ECE/CS 552: INTRODUCTION TO COMPUTER ARCHITECTURE

Project Description – Phase 1

Due on Friday, October 18, 2019, 11:59pm

In stage 1, the task is to design and implement a single cycle processor. The implementation should be in Verilog. Either Modelsim or Icarus should be used as the simulator to verify the design. Below, the WISC-F19 ISA specification will be introduced; and then design details and requirements will be discussed. You are required to follow the Verilog rules as specified by the rules document uploaded on canvas. NOTE: the only exception to this rule is the required use of inout (tri-state logic) in the register file. Do not use inout anywhere else in your design.

Some portions of the project requirements are covered in the homeworks. You are free to reuse those modules in your project.

1. WISC-F19 ISA Specifications

WISC-F19 contains a set of 16 instructions specified for a 16-bit data-path with load/store architecture.

The WISC-F19 memory is byte addressable, even though all accesses (instruction fetches, loads, stores) are restricted to half-word (2-byte), naturally-aligned accesses.

WISC-F19 has a register file, and a 3-bit FLAG register. The register file comprises sixteen 16-bit registers and has 2 read ports and 1 write port. Register $0 is hardwired to 0x0000. The FLAG register contains three bits: Zero (Z), Overflow (V), and Sign (N).

WISC-F19's instructions can be categorized into three major classes: Compute, Memory, and Control.

1.1 Compute Instructions

Six arithmetic and logical instructions belong to this category. They are ADD, PADDSB, SUB, XOR, SLL, SRA, ROR and RED.

The assembly level syntax for ADD, PADDSB, SUB, XOR and RED is:

Opcode rd, rs, rt

The two operands are[1] (rs) and (rt) and the result is written to the destination register rd.

The ADD, PADDSB, SUB and RED instructions operate on the two operands (rs, rt) in two's complement representation and save the result to register rd.

The ADD and SUB instructions will use saturating arithmetic. Meaning if a result exceeds the most positive number $2^{15} - 1$ (i.e., positive overflow), then the result is saturated to $2^{15} - 1$. Likewise, if the result is smaller than the most negative number -2^{15} (i.e., negative overflow) then the result is saturated to -2^{15}.

The XOR instruction performs bitwise XOR on the two operands and saves the result in register rd.

The PADDSB instruction performs four half-byte additions in parallel to realize sub-word parallelism. Specifically, each of the four half bytes (4-bits) will be treated as separate numbers stored in a single word as a byte vector. When PADDSB is performed, the four numbers will be added separately. To be more specific, let the contents in rs and rt be aaaa_bbbb_cccc_dddd and eeee_ffff_gggg_hhhh, respectively, where a, b, c, d, e, f, g and h in {0, 1}. Then after execution of PADDSB, the contents of rd will be {sat(aaaa+eeee), sat(bbbb+ffff), sat(cccc+gggg), sat(dddd+hhhh)}. The four half-bytes of the result should be saturated separately, meaning if a result exceeds the most positive number $2^3 - 1$, then the result is saturated to $2^3 - 1$. And if the result were to drop below the most negative number -

[1] (rx) stands for the contents of register rx

2^3, then the result is saturated to -2^3.

The RED instruction performs reduction on 4 byte-size operands (i.e., 2 bytes each from 2 registers). To be more specific, let the contents in rs and rt be aaaaaaaa_bbbbbbbb and cccccccc_dddddddd, respectively, where a, b, c and d in {0, 1}. Then after the execution of RED, the contents of rd will be the sign-extended value of ((aaaaaaaa+ cccccccc) + (bbbbbbbb+dddddddd)).

The SLL, SRA and ROR instructions perform logical left shift, arithmetic right shift and right rotation, respectively , of (rs) by the number of bits specified in the imm field and saves the result in register rd. For ROR, bits are rotated off the right (least significant) and are inserted into the vacated bit positions on the left (most significant).

They have the following assembly level syntax:

Opcode rd, rs, imm

The imm field is a 4-bit immediate operand in unsigned representation for the SLL, SRA and ROR instructions.

