ECE411 Computer Organization and Design
Mid-Term Exam 1

10/10/2019

NetID: __SOLUTION______

First Name:___________________

Last Name:___________________

Exam Guidelines:

1. This exam has **5 problems.** Make sure you have a complete exam before you begin **[27 pages + the cover page]**.
2. Write your name on every page in case pages become separated during grading.
3. You will have **3 hours** to complete this exam.
4. Write all of your answers on the exam itself. If you need more space to answer a given problem, continue on the back of the page, but clearly indicate that you have done so.
5. This exam is closed-book. You may use one sheet of notes. You may use a calculator.
6. **DO NOT** do anything that might be perceived as cheating. The minimum penalty will be a grade of zero.
7. Show all of your work on all problems. Correct answers that do not include work demonstrating how they were generated may not receive full credit, and answers that show no work cannot receive partial credit.
8. The exam is meant to test your understanding. Ample time has been provided. So be patient and read the questions/problems carefully before you answer.
9. **Good luck!**

Problem	Points
1. **10** pt	
2. **14** pt + 2 extra pt	
3. **9** pt	
4. **20** pt	
5. **13** pt	
Total: **66** pt + 2 extra points	

NetID:________________________

Question 1: ISA

Predication as an architectural feature allows for instructions to be "predicated" by tying them to a conditional code which decides whether they affect change to the architectural state. This approach serves as an alternative to the typical branch based conditional transfer of control you are most familiar with from working with RISC-V and LC3. The two tables below show the REG-REG ISA instructions both with and without predication. Note that comparisons are signed.

REG-REG: In this ISA, instructions operate directly on registers. Register \$0 will always be 0, like in your RISC-V ISA. You further have registers \$1 through \$7

MOV dest, src	Copies the data from register *src* to register *dest*
ADDimm dest, src, imm	Stores the sum of register *src* and the specified immediate into *dest*
ADD dest, src1,src2	Stores the sum of registers *src1* and *src2* into register *dest*
NOT dest, src	Stores the bitwise inversion of register *src* at register *dest*
LD dest, srcaddr	Loads the data from address *srcaddr* into register *dest*
ST ~~src, destaddr~~ destaddr, src	Stores the data of register *src* at address *destaddr*
BR[g/n/eq/geq] ~~*destaddr,src1,src2*~~ src1, src2, dest	If src1 > [g], == [eq], != [neq] or >= [geq] src2, jump to destaddr. All condition codes can be a combination of base conditions

REG-REG Predication: This ISA is very similar to the REG-REG one, but with a conditional register at the start of the instruction. If this register is \$0, the always-zero register, the instruction will unconditionally run. Otherwise, it will only run if the register is nonzero.

`(reg) MOV dest, src`	Copies the data from register src to register dest
`(reg) ADDimm dest, src, imm`	Stores the sum of register src and the specified immediate into dest
`(reg) ADD dest, src1,src2`	Stores the sum of registers src1 and src2 into register dest

`(reg) NOT dest, src`	Stores the bitwise inversion of register src at register dest
`(reg) LD dest, srcaddr`	Loads the data from address srcaddr into register dest
`(reg) ST src, destaddr`	Stores the data of register src at address destaddr
`(reg) SET [g/n/eq] dest,src1,src2`	Stores 1 in reg dest if reg src1 [g/n/eq] reg src2. (else stores 0)
`(reg) JMP destaddr`	Jumps to address destaddr

For the problem below, you will consider how predication effects efficiency in a pipelined processor. The pipelined processor you will consider below will, like your eventual MP3 processor, be a classic five stage processor with stages FETCH, DECODE, EXECUTE, MEM, and WRITEBACK. <u>Every cycle of this processor will take the same amount of time, including any memory accesses that must happen.</u> Assume that the memory model will not in any way impede this or cause the processor to stall (i.e., all memory accesses happen within these cycles). Further assume that the processor has the forwarding necessary to never lose a cycle due to data hazards.

