from the register file or the forwarded data from the forwarding unit into bus A and/or bus B to be operated by the ALU on the next clock cycle, it is also in charge of computing the branch, jump, jump register and jump addresses to be sent to the instruction fetch stage data path as well as the instruction address select control signal, finally it also implements the early branch detection unit which calculates whether a branch needs to be taken and if so instructs the instruction fetch stage data path to select the branch address to avoid branch delays.

ID/EX register: This module simply buffers the data coming from the Instruction decode stage to be sent into the instruction execute stage; this is done synchronously every clock cycle.

ALU Subsystem: Refer to single cycle design

Forwarding Unit: The forwarding unit keeps track of data dependencies between adjacent instructions (via the use of a shift register to store old instructions and instruction field comparison to detect dependencies) and forwards data early from the output of the ALU result into the instruction decode stage so that it is ready to be operated on by the ALU on the next clock cycle.

EX/MEM register: Like other registers this register simply clocks the data from the execution stage into the memory stage of the pipelined system on every clock cycle.

Memory Subsystem: The description of this module remains similar to the one expressed above for the single cycle CPU with a few additions, in this pipelined system the memory subsystem was instructed via a 2 bit control signal that indicated whether data should be retrieved from the SRAM or buffered from the ALU (for R-type operations) or simply kept low during program loading operation.

Data and Control Flow

For a diagram of the single cycle CPU showing the dataflow and control paths in the system please refer to figure 1 in the system description.

The following diagram presents a high level view of a pipelined CPU with the data path and control paths differentiated by black and blue coloring respectively. Our system is a slight modification of this system, without the write back stage, the forwarding unit shifted back one pipe stage with one less forwarding condition, and a multiplexor in place of a program counter.

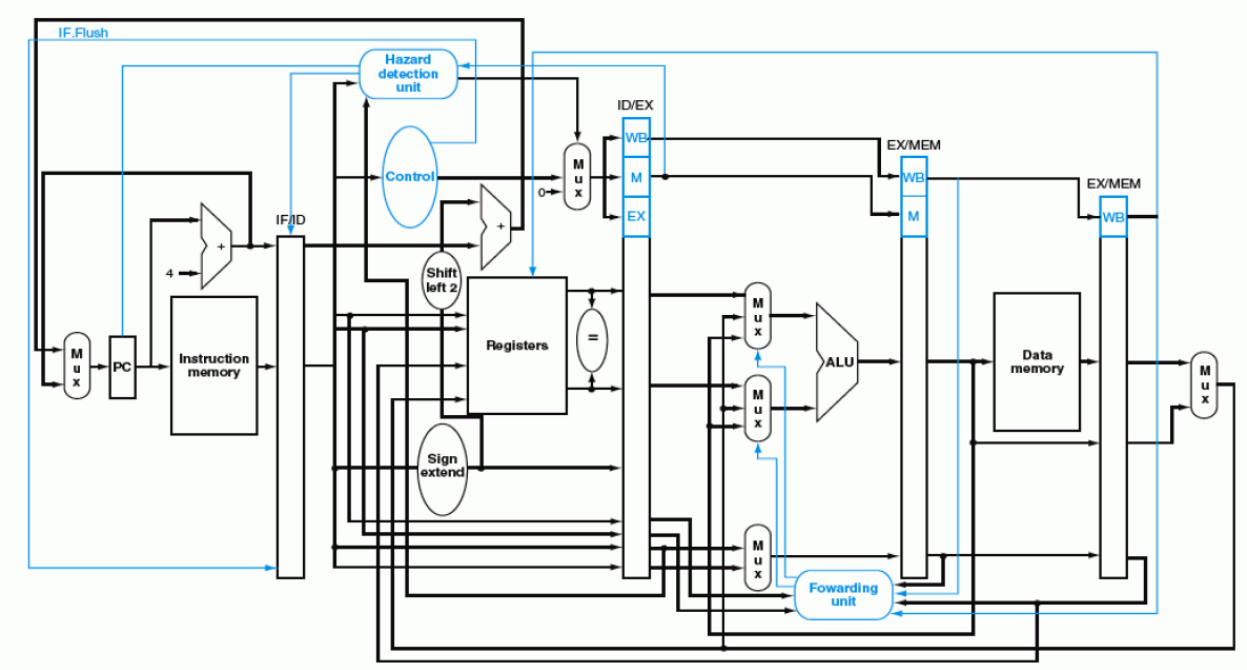


Figure 11: Data path (control) and control path (blue)

Hardware Implementation

Block Diagram

Discussion of Modules

Single Cycle CPU: (*Appendix A.8 – A.10)*

Program loader: The program loader module is implemented primarily with a software state machine. This is accomplished using a standard register array to hold the various state bits. However the program loader outputs a control signal when operational that forces the rest of the system to a no-op when active. The final state of the program loader is one in which the program loader turns off this control signal and sits in this state until a reset.

Instruction memory: The instruction memory was implemented as a two dimensional array of 128 32-bit registers. Multiplexors decode the input address and output the instruction on a read. The same multiplexor hardware is used for an input address and data to write to the instruction memory. A single input control signal is used as a positive edge trigger write for the instruction memory.

Data Path Subsystem: The data path subsystem is implemented primarily as multiplexors which use controls produced by the control subsystem as controls to determine the next instruction. This module requires an adder if the next instruction is simply an increment of the last, or outputs as the next address a branch or jump address. It will also not increment the address if the program loader control line is high by not adding or taking a branch/jump (done via a hardware multiplexor).

Program Counter: The program counter is a 32 bit shift register which simply shifts in the next instruction as determined by the data path subsystem every clock cycle (positive edge triggered registers). There is no logic in this module except for the reset condition to address 0

Bus Driver: The bus driver consists solely of a number of multiplexors which receive control signals from the control subsystem directing which data will appear on the system bus, and which module will be given control of the write signals for the RAM and the processor registers.

Memory Subsystem: The memory subsystem consists of a small amount of logic and multiplexors. An SRAM enable signal as well as a read (not write) signal are used to determine the output write signal for the SRAM. The module normally continually reads from the SRAM, though normally nothing is done with this data unless control signals indicate that something should be done with it. If the SRAM enable signal is high and the input read signal is low (indicating a write) then the module will toggle a write signal out to the SRAM while the system clock is low, in order to write the input data (assumed to be stable on the system bus by the time the clock signal becomes negative) into the SRAM. This module has a register to store an input address and input data so that it is stable one the lines when doing a write. This module also has a data out register to keep the previously read data stable on the lines.

Processor Registers: See the software implementation description for this module.

Control Subsystem: This module used multiplexors with sections of the current instruction as control. The input to these multiplexors where hardcoded control signals for all of the types of instructions we implemented in this CPU. These control signals are used in the other modules of this CPU and are described in the surrounding module's hardware descriptions.

ALU Subsystem: The arithmetic unit from the ALU implements a digital logic ALU described in our previous lab report "Designing an ALU" (Bowen, Lawrence, Tejeiro). Please reference this report for further explanation of the logic used to implement the addition, subtraction, and, or, xor, slt, and sll functions of the ALU.

Pipelined CPU: (*Appendix B.8 – B.10)*

***** Some software modules experienced very minor changes from the single cycle implementation; this is indicated in the module description

Program Loader: refer to single cycle design

Instruction Memory: refer to single cycle design

IFDPath: This module is essentially a multiplexor that multiplexes between a set of addresses determined in the instruction decode/ data retrieval pipe of our CPU. The control signal for this multiplexor is also determined in this pipe. This module also increments the last instruction outputted in the previous cycle, using a 32-bit adder. The output of this multiplexor is used as the input to the read address of the instruction memory module.

IF/ID register: This module is implemented with a simple 32 bit shift register. It accepts a new instruction into the register on every positive edge of the system clock.

Control Subsystem: This module is implemented with a series of multiplexors just as the control subsystem in the single cycle CPU was implemented. The control signals for this module are different however. This module also stores the previous instruction it accepted into a shift register, checking the current instruction's read address for the processor registers against the previous instruction's write register if it is a load word. It uses a bitwise comparator to do this. If such a hazard evaluates to true, then a hardcoded input multiplexor outputs a hazard signal to high, which changes the control signal to the IFDPath multiplexor so that the instruction does not increment, and forces a no-operation in its place through a multiplexor our of instruction memory.

Processor Registers: Refer to single cycle design

IDDPath: This module is a series of multiplexors which primarily determine which data goes into the execution stage into which operand slot. Control signals from both control subsystem and the forwarding unit determine the multiplexor's output. If the forwarding unit detects forwarding case, a signal from it allows data to be taken from the forwarding unit rather than out of the processor registers by way of multiplexors. It also reads data from the operand results directly in the case of branch greater than instructions, and performs a bitwise greater than

operation and sets a branch signal to high if a branch is taken (encoded into IFDPathControl signal). It also computes using adders and fields in the instruction or operands jump, jump register, and branch addresses.

ID/EX register: This module is implemented with a shift register.

ALU Subsystem: Refer to single cycle design

Forwarding Unit: The forwarding unit uses a shift register to store the previous instruction's write back address, and compares this with the current instructions read addresses using a bitwise comparison hardware. When either of the read addresses compare positively the previous instruction's write back address (and the read address is non zero) then a forwarding signal is output to high using a multiplexor and data from the execution unit is output to the IDDPath module.

EX/MEM register: Like other registers this module is simply a shift register with inputs and outputs.

Memory Subsystem: As discussed in the software implementation of this module, this module differs from the single cycle CPU implementation in that if a data read from the RAM is not being taken then data passed into this pipe is routed through this module as its output. This is accomplished by using a multiplexor that uses the SRAM enable signal as a control, and data from the RAM and data input into the modules as data inputs.

Design Choices

Most of the design choices made in this project were taken from the course textbook for design of a single cycle and pipelined CPU (see Patterson, Hennessy). The single cycle CPU was implemented exactly as the book directed.

For the pipelined CPU a few design choices were made by our team that differed from the book. The first was eliminating the final write back pipe stage altogether. We determined that a read from the RAM and into the processor registers could happen fast enough for the processor register outputs to propagate through IDDPath by the time the IDEX module clocks in this input. This design choice allowed us to cut out an entire data forwarding condition since data is no longer needed to be forwarded from the memory stage of the pipe. This design required us to forward data into the instruction decode/data retrieval stage rather than the execution stage, which only required us to multiplex output of the data forwarding unit with the output of the processor registers rather than the input of the CPU.

Also for the pipelined CPU, we chose to determine jump and branch conditions and addresses in the instruction decode/data retrieval stage and multiplex this directly with the determination of the next instruction address. This allows us to take a jump/branch immediately after such an instruction is executed, so that wrong instructions never enter the pipeline and thus

no data hazards present themselves for these instructions. This design choice depends on the assumption that necessary processor register/data forwarding data can be determined, multiplexed, added/compared, and multiplexed in IDDPath in one cycle. In our tests our system had no issue executing this logic in once clock cycle so we went with this design.

For the pipelined CPU we also used 16 bit words, sign extended as if they are positive integers since our testing never required the use of negative integers. This could be seen as a limitation of our system, or a positive aspect of our system if only positive, unsigned numbers are used. To implement signed numbers and unsigned numbers in 16 bit words, using every bit as part of the word, one could have choses certain blocks in the RAM as designated signed integer blocks and designated unsigned integer blocks, or handled this at compile time. For our system we determined that handling negative numbers was an unnecessary feature to meet the design specification for testing the module.

Testing

Test Plan

Single Cycle CPU:

To test the correct functioning of our Single cycle CPU, the most important factor that we will have to consider is that all instructions will have to be executed within a single clock cycle, once this timing requirement is met, we will have to proceed to test each of our processor sequential steps i.e. being able to fetch, decode, execute and write back, finally we will upload the complete compiled and assembled C test program (Appendix A.2) and make sure that our expected outputs match the values written back into memory at the end of the program execution

Pipelined CPU:

While the timing requirements for this implementation are more predictable than for the single cycle CPU due to the synchronous execution of each stage, we will still need to test that instructions propagate through the processor pipe in order. Further due to the parallel nature of this implementation we will need to make sure that all stages are constantly utilized while preventing stalls and bubbles. Finally just like before we will upload the complete compiled and assembled C test program (Appendix B.2) and make sure that our expected outputs match the values written back into memory at the end of the program execution

Test Specification

Single Cycle CPU:

As mentioned in the test plan for the single cycle CPU timing is crucial, however before we can meet the time requirements for this processor we need to have some test instructions to time the processor therefore the following test specifications will be executed:

- Test the correct functioning of the program loader by making sure the instructions and data being loaded into the instruction memory and main memory are what we expect
- Take a single instruction with known maximum propagation delay (Uses ALU, and accesses main memory) and make sure the timing requirements are met
- The next test here will be to identify the individual instruction being fetched, know the outputs of the different modules that will be involved in processing this instruction and make sure that these are correct
- Finally we will compare the stored execution results stored in main memory with those resulting from our hand calculations

Pipelined CPU:

The testing of the pipelined CPU will be somewhat similar to the testing done for the single cycle CPU since the loaded test programs will make use of a similar set of instructions; this is however with some caveats, the test specification is provided below:

- Test the correct functioning of the program loader by making sure the instructions and data being loaded into the instruction memory and main memory are what we expect
- Take a single initial lw instruction and trace its path as it progresses down the pipe, the instruction should propagate through each stage for every clock cycle, at the fourth clock cycle we should see the correct data being loaded into the register file
- Now we want to test the parallel capabilities of the system, so we will load multiple instructions and trace the first instruction as it progresses down the pipe (just like we did before), however at the 5^{th} and 6^{th} clock cycle we will make sure that the second and third instructions arrive at the memory stage, thus testing the pipelining capabilities of the system
- Next we will want to check the hazard detection unit and the forwarding unit, for this we will employ two initial simple tests, we will execute a code fragment (set of instructions) that includes a dependent R-type operation after a load word operation (hazard detection test) and then two interdependent R-type operations (forwarding unit test).
- Finally we will allow the system to run to complete execution and we will check that the data saved into memory agrees with our expectations.

Test Cases

Single Cycle CPU:

The following detailed test cases will be used to test our system:

1. Load the compiled program into the DE1 board, use signal tap to look at the signals coming out of the program loader to verify the correctness of the instructions. The following instructions should be written into the instruction memory (from Appendix A.2), use the machine code number in decimal format for comparison:

Instruction #	Assembly Instructions	Instruction Fields Format (decimal)	Machine Code (decimal)
0	lw $t0, $zero(0)	35,0,8,0	2349334528
1	lw $t1, $zero(1)	35,0,9,1	2349400065
2	lw $t2, $zero(2)	35,0,10,2	23439465602
3	lw $t3, $zero(3)	35,0,11,3	2349531139
4	lw $t4, $zero(4)	35,0,12,4	2349596676
5	sub $t5, $t0, $t1	0,8,9,13,0,34	17393698
6	lw $t6, $zero(5)	35,0,14,5	2349727749
7	bgt $t5, $t6, 3	5,13,14,3	363724803
8	sll $t2, $t2, 5	0,0,10,10,5,0	676160
9	lw $t4, $zero(24)	35,0,11,6	2349531142
10	j 56	2,13	134217741
11	lw $t2, $zero(7)	35,0,10,7	2349465607
12	sll $t3, $t3, 2	0,0,11,11,2,0	743552
13	sw $t0, $zero(0)	43,0,8,0	2886205440
14	sw $t1, $zero(1)	43,0,9,1	2886270977
15	sw $t2, $zero(2)	43,0,10,2	2886336514
16	sw $t3, $zero(3)	43,0,11,3	2886402051
17	sw $t4, $zero(4)	43,0,12,4	2886467588
18	lw $zero, $zero(0)	35,0,0,0	2348810240
19	lw $zero, $zero(1)	35,0,0,1	2348810241
20	lw $zero, $zero(2)	35,0,0,2	2348810242
21	lw $zero, $zero(3)	35,0,0,3	2348810243
22	lw $zero, $zero(4)	35,0,0,4	2348810244
23	jr 0	0,0,0,0,0,8	8
24	nop	0,0,0,0,0,0	0

Figure 12: Test instruction conversion table

2. Test the correctness of the data being stored into main memory, monitor the loader output lines using signal tap and check that the following numbers are being written into the SRAM (from Appendix A.3) at the indicated memory addresses:

MAIN MEMORY (Initial State)		
Memory Address	**Stored Data (Branch)**	**Stored Data (No Branch)**
0	9	7
1	5	5
2	2	2
3	4	4
4	3	3
5	3	3
6	7	7
7	6	6
8	0	0
9	0	0
10	0	0
11	0	0
12	0	0
13	0	0
14	0	0
15	0	0
16	0	0
17	0	0
18	0	0
19	0	0
20	0	0
21	0	0
22	0	0
23	0	0
24	0	0

Figure 13: Main memory initialization state

3. Now test that the first instruction coming from instruction memory (lw, uses ALU for address computation and accesses main memory for data) is executed on a single clock cycle, this will be done using signal tap with a trigger immediately after the loader has finished.
4. The next test will need to assess the correct execution of the instructions, this test will be done recursively starting with a set of instructions and then following with the next, in this case we will start with the initial load words instructions and focus our attention on both the register file write select line and the write data line, the values on these lines seen through signal tap should match the chosen register from the instruction destination fields (for write select) and the expected loaded value for the write data line, iterative testing and debugging over a series of instructions using this technique should yield satisfactory results.
5. Finally the last test to perform is to compare the values written back to the SRAM with those expected from our hand calculations, the expected updated values are the following (From Appendix A.4)

MAIN MEMORY (Final State)		
Memory Address	**Stored Data (Branch)**	**Stored Data (No Branch)**
0	9	7
1	5	5
2	6	64
3	16	7
4	3	3
5	3	3
6	7	7
7	6	6
8	0	0
9	0	0
10	0	0
11	0	0
12	0	0
13	0	0
14	0	0
15	0	0
16	0	0
17	0	0
18	0	0
19	0	0
20	0	0
21	0	0
22	0	0
23	0	0
24	0	0

Figure 14: Main memory final state

Pipelined CPU:

As mentioned before the steps required to test the pipelined CPU are somewhat similar to those mentioned before, the required test cases are the following:

1. The program loader should be tested as mentioned before; load the compiled program into the DE1 board, use signal tap to look at the signals coming out of the program loader to verify the correctness of the instructions and the data. The following instructions and data should be written into the instruction memory and main memory (from Appendix B.2), use the machine code number in hexadecimal format for comparison:
   Loaded Instructions:

				Instruction Fields							Machine Code
C Code	MIPS Code	IAdx	Type	OC	Rs	Rt	R/S/F/A				
int A =6;	sw 0(0), 6	0	PL	//Via Program Loader//							
int B =4;	sw 0(1), 4	1	PL								
int C =2;	sw 0(2), 2	2	PL								
int D =4;	sw 0(3), 4	3	PL								
int* dPtr =&D;	sw 0(4), 0(3)	4	PL								
unsigned int E =0x7676;	sw 0(5), 30326	5	PL								
unsigned int F =0xA5A5;	sw 0(6), 42405	6	PL								
unsigned int G =0xFF;	sw 0(7), 255	7	PL								
unsigned int H =0xC3;	sw 0(8), 195	8	PL								
int const0 =3	sw 0(9), 3	9	PL								
int const1 =4	sw 0(10), 4	10	PL								
int const2 =7	sw 0(11), 7	11	PL								
	noop	0	R	0	0	0	0	0	0	No Op:	
//Load A	lw $s0, 0(0)	1	I	35	0	16	0			Load Data:	8C100000
//Load B	lw $s1, 0(1)	2	I	35	0	17	1				8C110001
//Load C	lw $s2, 0(2)	3	I	35	0	18	2				8C120002
//Load D	lw $s3, 0(3)	4	I	35	0	19	3				8C130003
//Load E	lw $s4, 0(5)	5	I	35	0	20	5				8C140005
//Load F	lw $s5, 0(6)	6	I	35	0	21	6				8C150006
//Load H	lw $s7, 0(8)	7	I	35	0	23	8				8C170008
//Load Constant: 3	lw $t0, 0(9)	8	I	35	0	8	9				8C080009
//Load Constant: 4	lw $t1, 0(10)	9	I	35	0	9	10				8C09000A
//Load Constant: 7	lw $t2, 0(11)	10	I	35	0	10	11				8C0A000B
//Subtract: A - B	sub $t3, $s0, $s1	11	R	0	16	17	11	0	34	Main:	02115822
if((A-B)>3{	bgt $t3, $t1, IAdx[19]	12	I	5	11	8	17				15680011
C =C +4;	add $s2, $s2, $t1	13	R	0	18	9	18	0	32	Branch1:	02499020
D =C - 3;	sub $s3, $s2, $t0	14	R	0	18	8	19	0	34		02489822
G =E | F;	or $t4, $s4, $s5	15	R	0	20	21	12	0	37		02956025
//Jump out of Branch1	j IAdx[21]	16	J	2	20						08000014
C =C <<3;	sll $s2, $s2, 3	17	R	0	18	0	18	3	0	Branch2:	024090C0
*dPtr =7	add $s3, $zero, $t2	18	R	0	0	10	19	0	32		000A9820
G =E & F;}	and $t4, $s4, $s5	19	R	0	20	21	12	0	36		02956024
A =A +B;	add $s0, $s0, $s1	20	R	0	16	17	16	0	32	Main:	02118020
//XOR: E ^F	xor $t5, $s4, $s5	21	R	0	20	21	13	0	38		2956826
G =(E ^F) & H;	and $t5, $t5, $s7	22	R	0	13	23	13	0	36		01B76824
//Store A	sw 0(0), $s0	23	I	43	0	16	0			Store Data:	AC100000
//Store C	sw 0(2), $s2	24	I	43	0	18	2				AC120002
//Store D	sw 0(3), $s3	25	I	43	0	19	3				AC130003
//Store G	sw 0(7), $t5	26	I	43	0	13	7				AC0D0007

Figure 15: Instruction table

Loaded Data:

MAIN MEMORY (Initial State)		
Memory Address	**Stored Data (Branch)**	**Stored Data (No Branch)**
0	8	6
1	4	4
2	2	2
3	4	4
4	3	3
5	30326	30326
6	42405	42405
7	255	255
8	195	195
9	3	3
10	4	4
11	7	7
12	0	0
13	0	0
14	0	0
15	0	0
16	0	0
17	0	0
18	0	0
19	0	0
20	0	0
21	0	0
22	0	0
23	0	0
24	0	0
25	0	0
26	0	0

Figure 16: Loaded Data table

2. Now as expected, the following test will require us to take the second instruction (the lw, since the first one is a nop) and trace it as it progresses through the pipe, analyzing the correct fetching and updating of the instruction adx, the subsequent decoding by the control block and propagation of data by the IDDatapath, then in the next clock cycle we should see the ALU calculate the memory address (which in this case is just 0) that will be used to read from the SRAM, finally in the fourth clock cycle we should see the data at address 0 coming out of the sram and being written into address 16 in the processor registers. This test will provide a simple routine to follow when tracing data through the pipe.
3. For the following test we will perform the previous routine but instead of stopping after the first instruction we should make sure that during the 5th and 6th clock cycle the 2nd and 3rd instructions load the values 4 and 2 into locations 17 and 18 in the register file, after proving that this test works similar tests should be employed for subsequent instruction blocks.
4. The last test of checking the final values updated in main memory with those expected from our hand calculations certifies that our pipeline processor behaves correctly, the expected values updated in memory are the following (from Appendix B.4)

MAIN MEMORY (Final State)		
Memory Address	**Stored Data (Branch)**	**Stored Data (No Branch)**
0	12	10
1	4	4
2	16	6
3	7	3
4	3	3
5	30326	30326
6	42405	42405
7	195	195
8	195	195
9	3	3
10	4	4
11	7	7
12	0	0
13	0	0
14	0	0
15	0	0
16	0	0
17	0	0
18	0	0
19	0	0
20	0	0
21	0	0
22	0	0
23	0	0
24	0	0
25	0	0
26	0	0

Figure 17: Final memory state following testing

Results

Results

Relevant results for this project involve those gathered from testing for the correct operation of the program loader, those gathered from testing for the correct execution of instructions by the single CPU and the pipelined CPU, and finally the results of assessing the time taken for each

processor to execute a common set of instructions (CPU time). The following sections describe the results obtained from testing our systems.

Program Loader:

The results obtained from testing the program loaders in our system can be found in Appendix A.5 (single cycle CPU program loader) and in Appendix B.5 (Pipelined CPU program loader). As can be seen from the timing diagrams for both loaders the instruction writeadx signal is incremented by 1 sequentially and the appropriate instruction is written for every address, the same timing holds for loading the values to operate on to the SRAM; the SRAM memoryadx signal is incremented by 1 sequentially while the appropriate values are written into memory.

Single Cycle CPU:

The results from testing our single cycle CPU for both branching (on condition true) operations and non-branching (on condition false) operations are displayed in Appendix A.6 and Appendix A.7 respectively.

Branching: Focusing first on the branching timing diagram, it is helpful to recognize the correct sequential increases in the instruction adx with the respective instructions coming out of the instruction memory on every clock cycle. It is also important to notice the correct execution of the initial load instructions; this can be seen by looking at the values being written back into the processor registers as they are clocked in by the write enable signal. Finally but most importantly we can see the results from the subtraction R-type operation yielding the branch condition; this can be seen by looking at the "alu data" result from the subtraction of the values on Bus A and Bus B, further upon taking the branch the new instruction address specified in the instruction field can be seen displayed in the instruction adx line.

Non-Branching: Now taking a look at the non-branching timing diagram Appendix A.7, we can see that the execution of the first instructions remain unchanged as we would expect. The turning point however is upon the arrival to the subtract instruction where it can be seen that the alu data result (2) is smaller than the condition value of 3, therefore as we would expect the branch is not taken and execution remains sequential until the jump instruction is reached and we jump from instruction 10 to instruction 13.

Finally the results of executing the programs to be stored back into memory are highlighted at the end of the timing diagrams and displayed on the system bus line and can be compared with the expected results from the testing section.

Pipelined CPU:

The results from testing our pipelined CPU for both branching (on condittion true) operations and non-branching (on condition false) operations are displayed in Appendix B.6 and Appendix B.7 respectively.

Branching: Important results from testing the branching operation of our pipelined CPU can be seen by looking at the highlighted fields in the pipelined CPU Branching timing diagram (Appendix B.6). The first important timing sequence is found by noticing the sequential increments in the instructionADXPCI line with the respective instructions seen in the instruction_Out field. Another important result is being able to see that the instructions are executing in a pipelined manner, this can be seen by looking at the desired register write address in the BusBid finally propagate to the input of the register file write address line (in the write back stage) after two intermediate clock cycles. The next important result is seeing that the processor actually took the branch, this can be seen by looking at the green highlighted fields; the value in BusAid (4) which is forwarded (controlBusA = 1) from the previous subtraction ALU result is greater than that on BusBid(3) thus we take the branch as can be seen by the branch address and the updated instructionADXPCI value, another result that is relevant is found by looking at the forwarding between interdependent R-type instructions (instructionADXPCI = 23) where you can see the forwarding unit in action forwarding data from the XOR calculation result as one of the operands for the AND operation. The last important result and practically the most important is checking that the resultant updated values being written back to memory after the program is done are correct when compared to the expected values displayed in the testing section (this can be seen from looking at the highlighted 'memWriteDataO' signal at the end of the timing diagram).