The machine level encoding for each arithmetic/logic instruction is:

0aaa dddd ssss tttt

where 0aaa represents the opcode (see Table 2), and dddd and ssss represent the rd and rs registers, respectively. The tttt field represents either the rt register or the imm field.

1.2 Memory Instructions

There are four instructions of this category: LW, SW, LLB and LHB.

The first group of these instructions are LW (load word) and SW (save word). The assembly level syntax for LW and SW is:

Opcode rt, rs, offset

The LW instruction loads register rt with contents from the memory location specified by register rs plus the immediate offset. The signed value offset is shifted left by 1, sign-extended and added to the contents of register rs to compute the address of the memory location to load. The address is always even (the low-order bit will always be zero).

The SW instruction saves (rt) to the location specified by the register rs plus the immediate offset. The address of the memory location is computed as in LW.

The machine level encoding of these two instructions is:

100a tttt ssss oooo

where 100a specifies the opcode, tttt specifies rt, ssss specifies rs, and oooo is the offset in two's complement representation, but right-shifted by 1 bit (since the LSb will always be zero, there is no reason to encode that bit in the instruction word). The address is computed as addr = (Reg[ssss] & 0xFFFE) + (sign-extend(oooo) << 1).

The next two instructions are of the Load Immediate type: LLB (load lower byte) and LHB (load higher byte). The assembly level syntax for the LLB and LHB instructions is:

LLB rd, 0xYY

LHB rd, 0xYY

Register rd is the register being loaded into, and 0xYY is the 8-bit immediate value to load (specified in hexadecimal).

LLB loads the least significant 8 bits of register rd with the bits from the immediate field. The most significant 8 bits of the register rd are left unchanged. Conversely, LHB loads the most significant 8 bits of rd while the least significant remain unchanged.

Note: These two are not technically loading from memory but are grouped with memory instructions.

The machine level encoding for these instructions is

101a dddd uuuu uuuu

where 101a is the opcode, dddd specifies the destination register, and uuuuuuuu is the 8-bit immediate value.

Note that LLB/LHB must not overwrite the upper/lower half of Reg[dddd]. Since your register file design does not support partial register writes, you will have to implement them using a read-modify-write register transfer: Reg[dddd] = (Reg[dddd] & 0xFF00) | uuuuuuuu for LLB and Reg[dddd] = (Reg[dddd] & 0x00FF) | (uuuuuuuu << 8) for LHB.

1.3 Control Instructions

There are four instructions belonging to this category: B, BR, PCS and HLT.

The B (Branch) instruction conditionally jumps to the address obtained by adding the 9-bit immediate (signed) offset to the contents of the program counter+2 (i.e., address of B instruction + 2).

The assembly level syntax for this instruction is:

B ccc, Label

And the machine level encoding for this instruction is:

Opcode ccci iiii iiii

where ccc specifies the condition as in Table 1 and iiiiiiiii represents the 9-bit signed offset in two's complement representation. You will need to left-shift the offset by 1 (since you are accessing half-words; i.e., 2 bytes in a byte-addressable memory). The target is computed as: target = PC + 2 + (iiiiiiiii << 1).

The BR (Branch Register) instruction conditionally jumps to the address specified by (rs).

The assembly level syntax for this instruction is:

BR ccc, rs

And the machine level encoding for this instruction is:

Opcode cccx ssss xxxx

where ccc specifies the condition as in Table 1 and ssss encodes the source register rs.