This processor uses a static not-taken approach to branch prediction. That is to say that for the system without predication, a branch that is not taken does not merit a flush of the pipeline, while one taken (regardless of address branched to) does result in a flush of the pipeline. In the predication ISA, this similarly manifests in whether a jump (JMP) is taken or not. Both the jump-taken and branch-taken would be evaluated when the relevant instruction is in the EXECUTE stage, with the processor fetching the instruction as the pipeline is flushed next cycle.

a. (4 points) Consider the non-predication REG-REG code below:
This code iterates through an array of length 5 whose starting address begins in $1, and counts the number of non-negative integers within, storing the result at 0x0411.

```
     MOV $2, $0  //Set $2 as 0 to hold result of non-negative count
     ADDimm $5, $0, #5 //Initialize $5 to 5 as loop counter.
LOOP:
     LD $3, $1   //Uses $1 as the source address
     BR g $0, $3, SKIP_INCREMENT
     ADDimm $2, $2, #1
SKIP_INCREMENT:
     ADDimm $1, $1, #1  //Move to next item in array
     ADDimm $5, $5, #-1
     BR g $5, $0, LOOP
     ST 0x0411, $2
```

Write **commented** code of equivalent functionality in the Predication based ISA:
The first line has been provided to you to serve as an example of a predicated instruction. Your solution should (must) take no more than 9 lines, including the provided line. Labels do not contribute to your line count.
Your solution cannot use more than one JMP.

```
     ($0)  MOV  $2,  $0   //Unconditionally clears $2 to hold result
value
     ($0) ADDimm $5, $0, #5
LOOP:
     ($0) LD $3, $1
     ($0) SET geq $4, $3, $0 // Note that this uses the g + eq
     ($4) ADDimm $2, $2, #1
     ($0) ADDimm $1, $1, #1
     ($0) ADDimm $5, $5, #-1
     ($5) JMP LOOP
     ($0) ST 0x0411, $2
```

NetID:__________________________

b. (4 points) For both the provided code without predication and your code with predication, consider the first (MOV) instruction entering the pipeline of a processor (being fetched) at time cycle 0. In which cycle does your processor retire the last instruction (have it leave the WRITEBACK stage) if the arrays contents are -4, -1, -1, 411, -411?

Processor without Predication (cycles):	49 (Also accepted 48/50)
Processor with Predication (cycles):	45 (Also accepted 44/46)

c. (2 points) Based on your understanding of the typical pipelined processor, what kind of code might favor non-predicated branch oriented ISA over one with predication?

Code with large segments of code being chosen between by the control structure, as well as code for which a branch predictor would be highly accurate/effective. Furthermore, code that uses a large number of registers would have issues delegating some registers for use in predication, and so would benefit from working with a branch oriented ISA.

Question 2: Caches

Regina Register runs the following program on her correctly working MP2 computer (with a 2-way, 8-set, 256-bit line cache):

```
I1:  AND x1, x0, x0              ; x1: loop counter
I2:  ADDI x2, x0, 0x0020         ; x2: loop limit
I3:  LA x3, array_base           ; x3: data pointer
I4:  ADDI x4, x0, 1
I5:  SLLI x4, x4, 19             ; x4: data stride
loop:
I6:  LW x5, 0(x3)
I7:  ADDI x5, x5, 1
I8:  SW x5, 0(x3)
I9:  ADD x3, x3, x4
I10: ADDI x1, x1, 1
I11: BLT x1, x2, loop
halt:
I12: BEQ x0, x0, halt
I13: array_base                  ; 32 byte aligned
```

Suppose each instruction takes 5 cycles to complete given no cache misses and the memory access latency is 25 cycles (i.e. a program whose first instruction is NOP will take 30 cycles --- 5 cycles for the ~~pipeline~~ computation, and 25 cycles to fetch the data line into the empty cache).

a. (3 points) How many cycles does the 16'th iteration (counting from 1) of the loop (instructions `I6-I11`) require? Assume instruction `I1` is at address `0x060`.

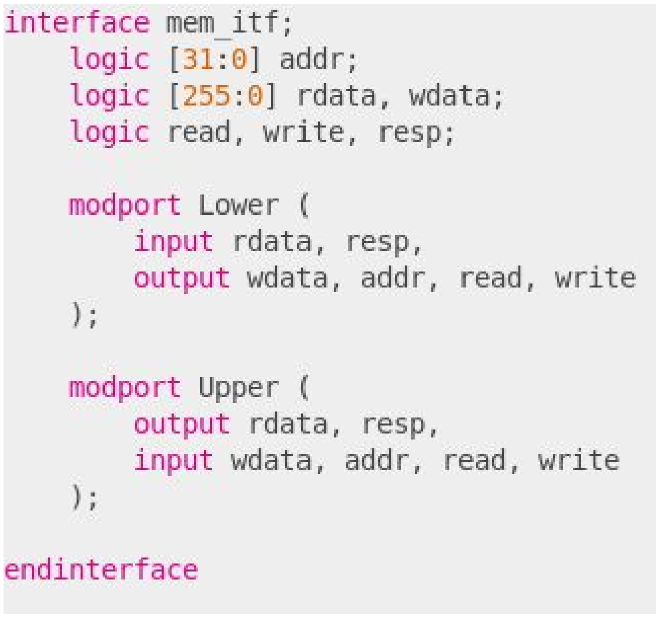
By the 16th iteration, the cache is certainly filled (note that we are only accessing a single set due to the data stride). Thus,