Non-Branching: The initial results from testing the non-branching execution of the pipelined CPU are similar to those under branching operation as well as the execution of the interdependent AND logical operation after the XOR operation, the only difference comes when the decision to take the branch needs to be taken. Looking at the timing diagram (Appendix B.7) we can see that the branch detection hazard promptly reacts using the forwarded data from the previous subtraction instruction to prevent the processor from taking the branch (refer to the green highlighted timing sequence while looking at the instructionADXCPI which does not branch and keeps incrementing sequentially). Finally again we can check the values being written back into memory (by looking at the 'memWriteDataO' bus highlighted at the end of the timing snapshot) and compare them with the expected updated values from the testing section to conclude that these agree and our results are correct.

Performance Measure:

Single Cycle Computer: The CPU time for a common program executed for this CPU is computed below:

$$SPI_{SC} = \frac{seconds}{instruction} = \frac{1}{\frac{1\,instruction}{cycle} * 25\,M\,cycles} = 40\,nS$$

$$CPU_{timeSC} = ¿\, instructions * SPI_{SC}$$

Where:

$$¿\, instructions = 25$$

For a common program executed on both CPU's

$$\therefore CPU_{timeSC} = 40\, nS * 25 = 1\, uS$$

Pipelined Computer: Running the same program in the pipelined processor without including unrecoverable hazards (dependent R-type operations after lw's) yields the following results:

The cycle time for the execution of the program in the pipelined CPU will be 4 clock cycles (4 pipelined stages) for the first instruction and a single clock cycle for every instruction thereafter, therefore the total number of cycles for executing this program.

$$TIC_{PP} = total\ instruction\ cycles = 4 + (¿\, instructions - 1) = 4 + 24 = 28$$

$$\therefore CPU_{timePP} = TIC_{PP} * T_{clk} = 28 * 40\, nS = 1.12\, uS$$

As can be seen from this reasoning if there were to be unrecoverable hazards our TICpp will increment by one for each of these hazards making the pipelined processor even less ideal. In spite of these drawbacks the pipelined processor has some other important advantages over the single cycle processor; In fact, at current clock speeds some types of operations (division, multiplication, floating point operations) are very hard to implement in a single cycle and designing a pipelined CPU for this kind of instructions might be desirable and might provide more predictable timing for the system.

Error Analysis

After rigorously testing the finished design, all apparent errors in the system have been addressed and no longer exist. During debugging, it was found that several of the systems previously designed contained minor design flaws that were previously undetected during testing of the other subsystems, but were needing to be corrected when integrating the IMEM and instruction data into the existing system. These previously undetected quirks in the subsystems were not bugs in the coding of the individual systems, but were just not fully synchronized to align with the integration of the new subsystem. Some tweaking of the subsystem coding was done to remedy this alignment synchronization issue. For example, we were using data that was stored in the SRAM as 32 bit words, this was changed to 16 bit words that were sign extended when they

were loaded into the registers for processing. This decreased the wait time required before the data was accessible in the registers.

One bug that posed a problem for us for a while prior to being resolved was: specifically for the XOR instruction, Quartus forced the Opcode to an unspecified value regardless of what we had programmed to represent the XOR operation. Due to this, the remaining instructions were never ran because the error would disrupt the running of the CPU. After several trial and error to resolve the issue or to drive the value to our desired output, we resigned to adjusting the value to that of what Quartus itself was assigning our Opcode to. This is fixed the issue and allowed the CPU to operate and execute the remaining instructions. We are still unclear as to why Quartus forced us to utilize the specific value for XOR since none of the other Opcodes were affected this way.

Summary

This project designed the instruction set for a single cycle computer with MIPS architecture CPU and pipelining the instructions by building upon the previously designed subsystems of the Register File, ALU, and Data Flow Bus modules, completing the overall hardware design and inserting a Forwarding Unit and Hazard Detection operations within the Control module. New instructions: LW, SW, J, JR, and BGT, were add to the already existing operations in the ALU: NOP, ADD, SUB, AND, OR, XOR, SLT, SLL, and the corresponding machine code for each new instruction and the corresponding control signals have been added and implemented into the overall design. The instructions used to be executed within the design were originally written in C Language code and were hand compiled into MIPS, then subsequently converted into machine code and integrated into the CPU.

Conclusion

In conclusion, the tying in of the instruction memory (IMEM), using hand compiled C code instructions, and pipelining the operations of the CPU posed a difficult task when attempting debugging. The objectives of this project were to use the previously designed and integrate all of the subsystems from the subsequent projects, still utilizing the Cyclone II FPGA chip on the Altera DE1 board with the Quartus IDE software environment, and implement a specific set of instructions through a pipelined CPU. Five new operations: Load Word (LW), Store Word (SW), Jump (J), Jump Register (JR), and Branch when Greater Than (BGT), were added to the existing ALU subsystem so we would be able to execute a series of instructions assigned to the new memory register array IMEM contained within the Control subsystem, as well as utilizing the Forwarding Unit and Hazard Detection functions. The instructions are originally designed in C Language, and were hand compiled into MIPS instructions and then converted into machine code instructions. The machine code was then loaded into the control system for execution.

Appendix A. Single Cycle CPU

A.1 Single Cycle CPU Instruction Set

				SINGLE CYCLE CPU INSTRUCTION SET					
Type	**Name**	**Mnemonic**	**Op Code (31:26)**	**Rs (25:21)**	**Rt (20:16)**	**Rd (15:11)**	**Shamt (10:6)**	**Function (5:0)**	**Operation**
	No operation	nop	0	0	0	0	0	0	no operation
R-type	Add	add	0	Rsource	Rt	Rdestination	0	32	Rd ⇐ Rs +Rt
R-type	Subtract	sub	0	Rsource	Rt	Rdestination	0	34	Rd ⇐ Rs - Rt
R-type	And	and	0	Rsource	Rt	Rdestination	0	36	Rd ⇐ Rs & Rt
R-type	Or	or	0	Rsource	Rt	Rdestination	0	37	Rd ⇐ Rs | Rt
R-type	Xor	xor	0	Rsource	Rt	Rdestination	0	38	Rd ⇐ Rs ^Rt
R-type	Set Less Than	slt	0	Rsource	Rt	Rdestination	0	42	if (Rs <Rt) Rd ⇐ 1 else Rd ⇐ 0
R-type	Shift Left Logical	sll	0	Rsource	0	Rdestination	shamt	0	Rd ⇐ Rs <<shamt
R-type	Jump Register	jr	0	Rsource	0	0	0	8	jump to adx stored in Rs
I-type	Load Word	lw	35	Rsource	Rt	Immediate Offset (16 bits)			load (Rs +offset) into Rt
I-type	Store Word	sw	43	Rsource	Rt	Immediate Offset (16 bits)			store Rt into (Rs +offset)
I-type	Branch Greater Than	bgt	5	Rsource	Rt	Immediate Address to Branch To (16 bits)			if (Rs >Rt) PC ⇐ PC +1 else branch to Address*
J-type	Jump	j	2	Desired Address to jump to (26 bits)					jump to adx passed in the instruction
* this implementation was reversed for the pipelined CPU									

A.2 Test Instructions/Test program loaded into instruction memory

		INSTRUCTION MEMORY		
Instruction #	**Assembly Instructions**	**Instruction Fields Format (decimal)**	**Machine Code (decimal)**	**Comments**
0	lw $t0, $zero(0)	35,0,8,0	2349334528	load A into $t0
1	lw $t1, $zero(1)	35,0,9,1	2349400065	load B into $t1
2	lw $t2, $zero(2)	35,0,10,2	23439465602	load C into $t2
3	lw $t3, $zero(3)	35,0,11,3	2349531139	load D into $t3
4	lw $t4, $zero(4)	35,0,12,4	2349596676	load D* into $t4
5	sub $t5, $t0, $t1	0,8,9,13,0,34	17393698	subtract A - B
6	lw $t6, $zero(5)	35,0,14,5	2349727749	load constant 3 into $t6
7	bgt $t5, $t6, 3	5,13,14,3	363724803	A-B > 3
8	sll $t2, $t2, 5	0,0,10,10,5,0	676160	C = C<<5
9	lw $t4, $zero(24)	35,0,11,6	2349531142	load constant 7 into $t4
10	j 56	2,13	134217741	jump out
11	lw $t2, $zero(7)	35,0,10,7	2349465607	load constant 6 into $t2
12	sll $t3, $t3, 2	0,0,11,11,2,0	743552	D = D<<2
13	sw $t0, $zero(0)	43,0,8,0	2886205440	Update A in main memory
14	sw $t1, $zero(1)	43,0,9,1	2886270977	Update B in main memory
15	sw $t2, $zero(2)	43,0,10,2	2886336514	Update C in main memory
16	sw $t3, $zero(3)	43,0,11,3	2886402051	Update D in main memory
17	sw $t4, $zero(4)	43,0,12,4	2886467588	Update D* in main memory
18	lw $zero, $zero(0)	35,0,0,0	2348810240	debugging Purposes Only
19	lw $zero, $zero(1)	35,0,0,1	2348810241	debugging Purposes Only
20	lw $zero, $zero(2)	35,0,0,2	2348810242	debugging Purposes Only
21	lw $zero, $zero(3)	35,0,0,3	2348810243	debugging Purposes Only
22	lw $zero, $zero(4)	35,0,0,4	2348810244	debugging Purposes Only
23	jr 0	0,0,0,0,0,8	8	jump back to start of program
24	nop	0,0,0,0,0,0	0	execute a nop

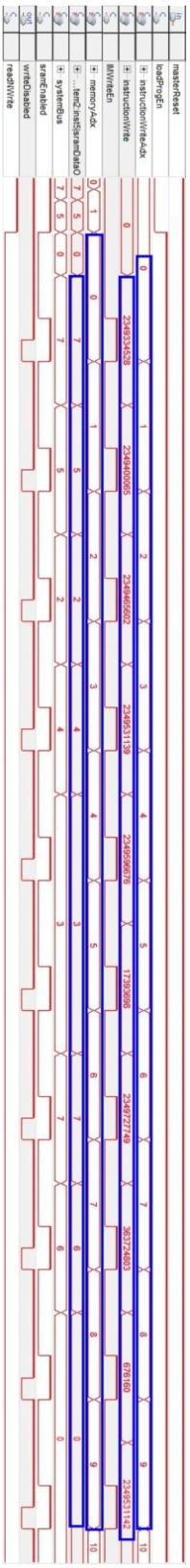
A.3 Test Data/values loaded into main memory

MAIN MEMORY (Initial State)		
Memory Address	**Stored Data (Branch)**	**Stored Data (No Branch)**
0	9	7
1	5	5
2	2	2
3	4	4
4	3	3
5	3	3
6	7	7
7	6	6
8	0	0
9	0	0
10	0	0
11	0	0
12	0	0
13	0	0
14	0	0
15	0	0
16	0	0
17	0	0
18	0	0
19	0	0
20	0	0
21	0	0
22	0	0
23	0	0
24	0	0

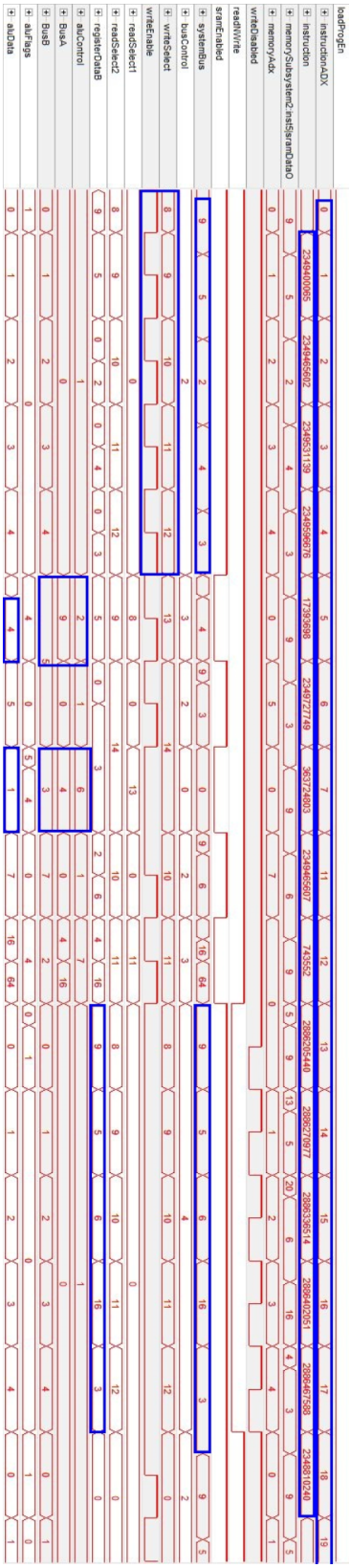
A.4 Expected Results / Final values stored in main memory

MAIN MEMORY (Final State)		
Memory Address	**Stored Data (Branch)**	**Stored Data (No Branch)**
0	9	7
1	5	5
2	6	64
3	16	7
4	3	3
5	3	3
6	7	7
7	6	6
8	0	0
9	0	0
10	0	0
11	0	0
12	0	0
13	0	0
14	0	0
15	0	0
16	0	0
17	0	0
18	0	0
19	0	0
20	0	0
21	0	0
22	0	0
23	0	0
24	0	0

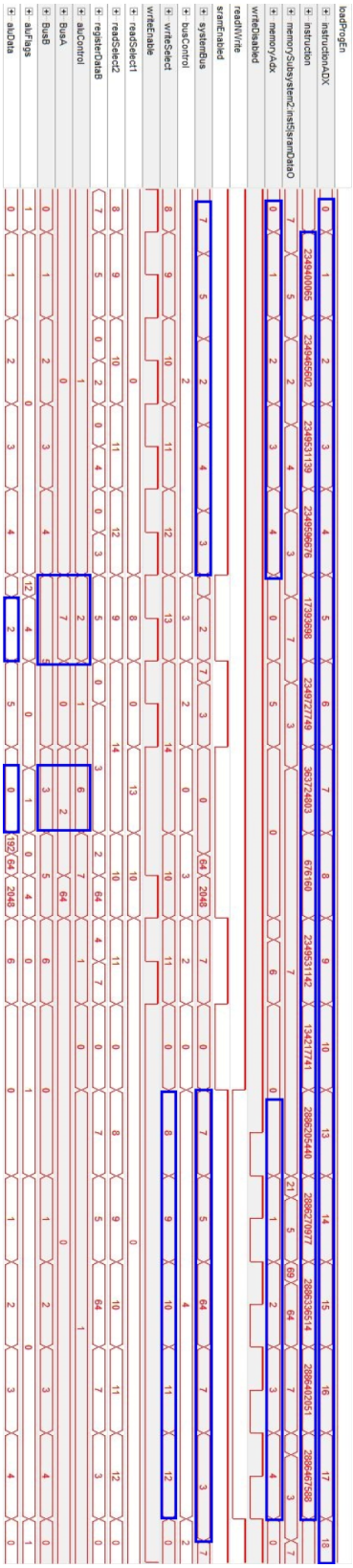
A.5 Timing Diagrams, Program Loader



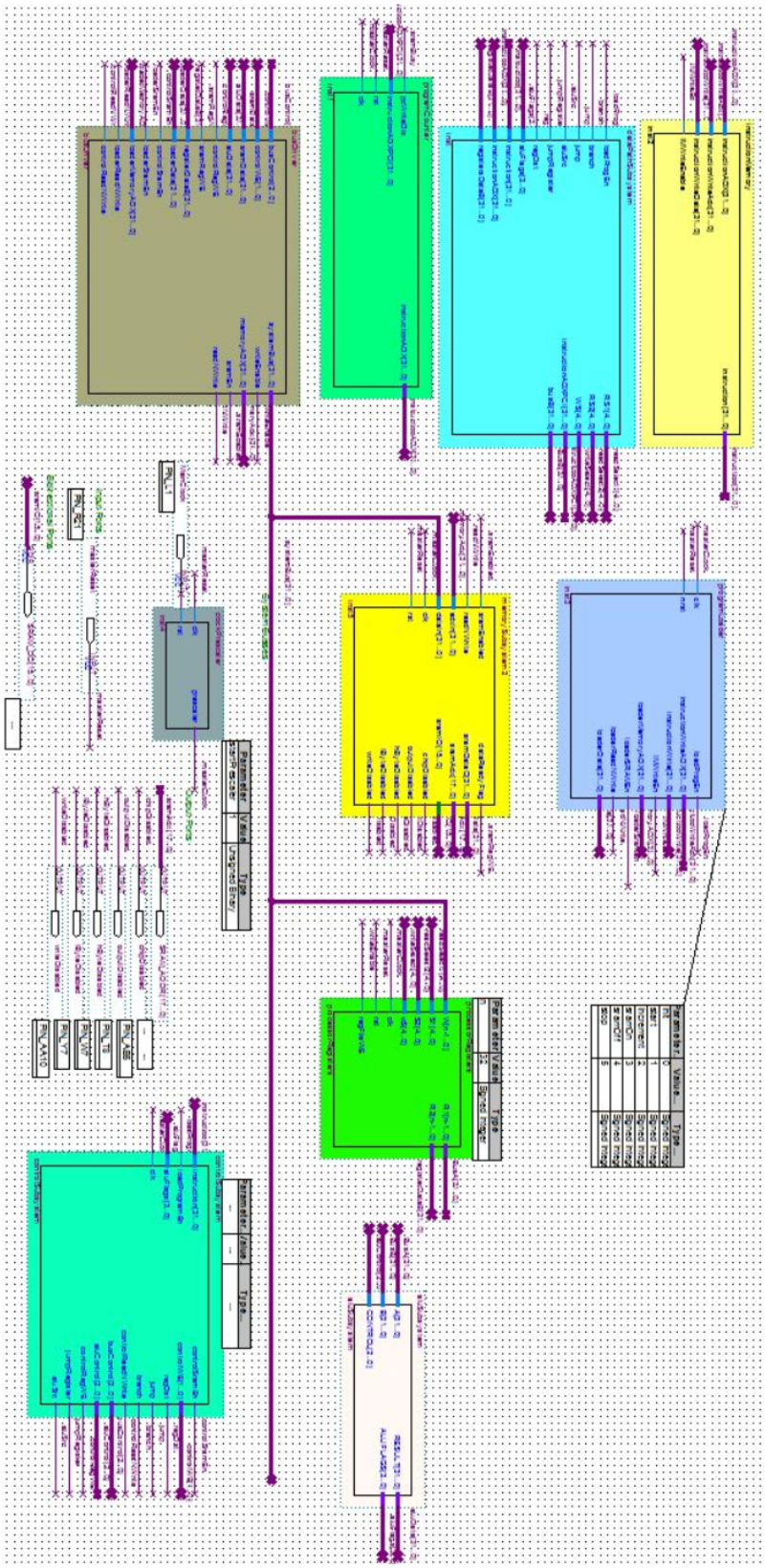
A.6 Timing Diagrams, Processor under Test (Branch)



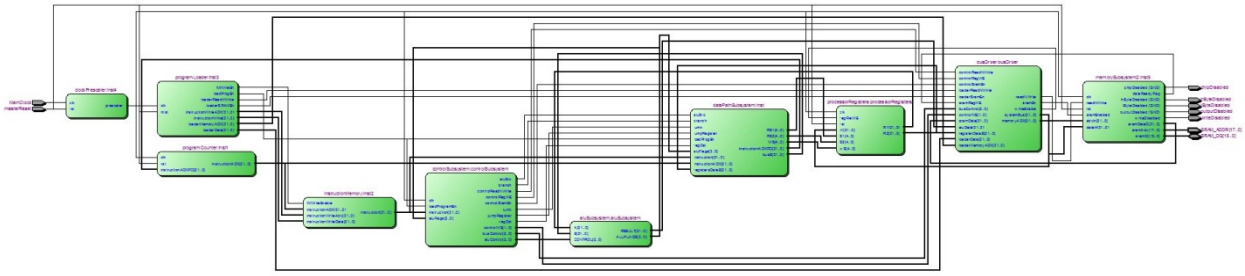
A.7 Timing Diagrams, Processor under Test (No Branch)



A.8 Top Level Block Diagram, Architecture

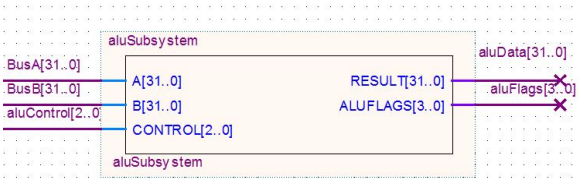


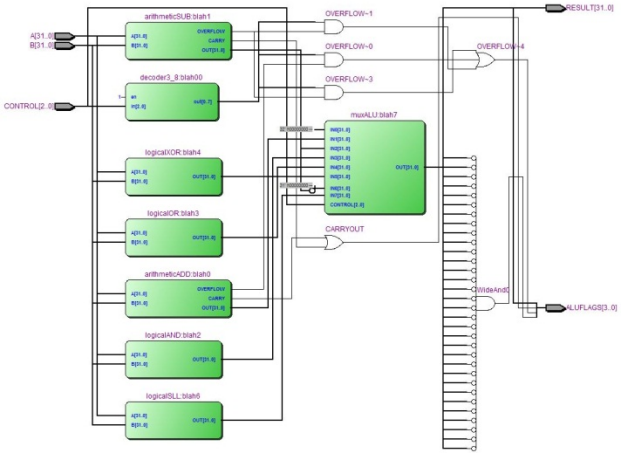
A.9 Top Level RTL Diagram



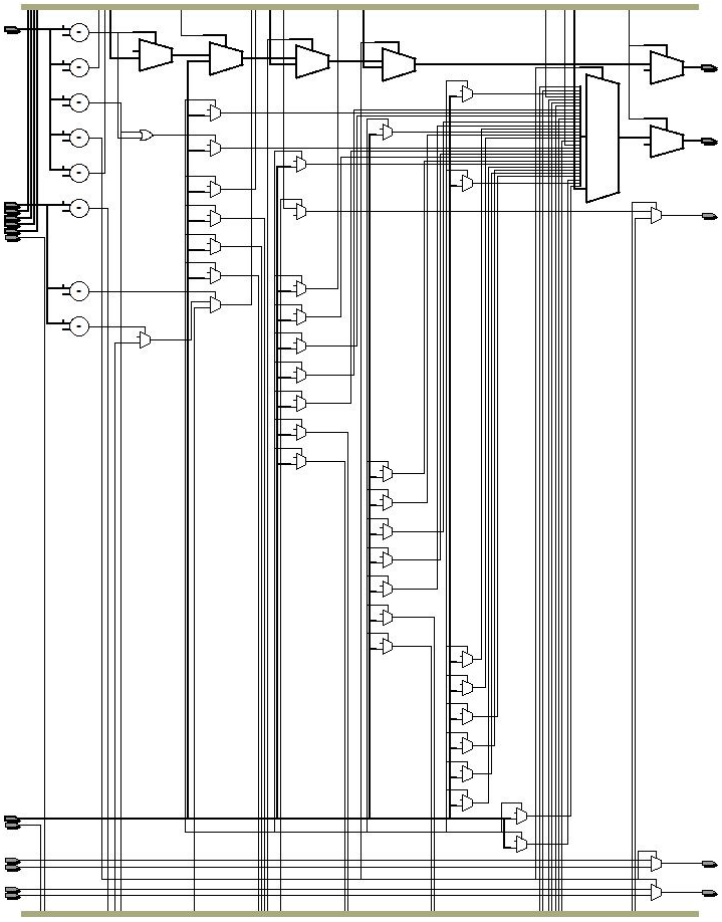
A.10 Individual Modules Block/RTL Diagram