Table 1: Encoding for branch conditions

ccc	Condition
000	Not Equal (Z = 0)
001	Equal (Z = 1)
010	Greater Than (Z = N = 0)
011	Less Than (N = 1)
100	Greater Than or Equal (Z = 1 or Z = N = 0)
101	Less Than or Equal (N = 1 or Z = 1)
110	Overflow (V = 1)
111	Unconditional

The eight possible conditions are Equal (EQ), Not Equal (NEQ), Greater Than (GT), Less Than (LT), Greater Than or Equal (GTE), Less Than or Equal (LTE), Overflow (OVFL) and Unconditional (UNCOND). Many of these conditions are determined based on the 3-bit flag N, V, and Z. The instructions that set these flags are outlined in Table 2 below:

Table 2: Flags set by instructions

Instruction	Flags Set
ADD	N, Z, V
SUB	N, Z, V
XOR	Z
SLL	Z
SRA	Z
ROR	Z

A true condition corresponds to a taken branch. The status of the condition is obtained from the FLAG register (the definition of each flag is in Section 3.3).

The PCS instruction saves the contents of the next program counter (address of the PCS instruction + 2) to the register rd and increments the PC.

The assembly level syntax for this instruction is:

PCS rd

The machine level encoding for this instruction is:

Opcode dddd xxxx xxxx

where dddd encodes register rd.

The HLT instruction freezes the whole machine by stopping the advancement of PC.

Opcode xxxx xxxx xxxx

The list of instructions and their opcodes are summarized in Table 3 below.

Table 3: Table of opcodes

Instruction	Opcode
ADD	0000
SUB	0001
XOR	0010
RED	0011
SLL	0100
SRA	0101
ROR	0110
PADDSB	0111
LW	1000
SW	1001
LLB	1010
LHB	1011
B	1100
BR	1101
PCS	1110
HLT	1111

2. Memory System

For this stage of the project, the processor will have a separate single-cycle instruction memory and data memory, which are both byte-addressable. The instruction memory has a 16-bit address input and a 16-bit data output. The data memory has a 16-bit address input, a 16-bit data input, a 16-bit data output, and a write enable signal. If the write signal is asserted, the memory will write the data input bits to the location specified by the input address. Both instruction and data memories are implemented as asynchronous memories.

Verilog modules are provided for both memories.

The instruction memory contains the binary machine code instructions to be executed on your processor.

3. Implementation

3.1 Design

As mentioned earlier, in this project phase, you will implement the ISA and design a single cycle processor. On each clock cycle, one instruction is first read from instruction memory, then it is executed and finally the results are stored. Each instruction takes only one cycle to execute. Required design specifications for specific modules are provided below:

1. ALU Adder: Carry lookahead adder (CLA)
2. Shifter: Use 3:1 muxes, a variant of the design with 2:1 muxes in the lecture slides
3. Register File: As specified in the homework
4. Reduction unit (for RED instruction): Use a tree of 4-bit carry lookahead adders. At the first level of the reduction tree, sum_{ab} = aaaaaaaa + bbbbbbbb needs an 8-bit adder to generate a 9-bit result, in which this 8-bit adder is constructed from two 4-bit CLAs. The same goes for sum_{cd} = cccccccc + dddddddd. Then at the second level of the tree, the final result sum_{ab} + sum_{cd} should perform 9-bit addition using three 4-bit CLAs .

3.2 Reset Sequence

WISC-F19 has an active low reset input (rst_n). Instructions are executed when rst_n is high. If rst_n goes low for one clock cycle, the contents of the state of the machine are reset and execution is restarted at address 0x0000.

3.3 Flags

Flag bits are stored in the FLAG register and used in conditional branches. There are three bits in the FLAG register: Zero (Z), Overflow (V), and Sign (N). Only the arithmetic instructions (except PADDSB and RED) can change the three flags (Z, V, N). The logical instructions (XOR, SLL, SRA, ROR) change the Z FLAG, but they do not change the N or V flag.

The Z flag is set if and only if the output of the operation is zero.

The V flag is set by the ADD and SUB instructions if and only if the operation results in an overflow. Overflow must be set based on treating the arithmetic values as 16-bit signed integers.

The N flag is set if and only if the result of the ADD or SUB instruction is negative.

Other Instructions, including load/store instructions and control instructions, do not change the contents of the FLAG register.

4. Interface

Your top level Verilog code should be in a file named cpu.v. It should have a simple 4-signal interface: clk, rst_n, hlt and pc[15:0].