6*5 = 30 cycles for computation
25 cycles for writeback on dirty miss
25 cycles for load on miss
80 cycles total

Regina is unhappy with this performance and decides to implement a "Eviction Write Buffer" (EWB), which stores a dirty line after eviction from the cache, but before write-back to memory. The EWB sits between the cache and memory and consists of the following components:

1. A register which stores the following structure:
```
struct {
        logic [31:5] address;
        logic [255:0] data;
        logic valid;
};
```
2. A state-machine;
3. A counter.

To facilitate transmission of signals from the cache to the EWB and from the EWB to the physical memory, Regina uses an interface described in Figure 2.1 below. Two instances of this interface are instantiated in the top level, with the Cache and EWB sharing one instance, and the EWB and memory sharing the second, as shown in Figure 2.2.



```
interface mem_itf;
    logic [31:0] addr;
    logic [255:0] rdata, wdata;
    logic read, write, resp;

    modport Lower (
        input rdata, resp,
        output wdata, addr, read, write
    );

    modport Upper (
        output rdata, resp,
        input wdata, addr, read, write
    );

endinterface
```

Figure 2.1: EWB interface definition

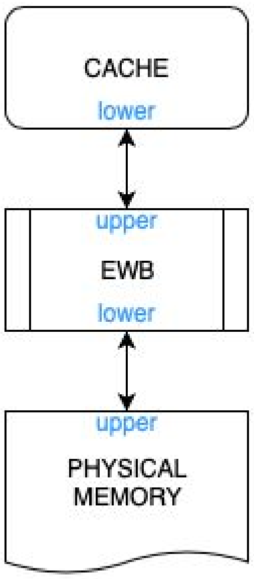


Figure 2.2: EWB Memory Hierarchy

On eviction of a dirty line due to a miss in the set associated with the access address, the cache takes the following actions:

1. If the EWB is full, or the cache is currently writing the EWB line, the cache stalls the processor until the line in the EWB is written back to memory, and then proceeds to step 2.
2. The dirty line is "evicted" into the EWB.
3. The new line is fetched from memory and loaded into the cache.
4. The Load/Store instruction is executed, the cache asserts its 'resp' signal.
5. After storing a line, if neither the Upper.read nor Upper.write signal are asserted, the EWB begins a countdown from 10 to 0. If it reaches 0, the EWB assumes there are unlikely to be subsequent reads from the cache, and thus begins to write back the stored line to memory.
6. When the EWB has a line stored in memory, and Upper.read is asserted, the EWB must first check if the line requested by the cache is stored in the EWB. If it is, it responds to the cache with the stored data line, and then begins the `COUNTDOWN` again.

The EWB state machine outputs the following signals:

`l.read, l.write, u.resp`

The EWB state machine takes the following signals as input:

`u.read, u.write, l.resp`

Additionally, the following signals are used between the EWB datapath and controller:

`hit, miss, load, set_valid, clear_valid, counter[3:0], reset_counter, dec_counter`

Note: u. and l.* correspond to the upper and lower interface ports, respectively.*

b. (6 points) Fill in the redacted state diagram (shown in the next page) with the conditions and outputs on each transition arrow. Also, using your completed state machine and the component description of EWB, draw the EWB's datapath. Any control signals used by the datapath must be added to the state machine diagram. Denote the bitwidth of wires in the datapath.