ALU Subsystem

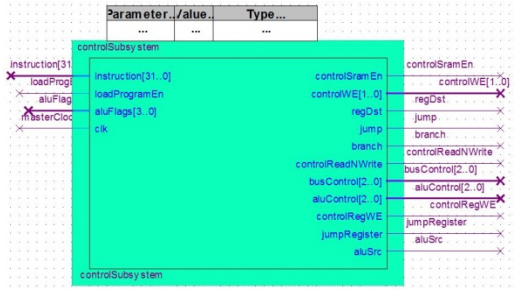


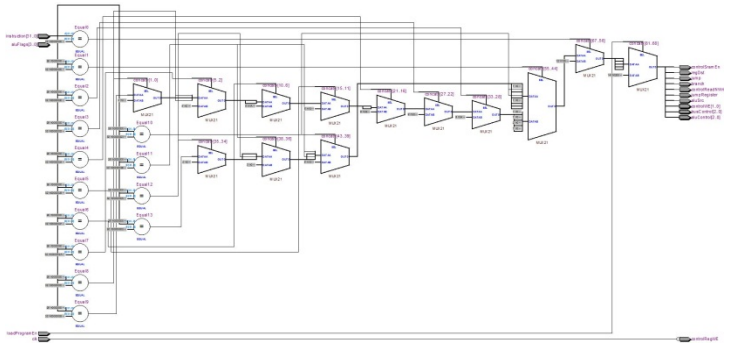


Bus Driver

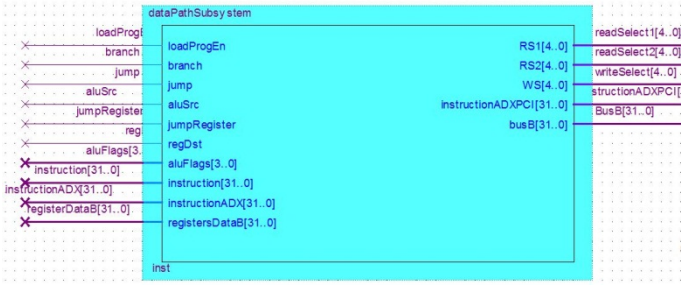


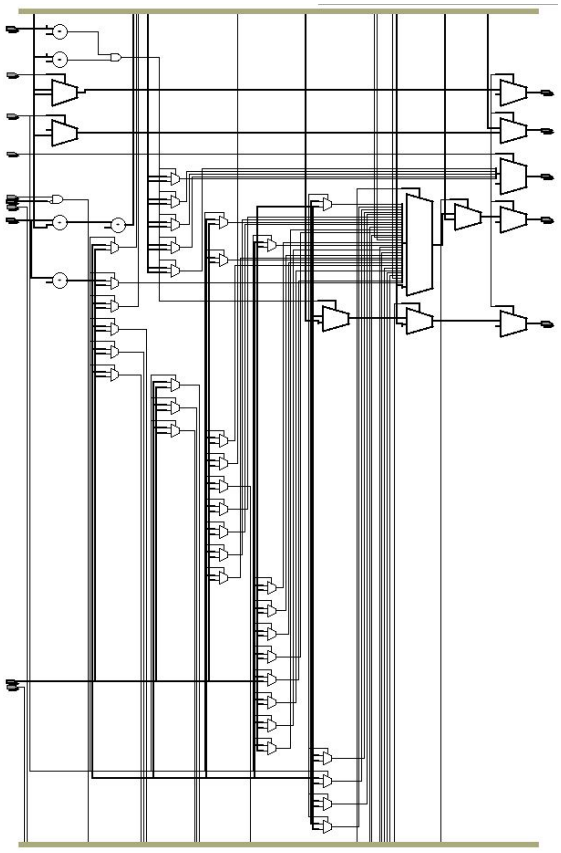
Control Subsystem



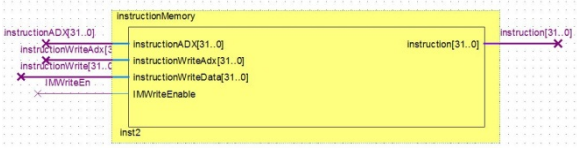


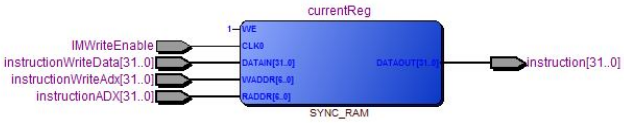
Data path Subsystem



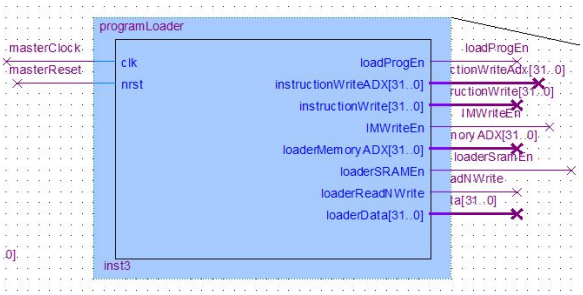


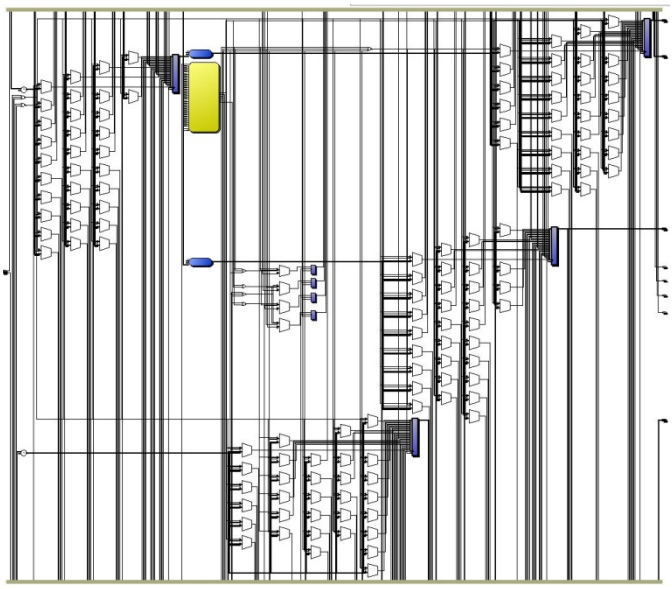
Instruction Memory



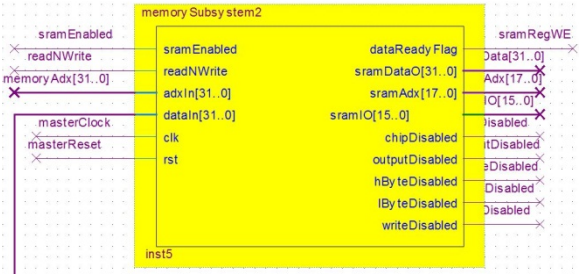


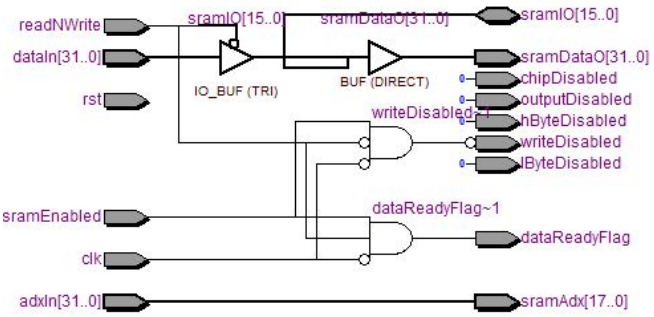
Program Loader



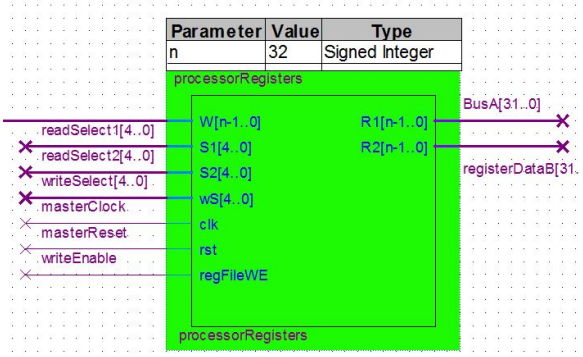


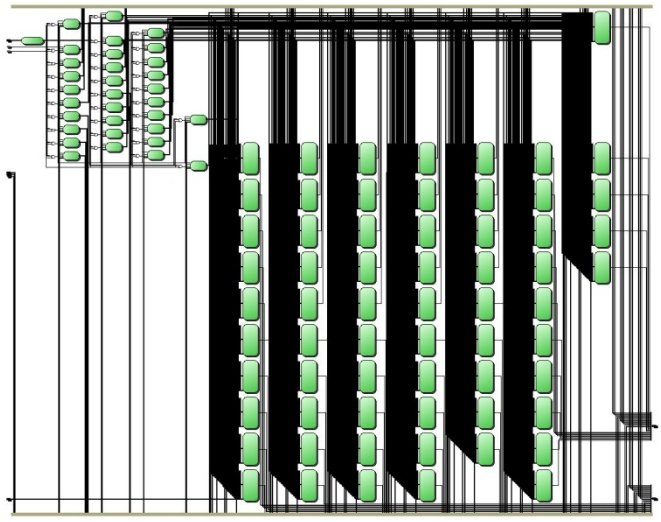
Memory Subsystem



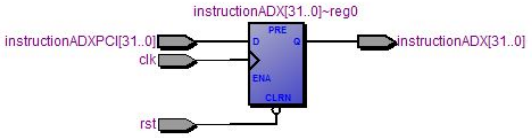


Processor Registers

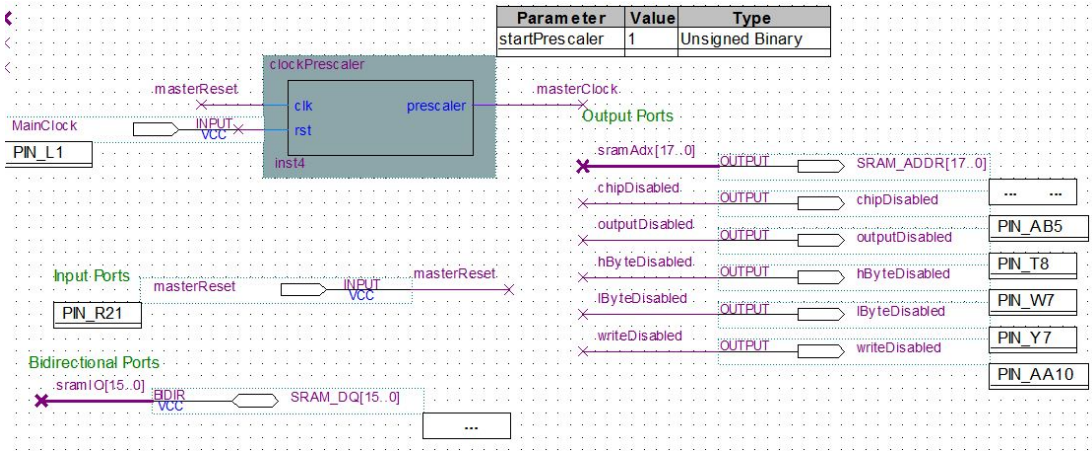




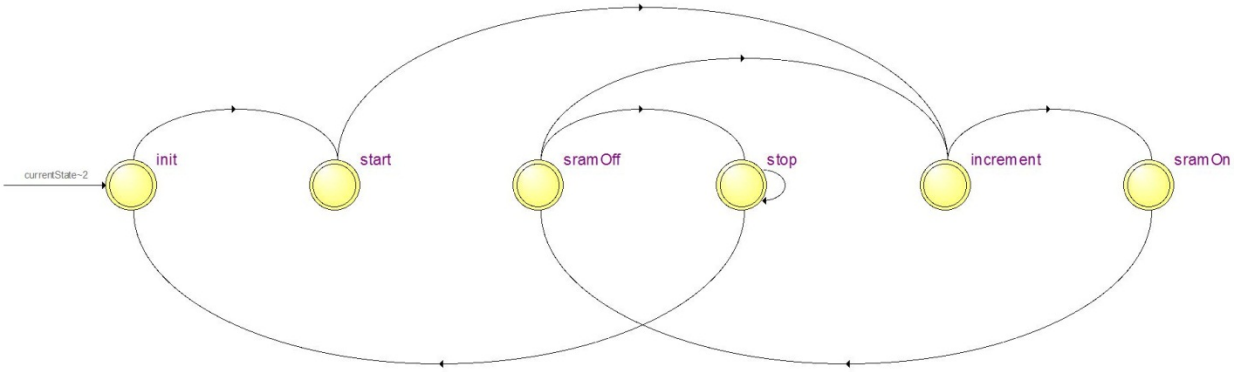
Program Counter



Input / Output Ports



A.11 Program Loader State Machine



Appendix B. Pipelined CPU

B.1 Pipelined CPU Instruction Set

				PIPELINED CPU INSTRUCTION SET					
Type	**Name**	**Mnemonic**	**Op Code (31:26)**	**Rs (25:21)**	**Rt (20:16)**	**Rd (15:11)**	**Shamt (10:6)**	**Function (5:0)**	**Operation**
	No operation	nop	0	0	0	0	0	0	no operation
R-type	Add	add	0	Rsource	Rt	Rdestination	0	32	Rd <= Rs +Rt
R-type	Subtract	sub	0	Rsource	Rt	Rdestination	0	34	Rd <= Rs - Rt
R-type	And	and	0	Rsource	Rt	Rdestination	0	36	Rd <= Rs & Rt
R-type	Or	or	0	Rsource	Rt	Rdestination	0	37	Rd <= Rs | Rt
R-type	Xor	xor	0	Rsource	Rt	Rdestination	0	38	Rd <= Rs ^Rt
R-type	Set Less Than	slt	0	Rsource	Rt	Rdestination	0	42	if (Rs <Rt) Rd <= 1 else Rd <= 0
R-type	Shift Left Logical	sll	0	Rsource	0	Rdestination	shamt	0	Rd <= Rs <<shamt
R-type	Jump Register	jr	0	Rsource	0	0	0	8	jump to adx stored in Rs
I-type	Load Word	lw	35	Rsource	Rt	Immediate Offset (16 bits)			load (Rs +offset) into Rt
I-type	Store Word	sw	43	Rsource	Rt	Immediate Offset (16 bits)			store Rt into (Rs +offset)
I-type	Branch Greater Than	bgt	5	Rsource	Rt	Immediate Address to Branch To (16 bits)			if (Rs >Rt) branch to adx else PC <= PC +1
J-type	Jump	j	2	Desired Address to jump to (26 bits)					jump to adx passed in the instruction

B.2 Test Instructions/Test program loaded into instruction memory

				Instruction Fields							Machine Code
C Code	MIPS Code	IAdx	Type	OC	Rs	Rt	R/S/F/A				
int A =6;	sw 0(0), 6	0	PL	//Via Program Loader//							
int B =4;	sw 0(1), 4	1	PL								
int C =2;	sw 0(2), 2	2	PL								
int D =4;	sw 0(3), 4	3	PL								
int* dPtr =&D;	sw 0(4), 0(3)	4	PL								
unsigned int E =0x7676;	sw 0(5), 30326	5	PL								
unsigned int F =0xA5A5;	sw 0(6), 42405	6	PL								
unsigned int G =0xFF;	sw 0(7), 255	7	PL								
unsigned int H =0xC3;	sw 0(8), 195	8	PL								
int const0 =3	sw 0(9), 3	9	PL								
int const1 =4	sw 0(10), 4	10	PL								
int const2 =7	sw 0(11), 7	11	PL								
	noop	0	R	0	0	0	0	0	0	No Op:	
//Load A	lw $s0, 0(0)	1	I	35	0	16	0			Load Data:	8C100000
//Load B	lw $s1, 0(1)	2	I	35	0	17	1				8C110001
//Load C	lw $s2, 0(2)	3	I	35	0	18	2				8C120002
//Load D	lw $s3, 0(3)	4	I	35	0	19	3				8C130003
//Load E	lw $s4, 0(5)	5	I	35	0	20	5				8C140005
//Load F	lw $s5, 0(6)	6	I	35	0	21	6				8C150006
//Load H	lw $s7, 0(8)	7	I	35	0	23	8				8C170008
//Load Constant: 3	lw $t0, 0(9)	8	I	35	0	8	9				8C080009
//Load Constant: 4	lw $t1, 0(10)	9	I	35	0	9	10				8C09000A
//Load Constant: 7	lw $t2, 0(11)	10	I	35	0	10	11				8C0A000B
//Subtract: A - B	sub $t3, $s0, $s1	11	R	0	16	17	11	0	34	Main:	02115822
if((A-B)>3{	bgt $t3, $t1, IAdx[19]	12	J	5	11	8	17				15680011
C =C +4;	add $s2, $s2, $t1	13	R	0	18	9	18	0	32	Branch1:	02499020
D =C - 3;	sub $s3, $s2, $t0	14	R	0	18	8	19	0	34		02489822
G =E|F;	or $t4, $s4, $s5	15	R	0	20	21	12	0	37		02956025
//Jump out of Branch1	j IAdx[21]	16	J	2	20						08000014
C =C <<3;	sll $s2, $s2, 3	17	R	0	18	0	18	3	0	Branch2:	024090C0
*dPtr =7	add $s3, $zero, $t2	18	R	0	0	10	19	0	32		000A9820
G =E & F; }	and $t4, $s4, $s5	19	R	0	20	21	12	0	36		02956024
A =A +B;	add $s0, $s0, $s1	20	R	0	16	17	16	0	32	Main:	02118020
//XOR: E ^F	xor $t5, $s4, $s5	21	R	0	20	21	13	0	38		2956826
G =(E ^F) & H;	and $t5, $t5, $s7	22	R	0	13	23	13	0	36		01B76824
//Store A	sw 0(0), $s0	23	I	43	0	16	0			Store Data:	AC100000
//Store C	sw 0(2), $s2	24	I	43	0	18	2				AC120002
//Store D	sw 0(3), $s3	25	I	43	0	19	3				AC130003
//Store G	sw 0(7), $t5	26	I	43	0	13	7				AC0D0007

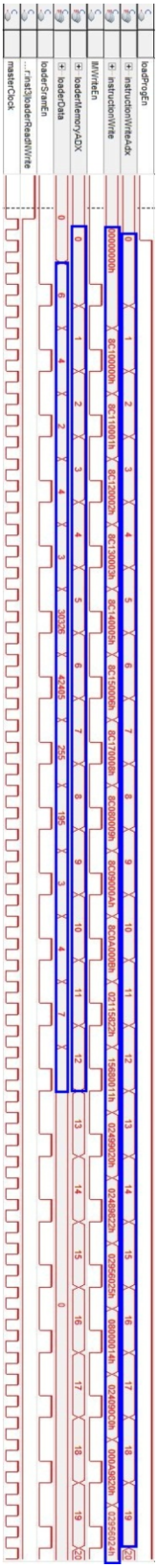
B.3 Test Data/values loaded into main memory

MAIN MEMORY (Initial State)		
Memory Address	**Stored Data (Branch)**	**Stored Data (No Branch)**
0	8	6
1	4	4
2	2	2
3	4	4
4	3	3
5	30326	30326
6	42405	42405
7	255	255
8	195	195
9	3	3
10	4	4
11	7	7
12	0	0
13	0	0
14	0	0
15	0	0
16	0	0
17	0	0
18	0	0
19	0	0
20	0	0
21	0	0
22	0	0
23	0	0
24	0	0
25	0	0
26	0	0

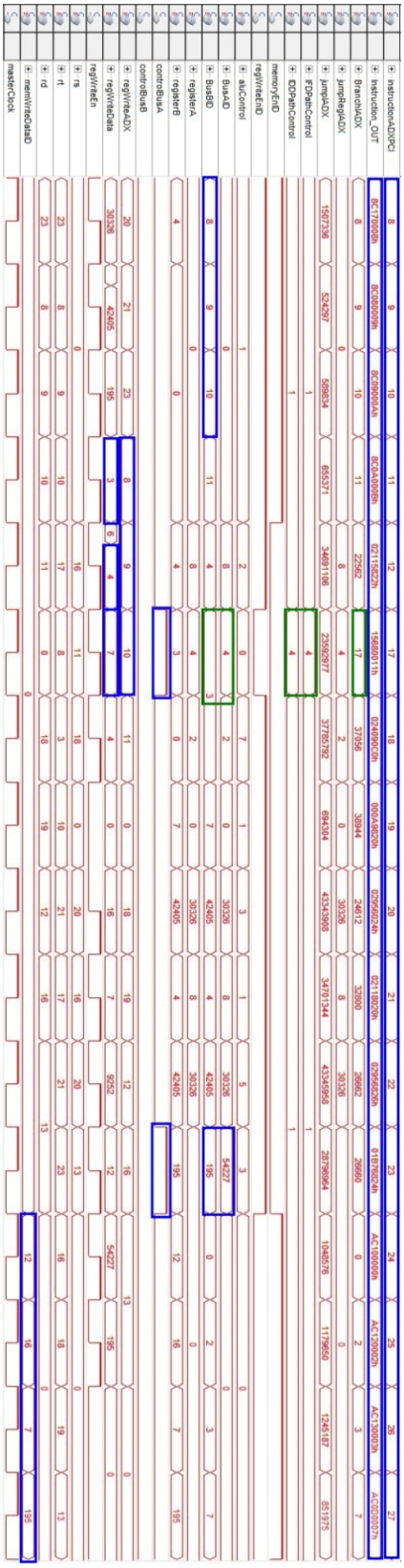
B.4 Expected Results / Final values stored in main memory

MAIN MEMORY (Final State)		
Memory Address	**Stored Data (Branch)**	**Stored Data (No Branch)**
0	12	10
1	4	4
2	16	6
3	7	3
4	3	3
5	30326	30326
6	42405	42405
7	195	195
8	195	195
9	3	3
10	4	4
11	7	7
12	0	0
13	0	0
14	0	0
15	0	0
16	0	0
17	0	0
18	0	0
19	0	0
20	0	0
21	0	0
22	0	0
23	0	0
24	0	0
25	0	0
26	0	0

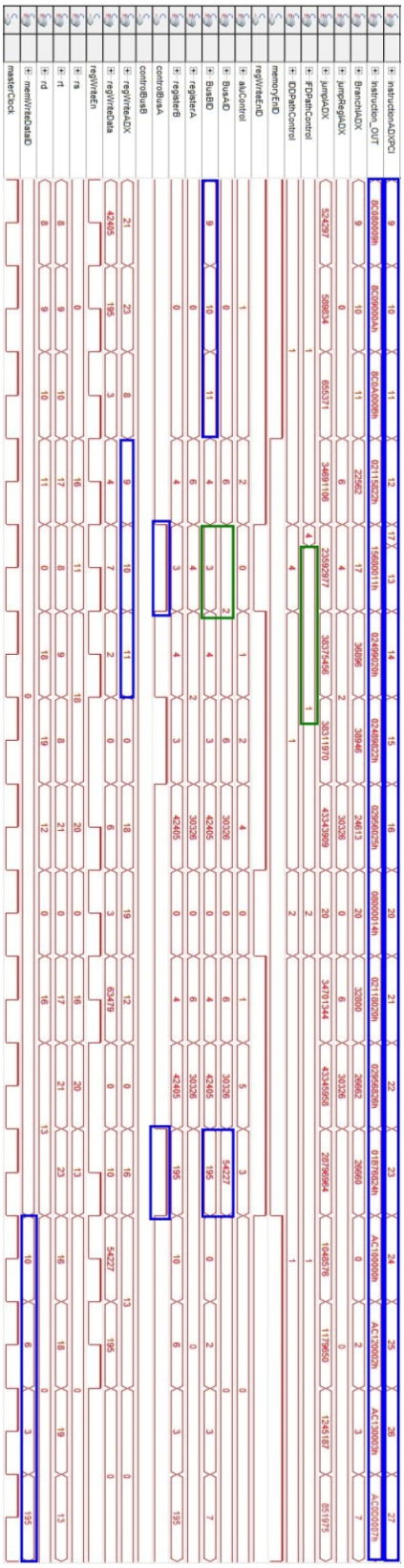
B.5 Timing Diagrams, Program Loader



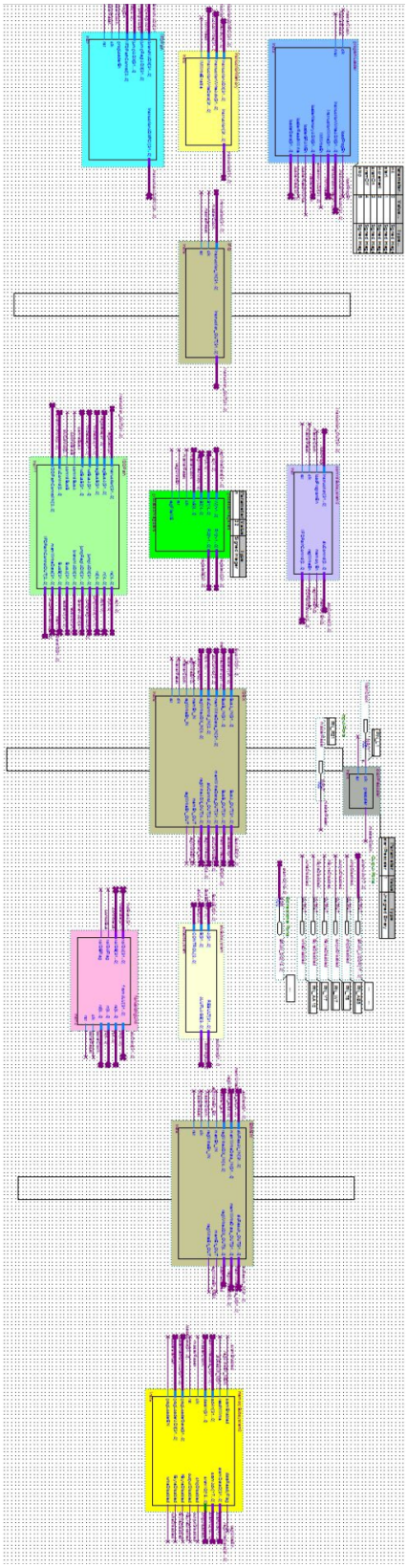
B.6 Timing Diagrams, Processor under Test (Branch)



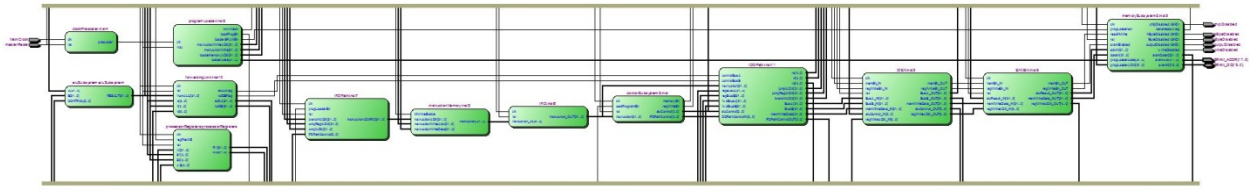
B.7 Timing Diagrams, Processor under Test (No Branch)



B.8 Top Level Block Diagram, Architecture

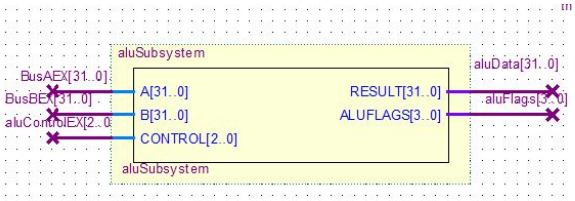


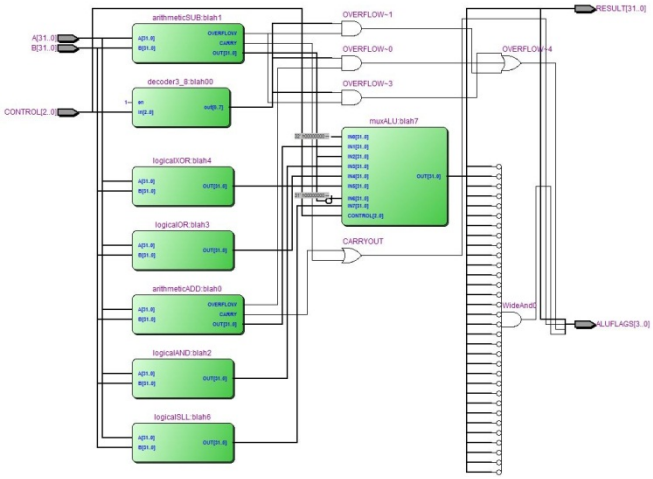
B.9 Top Level RTL Diagram



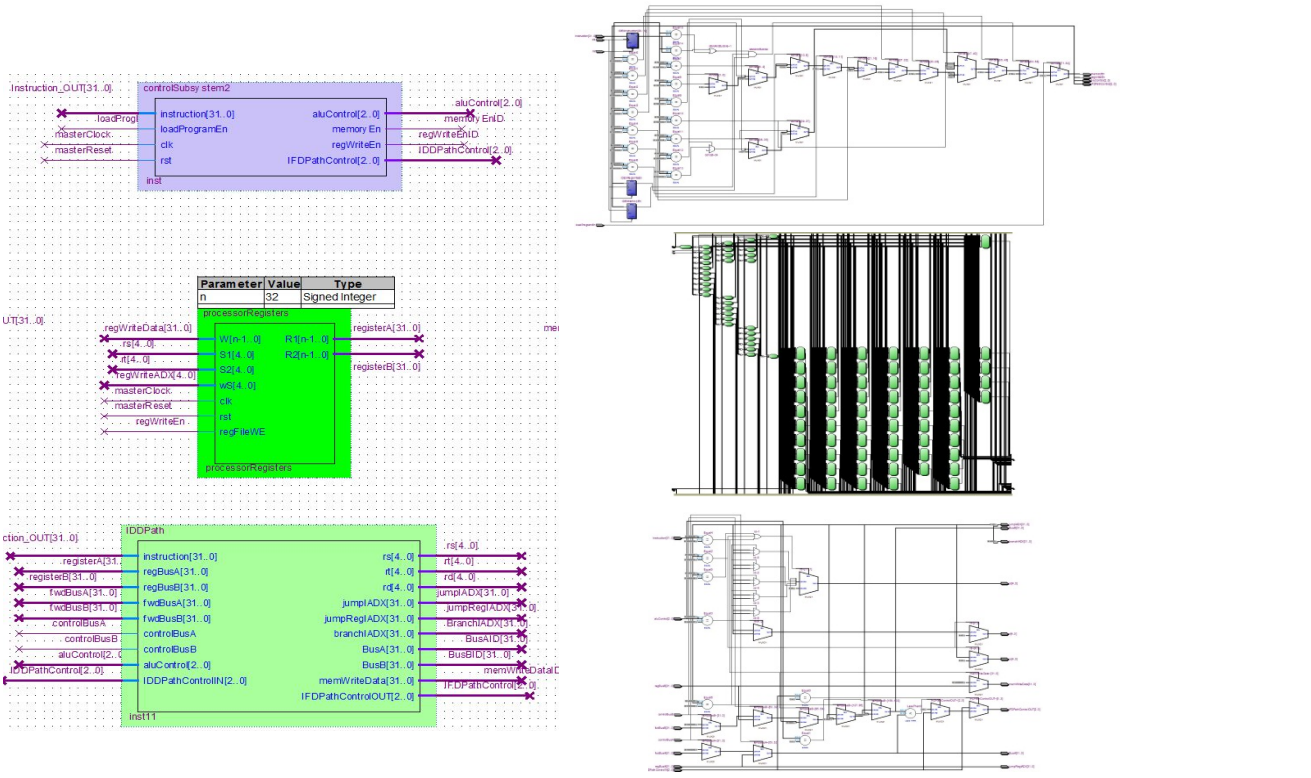
B.10 Individual Modules Block/RTL Diagram