Signal Interface of cpu.v		
Signal:	Direction:	Description:
clk	in	System clock
rst_n	in	Active low reset. A low on this signal resets the processor and causes execution to start at address 0x0000
hlt	out	When your processor encounters the HLT instruction it will assert this signal once it is finished processing the instruction prior to the HLT
pc[15:0]	out	PC value over the course of program execution

5. Submission Requirements

1. You are provided with an assembler to convert your text-level test cases into machine level instructions. You will also be provided with a global testbench and test case. The test case should be run with the testbench and the output (as a .txt file) should be submitted for Phase 1 evaluation.

2. You are also required to submit a zipped file containing: all the Verilog files of your design, all testbenches used and any other support files.

ECE/CS 552: INTRODUCTION TO COMPUTER ARCHITECTURE

Project Description – Phase 2

Due on Friday, November 15, 2019, 11:59pm

In Phase 2 of this project, the task is to design a 5-stage pipelined processor to implement the WISC-F19 ISA.

Functional blocks of the single-cycle realization in Phase 1 of this project should be reused when possible.

The main task in Phase 2 is the implementation of pipeline control. You will implement (a) control blocks for hazard (both control and data) detection and mitigation (stalls, flushes and forwarding), and (b) control and data path modifications for data forwarding, including register file bypassing. More details are described below.

Either Modelsim or Icrarus should be used as the simulator to verify the design. You are required to follow the Verilog rules as specified by the rules document uploaded on canvas. NOTE: the only exception to this rule is the required use of *inout* (tri-state logic) in the register file. Do not use *inout* anywhere else in your design.

1. WISC-F19 ISA Summary

WISC-F19 contains a set of 16 instructions specified for a 16-bit data-path with load/store architecture.

The WISC-F19 memory is byte addressable, even though all accesses (instruction fetches, loads, stores) are restricted to half-word (2-byte), naturally-aligned accesses.

WISC-F19 has a register file, and a 3-bit FLAG register. The register file comprises sixteen 16-bit registers and has 2 read ports and 1 write port. Register $0 is hardwired to 0x0000. The FLAG register contains three bits: Zero (Z), Overflow (V), and Sign (N).

The list of instructions and their opcodes are summarized in Table 1 below. Please refer to the Phase 1 handout for more details.

Table 1: Table of opcodes

Instruction	Opcode
ADD	0000
SUB	0001
XOR	0010
RED	0011
SLL	0100
SRA	0101
ROR	0110
PADDSB	0111
LW	1000
SW	1001
LLB	1010
LHB	1011
B	1100
BR	1101
PCS	1110
HLT	1111

2. Memory System

For this stage of the project, the memory modules are the same as in Phase 1. The processor will have separate single-cycle instruction and data memory, which are both byte-addressable. The instruction memory has a 16-bit address input and a 16-bit data output. The data memory has a 16-bit address input, a 16-bit data input, a 16-bit data output, and a write enable signal. If the write signal is asserted, the memory will write the data input bits to the location specified by the input address.

Verilog modules are provided for both memories.

The instruction memory contains the binary machine code instructions to be executed on your processor.

3. Implementation

3.1 Pipelined Design

Your design must use a five-stage pipeline (IF, ID, EX, MEM, WB) similar to that in the class material. The design will make use of Verilog modules that you developed as part of the homework assignments along with the modules developed for Phase 1 of the project.

You must implement hazard detection so that your pipeline correctly handles all dependences. You are required to implement full data forwarding and register bypassing, including MEM-to-MEM forwarding as described in HW5. You need to handle both data hazards and control hazards. You must implement predict-not-taken for branch instructions, which flushes instructions when branches are taken. You are not required to implement other optimizations to reduce branch delays such as dynamic branch prediction and branch delay slots. Branches should be resolved in the ID stage.

It is highly recommended to make a table that lists (for each instruction) the control signals needed at each pipeline stage. Also, make a table listing the input signals and output signals for the data and control hazard detection units (data hazard stalls and control hazard flushes). The data hazard detection unit should assume that data forwarding and register bypassing is available.