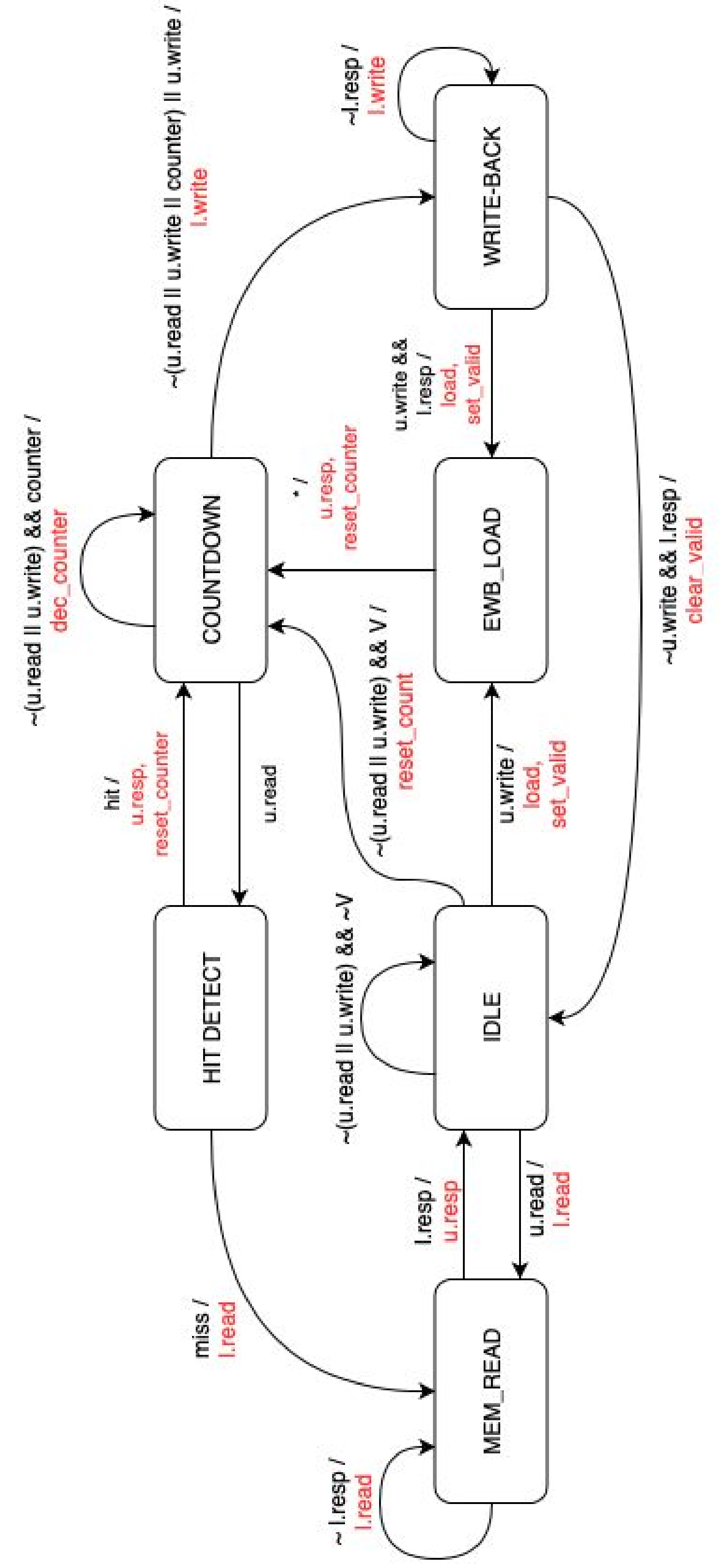
For full credit, datapath solutions required:

Data, Valid, Address, and Counter registers (Data, Valid, and address could be combined)

Combinational logic supporting the Counter

Data mux to select data to send to cache (either from Data register or from memory)

V = valid


WRITE-BACK
EWB_LOAD
COUNTDOWN
IDLE
HIT DETECT
MEM_READ
~l.resp / l.write
~(u.read || u.write || counter) || u.write / l.write
u.write && l.resp / load, set_valid
~(u.read || u.write) && counter / dec_counter
* / u.resp, reset_counter
~u.write && l.resp / clear_valid
~(u.read || u.write) && V / reset_count
hit / u.resp, reset_counter
u.read
u.write / load, set_valid
~(u.read || u.write) && ~V
l.resp / u.resp
u.read / l.read
miss / l.read
~ l.resp / l.read

c. After implementing and connecting the EWB to her MP2 design, Regina runs the same test code again.

   i. (3 points) How many clock cycles are required for the 16'th iteration (counting from 1) of the loop with the EWB installed and what is the speed-up of the 16'th iteration of the loop? Assume "Countdown" is set to 10.