Execute Stage

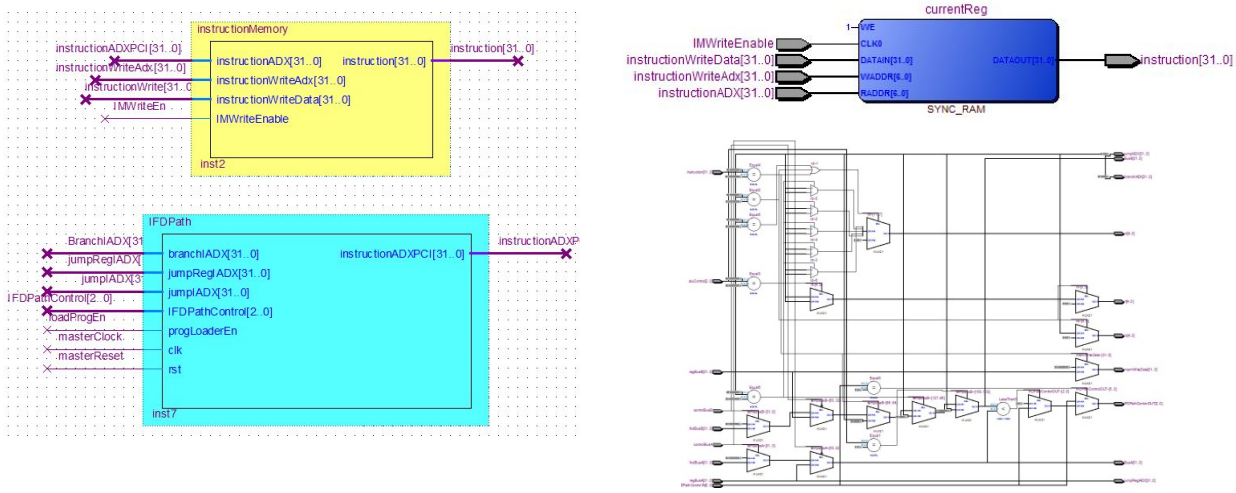




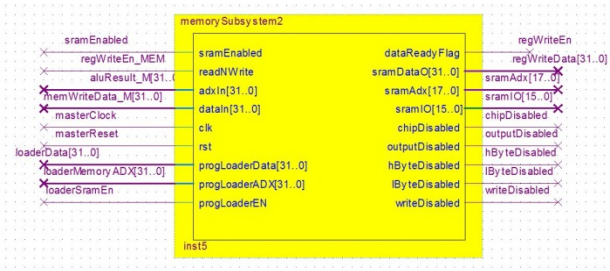
Instruction Decode Stage

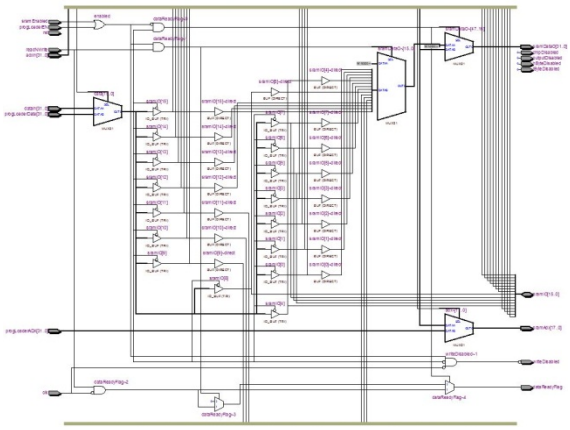


Instruction Fetch Stage

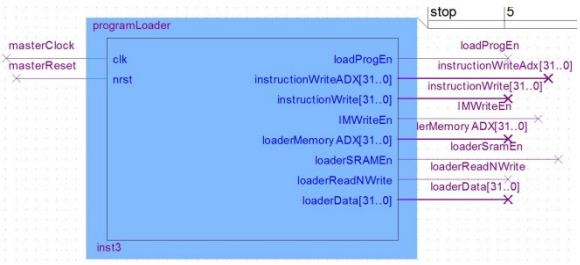


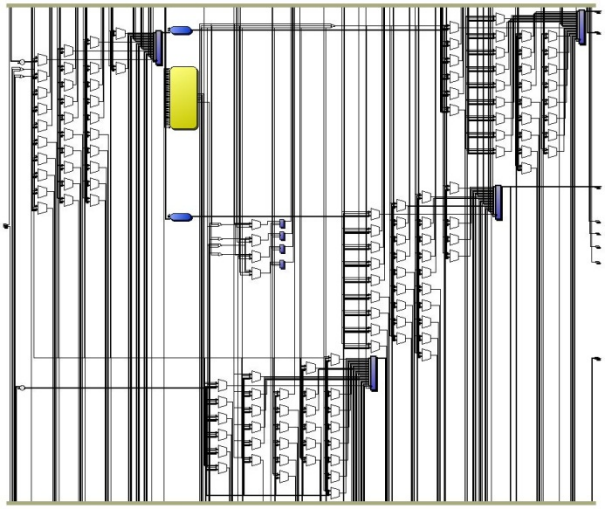
Memory Stage



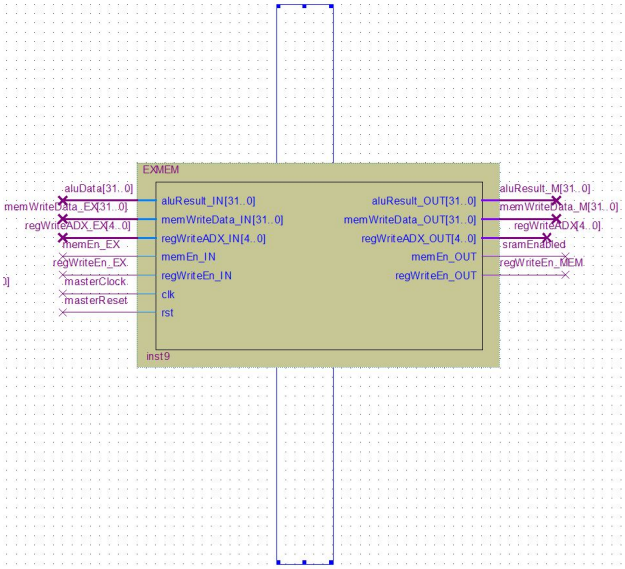


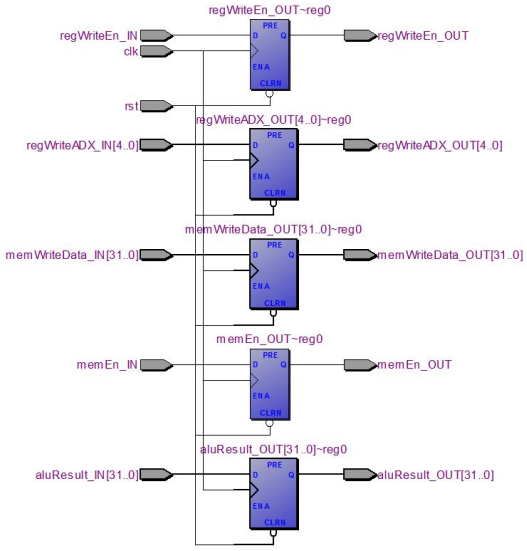
Program Loader



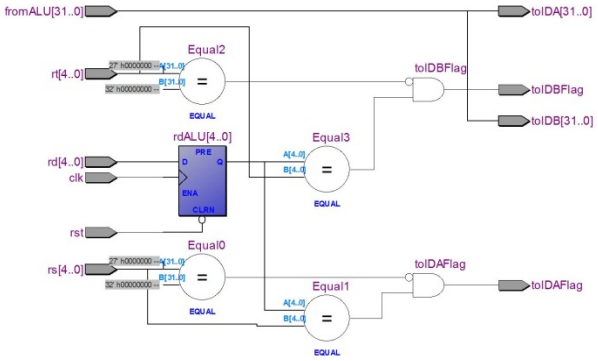


EX/MEM Register

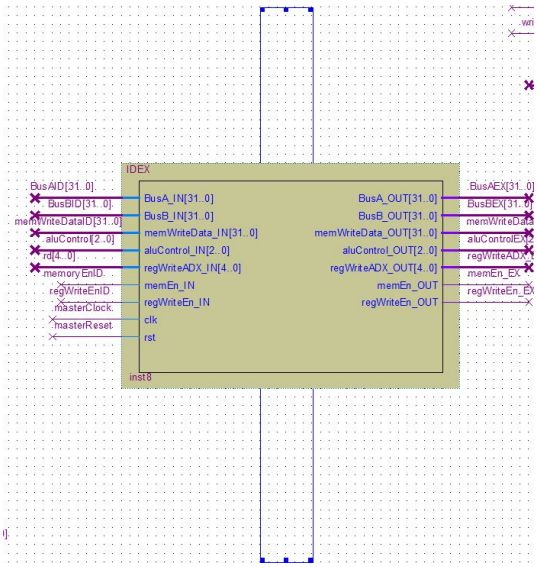


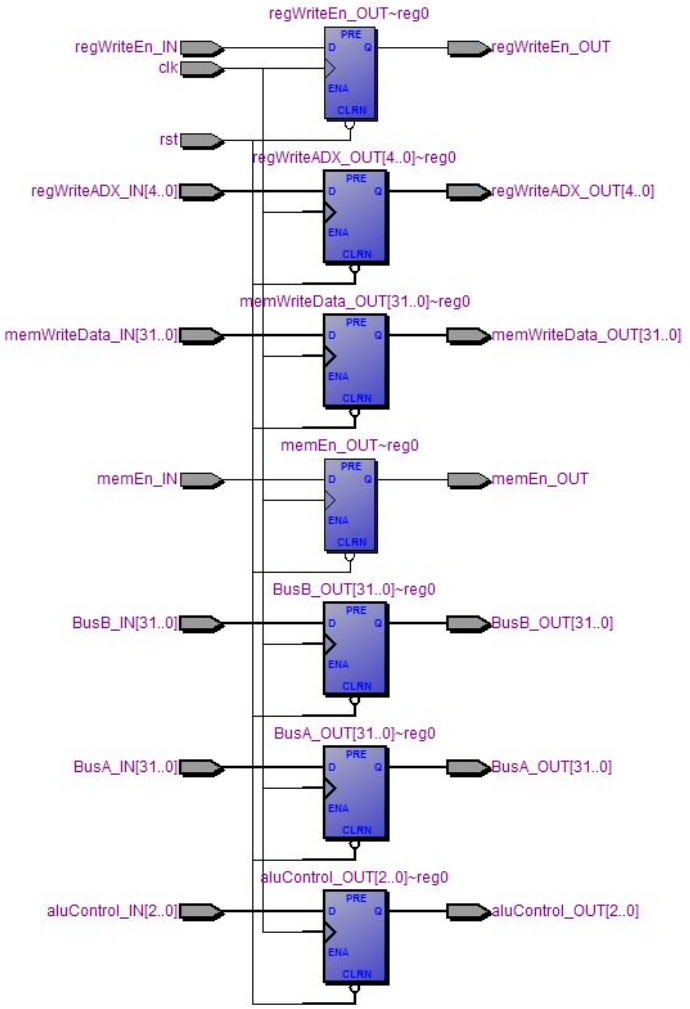


Forwarding Unit

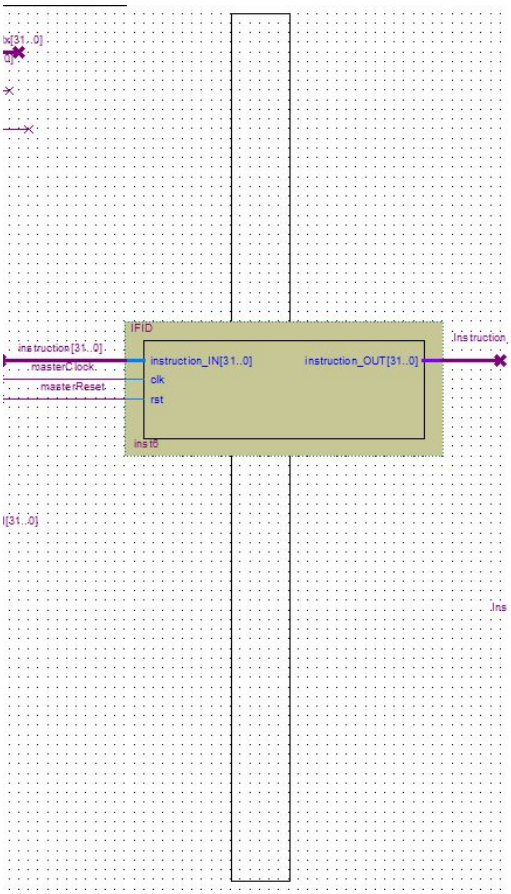


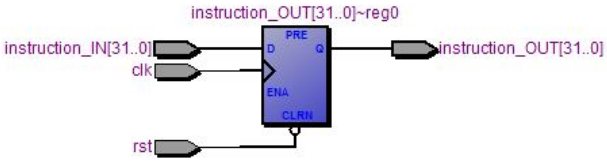
ID/EX Register



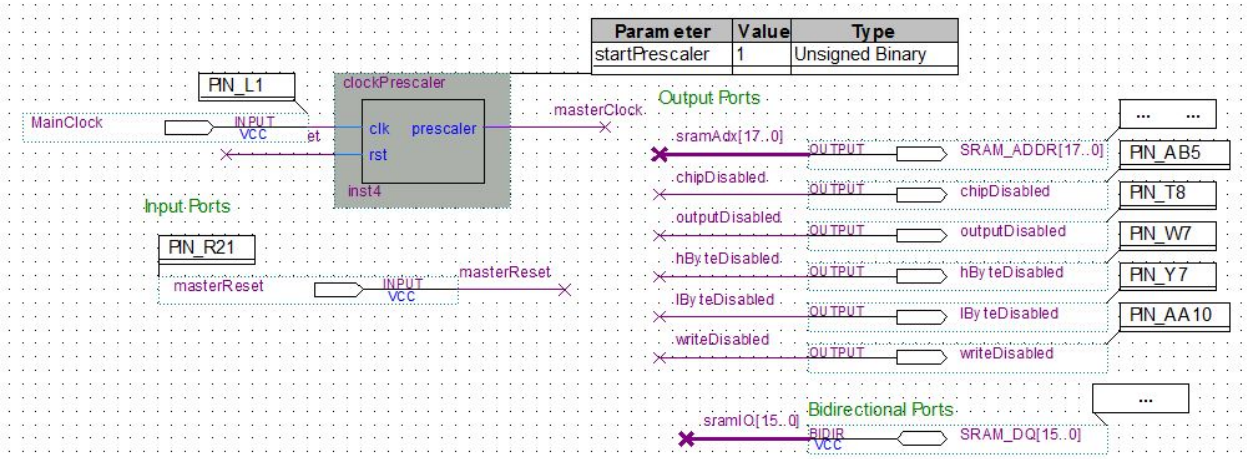


IF/ID Register

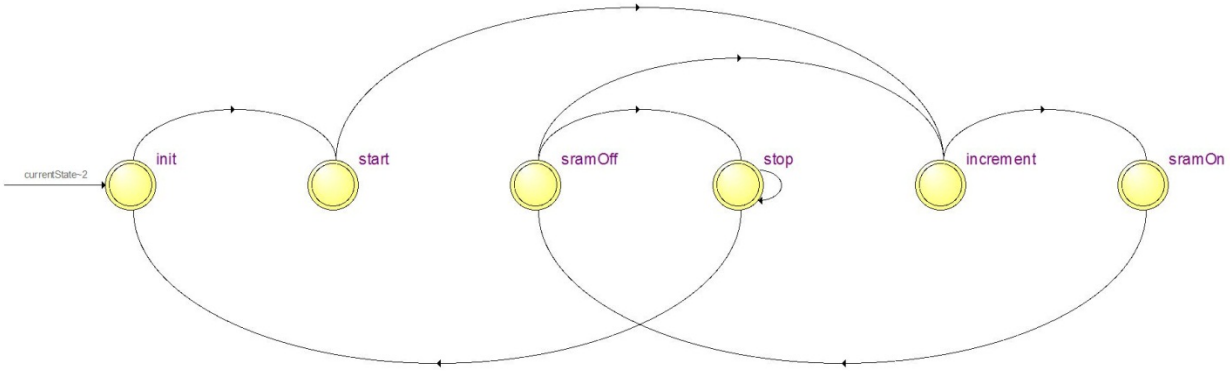




Input/output Ports



B.11 Program Loader State Machine



Appendix C. Single Cycle CPU Annotated Source Code

C.1 ALU Subsystem Module

```
/**
 * The ALU Subsystem takes care of arithmetic and logical operations, it also
sets
 * flags to indicate erroneous results (overflow) or provide results
information
 * (negative, zero, carryout)
 */
module aluSubsystem(RESULT, ALUFLAGS, A, B, CONTROL);

      /********************** I/O Ports **********************/
      output [31:0] RESULT;
      output [3:0] ALUFLAGS;
      input [31:0] A, B;
      input [2:0] CONTROL; //Op codes

      /******************* Wire/Reg Declarations *******************/
      wire ZERO, OVERFLOW, CARRYOUT, NEGATIVE;
      wire [31:0] wNOP, wADD, wSUB, wAND, wOR, wXOR, wSLT, wSLL;
      wire [7:0] decoded;
      wire OVERFLOWADD, OVERFLOWSUB, CARRYADD, CARRYSUB;

      // Logic to determine the flags set by the ALU
      assign ALUFLAGS = {NEGATIVE, CARRYOUT, OVERFLOW, ZERO};
      assign wSLT[31:1] = 31'b0;
      decoder3_8 blah00(decoded,1'b1,CONTROL);
      assign ZERO = &(~RESULT);
      assign OVERFLOW = OVERFLOWADD&decoded[1] | OVERFLOWSUB&decoded[2] |
OVERFLOWSUB&decoded[6];
      assign CARRYOUT = CARRYADD|CARRYSUB;
      assign NEGATIVE = RESULT[31];

      // AlU operations Combinational Logic
      assign wNOP = 32'b0;
                  //0
```

```
	arithmeticADD blah0(wADD,OVERFLOWADD,CARRYADD,A,B);//1
	arithmeticSUB blah1(wSUB,OVERFLOWSUB,CARRYSUB,A,B);//2
	logicalAND blah2(wAND, A, B);
	//3
	logicalOR blah3(wOR, A, B);
	//4
	logicalXOR blah4(wXOR, A, B);
	//5
	assign wSLT[0] = ~wSUB[31];
	//6
	logicalSLL blah6(wSLL, A, B);
	//7

	muxALU blah7(RESULT, wNOP, wADD, wSUB, wAND, wOR, wXOR, wSLT, wSLL,
CONTROL);

endmodule

// 8 to 1 mux used to output only  the desired alu operation
module muxALU(OUT, IN0,IN1,IN2,IN3,IN4,IN5,IN6,IN7, CONTROL);
	output [31:0] OUT;
	input [31:0] IN0,IN1,IN2,IN3,IN4,IN5,IN6,IN7;
	input [2:0] CONTROL;

	assign OUT = (CONTROL==0)?IN0:(CONTROL==1)?IN1:(CONTROL==2)?IN2:
(CONTROL==3)?IN3:(CONTROL==4)?IN4:(CONTROL==5)?IN5:(CONTROL==6)?IN6:IN7;

endmodule
```

C.2 Bus Driver Subsystem Module

```
/** module used to provide a safe bus sharing mechanism where multiple writers
get to use the
 *  buses only when instructed by a bus control signal coming from control
subsystem */

module busDriver (systemBus, writeEnable,memoryADX,
sramEn,readNWrite,busControl, controlWE, sramData, aluData,
	controlRegWE, sramRegWE,
registerDataB,loaderData,controlSramEn,loaderSramEn,loaderMemoryADX,loaderRead
NWrite,controlReadNWrite);

	// I/O port declarations
	output wire [31:0] systemBus, memoryADX;
	output wire writeEnable, sramEn, readNWrite;

	input wire [2:0] busControl;
	input wire [1:0] controlWE;
	input wire [31:0] sramData, aluData, registerDataB, loaderData,
loaderMemoryADX;
	input wire controlRegWE, sramRegWE,controlSramEn, loaderSramEn,
loaderReadNWrite,controlReadNWrite;

	// gate instantiations and combinational logic, in parallel
```

```
      assign systemBus [31:0] = (busControl==0)? 0:(busControl==1)?
loaderData:(busControl==2)? sramData:(busControl==3)? aluData:(busControl==4)?
registerDataB:0;
      assign memoryADX[31:0] = (busControl == 0)?0:(busControl==1)?
loaderMemoryADX:(busControl == 2 | busControl == 4)?aluData:0;
      assign writeEnable = (busControl==2)?sramRegWE:(controlWE==0)? 1'b0:
(controlWE==1)? controlRegWE:(controlWE==2)? sramRegWE:1'b0;
      assign sramEn = (busControl==1)?loaderSramEn:controlSramEn;
      assign readNWrite = (busControl==1)?loaderReadNWrite:controlReadNWrite;

endmodule
```

C.3 Control Subsystem Module

```
/**
 * The control subsystem module decodes the incoming instructions and sets the
respective flags to
 * control the datapath in order to be able to execute the desired instruction
 */
module
controlSubsystem(controlSramEn,controlWE,regDst,jump,branch,controlReadNWrite,
busControl,aluControl,controlRegWE,
jumpRegister,aluSrc,instruction,loadProgramEn,aluFlags,clk);

      /********************************* I/O PORTS
*******************************/
      output regDst, jump,branch,controlReadNWrite,jumpRegister,
aluSrc,controlRegWE;
      output controlSramEn;
      output [1:0] controlWE;
      output [2:0] aluControl, busControl;

      input [31:0] instruction;
      input loadProgramEn,clk;
      input [3:0] aluFlags;

      /********************************* Control Decode Logic
*****************************************/
      assign
{aluControl,regDst,jump,jumpRegister,branch,busControl,controlReadNWrite,contr
olWE,aluSrc} =
            (loadProgramEn)?14'b000_0_0_0_0_001_0_00_0:
            ((instruction == 0)?14'b000_0_0_0_0_000_1_00_0: //no OP
            ((instruction[31:26]==0)? //R type
            ((instruction[5:0]==32)?14'b001_1_0_0_0_011_1_01_0: //add
            (instruction[5:0]==34)?14'b010_1_0_0_0_011_1_01_0: //sub
            (instruction[5:0]==36)?14'b011_1_0_0_0_011_1_01_0: //and
            (instruction[5:0]==37)?14'b100_1_0_0_0_011_1_01_0: //or
            (instruction[5:0]==39)?14'b101_1_0_0_0_011_1_01_0: //xor
            (instruction[5:0]==42)?14'b110_1_0_0_0_011_1_01_0: //slt
            (instruction[5:0]==0)?14'b111_1_0_0_0_011_1_01_0: //sll
            (instruction[5:0]==8)?14'b000_0_0_1_0_000_1_00_0:14'b0): //jump
register
            (instruction[31:26]==35)?14'b001_0_0_0_0_010_1_10_1: //load word
            (instruction[31:26]==43)?14'b001_0_0_0_0_100_0_00_1: //store word
            (instruction[31:26]==5)?14'b110_0_0_0_1_000_1_00_0: //bgt
            (instruction[31:26]==2)?14'b000_0_1_0_0_000_1_00_0:14'b0)); //jump
```

```
      // assign the output wires the desired values
      assign controlSramEn = aluSrc;
      assign controlRegWE = ~clk;
endmodule
```

C.4 Data Path Subsystem Module

```
/* Datapath Subsystem
 * Includes all the logic that routes the data around the processor,
 * the control signals coming from control subsystem and the opcode
 * and function fields are used to control the multiplexers
 */

module
dataPathSubsystem(RS1,RS2,WS,instructionADXPCI,busB,loadProgEn,branch,jump,alu
Src,jumpRegister,

      regDst,aluFlags,instruction,instructionADX,registersDataB);
      /****************** Ports Definitions ******************/
      output [4:0] RS1,RS2,WS;
      output [31:0] instructionADXPCI, busB;
      input loadProgEn, branch,jump,aluSrc,jumpRegister,regDst;
      input [3:0] aluFlags;
      input [31:0] instruction, instructionADX, registersDataB;

      /*************** Datapath Combinational Logic *************/
      assign RS1 = (loadProgEn)?5'b0:(instruction[31:26] == 0 &
instruction[5:0] == 0)?instruction[20:16]:instruction[25:21];
      assign RS2 = (loadProgEn)?(jumpRegister)?
instruction[25:21]:5'b0:instruction[20:16];
      assign WS = (loadProgEn)?5'b0:((regDst)?
instruction[15:11]:instruction[20:16]);
      assign instructionADXPCI[31:0] = (loadProgEn)?0:(branch&~aluFlags[0])?
(instructionADX + instruction[15:0] + 1):(jump)?{6'd0,instruction[25:0]}:
(jumpRegister)?registersDataB:(instructionADX+1);
      assign busB[31:0] = (loadProgEn)?0:((aluSrc)?{16'd0,instruction[15:0]}:
(instruction[31:26] == 0 & instruction[5:0] == 0)?
{27'd0,instruction[10:6]}:registersDataB[31:0]);


endmodule
```

C.5 Instruction Memory Module

```
/* Instruction memory, 128*32 bit memory used to store the instructions to be
executed by the processor
 */
module instructionMemory(instruction, instructionADX, instructionWriteAdx,
instructionWriteData, IMWriteEnable);

      /*********** I/O Ports **********/
      output [31:0] instruction;
      input IMWriteEnable;
      input [31:0] instructionADX, instructionWriteAdx, instructionWriteData;
      reg [31:0] currentReg [127:0];
```

```
	/****** Sequential Execution *******/
	// alyaws write in data only on the positive edge of Instruction memory
write enable
	always@(posedge IMWriteEnable)
	begin
		currentReg[instructionWriteAdx[6:0]] = instructionWriteData[31:0];
	end

	/****** Parallel execution, reads are done continuously *******/
	assign instruction[31:0] = currentReg[instructionADX[6:0]];

endmodule
```