3.2 Stall/Flush

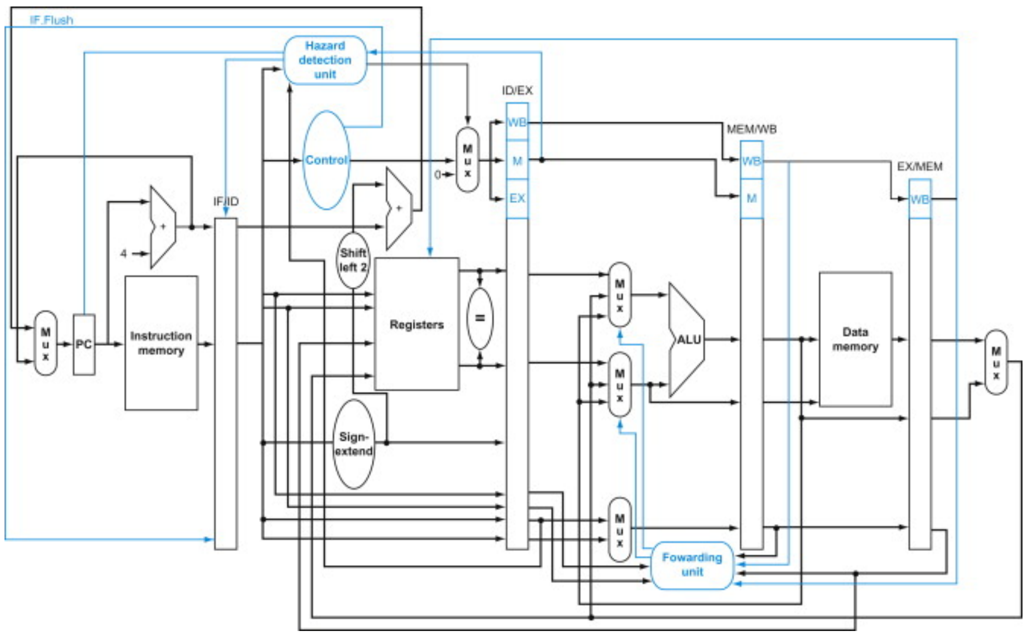
A stall is performed by using a global stall signal, which disables the write-enable signals to the D-FFs of the upstream pipeline registers (i.e., upstream from the stalling stage).
Flushes can be performed by converting the flushed operations into NOPs (i.e., preventing them from performing register/memory writes and from making changes to any other processor state, such as flags).

3.3 HLT Instruction

The HLT instruction should raise the 'hlt' signal only when it reaches the writeback stage. In the IF stage, you need to check for HLT instructions. If you fetch a HLT instruction, you must prevent the PC from being updated, thus stopping the program from fetching beyond the HLT instruction. The only exception is when the HLT instruction is fetched immediately after a taken branch: in this case, the HLT instruction is part of the wrong predict-not-taken branch path and will be flushed, thus the PC should be updated to the branch target address as usual.

3.4 Schematic

A summary diagram of the 5-stage pipeline is shown below (COD Figure 4.65):



Note: Not all hardware components and signals are shown here. There are more detailed diagrams of each stage of the pipeline in your textbook (along with details of how hazards are detected and resolved). You can refer to them for building the pipeline stages of the project.

3.5 Reset Sequence

WISC-F19 has an active low reset input (rst_n). Instructions are executed when rst_n is high. If rst_n goes low for one clock cycle, the contents of the state of the machine is reset and starts execution at address 0x0000.

4. Interface

Your top level Verilog code should be in file named *cpu.v*. It should have a simple 4-signal interface: *clk*, *rst_n*, *hlt* and *pc[15:0]*.