      6*5 = 30 cycles for computation
      2 cycles to write into EWB on dirty miss
         (IDLE → EWB_LOAD → COUNTDOWN)
      3 + 25 = 28 cycle to read from memory on miss (also miss in EWB)
         (COUNTDOWN → HIT DETECT → MEM_READ → IDLE + 25 M.A.T.)
      60 cycles total
      Speedup = 80/60 = 1.33x

   ii. (2 points) Give an example of a situation where the EWB provides little to no benefit. This may be an English description or a short code snippet.

      Any sequence of evictions that occur faster than writeback can complete (in this case, 10 cycles for countdown + 25 for memory access). The EWB enables the cache to perform non-blocking writes. If another eviction occurs, the EWB must empty itself before accepting the evicted data.

      Trivially, if there are no loads/stores or no evictions, then the EWB provides no benefit.

d. (Extra credit - 2 points) When waiting to write a value back to memory, a read request will be serviced by returning the buffered data if it exists in the EWB, or passing through from memory if it is not. After the read, we continue to write back the EWB data to memory. Why is it not sufficient to simply clear the EWB when the cache requests the data again without finishing the writeback to memory?

   The EWB does not have a mechanism for telling the cache that a line is still dirty. If the write-back is cancelled, the cache would need to understand that the data does not exist in memory and must be evicted instead of overwritten.

Question 3: Caches and Virtual Memory

a. (3 points) Given 4 KB pages, 512 bit data lines, and byte addressability, select the indices of tag, set, and offset bits of the physical address which maximize the number of sets in the cache while avoiding the synonym and homonym problems. (Hint: the fields must partition the address). Write your answer in the `paddr` union's fields struct. VIPT Cache

Virtual Address:

```
union {
  logic [31:0] addr;
  struct packed {
    logic [31:12] vpn;
    logic [11:0] offset;
  } fields;
} vaddr;
```

Physical Address:

```
union {
  logic [31:0] addr;
  struct packed {
    logic [ 31 : 12 ] tag;
    logic [ 11 :  6 ] set;
    logic [  5 :  0 ] offset;
  } fields;
} paddr;
```

To avoid the synonym and homonym problems, the physical address set and offset must fit entirely within the page offset. Thus, the tag is at least 31:12. The data lines are 512 bits = 64 bytes = 2^6, so offset is 5:0. To maximize the number of sets, the set field should be as wide as possible, i.e. 11:6 (it can't use bits >= 12 because of the synonym and homonym problems).

b. In x86 architectures, page table entries, used to manage virtual memory, are 32-bit words organized as follows:

```
union {
    logic [31:0] pte;
    struct packed {
        logic [31:12] paddr; // Physical address of page frame base
        logic [11:9] avail;  // Available for system programmer’s use
        logic [8:7] rsvd1;   // Reserved
        logic [6:6] D;       // Set to 1 by CPU on write to page
        logic [5:5] A;       // Set to 1 by CPU on read/write to page
        logic [4:3] rsvd2;   // Reserved
        logic [2:2] u_s;     // User / supervisor permissions
        logic [1:1] r_w;     // Read / write permissions
        logic [0:0] P;       // Present (not swapped to disk)
    } fields;
};
```

Ben Bitdiddle implements an x86 processor with a VIPT instruction cache, a VIPT data cache, and a 1-level direct mapped TLB. The caches suffer from neither the homonym problem nor the synonym problem. On a memory access, the appropriate PTE is loaded into the TLB (if not already present) and the ‘A’ and ‘D’ fields of the PTE are set in the TLB as appropriate. If the TLB changes either of these fields, an additional TLB-line metadata dirty bit is set causing the TLB to write-back the PTE on eviction.

To Ben’s surprise, his processor doesn’t work properly, causing crashes in his OS kernel and in application programs. He believes the problem lies somewhere in virtual memory (since the processor works correctly when virtual memory is disabled). He believes the issue is not in address translation, because he has unit tested the address translation of the TLB and it works properly, as does its memory write back mechanism. Ben examined the crash reports and notices something startling: the dirty bits of the PTEs in memory are occasionally not being set properly!! Ben confirms that the TLB is properly setting the dirty bits of the PTEs. Confused, Ben turns to you for help (note that this is a simplified version of a real problem faced by AMD in their Phenom processor):

NetID:________________________

i. (2 points) What is wrong with Ben's virtual memory implementation?