C.6 Memory Subsystem Module

```
/**
 * This module implements a driver to be able to read from the
 * SRAM Hardware, this is now implemented all in combinational logic
 * for enhanced acces time
 */
module memorySubsystem2(dataReadyFlag, sramDataO,
                            sramAdx, sramIO, chipDisabled, outputDisabled,
hByteDisabled, lByteDisabled, writeDisabled,
                            sramEnabled, readNWrite, adxIn, dataIn, clk,
rst);

	// I/O port declarations, exterior interface
	input wire sramEnabled, readNWrite, clk, rst;
	input wire [31:0] adxIn;
	input wire [31:0] dataIn;

	output dataReadyFlag;
	output [31:0] sramDataO;

	// these flags are used by the processor to know when data is stable and
in the output lines
	assign dataReadyFlag = readNWrite&(~clk)&sramEnabled; //occurs on second
half of cycle after dataIn hopefully stabilizes
	assign sramDataO = {{16{sramIO[15]}},sramIO[15:0]}; //if read, take sign
extension of sramIO, else take 0


	// Sram Hardware I/O port declarations
	output chipDisabled, outputDisabled, hByteDisabled, lByteDisabled;
	output writeDisabled;
	output [17:0] sramAdx;

	inout wire [15:0] sramIO;

	// not care about power, so just keep flags always low (active)
	assign chipDisabled = 0;
	assign outputDisabled = 0;
	assign hByteDisabled = 0;
	assign lByteDisabled = 0;
```

```
      // combinational logic that performs the reads and writes from/to the
SRAM
      assign sramAdx = adxIn[17:0];
      assign writeDisabled = ~((~readNWrite)&(~clk)&sramEnabled); //occurs on
second half of cycle after adx hopefully stabilizes
      assign sramIO [15:0] = (~readNWrite)?dataIn[15:0]:16'bZ;//if writing
take dataIn

endmodule
```

C.7 Processor Registers Module

```
/*    Processor registers used for fast local data access, for processor
operations
 * R1 = Read Data from S1 call
 *    R2 = Read Data from S2 call
 *    W = Data to be written to register called from wS call only if enabled
 *    S1 = call for register to be read
 * S2 = call for register to be read
 *    wS = call for register to write to
 *    regFileWE = regFileWEnables write function to register called by wS
 */
module processorRegisters(R1,R2,W,S1,S2,wS,clk,rst,regFileWE);
      parameter          n = 32;
      output      [n-1:0]     R1, R2;
      input [n-1:0]     W;
      input [4:0] S1, S2, wS;
      input              clk, rst, regFileWE;
      wire  [n-1:0]     C0,C1,C2,C3,C4,C5,C6,C7,C8,C9,C10,C11,
                              C12,C13,C14,C15,C16,C17,C18,C19,C20,C21,
                              C22,C23,C24,C25,C26,C27,C28,C29,C30,C31;
      wire  [n-1:0]     D0,D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,
                              D12,D13,D14,D15,D16,D17,D18,D19,D20,D21,
                              D22,D23,D24,D25,D26,D27,D28,D29,D30,D31,
                              W,adxW;
      wire
regFileWE0,regFileWE1,regFileWE2,regFileWE3,regFileWE4,regFileWE5,regFileWE6,r
egFileWE7,regFileWE8,regFileWE9,regFileWE10,
regFileWE11,regFileWE12,regFileWE13,regFileWE14,regFileWE15,regFileWE16,regFil
eWE17,regFileWE18,regFileWE19,


regFileWE20,regFileWE21,regFileWE22,regFileWE23,regFileWE24,regFileWE25,regFil
eWE26,regFileWE27,regFileWE28,regFileWE29,regFileWE30,regFileWE31;

      //builds array of n registers with individual enable functions. Rg0 set
to
      // be 0 at all times.
      regB  Rg0(32'd0, D0, clk, rst, regFileWE0),     Rg1(W, D1, clk, rst,
regFileWE1),
                  Rg2(W, D2, clk, rst, regFileWE2),   Rg3(W, D3, clk, rst,
regFileWE3),
                  Rg4(W, D4, clk, rst, regFileWE4),   Rg5(W, D5, clk, rst,
regFileWE5),
```

```
                Rg6(W, D6, clk, rst, regFileWE6),   Rg7(W, D7, clk, rst,
regFileWE7),
                Rg8(W, D8, clk, rst, regFileWE8),   Rg9(W, D9, clk, rst,
regFileWE9),

                Rg10(W, D10, clk, rst, regFileWE10), Rg11(W, D11, clk, rst,
regFileWE11),
                Rg12(W, D12, clk, rst, regFileWE12), Rg13(W, D13, clk, rst,
regFileWE13),
                Rg14(W, D14, clk, rst, regFileWE14), Rg15(W, D15, clk, rst,
regFileWE15),
                Rg16(W, D16, clk, rst, regFileWE16), Rg17(W, D17, clk, rst,
regFileWE17),
                Rg18(W, D18, clk, rst, regFileWE18), Rg19(W, D19, clk, rst,
regFileWE19),

                Rg20(W, D20, clk, rst, regFileWE20), Rg21(W, D21, clk, rst,
regFileWE21),
                Rg22(W, D22, clk, rst, regFileWE22), Rg23(W, D23, clk, rst,
regFileWE23),
                Rg24(W, D24, clk, rst, regFileWE24), Rg25(W, D25, clk, rst,
regFileWE25),
                Rg26(W, D26, clk, rst, regFileWE26), Rg27(W, D27, clk, rst,
regFileWE27),
                Rg28(W, D28, clk, rst, regFileWE28), Rg29(W, D29, clk, rst,
regFileWE29),

                Rg30(W, D30, clk, rst, regFileWE30), Rg31(W, D31, clk, rst,
regFileWE31);

      //Select signal sent to the decoder of individual register selection
      adx_decoder wDecode(adxW, wS);

      //regFileWEnable gates for the write function for each register
      and         a0(regFileWE0,adxW[0],regFileWE),
      a1(regFileWE1,adxW[1],regFileWE),
      a2(regFileWE2,adxW[2],regFileWE),
                        a3(regFileWE3,adxW[3],regFileWE),
      a4(regFileWE4,adxW[4],regFileWE),
      a5(regFileWE5,adxW[5],regFileWE),
                        a6(regFileWE6,adxW[6],regFileWE),
      a7(regFileWE7,adxW[7],regFileWE),
      a8(regFileWE8,adxW[8],regFileWE),
                        a9(regFileWE9,adxW[9],regFileWE),
      a10(regFileWE10,adxW[10],regFileWE),
      a11(regFileWE11,adxW[11],regFileWE),
                        a12(regFileWE12,adxW[12],regFileWE),
      a13(regFileWE13,adxW[13],regFileWE),
      a14(regFileWE14,adxW[14],regFileWE),
                        a15(regFileWE15,adxW[15],regFileWE),
      a16(regFileWE16,adxW[16],regFileWE),
      a17(regFileWE17,adxW[17],regFileWE),
                        a18(regFileWE18,adxW[18],regFileWE),
      a19(regFileWE19,adxW[19],regFileWE),
      a20(regFileWE20,adxW[20],regFileWE),
```

```
                        a21(regFileWE21,adxW[21],regFileWE),
      a22(regFileWE22,adxW[22],regFileWE),
      a23(regFileWE23,adxW[23],regFileWE),
                        a24(regFileWE24,adxW[24],regFileWE),
      a25(regFileWE25,adxW[25],regFileWE),
      a26(regFileWE26,adxW[26],regFileWE),
                        a27(regFileWE27,adxW[27],regFileWE),
      a28(regFileWE28,adxW[28],regFileWE),
      a29(regFileWE29,adxW[29],regFileWE),
                        a30(regFileWE30,adxW[30],regFileWE),
      a31(regFileWE31,adxW[31],regFileWE);

      // always @(S1,S2)
      // begin
      assign      C0 =
{D31[0],D30[0],D29[0],D28[0],D27[0],D26[0],D25[0],D24[0],D23[0],D22[0],

D21[0],D20[0],D19[0],D18[0],D17[0],D16[0],D15[0],D14[0],D13[0],D12[0],
                    D11[0],D10[0], D9[0], D8[0], D7[0], D6[0], D5[0], D4[0],
D3[0], D2[0],D1[0],D0[0]};
      assign      C1 =
{D31[1],D30[1],D29[1],D28[1],D27[1],D26[1],D25[1],D24[1],D23[1],D22[1],

D21[1],D20[1],D19[1],D18[1],D17[1],D16[1],D15[1],D14[1],D13[1],D12[1],
                    D11[1],D10[1], D9[1], D8[1], D7[1], D6[1], D5[1], D4[1],
D3[1], D2[1],D1[1],D0[1]};
      assign      C2 =
{D31[2],D30[2],D29[2],D28[2],D27[2],D26[2],D25[2],D24[2],D23[2],D22[2],

D21[2],D20[2],D19[2],D18[2],D17[2],D16[2],D15[2],D14[2],D13[2],D12[2],
                    D11[2],D10[2], D9[2], D8[2], D7[2], D6[2], D5[2], D4[2],
D3[2], D2[2],D1[2],D0[2]};
      assign      C3 =
{D31[3],D30[3],D29[3],D28[3],D27[3],D26[3],D25[3],D24[3],D23[3],D22[3],

D21[3],D20[3],D19[3],D18[3],D17[3],D16[3],D15[3],D14[3],D13[3],D12[3],
                    D11[3],D10[3], D9[3], D8[3], D7[3], D6[3], D5[3], D4[3],
D3[3], D2[3],D1[3],D0[3]};
      assign      C4 =
{D31[4],D30[4],D29[4],D28[4],D27[4],D26[4],D25[4],D24[4],D23[4],D22[4],

D21[4],D20[4],D19[4],D18[4],D17[4],D16[4],D15[4],D14[4],D13[4],D12[4],
                    D11[4],D10[4], D9[4], D8[4], D7[4], D6[4], D5[4], D4[4],
D3[4], D2[4],D1[4],D0[4]};
      assign      C5 =
{D31[5],D30[5],D29[5],D28[5],D27[5],D26[5],D25[5],D24[5],D23[5],D22[5],

D21[5],D20[5],D19[5],D18[5],D17[5],D16[5],D15[5],D14[5],D13[5],D12[5],
                    D11[5],D10[5], D9[5], D8[5], D7[5], D6[5], D5[5], D4[5],
D3[5], D2[5],D1[5],D0[5]};
      assign      C6 =
{D31[6],D30[6],D29[6],D28[6],D27[6],D26[6],D25[6],D24[6],D23[6],D22[6],

D21[6],D20[6],D19[6],D18[6],D17[6],D16[6],D15[6],D14[6],D13[6],D12[6],
```

```
                    D11[6],D10[6], D9[6], D8[6], D7[6], D6[6], D5[6], D4[6],
D3[6], D2[6],D1[6],D0[6]};
      assign      C7 =
{D31[7],D30[7],D29[7],D28[7],D27[7],D26[7],D25[7],D24[7],D23[7],D22[7],

D21[7],D20[7],D19[7],D18[7],D17[7],D16[7],D15[7],D14[7],D13[7],D12[7],
                    D11[7],D10[7], D9[7], D8[7], D7[7], D6[7], D5[7], D4[7],
D3[7], D2[7],D1[7],D0[7]};
      assign      C8 =
{D31[8],D30[8],D29[8],D28[8],D27[8],D26[8],D25[8],D24[8],D23[8],D22[8],

D21[8],D20[8],D19[8],D18[8],D17[8],D16[8],D15[8],D14[8],D13[8],D12[8],
                    D11[8],D10[8], D9[8], D8[8], D7[8], D6[8], D5[8], D4[8],
D3[8], D2[8],D1[8],D0[8]};
      assign      C9 =
{D31[9],D30[9],D29[9],D28[9],D27[9],D26[9],D25[9],D24[9],D23[9],D22[9],

D21[9],D20[9],D19[9],D18[9],D17[9],D16[9],D15[9],D14[9],D13[9],D12[9],
                    D11[9],D10[9], D9[9], D8[9], D7[9], D6[9], D5[9], D4[9],
D3[9], D2[9],D1[9],D0[9]};

      assign      C10 =
{D31[10],D30[10],D29[10],D28[10],D27[10],D26[10],D25[10],D24[10],D23[10],D22[1
0],

D21[10],D20[10],D19[10],D18[10],D17[10],D16[10],D15[10],D14[10],D13[10],D12[10
],
                    D11[10],D10[10], D9[10], D8[10], D7[10], D6[10], D5[10],
D4[10], D3[10], D2[10],D1[10],D0[10]};
      assign      C11 =
{D31[11],D30[11],D29[11],D28[11],D27[11],D26[11],D25[11],D24[11],D23[11],D22[1
1],

D21[11],D20[11],D19[11],D18[11],D17[11],D16[11],D15[11],D14[11],D13[11],D12[11
],
                    D11[11],D10[11], D9[11], D8[11], D7[11], D6[11], D5[11],
D4[11], D3[11], D2[11],D1[11],D0[11]};
      assign      C12 =
{D31[12],D30[12],D29[12],D28[12],D27[12],D26[12],D25[12],D24[12],D23[12],D22[1
2],

D21[12],D20[12],D19[12],D18[12],D17[12],D16[12],D15[12],D14[12],D13[12],D12[12
],
                    D11[12],D10[12], D9[12], D8[12], D7[12], D6[12], D5[12],
D4[12], D3[12], D2[12],D1[12],D0[12]};
      assign      C13 =
{D31[13],D30[13],D29[13],D28[13],D27[13],D26[13],D25[13],D24[13],D23[13],D22[1
3],

D21[13],D20[13],D19[13],D18[13],D17[13],D16[13],D15[13],D14[13],D13[13],D12[13
],
                    D11[13],D10[13], D9[13], D8[13], D7[13], D6[13], D5[13],
D4[13], D3[13], D2[13],D1[13],D0[13]};
```

```
      assign      C14 =
{D31[14],D30[14],D29[14],D28[14],D27[14],D26[14],D25[14],D24[14],D23[14],D22[1
4],

D21[14],D20[14],D19[14],D18[14],D17[14],D16[14],D15[14],D14[14],D13[14],D12[14
],
                    D11[14],D10[14], D9[14], D8[14], D7[14], D6[14], D5[14],
D4[14], D3[14], D2[14],D1[14],D0[14]};
      assign      C15 =
{D31[15],D30[15],D29[15],D28[15],D27[15],D26[15],D25[15],D24[15],D23[15],D22[1
5],

D21[15],D20[15],D19[15],D18[15],D17[15],D16[15],D15[15],D14[15],D13[15],D12[15
],
                    D11[15],D10[15], D9[15], D8[15], D7[15], D6[15], D5[15],
D4[15], D3[15], D2[15],D1[15],D0[15]};
      assign      C16 =
{D31[16],D30[16],D29[16],D28[16],D27[16],D26[16],D25[16],D24[16],D23[16],D22[1
6],

D21[16],D20[16],D19[16],D18[16],D17[16],D16[16],D15[16],D14[16],D13[16],D12[16
],
                    D11[16],D10[16], D9[16], D8[16], D7[16], D6[16], D5[16],
D4[16], D3[16], D2[16],D1[16],D0[16]};
      assign      C17 =
{D31[17],D30[17],D29[17],D28[17],D27[17],D26[17],D25[17],D24[17],D23[17],D22[1
7],

D21[17],D20[17],D19[17],D18[17],D17[17],D16[17],D15[17],D14[17],D13[17],D12[17
],
                    D11[17],D10[17], D9[17], D8[17], D7[17], D6[17], D5[17],
D4[17], D3[17], D2[17],D1[17],D0[17]};
      assign      C18 =
{D31[18],D30[18],D29[18],D28[18],D27[18],D26[18],D25[18],D24[18],D23[18],D22[1
8],

D21[18],D20[18],D19[18],D18[18],D17[18],D16[18],D15[18],D14[18],D13[18],D12[18
],
                    D11[18],D10[18], D9[18], D8[18], D7[18], D6[18], D5[18],
D4[18], D3[18], D2[18],D1[18],D0[18]};
      assign      C19 =
{D31[19],D30[19],D29[19],D28[19],D27[19],D26[19],D25[19],D24[19],D23[19],D22[1
9],

D21[19],D20[19],D19[19],D18[19],D17[19],D16[19],D15[19],D14[19],D13[19],D12[19
],
                    D11[19],D10[19], D9[19], D8[19], D7[19], D6[19], D5[19],
D4[19], D3[19], D2[19],D1[19],D0[19]};

      assign      C20 =
{D31[20],D30[20],D29[20],D28[20],D27[20],D26[20],D25[20],D24[20],D23[20],D22[2
0],

D21[20],D20[20],D19[20],D18[20],D17[20],D16[20],D15[20],D14[20],D13[20],D12[20
],
```

```
                    D11[20],D10[20], D9[20], D8[20], D7[20], D6[20], D5[20],
D4[20], D3[20], D2[20],D1[20],D0[20]};
      assign      C21 =
{D31[21],D30[21],D29[21],D28[21],D27[21],D26[21],D25[21],D24[21],D23[21],D22[2
1],

D21[21],D20[21],D19[21],D18[21],D17[21],D16[21],D15[21],D14[21],D13[21],D12[21
],
                    D11[21],D10[21], D9[21], D8[21], D7[21], D6[21], D5[21],
D4[21], D3[21], D2[21],D1[21],D0[21]};
      assign      C22 =
{D31[22],D30[22],D29[22],D28[22],D27[22],D26[22],D25[22],D24[22],D23[22],D22[2
2],

D21[22],D20[22],D19[22],D18[22],D17[22],D16[22],D15[22],D14[22],D13[22],D12[22
],
                    D11[22],D10[22], D9[22], D8[22], D7[22], D6[22], D5[22],
D4[22], D3[22], D2[22],D1[22],D0[22]};
      assign      C23 =
{D31[23],D30[23],D29[23],D28[23],D27[23],D26[23],D25[23],D24[23],D23[23],D22[2
3],

D21[23],D20[23],D19[23],D18[23],D17[23],D16[23],D15[23],D14[23],D13[23],D12[23
],
                    D11[23],D10[23], D9[23], D8[23], D7[23], D6[23], D5[23],
D4[23], D3[23], D2[23],D1[23],D0[23]};
      assign      C24 =
{D31[24],D30[24],D29[24],D28[24],D27[24],D26[24],D25[24],D24[24],D23[24],D22[2
4],

D21[24],D20[24],D19[24],D18[24],D17[24],D16[24],D15[24],D14[24],D13[24],D12[24
],
                    D11[24],D10[24], D9[24], D8[24], D7[24], D6[24], D5[24],
D4[24], D3[24], D2[24],D1[24],D0[24]};
      assign      C25 =
{D31[25],D30[25],D29[25],D28[25],D27[25],D26[25],D25[25],D24[25],D23[25],D22[2
5],

D21[25],D20[25],D19[25],D18[25],D17[25],D16[25],D15[25],D14[25],D13[25],D12[25
],
                    D11[25],D10[25], D9[25], D8[25], D7[25], D6[25], D5[25],
D4[25], D3[25], D2[25],D1[25],D0[25]};
      assign      C26 =
{D31[26],D30[26],D29[26],D28[26],D27[26],D26[26],D25[26],D24[26],D23[26],D22[2
6],

D21[26],D20[26],D19[26],D18[26],D17[26],D16[26],D15[26],D14[26],D13[26],D12[26
],
                    D11[26],D10[26], D9[26], D8[26], D7[26], D6[26], D5[26],
D4[26], D3[26], D2[26],D1[26],D0[26]};
      assign      C27 =
{D31[27],D30[27],D29[27],D28[27],D27[27],D26[27],D25[27],D24[27],D23[27],D22[2
7],
```

```
D21[27],D20[27],D19[27],D18[27],D17[27],D16[27],D15[27],D14[27],D13[27],D12[27
],
                    D11[27],D10[27], D9[27], D8[27], D7[27], D6[27], D5[27],
D4[27], D3[27], D2[27],D1[27],D0[27]};
      assign      C28 =
{D31[28],D30[28],D29[28],D28[28],D27[28],D26[28],D25[28],D24[28],D23[28],D22[2
8],

D21[28],D20[28],D19[28],D18[28],D17[28],D16[28],D15[28],D14[28],D13[28],D12[28
],
                    D11[28],D10[28], D9[28], D8[28], D7[28], D6[28], D5[28],
D4[28], D3[28], D2[28],D1[28],D0[28]};
      assign      C29 =
{D31[29],D30[29],D29[29],D28[29],D27[29],D26[29],D25[29],D24[29],D23[29],D22[2
9],

D21[29],D20[29],D19[29],D18[29],D17[29],D16[29],D15[29],D14[29],D13[29],D12[29
],
                    D11[29],D10[29], D9[29], D8[29], D7[29], D6[29], D5[29],
D4[29], D3[29], D2[29],D1[29],D0[29]};

      assign      C30 =
{D31[30],D30[30],D29[30],D28[30],D27[30],D26[30],D25[30],D24[30],D23[30],D22[3
0],

D21[30],D20[30],D19[30],D18[30],D17[30],D16[30],D15[30],D14[30],D13[30],D12[30
],
                    D11[30],D10[30], D9[30], D8[30], D7[30], D6[30], D5[30],
D4[30], D3[30], D2[30],D1[30],D0[30]};
      assign      C31 =
{D31[31],D30[31],D29[31],D28[31],D27[31],D26[31],D25[31],D24[31],D23[31],D22[3
1],

D21[31],D20[31],D19[31],D18[31],D17[31],D16[31],D15[31],D14[31],D13[31],D12[31
],
                    D11[31],D10[31], D9[31], D8[31], D7[31], D6[31], D5[31],
D4[31], D3[31], D2[31],D1[31],D0[31]};
      // end


      //multiplexes outputs to R1 and R2 from selected registers
      mux32to1v1        mxR10(C0,S1,R1[0]),     mxR11(C1,S1,R1[1]),
                        mxR12(C2,S1,R1[2]),     mxR13(C3,S1,R1[3]),
                        mxR14(C4,S1,R1[4]),     mxR15(C5,S1,R1[5]),
                        mxR16(C6,S1,R1[6]),     mxR17(C7,S1,R1[7]),
                        mxR18(C8,S1,R1[8]),     mxR19(C9,S1,R1[9]),

                        mxR110(C10,S1,R1[10]), mxR111(C11,S1,R1[11]),
                        mxR112(C12,S1,R1[12]), mxR113(C13,S1,R1[13]),
                        mxR114(C14,S1,R1[14]), mxR115(C15,S1,R1[15]),
                        mxR116(C16,S1,R1[16]), mxR117(C17,S1,R1[17]),
                        mxR118(C18,S1,R1[18]), mxR119(C19,S1,R1[19]),

                        mxR120(C20,S1,R1[20]), mxR121(C21,S1,R1[21]),
                        mxR122(C22,S1,R1[22]), mxR123(C23,S1,R1[23]),
```

```
                        mxR124(C24,S1,R1[24]), mxR125(C25,S1,R1[25]),
                        mxR126(C26,S1,R1[26]), mxR127(C27,S1,R1[27]),
                        mxR128(C28,S1,R1[28]), mxR129(C29,S1,R1[29]),

                        mxR130(C30,S1,R1[30]), mxR131(C31,S1,R1[31]),

                        mxR20(C0,S2,R2[0]),     mxR21(C1,S2,R2[1]),
                        mxR22(C2,S2,R2[2]),     mxR23(C3,S2,R2[3]),
                        mxR24(C4,S2,R2[4]),     mxR25(C5,S2,R2[5]),
                        mxR26(C6,S2,R2[6]),     mxR27(C7,S2,R2[7]),
                        mxR28(C8,S2,R2[8]),     mxR29(C9,S2,R2[9]),

                        mxR210(C10,S2,R2[10]), mxR211(C11,S2,R2[11]),
                        mxR212(C12,S2,R2[12]), mxR213(C13,S2,R2[13]),
                        mxR214(C14,S2,R2[14]), mxR215(C15,S2,R2[15]),
                        mxR216(C16,S2,R2[16]), mxR217(C17,S2,R2[17]),
                        mxR218(C18,S2,R2[18]), mxR219(C19,S2,R2[19]),

                        mxR220(C20,S2,R2[20]), mxR221(C21,S2,R2[21]),
                        mxR222(C22,S2,R2[22]), mxR223(C23,S2,R2[23]),
                        mxR224(C24,S2,R2[24]), mxR225(C25,S2,R2[25]),
                        mxR226(C26,S2,R2[26]), mxR227(C27,S2,R2[27]),
                        mxR228(C28,S2,R2[28]), mxR229(C29,S2,R2[29]),

                        mxR230(C30,S2,R2[30]), mxR231(C31,S2,R2[31]);
endmodule
```

C.9 Program Counter Module

```
/* Program Counter module, simply a register that puts out the
 * instruction address on the positive edge of the clock
 */
module programCounter(instructionADX, pcWriteDis, instructionADXPCI,rst, clk);

      /***************** I/O Ports **********************/
      output reg [31:0] instructionADX /* synthesis keep */;
      input pcWriteDis, clk,rst;
      input [31:0] instructionADXPCI;

      /******* Sequential Block ************/
      // executes on the clock
      always@(posedge clk or negedge rst)
      begin
            if(!rst) instructionADX[31:0] = 0;
            else instructionADX[31:0] = instructionADXPCI;
      end

endmodule
```