Signal Interface of *cpu.v*		
Signal:	Direction:	Description:
clk	in	System clock
rst_n	in	Active low reset. A low on this signal resets the processor and causes execution to start at address 0x0000
hlt	out	When your processor encounters the HLT instruction, it will assert this signal once it has finished processing the last instruction before the HLT
pc[15:0]	out	PC value over the course of program execution

5. Submission Requirements

1. You are provided with an assembler to convert your text-level test cases into machine level instructions. You will also be provided with a global testbench and test case. The test case should be run with the testbench and the output (as a .txt file) should be submitted for Phase 2 evaluation.

2. You are also required to submit a zipped file containing: all the Verilog files of your design, all testbenches used and any other support files.

ECE/CS 552: INTRODUCTION TO COMPUTER ARCHITECTURE

Project Description – Phase 3

Due on Friday, December 11, 2019, 11:59pm
(Demos on December 7 and 8, 2019)

In Phase 3 of this project, the task is to add a cache hierarchy to interface with the 5-stage pipeline from Phase 2.

The major work in Phase 3 is the implementation of the cache modules (I-cache and D-cache) and the cache controller that allows interaction between the caches and the processor pipeline, as well as between the caches and the memory.

Either Modelsim or Icarus should be used as the simulator to verify the design. You are required to follow the Verilog rules as specified by the rules document uploaded on canvas. NOTE: the only exception to this rule is the required use of *inout* (tri-state logic) in the register file. Do not use *inout* anywhere else in your design.

1. WISC-F19 ISA Summary

WISC-F19 contains a set of 16 instructions specified for a 16-bit data-path with load/store architecture.

The WISC-F19 memory is byte addressable, even though all accesses (instruction fetches, loads, stores) are restricted to half-word (2-byte), naturally-aligned accesses.

WISC-F19 has a register file, and a 3-bit FLAG register. The register file comprises sixteen 16-bit registers and has 2 read ports and 1 write port. Register $0 is hardwired to 0x0000. The FLAG register contains three bits: Zero (Z), Overflow (V), and Sign (N).

The list of instructions and their opcodes are summarized in Table 1 below. Please refer to the Phase 1 handout for more details.

Table 1: Table of opcodes

Instruction	Opcode
ADD	0000
SUB	0001
XOR	0010
RED	0011
SLL	0100
SRA	0101
ROR	0110
PADDSB	0111
LW	1000
SW	1001
LLB	1010
LHB	1011
B	1100
BR	1101
PCS	1110
HLT	1111

2. Memory System

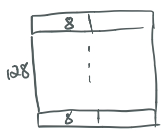
For this stage of the project, you are required to design a) I-cache and D-cache modules, b) cache controllers for reading and writing to the caches, c) interface between the caches and memory, d) interface between the I-cache and the IF pipeline stage, e) interface between the D-cache and the MEM pipeline stage.

Verilog modules are provided for: a) multi-cycle main memory, b) cache data array, c) cache meta-data array
.

You will load main memory with the binary machine code instructions to be executed on your design (not your I-cache). You will use one instance of a data array and a meta-data array for the I-cache and another instance each for the D-cache.

3. Implementation

3.1 Cache/Memory Specification



a. The processor will have separate single-cycle instruction and data caches, which are byte-addressable. Your caches will be 2KB (i.e,. 2048B) in size, 2-way set-associative, with cache blocks of 16B each. Correspondingly, the data array would have 128 lines in total, each being 16 bytes wide. The meta-data array would have 128 total entries composed of 64 sets with 2 ways each. Each entry in the meta-data array should contain the tag bits, the valid bit and one bit for LRU replacement.

b. The cache write policy is write-through and write-allocate. This means that on hits, it writes to the cache and main memory in parallel. On misses it finds the block in main memory and brings that block to the cache, and then re-performs the write (which will now be a cache hit; thus it will write to both cache and memory in parallel).

c. The cache read policy is to read from the cache for a hit; on a miss, the data is brought back from main memory to the cache and then the required word (from the block) is read out.

d. The memory module is the same as before, except for the longer read latency and a "data_valid" output bit. A 2-byte write to memory from the processor will take one cycle, while a 2-byte read from memory will take 4 cycles. Memory is pipelined, so read requests can be issued to memory on every cycle.

e. The cache modules will have the following interface: one 16-bit (2-byte) data input port, one 16-bit (2-byte) data output port, one 16-bit (2-byte) address input port and a one-bit write-enable input signal.

f. Note that the interaction between the memory and caches should occur at cache block (16-byte) granularity. Considering that the data ports are only 2 bytes wide, this would require a burst of 8 consecutive data transfers.