There is a coherence issue. Data may be simultaneously read and modified in the TLB and cache/memory. A modified PTE located in the TLB will not be present in memory, so if a program or system call attempts to modify or check it, a stale value will be returned from memory.

ii. (2 points) What is a simple hardware solution to this problem? Describe the downsides (qualitative) of this solution.

A) Disable the TLB. All address translation must walk the page directory structure to find the PTE in memory. This reduces performance dramatically, especially when multi-level paging structures are used.

B) Modify the TLB to use a write-through policy rather than write-back. This will ensure modified PTEs are updated immediately in memory (as long as the TLB is reading from the same memory as the processor, i.e. L1D cache, or has an invalidation mechanism). Simultaneously, restrict PTEs from entering the cache (i.e enforce atomic accesses to the PTEs). This reduces the benefit of having cached PTEs on write and increases the complexity of the design.

iii. (2 points) What is a simple software solution to this problem? Describe the downsides (qualitative) of this solution.

Flush the TLB any time a PTE is accessed. If a program wishes to read a PTE, it must first flush the TLB to ensure any updates are present in memory. This reduces the efficacy of the TLB as a cache, since it will contain valid translations less often.

Alternatively, the TLB could be disabled via firmware as AMD did to solve the Phenom bug.

Question 4: Pipelining

Given the 5 stage pipeline shown in figure 4.1. For memory instructions, the compiler sets the VPN in the register and the page offset as the immediate value. For example,

```
lw x5, 32(x4) # x4 has the VPN and 32 is the page offset
```

Predictor value: 0 = 'Not Taken'

Consider BHT with 0-bit width (ie. no BHT accessing the PHT with the address directly)

Consider the following:

- Fetch stage combinational delay = 4ns
- Decode stage combinational delay = 5ns
- Execute stage combinational delay = 5ns
- Writeback stage combinational delay = 4ns
- Data array access in D-Cache = 3ns
- Tage comparison combinational delay is 1ns
- Address Translation = 6ns (if needed)
- Register delay is 1ns
- The processor operates at the maximum possible safe frequency
- 100% cache hit rates.
- Both I-cache and D-cache are VIVT caches

Code:

```
     lw   x4, data0(x0)
     lw   x5, data1(x0)
     lw   x6, data2(x0)
     addi x1, x4, 2
     addi x2, x5, 6
     addi x3, x6, 8
     sw   x1, data3(x0)
     sw   x2, data4(x0)
     sw   x3, data5(x0)
     .
     .
data0: .word 0x2
data1: .word 0x3
data2: .word 0x4
data3: .word 0x0
data4: .word 0x0
data5: .word 0x0
```