C.10 Program Loader Module

```
/* Program Loader, Loads the program (set of instructions) into instruction
memory and the data
 * into main memory (SRAM), after completed flags the processor to start
executing the
```

```
 * instructions
 */
module programLoader(loadProgEn, instructionWriteADX, instructionWrite,
IMWriteEn, loaderMemoryADX, loaderSRAMEn, loaderReadNWrite, loaderData, clk,
nrst);

            /********************************* Module Parameters
****************************************/
            // States Parameters
            parameter init = 3'd0, start = 3'd1, increment = 3'd2, sramOn =
3'd3, sramOff = 3'd4, stop = 3'd5;
            //INSTRUCTION Op Codes
            parameter rOP = 6'h0, lwOP = 6'h23, swOP = 6'h2B, jumpOP = 6'h2,
bgtOP = 6'h5, addF = 6'h20, subF = 6'h22, andF = 6'h24, orF = 6'h25, xorF =
6'h26, sltF = 6'h2A, sllF = 6'h0, jregF = 6'h8;

            /************************** I/O Ports
*****************************/
            output reg loadProgEn, IMWriteEn, loaderSRAMEn, loaderReadNWrite;
            output reg [31:0] instructionWriteADX, instructionWrite,
loaderData, loaderMemoryADX;
            input clk, nrst;

            /************* Reg & Wires Declarations ************/
            reg [2:0] currentState;

            /************* Regs Initializations *****************/
            initial
            begin
                  currentState = init;
            end

            /************* Sequential Block **************/
            // FSM that loads instructions and data into
            // instruction memory and main memory
            always@(posedge clk or negedge nrst)
            begin
                  if(!nrst) currentState <= init;
                  else
                  begin
                        case(currentState)
                              init: begin
                                    loadProgEn <= 0;
                                    instructionWriteADX <= 0;
                                    instructionWrite <= 0;
                                    IMWriteEn <= 0;
                                    loaderMemoryADX <= 0;
                                    loaderSRAMEn <= 0;
                                    loaderReadNWrite <= 0;
                                    loaderData <= 0;
                                    currentState <= start;
                              end
                              start: begin
                                    loadProgEn <= 1;
                                    loaderReadNWrite <= 0;
                                    currentState <= increment;
```

```
                        end
                        increment: begin
                              case(loaderMemoryADX[4:0])
                                    0: begin
                                          instructionWrite <= {lwOP,
5'd0, 5'd8, 16'd0};//2349334528;
                                          loaderData <= 9;   // A = 7
(No Branch) A = 9 (Branch)
                                    end
                                    1:  begin
                                          instructionWrite <= {lwOP,
5'd0, 5'd9, 16'd1};//2349400065;
                                          loaderData <= 5;
                                    end
                                    2:  begin
                                          instructionWrite <= {lwOP,
5'd0, 5'd10, 16'd2};//2349465602;
                                          loaderData <= 2;
                                    end
                                    3:  begin
                                          instructionWrite <= {lwOP,
5'd0, 5'd11, 16'd3};//2349531139;
                                          loaderData <= 4;
                                    end
                                    4:  begin
                                          instructionWrite <= {lwOP,
5'd0, 5'd12, 16'd4};//2349596676;
                                          loaderData <= 3;  // adx of D
in main memory is 3
                                    end
                                    5:  begin
                                          instructionWrite <= {rOP,
5'd8, 5'd9, 5'd13, 5'd0, subF};//17393698;
                                          loaderData <= 3;
                                    end
                                    6:  begin
                                          instructionWrite <= {lwOP,
5'd0, 5'd14, 16'd5};//2349727749;
                                          loaderData <= 7;
                                    end
                                    7: begin
                                          instructionWrite <= {bgtOP,
5'd13, 5'd14, 16'd3};//363724803;
                                          loaderData <= 6;
                                    end
                                    8: begin
                                          instructionWrite <= {rOP,
5'd0, 5'd10, 5'd10, 5'd5, sllF};//676160;
                                          loaderData <= 0;
                                    end
                                    9: begin
                                          instructionWrite <= {lwOP,
5'd0, 5'd11, 16'd6};//2349531142;
                                          loaderData <= 0;
                                    end
                                    10: begin
```

```
26'd13};//134217741;
                                                    instructionWrite <= {jumpOP,
                                                    loaderData <= 0;
                                                end
                                                11: begin
                                                    instructionWrite <= {lwOP,
5'd0, 5'd10, 16'd7};//2349465607;
                                                    loaderData <= 0;
                                                end
                                                12: begin
                                                    instructionWrite <= {rOP,
5'd0, 5'd11, 5'd11, 5'd2, sllF};//743552;
                                                    loaderData <= 0;
                                                end
                                                13: begin
                                                    instructionWrite <= {swOP,
5'd0, 5'd8, 16'd0};//2886205440;
                                                    loaderData <= 0;
                                                end
                                                14: begin
                                                    instructionWrite <= {swOP,
5'd0, 5'd9, 16'd1};//2886270977;
                                                    loaderData <= 0;
                                                end
                                                15: begin
                                                    instructionWrite <= {swOP,
5'd0, 5'd10, 16'd2};//2886336514;
                                                    loaderData <= 0;
                                                end
                                                16: begin
                                                    instructionWrite <= {swOP,
5'd0, 5'd11, 16'd3};//2886402051;
                                                    loaderData <= 0;
                                                end
                                                17: begin
                                                    instructionWrite <= {swOP,
5'd0, 5'd12, 16'd4};//2886467588;
                                                    loaderData <= 0;
                                                end
                                                18: begin
                                                    instructionWrite <= {lwOP,
5'd0, 5'd0, 16'd0};//2348810240;
                                                    loaderData <= 0;
                                                end
                                                19: begin
                                                    instructionWrite <= {lwOP,
5'd0, 5'd0, 16'd1};//2348810241;
                                                    loaderData <= 0;
                                                end
                                                20: begin
                                                    instructionWrite <= {lwOP,
5'd0, 5'd0, 16'd2};//2348810242;
                                                    loaderData <= 0;
                                                end
                                                21: begin
```

```
                                        instructionWrite <= {lwOP,
5'd0, 5'd0, 16'd3};//2348810243;
                                        loaderData <= 0;
                                    end
                                    22: begin
                                        instructionWrite <= {lwOP,
5'd0, 5'd0, 16'd4};//2348810244;
                                        loaderData <= 0;
                                    end
                                    23: begin
                                        instructionWrite <= {rOP,5'b0,
5'd0, 5'd0, 5'd0, jregF};
                                        loaderData <= 0;
                                    end
                                    default: begin
                                        instructionWrite <= 0;
                                        loaderData <= 0;
                                    end
                                endcase
                                currentState <= sramOn;
                            end
                            sramOn: begin
                                loaderSRAMEn <= 1;
                                IMWriteEn <= 1;
                                currentState <= sramOff;
                            end
                            sramOff: begin
                                loaderMemoryADX <= loaderMemoryADX + 1;
                                instructionWriteADX <= instructionWriteADX
+ 1;
                                loaderSRAMEn <= 0;
                                IMWriteEn <= 0;
                                if(loaderMemoryADX >= 24) currentState <=
stop;
                                else currentState <= increment;
                            end
                            stop: begin
                                loadProgEn <= 0;
                                instructionWriteADX <= 0;
                                instructionWrite <= 0;
                                IMWriteEn <= 0;
                                loaderMemoryADX <= 0;
                                loaderSRAMEn <= 0;
                                loaderReadNWrite <= 1;
                                loaderData <= 0;
                                if(!nrst) currentState <= init;
                                else currentState <= stop;
                            end
                            default: currentState <= stop;
                        endcase
                    end
                end

endmodule
```

Appendix D. Pipelined CPU Annotated Source Code

D.1 ALU Subsystem

```
/**
 * The arithmetic and logic unit used to execute the instructions
 */
module aluSubsystem(RESULT, ALUFLAGS, A, B, CONTROL);
      output [31:0] RESULT;
      output [3:0] ALUFLAGS;
      input [31:0] A, B;
      input [2:0] CONTROL; //Op codes


      wire ZERO, OVERFLOW, CARRYOUT, NEGATIVE;
      wire [31:0] wNOP, wADD, wSUB, wAND, wOR, wXOR, wSLT, wSLL;
      wire [7:0] decoded;
      wire OVERFLOWADD, OVERFLOWSUB, CARRYADD, CARRYSUB;

      assign ALUFLAGS = {NEGATIVE, CARRYOUT, OVERFLOW, ZERO};
      assign wSLT[31:1] = 31'b0;
      decoder3_8 blah00(decoded,1'b1,CONTROL);

      assign ZERO = &(~RESULT);
      assign OVERFLOW = OVERFLOWADD&decoded[1] | OVERFLOWSUB&decoded[2] |
OVERFLOWSUB&decoded[6];
      assign CARRYOUT = CARRYADD|CARRYSUB;
      assign NEGATIVE = RESULT[31];

      assign wNOP = 32'b0;
                  //0
      arithmeticADD blah0(wADD,OVERFLOWADD,CARRYADD,A,B);//1
      arithmeticSUB blah1(wSUB,OVERFLOWSUB,CARRYSUB,A,B);//2
      logicalAND blah2(wAND, A, B);
      //3
      logicalOR blah3(wOR, A, B);
      //4
      logicalXOR blah4(wXOR, A, B);
      //5
      assign wSLT[0] = ~wSUB[31];
      //6
      logicalSLL blah6(wSLL, A, B);
      //7

      muxALU blah7(RESULT, wNOP, wADD, wSUB, wAND, wOR, wXOR, wSLT, wSLL,
CONTROL);


endmodule

module muxALU(OUT, IN0,IN1,IN2,IN3,IN4,IN5,IN6,IN7, CONTROL);
      output [31:0] OUT;
      input [31:0] IN0,IN1,IN2,IN3,IN4,IN5,IN6,IN7;
      input [2:0] CONTROL;
```

```
      assign OUT = (CONTROL==0)?IN0:(CONTROL==1)?IN1:(CONTROL==2)?IN2:
(CONTROL==3)?IN3:(CONTROL==4)?IN4:(CONTROL==5)?IN5:(CONTROL==6)?IN6:IN7;

endmodule
```

D.2 Control Subsystem

```
/**
 * Control subsystem, used to control the ID Datapath and also includes
 * the hazard detection unit
 */
module controlSubsystem2(aluControl,memoryEn,regWriteEn, IFDPathControl,
                                                          instruction,
loadProgramEn, clk, rst);

      input [31:0] instruction;
      input loadProgramEn;
      reg IDEXMemoryEn;
      reg IDEXRegWriteEn;
      input wire clk, rst;
      output [2:0] aluControl;
      output memoryEn,regWriteEn;
      output wire [2:0] IFDPathControl;

      (* keep *) reg [31:0] IDEXInstruction; /* Synthesis keep */
      wire stallAndBubble;

      // combinational logic used to determine the control lines from
instruction fields
      assign {IFDPathControl, aluControl,memoryEn,regWriteEn} =
            (loadProgramEn == 1)? 8'b000_000_0_0: //program loader = noOP
            (stallAndBubble == 1)? 8'b101_000_0_0: // there was a stall
            (instruction == 0)?8'b001_000_0_0: //no OP
            (instruction[31:26]==0)? //R type
            ((instruction[5:0]==32)?8'b001_001_0_1: //add
            (instruction[5:0]==34)?8'b001_010_0_1: //sub
            (instruction[5:0]==36)?8'b001_011_0_1: //and
            (instruction[5:0]==37)?8'b001_100_0_1: //or
            (instruction[5:0]==38)?8'b001_101_0_1: //xor
            (instruction[5:0]==42)?8'b001_110_0_1: //slt
            (instruction[5:0]==0)?8'b001_111_0_1: //sll
            (instruction[5:0]==8)?8'b011_000_0_0:8'd0): //jump
register:Unknown instruction = noOP
            (instruction[31:26]==35)?8'b001_001_1_1: //load word
            (instruction[31:26]==43)?8'b001_000_1_0: //store word
            (instruction[31:26]==5)?8'b100_000_0_0: //bgt
            (instruction[31:26]==2)?8'b010_000_0_0:8'd0; //jump:Unknown
instruction = noOP

      // shift register used to keep track of old instructions and old flags,
      // these are used by the hazard detection unit to determine whether
there
      // is a hazard taking place
      always@(posedge clk or negedge rst) begin
            if(!rst) begin
                  IDEXInstruction = 0;
                  IDEXMemoryEn = 0;
```

```
                IDEXRegWriteEn = 0;
            end else begin
                IDEXInstruction = instruction;
                IDEXMemoryEn = memoryEn;
                IDEXRegWriteEn = regWriteEn;
            end
        end

        // Hazard detection unit logic
        assign stallAndBubble = (((IDEXMemoryEn && IDEXRegWriteEn) &&
((IDEXInstruction [20:16] == instruction[25:21])|||(IDEXInstruction [20:16] ==
instruction[20:16])))== 1);

endmodule
```

D.3 EX/MEM Register

```
module EXMEM(aluResult_IN, memWriteData_IN, regWriteADX_IN, memEn_IN,
regWriteEn_IN, aluResult_OUT, memWriteData_OUT, regWriteADX_OUT, memEn_OUT,
regWriteEn_OUT, clk, rst);
        input [4:0] regWriteADX_IN;
        input [31:0] aluResult_IN, memWriteData_IN;
        input memEn_IN, regWriteEn_IN;
        input clk, rst;
        output reg [4:0] regWriteADX_OUT;
        output reg [31:0] aluResult_OUT, memWriteData_OUT; /* Synthesis Keep */
        output reg memEn_OUT, regWriteEn_OUT; /* Synthesis Keep */

        always@(posedge clk or negedge rst) begin
            if(!rst) begin
                aluResult_OUT = 0;
                 memWriteData_OUT = 0;
                regWriteADX_OUT = 0;
                memEn_OUT = 0;
                regWriteEn_OUT = 0;
            end else begin
                aluResult_OUT = aluResult_IN;
                 memWriteData_OUT = memWriteData_IN;
                regWriteADX_OUT = regWriteADX_IN;
                memEn_OUT = memEn_IN;
                regWriteEn_OUT = regWriteEn_IN;
            end

        end


endmodule
```

D.4 Forwarding Unit Module

```
/**
 * The forwarding unit used to pass the data early if there is a hazard
 */
module forwardingUnit(toIDA, toIDB, toIDAFlag, toIDBFlag, fromALU, rs, rt, rd,
clk, rst);
        input [31:0] fromALU;
        input [4:0] rs, rt, rd;
```

```
	input clk, rst;
	output [31:0] toIDA, toIDB;
	output toIDAFlag, toIDBFlag;

	reg [4:0] rdALU;

	always@(posedge clk or negedge rst) begin
		if(!rst) begin
			rdALU = 0;
		end else begin
			rdALU <= rd;
		end
	end

	// the forwarded data
	assign toIDA = fromALU;
	assign toIDB = fromALU;

	// flag the datapath from the instruction decode stage to forward the
data when needed
	assign toIDAFlag = (!(rs == 0)&&(rs == rdALU))?1'b1:1'b0;
	assign toIDBFlag = (!(rt == 0)&&(rt == rdALU))?1'b1:1'b0;

endmodule
```

D.5 Instruction Decode Stage Data path

```
/**
 * Instruction Decode Datapath, Used to route data around the
 * instruction decode stage, controls the multiplexers to choose
 * the data onto buses A and B which can be coming forwarded
 * from the next stages or simply from the reg file
 *
 */
module IDDPath(rs, rt, rd, jumpIADX, jumpRegIADX, branchIADX, BusA, BusB,
memWriteData, IFDPathControlOUT,
	instruction, regBusA, regBusB, fwdBusA, fwdBusB,  controlBusA,
controlBusB, aluControl, IDDPathControlIN);

	output wire [31:0] jumpIADX, jumpRegIADX, branchIADX, BusA, BusB,
memWriteData;
	output wire [4:0] rs, rt, rd;
	output wire [2:0] IFDPathControlOUT;
	input wire [31:0] instruction, regBusA, regBusB, fwdBusA, fwdBusB;
	input wire [2:0] aluControl, IDDPathControlIN;
	input wire controlBusA, controlBusB;


	wire [31:0] tempBusA, SLLBusB, tempBusB;

	assign SLLBusB = {27'b0, instruction[10:6]};

	assign jumpIADX = {6'b000000, instruction[25:0]};

	assign jumpRegIADX = regBusA;
```

```
      assign branchIADX = instruction[15:0];
      assign IFDPathControlOUT = (IDDPathControlIN == 3'b100)?((tempBusA >
tempBusB)?IDDPathControlIN:3'b001):IDDPathControlIN;

      assign tempBusA =
            (controlBusA == 0)? regBusA :
            (controlBusA == 1)? fwdBusA : 32'b0;

      assign BusA = tempBusA;

      assign tempBusB =
            (aluControl == 3'b111)? SLLBusB:
            (instruction[31:26] == 35)? {16'b0, instruction[15:0]}:
            (instruction[31:26] == 43)? {16'b0, instruction[15:0]}:
            ((controlBusB == 0)? regBusB:
            (controlBusB == 1)? fwdBusB: 32'b0);

      assign BusB = tempBusB;

      assign rs =
            (instruction[31:26] == 2)? 5'd0: instruction[25:21];

      assign rt =
            (instruction[31:26] == 2)? 5'd0:
            (aluControl == 7)? instruction[10:6]: instruction[20:16];

      assign rd =
            ((instruction[31:26] == 2) ||
            (instruction[31:26] == 43) ||
            (instruction[31:26] == 5))? 5'd0:(instruction[31:26] == 35)?
instruction[20:16]:instruction[15:11];

      assign memWriteData = (instruction[31:26] == 43)?regBusB:0;

      endmodule
```

D.6 ID/EX Register

```
/**
 * The IDEX is the register used to divide the Instruction decode stage and
the
 * Execute stage (i.e. the alu)
 */
module IDEX(BusA_IN, BusB_IN, memWriteData_IN, aluControl_IN, regWriteADX_IN,
memEn_IN, regWriteEn_IN, BusA_OUT, BusB_OUT, memWriteData_OUT, aluControl_OUT,
regWriteADX_OUT, memEn_OUT, regWriteEn_OUT, clk, rst);
      input [31:0] BusA_IN, BusB_IN, memWriteData_IN;
      input [4:0] regWriteADX_IN;
      input [2:0] aluControl_IN;
      input memEn_IN, regWriteEn_IN;
      input clk, rst;
      output reg [31:0] BusA_OUT, BusB_OUT, memWriteData_OUT; /* Synthesis
Keep */
      output reg [4:0] regWriteADX_OUT;
      output reg [2:0] aluControl_OUT;
      output reg memEn_OUT, regWriteEn_OUT; /* Synthesis Keep */
```

```
	// shift the data out on the clock
	always@(posedge clk or negedge rst) begin
		if(!rst) begin
			BusA_OUT = 0;
			BusB_OUT = 0;
			memWriteData_OUT = 0;
			regWriteADX_OUT = 0;
			memEn_OUT = 0;
			regWriteEn_OUT = 0;
			aluControl_OUT = 0;
		end else begin
			BusA_OUT = BusA_IN;
			BusB_OUT = BusB_IN;
			 memWriteData_OUT = memWriteData_IN;
			regWriteADX_OUT = regWriteADX_IN;
			memEn_OUT = memEn_IN;
			regWriteEn_OUT = regWriteEn_IN;
			aluControl_OUT = aluControl_IN;
		end
	end

endmodule
```

D.7 Instruction Fetch Stage Data path

```
/**
 * Data path module for the Instruction Fetch stage,
 * This datapath is a series of multiplexers to chose from
 * the branch address, the jump address, the jump register address
 * or simply the next sequential address
 */
module IFDPath(instructionADXPCI, branchIADX, jumpRegIADX, jumpIADX,
IFDPathControl, progLoaderEn, clk, rst);
	input wire [31:0] branchIADX, jumpRegIADX, jumpIADX;
	input wire [2:0] IFDPathControl;
	input wire progLoaderEn;
	input wire clk, rst;
	output wire [31:0] instructionADXPCI;
	reg [31:0] lastInstructionADX;

	wire [2:0] control;

	assign control = progLoaderEn?3'd0:IFDPathControl;

	// choose which address to put on the lines based on
	// the control signal
	assign instructionADXPCI =
		(control == 0)? 32'b0:
		(control == 1)? (lastInstructionADX + 1):
		(control == 2)? jumpIADX:
		(control == 3)? jumpRegIADX:
		(control == 4)? branchIADX:
		(control == 5)? lastInstructionADX: 32'b0;

	// simply a shift reg to keep track of old and new addresses
```

```
	always@(posedge clk or negedge rst)
	begin
		if(!rst)
		begin
			lastInstructionADX = 0;
		end
		else
		begin
		lastInstructionADX = instructionADXPCI;
		end

	end

endmodule
```

D.8 IF/ID Register

```
/**
 * The IFID register used to clock the Fetched instruction into
 * the Instruction decode stage on the next clock cycle
 */
module IFID(instruction_IN, instruction_OUT, clk, rst);
	input [31:0] instruction_IN;
	input clk, rst;
	output reg [31:0] instruction_OUT; /* Synthesis Keep */

	always@(posedge clk or negedge rst)
	begin
		if(!rst) begin
			instruction_OUT = 0;
		end else begin
			instruction_OUT = instruction_IN;
		end
	end

endmodule
```

D.9 Instruction Memory Module

```
/**
 * The 128X32 memory used to store the program instructions
 */
module instructionMemory(instruction, instructionADX, instructionWriteAdx,
instructionWriteData, IMWriteEnable);
	output [31:0] instruction;
	input IMWriteEnable;
	input [31:0] instructionADX, instructionWriteAdx, instructionWriteData;
	(* keep *) reg [31:0] currentReg [127:0]; /* Synthesis keep */

	// write on the positive edge of write enable
	always@(posedge IMWriteEnable)
	begin
		currentReg[instructionWriteAdx[6:0]] = instructionWriteData[31:0];
	end

	// read continuously
	assign instruction[31:0] = currentReg[instructionADX[6:0]];
```

```
endmodule
```

D.10 Memory Subsystem

```
/**
 * This module implements an efficient combinational logic circuit
 * to improve the access times to a single clock cycle, data width (16 bits)
 */
module memorySubsystem2(dataReadyFlag, sramDataO,
                                sramAdx, sramIO, chipDisabled, outputDisabled,
hByteDisabled, lByteDisabled, writeDisabled,
                                sramEnabled, readNWrite, adxIn, dataIn, clk,
rst, progLoaderData, progLoaderADX, progLoaderEN);

      // I/O port declarations, exterior interface
      input wire sramEnabled, progLoaderEN, readNWrite, clk, rst;
      input wire [31:0] adxIn, progLoaderADX;
      input wire [31:0] dataIn, progLoaderData;

      output dataReadyFlag;
      output [31:0] sramDataO;

      wire enabled;
      wire [31:0] data, adx;

      assign enabled = progLoaderEN|sramEnabled;
      assign data = (progLoaderEN)?progLoaderData:dataIn;
      assign adx = (progLoaderEN)?progLoaderADX:adxIn;

      assign dataReadyFlag =
      (enabled == 0 && readNWrite == 1)? readNWrite&(~clk):
      (enabled == 1 && readNWrite == 1)? readNWrite&(~clk): 1'b0;

      assign sramDataO =
      (enabled == 0 && readNWrite == 1)? adxIn:
      (enabled == 1 && readNWrite == 1)? {16'b0,sramIO[15:0]}:32'b0;

      // sram I/O port declarations
      output chipDisabled, outputDisabled, hByteDisabled, lByteDisabled;
      output writeDisabled;
      output [17:0] sramAdx;

      inout wire [15:0] sramIO;

      // since we are going for speed (single cycle access time) keep all
other signals active low
      assign chipDisabled = 0;
      assign outputDisabled = 0;
      assign hByteDisabled = 0;
      assign lByteDisabled = 0;
      assign sramAdx = adx[17:0];
      assign writeDisabled = ~((~readNWrite)&(~clk)&enabled); //occurs on
second half of cycle after adx hopefully stabilizes
      assign sramIO [15:0] = (~readNWrite)?data[15:0]:16'bZ;//if writing take
dataIn
```

```
endmodule
```