3.2 Cache Hits Reads/Writes

a. Cache hits take only one cycle to execute. Once the data array lines and the meta-data array entries are identified based on the index and offset bits of the address, data is read/written from/to the data array and in parallel the tag match is performed.

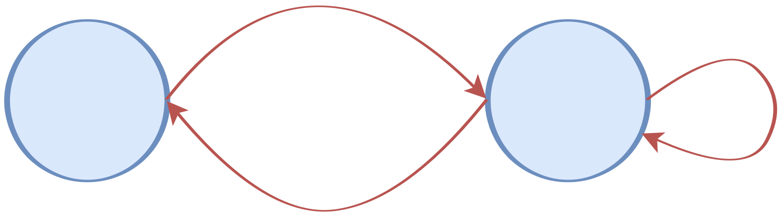
b. If the tag matches (i.e., hit), the read/write from/to the data array is valid. This data is returned via the cache data port in case of a read.

c. Being a write-through cache, all writes are written to main memory as well. As mentioned earlier, memory writes take only 1 cycle, so if it is a write hit, the memory write will complete in parallel to the cache write.

d. In case of a tag mismatch (i.e., miss), the miss handler is triggered and the pipeline is stalled (if the data cache misses, all upstream instructions must stall) or NOPs are inserted (if it is an instruction cache miss).

3.3 Cache Miss Handler FSM

The figure below shows a simple FSM for retrieving a block from memory upon a cache miss. The blue refers to the state, the green refers to the condition for changing state, and the red refers to actions performed on state transitions.



This is only a simple FSM for sequentially requesting and receiving each 2-byte chunk from memory. You are required to enhance this FSM to support pipelined memory requests. Since memory is pipelined, read requests can be issued to memory every cycle, and each 2-byte chunk should be received in consecutive cycles.

Points to note:

a. The cache is write-allocate. On both read and write misses, you need to bring in the correct block from memory to the cache.

b. The cache is write-through, so there is no data to be written back to memory upon an eviction.

c. The cache controller stalls the processor on a miss, and only after the entire cache block is brought into the cache, the new tag is written into the meta-data array, the valid and LRU bits are set and the stall is deasserted.

3.4 Memory Contention on Cache Misses

The mechanism described above for handling misses is independent for the I-cache and D-cache. If requests from both caches are sent to memory, only one can be handled at a time. You are required to implement an arbitration mechanism that selects either one of the competing requests and gives it a grant to go through to memory. The other request stalls for an extended period. Note that you will never have multiple misses from the I-cache at one time nor will you ever have multiple misses from the D-cache at one time; at most you can have one of each type in parallel.

4. Interface

Your top level Verilog code should be in a file named *cpu.v*. It should have a simple 4-signal interface: *clk*, *rst_n*, *hlt* and *pc[15:0]*.

Signal Interface of *cpu.v*		
Signal:	Direction:	Description:
clk	in	System clock
rst_n	in	Active low reset. A low on this signal resets the processor and causes execution to start at address 0x0000
hlt	out	When your processor encounters the HLT instruction, it will assert this signal once it has finished processing the instruction prior to the HLT
pc[15:0]	Out	PC value over the course of program execution

5. Submission Requirements

1. You are provided with an assembler to convert your text-level test cases into machine-level instructions. You will also be provided with a testbench and test cases. The test cases should be run with the test bench and demonstrated at your demo timeslot. More details will be announced later.

2. You are also required to submit a zipped file containing: all the Verilog files of your design, all testbenches used and any other support files.

3. You must also submit the project final report; see the guidelines for the report on Canvas.