D.11 Processor Registers

```
/*    R1 = Read Data from S1 call
*     R2 = Read Data from S2 call
*     W = Data to be written to register called from wS call only if enabled
*     S1 = call for register to be read
*     S2 = call for register to be read
*     wS = call for register to write to
*     regFileWE = regFileWEnables write function to register called by wS
*/
module processorRegisters(R1,R2,W,S1,S2,wS,clk,rst,regFileWE);
      parameter         n = 32;
      output      [n-1:0]     R1, R2;
      input [n-1:0]     W;
      input [4:0] S1, S2, wS;
      input             clk, rst, regFileWE;
      wire  [n-1:0]     C0,C1,C2,C3,C4,C5,C6,C7,C8,C9,C10,C11,
                        C12,C13,C14,C15,C16,C17,C18,C19,C20,C21,
                        C22,C23,C24,C25,C26,C27,C28,C29,C30,C31;
      wire  [n-1:0]     D0,D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,
                        D12,D13,D14,D15,D16,D17,D18,D19,D20,D21,
                        D22,D23,D24,D25,D26,D27,D28,D29,D30,D31,
                        W,adxW;
      wire
regFileWE0,regFileWE1,regFileWE2,regFileWE3,regFileWE4,regFileWE5,regFileWE6,r
egFileWE7,regFileWE8,regFileWE9,regFileWE10,
regFileWE11,regFileWE12,regFileWE13,regFileWE14,regFileWE15,regFileWE16,regFil
eWE17,regFileWE18,regFileWE19,


regFileWE20,regFileWE21,regFileWE22,regFileWE23,regFileWE24,regFileWE25,regFil
eWE26,regFileWE27,regFileWE28,regFileWE29,regFileWE30,regFileWE31;

      //builds array of n registers with individual enable functions. Rg0 set
to
      // be 0 at all times.
      regB  Rg0(32'd0, D0, clk, rst, regFileWE0),     Rg1(W, D1, clk, rst,
regFileWE1),
                  Rg2(W, D2, clk, rst, regFileWE2),   Rg3(W, D3, clk, rst,
regFileWE3),
                  Rg4(W, D4, clk, rst, regFileWE4),   Rg5(W, D5, clk, rst,
regFileWE5),
                  Rg6(W, D6, clk, rst, regFileWE6),   Rg7(W, D7, clk, rst,
regFileWE7),
                  Rg8(W, D8, clk, rst, regFileWE8),   Rg9(W, D9, clk, rst,
regFileWE9),

                  Rg10(W, D10, clk, rst, regFileWE10), Rg11(W, D11, clk, rst,
regFileWE11),
                  Rg12(W, D12, clk, rst, regFileWE12), Rg13(W, D13, clk, rst,
regFileWE13),
                  Rg14(W, D14, clk, rst, regFileWE14), Rg15(W, D15, clk, rst,
regFileWE15),
```

```
                Rg16(W, D16, clk, rst, regFileWE16), Rg17(W, D17, clk, rst,
regFileWE17),
                Rg18(W, D18, clk, rst, regFileWE18), Rg19(W, D19, clk, rst,
regFileWE19),

                Rg20(W, D20, clk, rst, regFileWE20), Rg21(W, D21, clk, rst,
regFileWE21),
                Rg22(W, D22, clk, rst, regFileWE22), Rg23(W, D23, clk, rst,
regFileWE23),
                Rg24(W, D24, clk, rst, regFileWE24), Rg25(W, D25, clk, rst,
regFileWE25),
                Rg26(W, D26, clk, rst, regFileWE26), Rg27(W, D27, clk, rst,
regFileWE27),
                Rg28(W, D28, clk, rst, regFileWE28), Rg29(W, D29, clk, rst,
regFileWE29),

                Rg30(W, D30, clk, rst, regFileWE30), Rg31(W, D31, clk, rst,
regFileWE31);

      //Select signal sent to the decoder of individual register selection
      adx_decoder wDecode(adxW, wS);

      //regFileWEnable gates for the write function for each register
      and         a0(regFileWE0,adxW[0],regFileWE),
      a1(regFileWE1,adxW[1],regFileWE),
      a2(regFileWE2,adxW[2],regFileWE),
                        a3(regFileWE3,adxW[3],regFileWE),
      a4(regFileWE4,adxW[4],regFileWE),
      a5(regFileWE5,adxW[5],regFileWE),
                        a6(regFileWE6,adxW[6],regFileWE),
      a7(regFileWE7,adxW[7],regFileWE),
      a8(regFileWE8,adxW[8],regFileWE),
                        a9(regFileWE9,adxW[9],regFileWE),
      a10(regFileWE10,adxW[10],regFileWE),
      a11(regFileWE11,adxW[11],regFileWE),
                        a12(regFileWE12,adxW[12],regFileWE),
      a13(regFileWE13,adxW[13],regFileWE),
      a14(regFileWE14,adxW[14],regFileWE),
                        a15(regFileWE15,adxW[15],regFileWE),
      a16(regFileWE16,adxW[16],regFileWE),
      a17(regFileWE17,adxW[17],regFileWE),
                        a18(regFileWE18,adxW[18],regFileWE),
      a19(regFileWE19,adxW[19],regFileWE),
      a20(regFileWE20,adxW[20],regFileWE),
                        a21(regFileWE21,adxW[21],regFileWE),
      a22(regFileWE22,adxW[22],regFileWE),
      a23(regFileWE23,adxW[23],regFileWE),
                        a24(regFileWE24,adxW[24],regFileWE),
      a25(regFileWE25,adxW[25],regFileWE),
      a26(regFileWE26,adxW[26],regFileWE),
                        a27(regFileWE27,adxW[27],regFileWE),
      a28(regFileWE28,adxW[28],regFileWE),
      a29(regFileWE29,adxW[29],regFileWE),
                        a30(regFileWE30,adxW[30],regFileWE),
      a31(regFileWE31,adxW[31],regFileWE);
```

```
      // always @(S1,S2)
      // begin
      assign      C0 =
{D31[0],D30[0],D29[0],D28[0],D27[0],D26[0],D25[0],D24[0],D23[0],D22[0],

D21[0],D20[0],D19[0],D18[0],D17[0],D16[0],D15[0],D14[0],D13[0],D12[0],
                    D11[0],D10[0], D9[0], D8[0], D7[0], D6[0], D5[0], D4[0],
D3[0], D2[0],D1[0],D0[0]};
      assign      C1 =
{D31[1],D30[1],D29[1],D28[1],D27[1],D26[1],D25[1],D24[1],D23[1],D22[1],

D21[1],D20[1],D19[1],D18[1],D17[1],D16[1],D15[1],D14[1],D13[1],D12[1],
                    D11[1],D10[1], D9[1], D8[1], D7[1], D6[1], D5[1], D4[1],
D3[1], D2[1],D1[1],D0[1]};
      assign      C2 =
{D31[2],D30[2],D29[2],D28[2],D27[2],D26[2],D25[2],D24[2],D23[2],D22[2],

D21[2],D20[2],D19[2],D18[2],D17[2],D16[2],D15[2],D14[2],D13[2],D12[2],
                    D11[2],D10[2], D9[2], D8[2], D7[2], D6[2], D5[2], D4[2],
D3[2], D2[2],D1[2],D0[2]};
      assign      C3 =
{D31[3],D30[3],D29[3],D28[3],D27[3],D26[3],D25[3],D24[3],D23[3],D22[3],

D21[3],D20[3],D19[3],D18[3],D17[3],D16[3],D15[3],D14[3],D13[3],D12[3],
                    D11[3],D10[3], D9[3], D8[3], D7[3], D6[3], D5[3], D4[3],
D3[3], D2[3],D1[3],D0[3]};
      assign      C4 =
{D31[4],D30[4],D29[4],D28[4],D27[4],D26[4],D25[4],D24[4],D23[4],D22[4],

D21[4],D20[4],D19[4],D18[4],D17[4],D16[4],D15[4],D14[4],D13[4],D12[4],
                    D11[4],D10[4], D9[4], D8[4], D7[4], D6[4], D5[4], D4[4],
D3[4], D2[4],D1[4],D0[4]};
      assign      C5 =
{D31[5],D30[5],D29[5],D28[5],D27[5],D26[5],D25[5],D24[5],D23[5],D22[5],

D21[5],D20[5],D19[5],D18[5],D17[5],D16[5],D15[5],D14[5],D13[5],D12[5],
                    D11[5],D10[5], D9[5], D8[5], D7[5], D6[5], D5[5], D4[5],
D3[5], D2[5],D1[5],D0[5]};
      assign      C6 =
{D31[6],D30[6],D29[6],D28[6],D27[6],D26[6],D25[6],D24[6],D23[6],D22[6],

D21[6],D20[6],D19[6],D18[6],D17[6],D16[6],D15[6],D14[6],D13[6],D12[6],
                    D11[6],D10[6], D9[6], D8[6], D7[6], D6[6], D5[6], D4[6],
D3[6], D2[6],D1[6],D0[6]};
      assign      C7 =
{D31[7],D30[7],D29[7],D28[7],D27[7],D26[7],D25[7],D24[7],D23[7],D22[7],

D21[7],D20[7],D19[7],D18[7],D17[7],D16[7],D15[7],D14[7],D13[7],D12[7],
                    D11[7],D10[7], D9[7], D8[7], D7[7], D6[7], D5[7], D4[7],
D3[7], D2[7],D1[7],D0[7]};
      assign      C8 =
{D31[8],D30[8],D29[8],D28[8],D27[8],D26[8],D25[8],D24[8],D23[8],D22[8],

D21[8],D20[8],D19[8],D18[8],D17[8],D16[8],D15[8],D14[8],D13[8],D12[8],
```

```
                    D11[8],D10[8], D9[8], D8[8], D7[8], D6[8], D5[8], D4[8],
D3[8], D2[8],D1[8],D0[8]};
      assign      C9 =
{D31[9],D30[9],D29[9],D28[9],D27[9],D26[9],D25[9],D24[9],D23[9],D22[9],

D21[9],D20[9],D19[9],D18[9],D17[9],D16[9],D15[9],D14[9],D13[9],D12[9],
                    D11[9],D10[9], D9[9], D8[9], D7[9], D6[9], D5[9], D4[9],
D3[9], D2[9],D1[9],D0[9]};

      assign      C10 =
{D31[10],D30[10],D29[10],D28[10],D27[10],D26[10],D25[10],D24[10],D23[10],D22[1
0],

D21[10],D20[10],D19[10],D18[10],D17[10],D16[10],D15[10],D14[10],D13[10],D12[10
],
                    D11[10],D10[10], D9[10], D8[10], D7[10], D6[10], D5[10],
D4[10], D3[10], D2[10],D1[10],D0[10]};
      assign      C11 =
{D31[11],D30[11],D29[11],D28[11],D27[11],D26[11],D25[11],D24[11],D23[11],D22[1
1],

D21[11],D20[11],D19[11],D18[11],D17[11],D16[11],D15[11],D14[11],D13[11],D12[11
],
                    D11[11],D10[11], D9[11], D8[11], D7[11], D6[11], D5[11],
D4[11], D3[11], D2[11],D1[11],D0[11]};
      assign      C12 =
{D31[12],D30[12],D29[12],D28[12],D27[12],D26[12],D25[12],D24[12],D23[12],D22[1
2],

D21[12],D20[12],D19[12],D18[12],D17[12],D16[12],D15[12],D14[12],D13[12],D12[12
],
                    D11[12],D10[12], D9[12], D8[12], D7[12], D6[12], D5[12],
D4[12], D3[12], D2[12],D1[12],D0[12]};
      assign      C13 =
{D31[13],D30[13],D29[13],D28[13],D27[13],D26[13],D25[13],D24[13],D23[13],D22[1
3],

D21[13],D20[13],D19[13],D18[13],D17[13],D16[13],D15[13],D14[13],D13[13],D12[13
],
                    D11[13],D10[13], D9[13], D8[13], D7[13], D6[13], D5[13],
D4[13], D3[13], D2[13],D1[13],D0[13]};
      assign      C14 =
{D31[14],D30[14],D29[14],D28[14],D27[14],D26[14],D25[14],D24[14],D23[14],D22[1
4],

D21[14],D20[14],D19[14],D18[14],D17[14],D16[14],D15[14],D14[14],D13[14],D12[14
],
                    D11[14],D10[14], D9[14], D8[14], D7[14], D6[14], D5[14],
D4[14], D3[14], D2[14],D1[14],D0[14]};
      assign      C15 =
{D31[15],D30[15],D29[15],D28[15],D27[15],D26[15],D25[15],D24[15],D23[15],D22[1
5],

D21[15],D20[15],D19[15],D18[15],D17[15],D16[15],D15[15],D14[15],D13[15],D12[15
],
```

```
                    D11[15],D10[15], D9[15], D8[15], D7[15], D6[15], D5[15],
D4[15], D3[15], D2[15],D1[15],D0[15]};
      assign      C16 =
{D31[16],D30[16],D29[16],D28[16],D27[16],D26[16],D25[16],D24[16],D23[16],D22[1
6],

D21[16],D20[16],D19[16],D18[16],D17[16],D16[16],D15[16],D14[16],D13[16],D12[16
],
                    D11[16],D10[16], D9[16], D8[16], D7[16], D6[16], D5[16],
D4[16], D3[16], D2[16],D1[16],D0[16]};
      assign      C17 =
{D31[17],D30[17],D29[17],D28[17],D27[17],D26[17],D25[17],D24[17],D23[17],D22[1
7],

D21[17],D20[17],D19[17],D18[17],D17[17],D16[17],D15[17],D14[17],D13[17],D12[17
],
                    D11[17],D10[17], D9[17], D8[17], D7[17], D6[17], D5[17],
D4[17], D3[17], D2[17],D1[17],D0[17]};
      assign      C18 =
{D31[18],D30[18],D29[18],D28[18],D27[18],D26[18],D25[18],D24[18],D23[18],D22[1
8],

D21[18],D20[18],D19[18],D18[18],D17[18],D16[18],D15[18],D14[18],D13[18],D12[18
],
                    D11[18],D10[18], D9[18], D8[18], D7[18], D6[18], D5[18],
D4[18], D3[18], D2[18],D1[18],D0[18]};
      assign      C19 =
{D31[19],D30[19],D29[19],D28[19],D27[19],D26[19],D25[19],D24[19],D23[19],D22[1
9],

D21[19],D20[19],D19[19],D18[19],D17[19],D16[19],D15[19],D14[19],D13[19],D12[19
],
                    D11[19],D10[19], D9[19], D8[19], D7[19], D6[19], D5[19],
D4[19], D3[19], D2[19],D1[19],D0[19]};

      assign      C20 =
{D31[20],D30[20],D29[20],D28[20],D27[20],D26[20],D25[20],D24[20],D23[20],D22[2
0],

D21[20],D20[20],D19[20],D18[20],D17[20],D16[20],D15[20],D14[20],D13[20],D12[20
],
                    D11[20],D10[20], D9[20], D8[20], D7[20], D6[20], D5[20],
D4[20], D3[20], D2[20],D1[20],D0[20]};
      assign      C21 =
{D31[21],D30[21],D29[21],D28[21],D27[21],D26[21],D25[21],D24[21],D23[21],D22[2
1],

D21[21],D20[21],D19[21],D18[21],D17[21],D16[21],D15[21],D14[21],D13[21],D12[21
],
                    D11[21],D10[21], D9[21], D8[21], D7[21], D6[21], D5[21],
D4[21], D3[21], D2[21],D1[21],D0[21]};
      assign      C22 =
{D31[22],D30[22],D29[22],D28[22],D27[22],D26[22],D25[22],D24[22],D23[22],D22[2
2],
```

```
D21[22],D20[22],D19[22],D18[22],D17[22],D16[22],D15[22],D14[22],D13[22],D12[22
],
                    D11[22],D10[22], D9[22], D8[22], D7[22], D6[22], D5[22],
D4[22], D3[22], D2[22],D1[22],D0[22]};
      assign      C23 =
{D31[23],D30[23],D29[23],D28[23],D27[23],D26[23],D25[23],D24[23],D23[23],D22[2
3],

D21[23],D20[23],D19[23],D18[23],D17[23],D16[23],D15[23],D14[23],D13[23],D12[23
],
                    D11[23],D10[23], D9[23], D8[23], D7[23], D6[23], D5[23],
D4[23], D3[23], D2[23],D1[23],D0[23]};
      assign      C24 =
{D31[24],D30[24],D29[24],D28[24],D27[24],D26[24],D25[24],D24[24],D23[24],D22[2
4],

D21[24],D20[24],D19[24],D18[24],D17[24],D16[24],D15[24],D14[24],D13[24],D12[24
],
                    D11[24],D10[24], D9[24], D8[24], D7[24], D6[24], D5[24],
D4[24], D3[24], D2[24],D1[24],D0[24]};
      assign      C25 =
{D31[25],D30[25],D29[25],D28[25],D27[25],D26[25],D25[25],D24[25],D23[25],D22[2
5],

D21[25],D20[25],D19[25],D18[25],D17[25],D16[25],D15[25],D14[25],D13[25],D12[25
],
                    D11[25],D10[25], D9[25], D8[25], D7[25], D6[25], D5[25],
D4[25], D3[25], D2[25],D1[25],D0[25]};
      assign      C26 =
{D31[26],D30[26],D29[26],D28[26],D27[26],D26[26],D25[26],D24[26],D23[26],D22[2
6],

D21[26],D20[26],D19[26],D18[26],D17[26],D16[26],D15[26],D14[26],D13[26],D12[26
],
                    D11[26],D10[26], D9[26], D8[26], D7[26], D6[26], D5[26],
D4[26], D3[26], D2[26],D1[26],D0[26]};
      assign      C27 =
{D31[27],D30[27],D29[27],D28[27],D27[27],D26[27],D25[27],D24[27],D23[27],D22[2
7],

D21[27],D20[27],D19[27],D18[27],D17[27],D16[27],D15[27],D14[27],D13[27],D12[27
],
                    D11[27],D10[27], D9[27], D8[27], D7[27], D6[27], D5[27],
D4[27], D3[27], D2[27],D1[27],D0[27]};
      assign      C28 =
{D31[28],D30[28],D29[28],D28[28],D27[28],D26[28],D25[28],D24[28],D23[28],D22[2
8],

D21[28],D20[28],D19[28],D18[28],D17[28],D16[28],D15[28],D14[28],D13[28],D12[28
],
                    D11[28],D10[28], D9[28], D8[28], D7[28], D6[28], D5[28],
D4[28], D3[28], D2[28],D1[28],D0[28]};
      assign      C29 =
{D31[29],D30[29],D29[29],D28[29],D27[29],D26[29],D25[29],D24[29],D23[29],D22[2
9],
```

```
D21[29],D20[29],D19[29],D18[29],D17[29],D16[29],D15[29],D14[29],D13[29],D12[29
],
                    D11[29],D10[29], D9[29], D8[29], D7[29], D6[29], D5[29],
D4[29], D3[29], D2[29],D1[29],D0[29]};

      assign      C30 =
{D31[30],D30[30],D29[30],D28[30],D27[30],D26[30],D25[30],D24[30],D23[30],D22[3
0],

D21[30],D20[30],D19[30],D18[30],D17[30],D16[30],D15[30],D14[30],D13[30],D12[30
],
                    D11[30],D10[30], D9[30], D8[30], D7[30], D6[30], D5[30],
D4[30], D3[30], D2[30],D1[30],D0[30]};
      assign      C31 =
{D31[31],D30[31],D29[31],D28[31],D27[31],D26[31],D25[31],D24[31],D23[31],D22[3
1],

D21[31],D20[31],D19[31],D18[31],D17[31],D16[31],D15[31],D14[31],D13[31],D12[31
],
                    D11[31],D10[31], D9[31], D8[31], D7[31], D6[31], D5[31],
D4[31], D3[31], D2[31],D1[31],D0[31]};
      // end


      //multiplexes outputs to R1 and R2 from selected registers
      mux32to1v1        mxR10(C0,S1,R1[0]),     mxR11(C1,S1,R1[1]),
                        mxR12(C2,S1,R1[2]),     mxR13(C3,S1,R1[3]),
                        mxR14(C4,S1,R1[4]),     mxR15(C5,S1,R1[5]),
                        mxR16(C6,S1,R1[6]),     mxR17(C7,S1,R1[7]),
                        mxR18(C8,S1,R1[8]),     mxR19(C9,S1,R1[9]),

                        mxR110(C10,S1,R1[10]), mxR111(C11,S1,R1[11]),
                        mxR112(C12,S1,R1[12]), mxR113(C13,S1,R1[13]),
                        mxR114(C14,S1,R1[14]), mxR115(C15,S1,R1[15]),
                        mxR116(C16,S1,R1[16]), mxR117(C17,S1,R1[17]),
                        mxR118(C18,S1,R1[18]), mxR119(C19,S1,R1[19]),

                        mxR120(C20,S1,R1[20]), mxR121(C21,S1,R1[21]),
                        mxR122(C22,S1,R1[22]), mxR123(C23,S1,R1[23]),
                        mxR124(C24,S1,R1[24]), mxR125(C25,S1,R1[25]),
                        mxR126(C26,S1,R1[26]), mxR127(C27,S1,R1[27]),
                        mxR128(C28,S1,R1[28]), mxR129(C29,S1,R1[29]),

                        mxR130(C30,S1,R1[30]), mxR131(C31,S1,R1[31]),

                        mxR20(C0,S2,R2[0]),     mxR21(C1,S2,R2[1]),
                        mxR22(C2,S2,R2[2]),     mxR23(C3,S2,R2[3]),
                        mxR24(C4,S2,R2[4]),     mxR25(C5,S2,R2[5]),
                        mxR26(C6,S2,R2[6]),     mxR27(C7,S2,R2[7]),
                        mxR28(C8,S2,R2[8]),     mxR29(C9,S2,R2[9]),

                        mxR210(C10,S2,R2[10]), mxR211(C11,S2,R2[11]),
                        mxR212(C12,S2,R2[12]), mxR213(C13,S2,R2[13]),
                        mxR214(C14,S2,R2[14]), mxR215(C15,S2,R2[15]),
```

```
                        mxR216(C16,S2,R2[16]), mxR217(C17,S2,R2[17]),
                        mxR218(C18,S2,R2[18]), mxR219(C19,S2,R2[19]),

                        mxR220(C20,S2,R2[20]), mxR221(C21,S2,R2[21]),
                        mxR222(C22,S2,R2[22]), mxR223(C23,S2,R2[23]),
                        mxR224(C24,S2,R2[24]), mxR225(C25,S2,R2[25]),
                        mxR226(C26,S2,R2[26]), mxR227(C27,S2,R2[27]),
                        mxR228(C28,S2,R2[28]), mxR229(C29,S2,R2[29]),

                        mxR230(C30,S2,R2[30]), mxR231(C31,S2,R2[31]);
endmodule
```

D.12 Program Loader

```
/**
 * The loader used to load the instructions into instruction memory and data
into main memory
 */
module programLoader(loadProgEn, instructionWriteADX, instructionWrite,
IMWriteEn, loaderMemoryADX, loaderSRAMEn, loaderReadNWrite, loaderData, clk,
nrst);
            (* keep *) output reg loadProgEn, IMWriteEn, loaderSRAMEn,
loaderReadNWrite;
            (* keep *) output reg [31:0] instructionWriteADX,
instructionWrite,  loaderData, loaderMemoryADX;
            input clk, nrst;

            (* keep *) reg [2:0] currentState;
            parameter init = 3'd0, start = 3'd1, increment = 3'd2, sramOn =
3'd3, sramOff = 3'd4, stop = 3'd5;

            //INSTRUCTION PARAMETERS
            parameter rOP = 6'h0, lwOP = 6'h23, swOP = 6'h2B, jumpOP = 6'h2,
bgtOP = 6'h5, addF = 6'h20, subF = 6'h22, andF = 6'h24, orF = 6'h25, xorF =
6'h26, sltF = 6'h2A, sllF = 6'h0, jregF = 6'h8;

            initial
            begin
                  currentState = init;
            end

            // the FSM used to load instructions and data
            always@(posedge clk or negedge nrst)
            begin
                  if(!nrst) currentState <= init;
                  else
                  begin
                        case(currentState)
                              init: begin
                                    loadProgEn <= 0;
                                    instructionWriteADX <= 0;
                                    instructionWrite <= 0;
                                    IMWriteEn <= 0;
                                    loaderMemoryADX <= 0;
                                    loaderSRAMEn <= 0;
                                    loaderReadNWrite <= 0;
```

```
                                    loaderData <= 0;
                                    currentState <= start;
                              end
                              start: begin
                                    loadProgEn <= 1;
                                    loaderReadNWrite <= 0;
                                    currentState <= increment;
                              end
                              increment: begin
                                    case(loaderMemoryADX[4:0])
                                          0: begin
                                          instructionWrite <= {rOP, 5'd0,
5'd0, 5'd0, 5'd0, 6'd0}; //0
                                          loaderData <= 8;  //ANoBranch = 6
      ABranch = 8;
                                          end
                                          1: begin
                                          instructionWrite <= {lwOP, 5'd0,
5'd16, 16'd0}; //8c100000
                                          loaderData <= 4;
                                          end
                                          2: begin
                                          instructionWrite <= {lwOP, 5'd0,
5'd17, 16'd1}; //8c110001
                                          loaderData <= 2;
                                          end
                                          3: begin
                                          instructionWrite <= {lwOP, 5'd0,
5'd18, 16'd2}; //8c120002
                                          loaderData <= 4;
                                          end
                                          4: begin
                                          instructionWrite <= {lwOP, 5'd0,
5'd19, 16'd3}; //8c130003
                                          loaderData <= 3;
                                          end
                                          5: begin
                                          instructionWrite <= {lwOP, 5'd0,
5'd20, 16'd5}; //8c140005
                                          loaderData <= 30326;
                                          end
                                          6: begin
                                          instructionWrite <= {lwOP, 5'd0,
5'd21, 16'd6}; //8c150006
                                          loaderData <= 42405;
                                          end
                                          7: begin
                                          instructionWrite <= {lwOP, 5'd0,
5'd23, 16'd8}; //8c170008
                                          loaderData <= 255;
                                          end
                                          8: begin
                                          instructionWrite <= {lwOP, 5'd0,
5'd8, 16'd9}; //8c080009
                                          loaderData <= 195;
                                          end
```

```
5'd9, 16'd10}; //8c09000a

5'd10, 16'd11}; //8c0a000b

5'd17, 5'd11, 5'd0, subF}; //2110022

5'd8, 16'd17}; //15680011

5'd9, 5'd18, 5'd0, addF}; //2490020

5'd8, 5'd19, 5'd0, subF}; //2480022

5'd21, 5'd12, 5'd0, orF}; //2950025

26'd20}; //8000014

5'd0, 5'd18, 5'd3, sllF}; //24000c0

5'd10, 5'd19, 5'd0, addF}; //26a0020

5'd21, 5'd12, 5'd0, andF}; //2950024
```

```
9: begin
instructionWrite <= {lwOP, 5'd0,

loaderData <= 3;
end
10: begin
instructionWrite <= {lwOP, 5'd0,

loaderData <= 4;
end
11: begin
instructionWrite <= {rOP, 5'd16,

loaderData <= 7;
end
12: begin
instructionWrite <= {bgtOP, 5'd11,

loaderData <= 0;
end
13: begin
instructionWrite <= {rOP, 5'd18,

loaderData <= 0;
end
14: begin
instructionWrite <= {rOP, 5'd18,

loaderData <= 0;
end
15: begin
instructionWrite <= {rOP, 5'd20,

loaderData <= 0;
end
16: begin
instructionWrite <= {jumpOP,

loaderData <= 0;
end
17: begin
instructionWrite <= {rOP, 5'd18,

loaderData <= 0;
end
18: begin
instructionWrite <= {rOP, 5'd0,

loaderData <= 0;
end
19: begin
instructionWrite <= {rOP, 5'd20,

loaderData <= 0;
end
20: begin
```

```
                        instructionWrite <= {rOP, 5'd16,
5'd17, 5'd16, 5'd0, addF}; //2110020
                        loaderData <= 0;
                        end
                        21: begin
                        instructionWrite <= {rOP, 5'd20,
5'd21, 5'd13, 5'd0, xorF}; //2950026
                        loaderData <= 0;
                        end
                        22: begin
                        instructionWrite <= {rOP, 5'd13,
5'd23, 5'd13, 5'd0, andF}; //1b70024
                        loaderData <= 0;
                        end
                        23: begin
                        instructionWrite <= {swOP, 5'd0,
5'd16, 16'd0}; //ac100000
                        loaderData <= 0;
                        end
                        24: begin
                        instructionWrite <= {swOP, 5'd0,
5'd18, 16'd2}; //ac120002
                        loaderData <= 0;
                        end
                        25: begin
                        instructionWrite <= {swOP, 5'd0,
5'd19, 16'd3}; //ac130003
                        loaderData <= 0;
                        end
                        26: begin
                        instructionWrite <= {swOP, 5'd0,
5'd13, 16'd7}; //ac0d0007
                        loaderData <= 0;
                        end
                        default: begin
                              instructionWrite <= 0;
                              loaderData <= 0;
                        end
                  endcase
                  currentState <= sramOn;
            end
            sramOn: begin
                  loaderSRAMEn <= 1;
                  IMWriteEn <= 1;
                  currentState <= sramOff;
            end
            sramOff: begin
                  loaderMemoryADX <= loaderMemoryADX + 1;
                  instructionWriteADX <= instructionWriteADX
+ 1;
                  loaderSRAMEn <= 0;
                  IMWriteEn <= 0;
                  if(loaderMemoryADX >= 31) currentState <=
stop;
                  else currentState <= increment;
            end
```

```
                              stop: begin
                                    loadProgEn <= 0;
                                    instructionWriteADX <= 0;
                                    instructionWrite <= 0;
                                    IMWriteEn <= 0;
                                    loaderMemoryADX <= 0;
                                    loaderSRAMEn <= 0;
                                    loaderReadNWrite <= 1;
                                    loaderData <= 0;
                                    if(!nrst) currentState <= init;
                                    else currentState <= stop;
                              end
                              default: currentState <= stop;
                        endcase
                  end
            end

endmodule
```

Appendix E. Common CPU Support Source Code

E.1 ALU Support Modules

```
/*    AdderFunction Module
*     Description:      Add Function for ALU using a Carry Lookahead
architechture
*                             via calling a 4bit Adder several times.
*     Inputs:                 X = 32bit input signal 1
*                             Y = 32bit input signal 2
*     Output:                 S = 32bit output sum of signals 1 and 2
*                             V = Overflow bit
*                             C = Carry out bit
*/
module arithmeticADD(OUT,OVERFLOW,CARRY,A,B);
      parameter         n = 32;
      input [n-1:0] A, B;
      output  [n-1:0] OUT;
      output                  OVERFLOW,CARRY;
      wire C0, C1, C2, C3, C4, C5, C6;
      wire nA, nB, nOUT, V0, V1;

      Add4bit     a4b0(A[3:0],B[3:0],1'b0,OUT[3:0],C0);
      Add4bit     a4b1(A[7:4],B[7:4],C0,OUT[7:4],C1);
      Add4bit     a4b2(A[11:8],B[11:8],C1,OUT[11:8],C2);
      Add4bit     a4b3(A[15:12],B[15:12],C2,OUT[15:12],CARRY);
      Add4bit     a4b4(A[19:16],B[19:16],CARRY,OUT[19:16],C4);
      Add4bit     a4b5(A[23:20],B[23:20],C4,OUT[23:20],C5);
      Add4bit     a4b6(A[27:24],B[27:24],C5,OUT[27:24],C6);
      Add4bit     a4b7(A[31:28],B[31:28],C6,OUT[31:28],C3);

      not n0(nA,A[15]);
      not n1(nB,B[15]);
      not n2(nOUT,OUT[15]);
```

```
      and a0(V0,A[15],B[15],nOUT);
      and a1(V1,nA,nB,OUT[15]);
      or val(OVERFLOW,V0,V1);
endmodule

/*    4-bit Carry Lookahead Adder (CLA)
*     Description:      This code combines two 4-bit signals via addition
*     Inputs:                 ci = Carry-in: the remainder from the previous
adding of two seperate bits together
*                             x = 1st n-bit input to be added to y
*                             y = 2nd n-bit input to be added to x
*     Outputs:          s = Sum of the addition of x and y inputs
*                             co = Carry-out/Overflow: the remainder of the
last two bits added together
*/
module Add4bit(x,y,ci,s,co);
      input             [3:0] x,y;
      input                   ci;
      output            [3:0] s;
      output                        co;
      wire        [3:0] c,p,g,r;
      wire        [9:0] h;

      assign                        c[0] = ci;
      xor                     x0(p[0],x[0],y[0]),x1(p[1],x[1],y[1]),
                              x2(p[2],x[2],y[2]),x3(p[3],x[3],y[3]),
                              x4(s[0],c[0],p[0]),x5(s[1],c[1],p[1]),
                              x6(s[2],c[2],p[2]),x7(s[3],c[3],p[3]);
      and                     a0(g[0],x[0],y[0]),a1(g[1],x[1],y[1]),
                              a2(g[2],x[2],y[2]),a3(g[3],x[3],y[3]),

      a4(h[0],p[0],c[0]),a5(h[1],p[1],p[0],c[0]),

      a6(h[2],p[1],g[0]),a7(h[3],p[2],p[1],p[0],c[0]),

      a8(h[4],p[2],p[1],g[0]),a9(h[5],p[2],g[1]),
                              a10(h[6],p[3],p[2],p[1],p[0],c[0]),
                              a11(h[7],p[3],p[2],p[1],g[0]),

      a12(h[8],p[3],p[2],g[1]),a13(h[9],p[3],g[2]);
      or
      r0(c[1],h[0],g[0]),r1(c[2],h[1],h[2],g[1]),
                              r2(c[3],h[3],h[4],h[5],g[2]),
                              r3(co,h[6],h[7],h[8],h[9],g[3]);
endmodule
```

```
module arithmeticSUB(OUT,OVERFLOW,CARRY,A,B);
      parameter         n = 32;
      input [n-1:0] A,B;
      output      [n-1:0] OUT;
      output                  OVERFLOW,CARRY;
      wire  [n-1:0]     nB;
      wire tmpflow;
```

```
      assign nB = ~B + 1;
      arithmeticADD     Sub0(OUT,tmpflow,CARRY,A,nB);
      assign OVERFLOW = tmpflow | (nB[15]&B[15]);
endmodule
```

```
module logicalSLT(OUT, A, B);
      output wire [31:0] OUT;
      input wire [31:0] A, B;
      wire
w0,w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,w11,w12,w13,w14,w15,w16,w17,w18,w19,w20,w21,
w22,w23,w24,w25,w26,w27,w28,w29,w30;
      logicalSLTunit blah0(w0,A[0],B[0],1'b1);
      logicalSLTunit blah1(w1,A[1],B[1],w0);
      logicalSLTunit blah2(w2,A[2],B[2],w1);
      logicalSLTunit blah3(w3,A[3],B[3],w2);
      logicalSLTunit blah4(w4,A[4],B[4],w3);
      logicalSLTunit blah5(w5,A[5],B[5],w4);
      logicalSLTunit blah6(w6,A[6],B[6],w5);
      logicalSLTunit blah7(w7,A[7],B[7],w6);
      logicalSLTunit blah8(w8,A[8],B[8],w7);
      logicalSLTunit blah9(w9,A[9],B[9],w8);
      logicalSLTunit blah10(w10,A[10],B[10],w9);
      logicalSLTunit blah11(w11,A[11],B[11],w10);
      logicalSLTunit blah12(w12,A[12],B[12],w11);
      logicalSLTunit blah13(w13,A[13],B[13],w12);
      logicalSLTunit blah14(w14,A[14],B[14],w13);
      logicalSLTunit blah15(w15,A[15],B[15],w14);
      logicalSLTunit blah16(w16,A[16],B[16],w15);
      logicalSLTunit blah17(w17,A[17],B[17],w16);
      logicalSLTunit blah18(w18,A[18],B[18],w17);
      logicalSLTunit blah19(w19,A[19],B[19],w18);
      logicalSLTunit blah20(w20,A[20],B[20],w19);
      logicalSLTunit blah21(w21,A[21],B[21],w20);
      logicalSLTunit blah22(w22,A[22],B[22],w21);
      logicalSLTunit blah23(w23,A[23],B[23],w22);
      logicalSLTunit blah24(w24,A[24],B[24],w23);
      logicalSLTunit blah25(w25,A[25],B[25],w24);
      logicalSLTunit blah26(w26,A[26],B[26],w25);
      logicalSLTunit blah27(w27,A[27],B[27],w26);
      logicalSLTunit blah28(w28,A[28],B[28],w27);
      logicalSLTunit blah29(w29,A[29],B[29],w28);
      logicalSLTunit blah30(w30,A[30],B[30],w29);
      logicalSLTunit blah31(OUT[0],A[31],B[31],w30);
      assign OUT[31:1] = 31'b0;
endmodule

module logicalSLTunit(OUT,A,B,CARRY);
      output OUT;
      input A,B,CARRY;
      wire w0,w1,w2,w3,w4,w5;

      not blah0(w0,A);
      not blah1(w1,B);
      xor blah2(w2,A,B);
      not blah3(w3,w2);
```

```
      and blah4(w4,w3,CARRY);
      and blah5(w5,w0,B);
      or blah6(OUT,w4,w5);
endmodule
```

```
module logicalXOR(OUT, A, B);
      output [31:0] OUT;
      input [31:0] A, B;

      xor blah0(OUT[0], A[0] , B[0]);
      xor blah1(OUT[1], A[1] , B[1]);
      xor blah2(OUT[2], A[2] , B[2]);
      xor blah3(OUT[3], A[3] , B[3]);
      xor blah4(OUT[4], A[4] , B[4]);
      xor blah5(OUT[5], A[5] , B[5]);
      xor blah6(OUT[6], A[6] , B[6]);
      xor blah7(OUT[7], A[7] , B[7]);
      xor blah8(OUT[8], A[8] , B[8]);
      xor blah9(OUT[9], A[9] , B[9]);
      xor blah10(OUT[10], A[10] , B[10]);
      xor blah11(OUT[11], A[11] , B[11]);
      xor blah12(OUT[12], A[12] , B[12]);
      xor blah13(OUT[13], A[13] , B[13]);
      xor blah14(OUT[14], A[14] , B[14]);
      xor blah15(OUT[15], A[15] , B[15]);
      xor blah16(OUT[16], A[16] , B[16]);
      xor blah17(OUT[17], A[17] , B[17]);
      xor blah18(OUT[18], A[18] , B[18]);
      xor blah19(OUT[19], A[19] , B[19]);
      xor blah20(OUT[20], A[20] , B[20]);
      xor blah21(OUT[21], A[21] , B[21]);
      xor blah22(OUT[22], A[22] , B[22]);
      xor blah23(OUT[23], A[23] , B[23]);
      xor blah24(OUT[24], A[24] , B[24]);
      xor blah25(OUT[25], A[25] , B[25]);
      xor blah26(OUT[26], A[26] , B[26]);
      xor blah27(OUT[27], A[27] , B[27]);
      xor blah28(OUT[28], A[28] , B[28]);
      xor blah29(OUT[29], A[29] , B[29]);
      xor blah30(OUT[30], A[30] , B[30]);
      xor blah31(OUT[31], A[31] , B[31]);
endmodule
```

```
module logicalAND(OUT, A, B);
      output [31:0] OUT;
      input [31:0] A, B;

      and blah0(OUT[0], A[0] , B[0]);
      and blah1(OUT[1], A[1] , B[1]);
      and blah2(OUT[2], A[2] , B[2]);
      and blah3(OUT[3], A[3] , B[3]);
      and blah4(OUT[4], A[4] , B[4]);
      and blah5(OUT[5], A[5] , B[5]);
```

```
      and blah6(OUT[6], A[6] , B[6]);
      and blah7(OUT[7], A[7] , B[7]);
      and blah8(OUT[8], A[8] , B[8]);
      and blah9(OUT[9], A[9] , B[9]);
      and blah10(OUT[10], A[10] , B[10]);
      and blah11(OUT[11], A[11] , B[11]);
      and blah12(OUT[12], A[12] , B[12]);
      and blah13(OUT[13], A[13] , B[13]);
      and blah14(OUT[14], A[14] , B[14]);
      and blah15(OUT[15], A[15] , B[15]);
      and blah16(OUT[16], A[16] , B[16]);
      and blah17(OUT[17], A[17] , B[17]);
      and blah18(OUT[18], A[18] , B[18]);
      and blah19(OUT[19], A[19] , B[19]);
      and blah20(OUT[20], A[20] , B[20]);
      and blah21(OUT[21], A[21] , B[21]);
      and blah22(OUT[22], A[22] , B[22]);
      and blah23(OUT[23], A[23] , B[23]);
      and blah24(OUT[24], A[24] , B[24]);
      and blah25(OUT[25], A[25] , B[25]);
      and blah26(OUT[26], A[26] , B[26]);
      and blah27(OUT[27], A[27] , B[27]);
      and blah28(OUT[28], A[28] , B[28]);
      and blah29(OUT[29], A[29] , B[29]);
      and blah30(OUT[30], A[30] , B[30]);
      and blah31(OUT[31], A[31] , B[31]);
endmodule
```

```
module logicalOR(OUT, A, B);
      output [31:0] OUT;
      input [31:0] A, B;

      or blah0(OUT[0], A[0] , B[0]);
      or blah1(OUT[1], A[1] , B[1]);
      or blah2(OUT[2], A[2] , B[2]);
      or blah3(OUT[3], A[3] , B[3]);
      or blah4(OUT[4], A[4] , B[4]);
      or blah5(OUT[5], A[5] , B[5]);
      or blah6(OUT[6], A[6] , B[6]);
      or blah7(OUT[7], A[7] , B[7]);
      or blah8(OUT[8], A[8] , B[8]);
      or blah9(OUT[9], A[9] , B[9]);
      or blah10(OUT[10], A[10] , B[10]);
      or blah11(OUT[11], A[11] , B[11]);
      or blah12(OUT[12], A[12] , B[12]);
      or blah13(OUT[13], A[13] , B[13]);
      or blah14(OUT[14], A[14] , B[14]);
      or blah15(OUT[15], A[15] , B[15]);
      or blah16(OUT[16], A[16] , B[16]);
      or blah17(OUT[17], A[17] , B[17]);
      or blah18(OUT[18], A[18] , B[18]);
      or blah19(OUT[19], A[19] , B[19]);
      or blah20(OUT[20], A[20] , B[20]);
      or blah21(OUT[21], A[21] , B[21]);
      or blah22(OUT[22], A[22] , B[22]);
```

```
      or blah23(OUT[23], A[23] , B[23]);
      or blah24(OUT[24], A[24] , B[24]);
      or blah25(OUT[25], A[25] , B[25]);
      or blah26(OUT[26], A[26] , B[26]);
      or blah27(OUT[27], A[27] , B[27]);
      or blah28(OUT[28], A[28] , B[28]);
      or blah29(OUT[29], A[29] , B[29]);
      or blah30(OUT[30], A[30] , B[30]);
      or blah31(OUT[31], A[31] , B[31]);
endmodule
```

```
//OUT is A shifted by B[3:0] units
module logicalSLL(OUT, A, B);
      output [31:0] OUT;
      input [31:0] A, B;
      wire [31:0] w0,w1,w2;

      muxArray blah0(w0,{A[30:0],1'b0},A, B[0]);
      muxArray blah1(w1,{w0[29:0],2'b00},w0, B[1]);
      muxArray blah2(w2,{w1[27:0],4'b0000},w1, B[2]);
      muxArray blah3(OUT,{w2[23:0],8'b00000000},w2, B[3]);

endmodule

// if ctrl is 1 take A, 0 take B
module muxArray(OUT, A, B, CTRL);
      output [31:0] OUT;
      input [31:0] A, B;
      input CTRL;
      mux2to1 blah0(OUT[0],{A[0],B[0]},CTRL);
      mux2to1 blah1(OUT[1],{A[1],B[1]},CTRL);
      mux2to1 blah2(OUT[2],{A[2],B[2]},CTRL);
      mux2to1 blah3(OUT[3],{A[3],B[3]},CTRL);
      mux2to1 blah4(OUT[4],{A[4],B[4]},CTRL);
      mux2to1 blah5(OUT[5],{A[5],B[5]},CTRL);
      mux2to1 blah6(OUT[6],{A[6],B[6]},CTRL);
      mux2to1 blah7(OUT[7],{A[7],B[7]},CTRL);
      mux2to1 blah8(OUT[8],{A[8],B[8]},CTRL);
      mux2to1 blah9(OUT[9],{A[9],B[9]},CTRL);
      mux2to1 blah10(OUT[10],{A[10],B[10]},CTRL);
      mux2to1 blah11(OUT[11],{A[11],B[11]},CTRL);
      mux2to1 blah12(OUT[12],{A[12],B[12]},CTRL);
      mux2to1 blah13(OUT[13],{A[13],B[13]},CTRL);
      mux2to1 blah14(OUT[14],{A[14],B[14]},CTRL);
      mux2to1 blah15(OUT[15],{A[15],B[15]},CTRL);
      mux2to1 blah16(OUT[16],{A[16],B[16]},CTRL);
      mux2to1 blah17(OUT[17],{A[17],B[17]},CTRL);
      mux2to1 blah18(OUT[18],{A[18],B[18]},CTRL);
      mux2to1 blah19(OUT[19],{A[19],B[19]},CTRL);
      mux2to1 blah20(OUT[20],{A[20],B[20]},CTRL);
      mux2to1 blah21(OUT[21],{A[21],B[21]},CTRL);
      mux2to1 blah22(OUT[22],{A[22],B[22]},CTRL);
      mux2to1 blah23(OUT[23],{A[23],B[23]},CTRL);
      mux2to1 blah24(OUT[24],{A[24],B[24]},CTRL);
      mux2to1 blah25(OUT[25],{A[25],B[25]},CTRL);
```

```
      mux2to1 blah26(OUT[26],{A[26],B[26]},CTRL);
      mux2to1 blah27(OUT[27],{A[27],B[27]},CTRL);
      mux2to1 blah28(OUT[28],{A[28],B[28]},CTRL);
      mux2to1 blah29(OUT[29],{A[29],B[29]},CTRL);
      mux2to1 blah30(OUT[30],{A[30],B[30]},CTRL);
      mux2to1 blah31(OUT[31],{A[31],B[31]},CTRL);
endmodule
```

E.2 Clock Prescaler Module

```
/**
 * this is the precaler module used to scale down
 * the frequencie of the main clock oscillator
 * it is simply a 32 bit synchronous down counter
 */

module clockPrescaler(prescaler, clk, rst);

      // parameter declarations
      parameter startPrescaler = 1'b1;

      // in/out delaractions
      input clk, rst;
      output wire prescaler;
      wire [31:0] psc;

      // reg/wires declaration
      wire clk, rst;
      wire [31:1] t;

      // concurrent statements, modules instantiation
      tFlipFlop tff0(psc[0], , startPrescaler, clk, rst);
      tFlipFlop tff1(psc[1], , psc[0] , clk, rst);
      tFlipFlop tff2(psc[2], , t[1], clk, rst);
      tFlipFlop tff3(psc[3], , t[2], clk, rst);
      tFlipFlop tff4(psc[4], , t[3], clk, rst);
      tFlipFlop tff5(psc[5], , t[4], clk, rst);
      tFlipFlop tff6(psc[6], , t[5], clk, rst);
      tFlipFlop tff7(psc[7], , t[6], clk, rst);
      tFlipFlop tff8(psc[8], , t[7], clk, rst);
      tFlipFlop tff9(psc[9], , t[8], clk, rst);
      tFlipFlop tff10(psc[10], , t[9], clk, rst);
      tFlipFlop tff11(psc[11], , t[10], clk, rst);
      tFlipFlop tff12(psc[12], , t[11], clk, rst);
      tFlipFlop tff13(psc[13], , t[12], clk, rst);
      tFlipFlop tff14(psc[14], , t[13], clk, rst);
      tFlipFlop tff15(psc[15], , t[14], clk, rst);
      tFlipFlop tff16(psc[16], , t[15], clk, rst);
      tFlipFlop tff17(psc[17], , t[16], clk, rst);
      tFlipFlop tff18(psc[18], , t[17], clk, rst);
      tFlipFlop tff19(psc[19], , t[18], clk, rst);
      tFlipFlop tff20(psc[20], , t[19], clk, rst);
      tFlipFlop tff21(psc[21], , t[20], clk, rst);
      tFlipFlop tff22(psc[22], , t[21], clk, rst);
      tFlipFlop tff23(psc[23], , t[22], clk, rst);
```

```
        tFlipFlop tff24(psc[24], , t[23], clk, rst);
        tFlipFlop tff25(psc[25], , t[24], clk, rst);
        tFlipFlop tff26(psc[26], , t[25], clk, rst);
        tFlipFlop tff27(psc[27], , t[26], clk, rst);
        tFlipFlop tff28(psc[28], , t[27], clk, rst);
        tFlipFlop tff29(psc[29], , t[28], clk, rst);
        tFlipFlop tff30(psc[30], , t[29], clk, rst);
        tFlipFlop tff31(psc[31], , t[30], clk, rst);

        // gate instantiations
        and and0(t[1], psc[0], psc[1]);
        and and1(t[2], t[1], psc[2]);
        and and2(t[3], t[2], psc[3]);
        and and3(t[4], t[3], psc[4]);
        and and4(t[5], t[4], psc[5]);
        and and5(t[6], t[5], psc[6]);
        and and6(t[7], t[6], psc[7]);
        and and7(t[8], t[7], psc[8]);
        and and8(t[9], t[8], psc[9]);
        and and9(t[10], t[9], psc[10]);
        and and10(t[11], t[10], psc[11]);
        and and11(t[12], t[11], psc[12]);
        and and12(t[13], t[12], psc[13]);
        and and13(t[14], t[13], psc[14]);
        and and14(t[15], t[14], psc[15]);
        and and15(t[16], t[15], psc[16]);
        and and16(t[17], t[16], psc[17]);
        and and17(t[18], t[17], psc[18]);
        and and18(t[19], t[18], psc[19]);
        and and19(t[20], t[19], psc[20]);
        and and20(t[21], t[20], psc[21]);
        and and21(t[22], t[21], psc[22]);
        and and22(t[23], t[22], psc[23]);
        and and23(t[24], t[23], psc[24]);
        and and24(t[25], t[24], psc[25]);
        and and25(t[26], t[25], psc[26]);
        and and26(t[27], t[26], psc[27]);
        and and27(t[28], t[27], psc[28]);
        and and28(t[29], t[28], psc[29]);
        and and29(t[30], t[29], psc[30]);

        assign prescaler = psc[1];

endmodule
```

E.3 Address Register File Decoder

```
module adx_decoder(reg_select, adx_in);
output [31:0] reg_select;
input [4:0] adx_in;
wire not_adx_in_4;
not(not_adx_in_4, adx_in[4]);

decoder4_16 blah(reg_select[15:0], not_adx_in_4, adx_in[3:0]);
decoder4_16 blah1(reg_select[31:16], adx_in[4], adx_in[3:0]);
```

```
endmodule

module decoder1_2(out, en, in);
output [1:0] out;
input en, in;
wire not_in;
not(not_in, in);
and(out[0], en, not_in);
and(out[1], en, in);
endmodule

module decoder2_4(out, en, in);
output [3:0] out;
input [1:0] in;
input en;
wire [1:0] tmp;
wire not_in_1;
not(not_in_1, in[1]);
and(tmp[0], en, not_in_1);
and(tmp[1], en, in[1]);

decoder1_2 blah(out[1:0], tmp[0], in[0]);
decoder1_2 blah1(out[3:2], tmp[1], in[0]);
endmodule

module decoder3_8(out, en, in);
output [7:0] out;
input [2:0] in;
input en;
wire [1:0] tmp;
wire not_in_2;
not(not_in_2, in[2]);
and(tmp[0], en, not_in_2);
and(tmp[1], en, in[2]);

decoder2_4 blah(out[3:0], tmp[0], in[1:0]);
decoder2_4 blah1(out[7:4], tmp[1], in[1:0]);
endmodule

module decoder4_16(out, en, in);
output [15:0] out;
input [3:0] in;
input en;
wire [1:0] tmp;
wire not_in_3;
not(not_in_3, in[3]);
and(tmp[0], en, not_in_3);
and(tmp[1], en, in[3]);

decoder3_8 blah(out[7:0], tmp[0], in[2:0]);
decoder3_8 blah1(out[15:8], tmp[1], in[2:0]);

endmodule
```

E.4 Flip Flop Modules

```
module D_FlipFlop(q, qBar, D, clk, rst, E);
      input D, clk, rst, E;
      output q, qBar;
      reg q;

      not n1 (qBar, q);

      always@ (negedge rst or posedge E)  //Rest Asynchronously
      begin                               //Clocking Synchronously
            if(rst == 0)                  //Reseting DFF if rst = 0
                  q = 0;
            else
            begin
                        q = D;
            end
      end
endmodule
```

```
/**
 * D flip flop module
 */
module dFlipFlop(q, qBar, d, clk, rst);

      // in/out declarations
      input d, clk, rst;
      output q, qBar;

      // reg/wire declarations
      reg q;
      wire d, clk, rst, qBar;

      // procedural statements
      always@ (negedge rst or posedge clk)
      begin
            if(rst == 0)
                  q = 1'b0;
            else
                  q = d;
      end

      // gate instantiations
      not n1 (qBar, q);

endmodule
```

```
/**
 * T flipflop created from an xor gate and a d flipflop
 */
module tFlipFlop(q, qBar, t, clk, rst);

      // in/out delaractions
      input t, clk, rst;
```

```
      output qBar, q;

      // reg/wire declaration
      wire t, clk, rst, qBar, q;
      wire d;

      // concurrent statements, modules instantiation
      dFlipFlop dff(q, qBar, d, clk, rst);

      // gate instatiations
      xor xor0 (d, q, t);

endmodule
```

E.5 Multiplexer Modules

```
/**********************
* Multiplexer Modules
**********************/
//32-to-1 Multiplexer
module mux32to1(OUT,IN,S);
      input [31:0] IN;
      input [4:0] S;
      output OUT;
      wire [1:0] tmp;


      mux16to1 blah0(tmp[0],IN[15:0], S[3:0]);
      mux16to1 blah1(tmp[1],IN[31:16], S[3:0]);
      mux2to1 blah2(OUT, tmp,S[4]);
endmodule

module mux16to1(OUT,IN,S);
      input [15:0] IN;
      input [3:0] S;
      output OUT;
      wire [1:0] tmp;

      mux8to1 blah0(tmp[0],IN[7:0],S[2:0]);
      mux8to1 blah1(tmp[1],IN[15:8],S[2:0]);
      mux2to1 blah2(OUT, tmp, S[3]);
endmodule

module mux8to1(OUT,IN,S);
      input [7:0] IN;
      input [2:0] S;
      output OUT;
      wire [1:0] tmp;

      mux4to1 blah0(tmp[0],IN[3:0],S[1:0]);
      mux4to1 blah1(tmp[1],IN[7:4],S[1:0]);
      mux2to1 blah2(OUT, tmp, S[2]);
endmodule

module mux4to1(OUT,IN,S);
```

```
      input [3:0] IN;
      input [1:0] S;
      output OUT;
      wire [1:0] tmp;

      mux2to1 blah0(tmp[0],IN[1:0],S[0]);
      mux2to1 blah1(tmp[1],IN[3:2],S[0]);
      mux2to1 blah2(OUT, tmp, S[1]);
endmodule

module mux2to1(OUT,IN,S);
      input [1:0] IN;
      input S;
      output OUT;

      wire SN;
      wire tmp1, tmp2;

      not blah0(SN,S);
      and blah1(tmp1, IN[0], SN);
      and blah2(tmp2, IN[1], S);
      or blah3(OUT, tmp1,tmp2);
endmodule
```

```
/***********************
* Multiplexer Modules
***********************/
//32-to-1 Multiplexer
module mux32to1v1(X,S,f);
      input [31:0]      X;
      input [4:0] S;
      output      reg         f;

      always @(S,X)
      begin
      case(S)
            5'b00000:   f = X[0];
            5'b00001:   f = X[1];
            5'b00010:   f = X[2];
            5'b00011:   f = X[3];
            5'b00100:   f = X[4];
            5'b00101:   f = X[5];
            5'b00110:   f = X[6];
            5'b00111:   f = X[7];
            5'b01000:   f = X[8];
            5'b01001:   f = X[9];

            5'b01010:   f = X[10];
            5'b01011:   f = X[11];
            5'b01100:   f = X[12];
            5'b01101:   f = X[13];
            5'b01110:   f = X[14];
            5'b01111:   f = X[15];
            5'b10000:   f = X[16];
```

```
            5'b10001:   f = X[17];
            5'b10010:   f = X[18];
            5'b10011:   f = X[19];

            5'b10100:   f = X[20];
            5'b10101:   f = X[21];
            5'b10110:   f = X[22];
            5'b10111:   f = X[23];
            5'b11000:   f = X[24];
            5'b11001:   f = X[25];
            5'b11010:   f = X[26];
            5'b11011:   f = X[27];
            5'b11100:   f = X[28];
            5'b11101:   f = X[29];

            5'b11110:   f = X[30];
            5'b11111:   f = X[31];
            default:    f = 1'bz;
      endcase
      end
endmodule
```

E.6 Register Builder Module

```
//Creates an n sized register out of DFF's. Defaulted to n = 32.
module regB(D, Q, clk, rst, E);
      parameter         n = 32;                 //Defines vector size of the
buses
      output      [n-1:0]     Q;                      //Output data from each
DFF
      input [n-1:0] D;                    //Input data to each DFF
      input             clk, rst, E;
      wire  [n-1:0]     qBar;

      //builds n-bit register with parallel inputs with enable function
      D_FlipFlop  DFlop0(Q[0], qBar[0], D[0], clk, rst, E),
                        DFlop1(Q[1], qBar[1], D[1], clk, rst, E),
                        DFlop2(Q[2], qBar[2], D[2], clk, rst, E),
                        DFlop3(Q[3], qBar[3], D[3], clk, rst, E),
                        DFlop4(Q[4], qBar[4], D[4], clk, rst, E),
                        DFlop5(Q[5], qBar[5], D[5], clk, rst, E),
                        DFlop6(Q[6], qBar[6], D[6], clk, rst, E),
                        DFlop7(Q[7], qBar[7], D[7], clk, rst, E),
                        DFlop8(Q[8], qBar[8], D[8], clk, rst, E),
                        DFlop9(Q[9], qBar[9], D[9], clk, rst, E),

                        DFlop10(Q[10], qBar[10], D[10], clk, rst, E),
                        DFlop11(Q[11], qBar[11], D[11], clk, rst, E),
                        DFlop12(Q[12], qBar[12], D[12], clk, rst, E),
                        DFlop13(Q[13], qBar[13], D[13], clk, rst, E),
                        DFlop14(Q[14], qBar[14], D[14], clk, rst, E),
                        DFlop15(Q[15], qBar[15], D[15], clk, rst, E),
                        DFlop16(Q[16], qBar[16], D[16], clk, rst, E),
                        DFlop17(Q[17], qBar[17], D[17], clk, rst, E),
                        DFlop18(Q[18], qBar[18], D[18], clk, rst, E),
                        DFlop19(Q[19], qBar[19], D[19], clk, rst, E),
```

```
                    DFlop20(Q[20], qBar[20], D[20], clk, rst, E),
                    DFlop21(Q[21], qBar[21], D[21], clk, rst, E),
                    DFlop22(Q[22], qBar[22], D[22], clk, rst, E),
                    DFlop23(Q[23], qBar[23], D[23], clk, rst, E),
                    DFlop24(Q[24], qBar[24], D[24], clk, rst, E),
                    DFlop25(Q[25], qBar[25], D[25], clk, rst, E),
                    DFlop26(Q[26], qBar[26], D[26], clk, rst, E),
                    DFlop27(Q[27], qBar[27], D[27], clk, rst, E),
                    DFlop28(Q[28], qBar[28], D[28], clk, rst, E),
                    DFlop29(Q[29], qBar[29], D[29], clk, rst, E),

                    DFlop30(Q[30], qBar[30], D[30], clk, rst, E),
                    DFlop31(Q[31], qBar[31], D[31], clk, rst, E);

endmodule
```