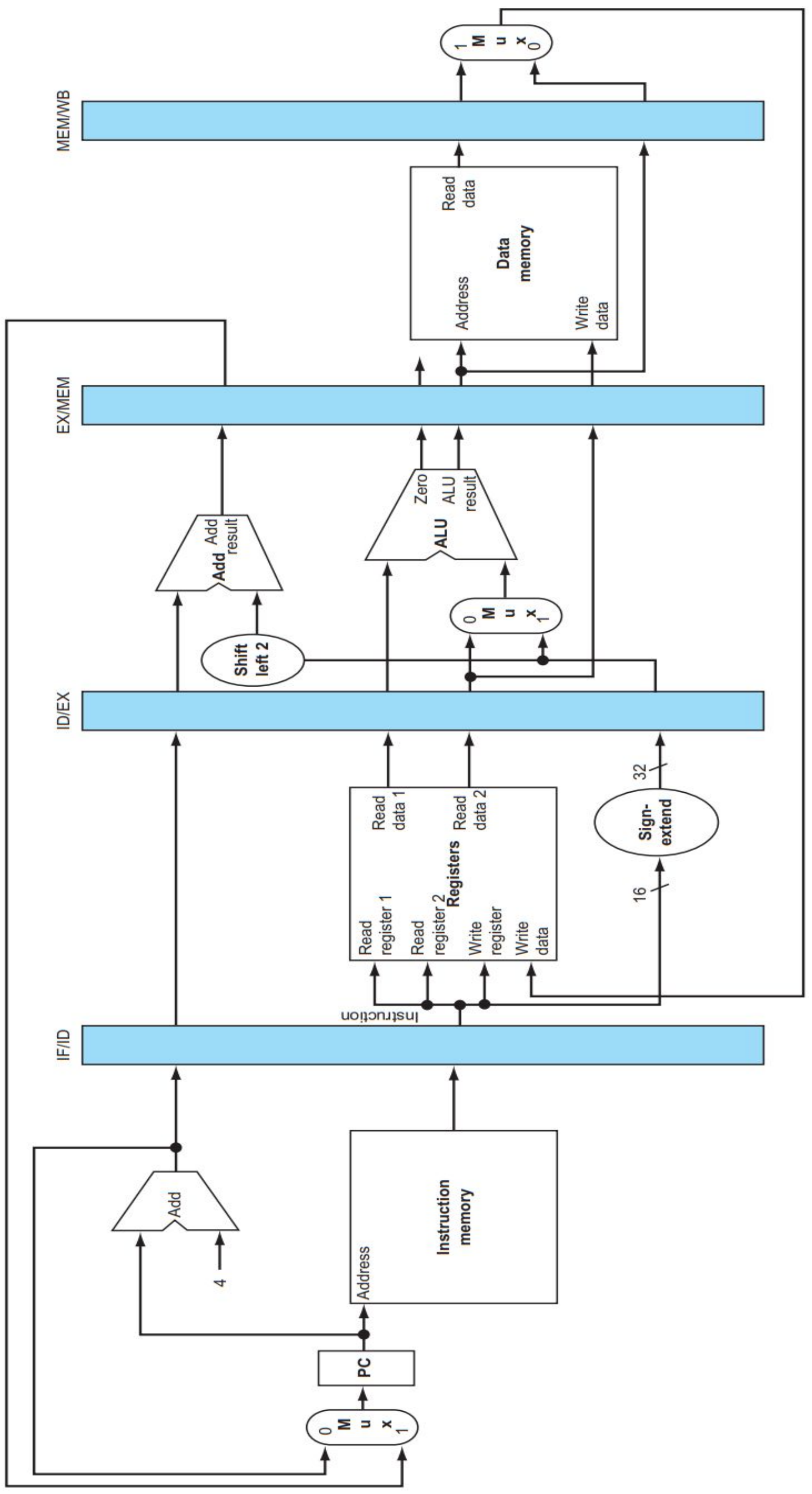


Fig 4.1 5-stage pipeline (faulty)

a. (4 points) Calculate how long it will take (in ns) to run the given code

Assuming no hazard detection: 13 x 6ns = 78ns
Assuming hazard detection: 15 x 6ns = 90ns

b. (4 points) There are two issues with the given pipeline that prevent the given code from producing the correct results. One issue is related to a HW bug (hint: wiring error). Explain those issues and provide **one** solution for the HW bug and at least **two** solutions for the other issue.

Write Register to the register file should be from WB

Data Hazard between WB and ID:

- Insert nop as following:

```
lw   x6, data2(x0)
nop  <--------
addi x1, x4, 2
addi x2, x5, 6
addi x3, x6, 8
nop  <--------
sw   x1, data4(x0)
```

- Forwarding logic between WB and ID
- HW hazard detection (stall for one cycle)

c. (3 points) To avoid the limitations of VIVT, choose the best scheme (among `PIVT`, `VIPT`, and `PIPT`) for the D-cache for the given pipeline. Explain why?

PIPT: translate in Ex then access the D-cache in Mem stage with the physical address

d. Assume the following code snippet:

```
c = 0;
for (int i = 0; i < 1000; i++) {
     for  (int j = 0; j < 4; j++) {
               c++;
     }
}
```

The compiler generates this assembly:

```
     addi x1, x0, 0       # c = 0 (x1)
     addi x2, x0, 0       # i = 0 (x2)
     addi x7, x0, 4       # x7 = 4
     addi x8, x0, 1000    # x8 = 1000

Outer:
     addi x3, x0, 0       # j = 0 (x3)
Inner:
     addi x4, x1, 0       # x4 = c
     addi x3, x3, 1       # j++
     addi x4, x4, 1       # x4++
     addi x1, x4, 0       # c = x4
     bne  x4, x7, Inner   # if j < 4 then goto Inner
     addi x2, x2, 1       # i++
     bne  x2, x8, Outer   # if i < 1000 then goto Outer
```

Assume a Per-Address (PA) ~~Two-level~~ branch predictor (~~BHT and~~ PHT), with no conflicts between branch address bits, and with all entries initialized to 0. For the given code:

i. (3 points) How many conditional branches would be mispredicted when PHT's entries are 1 bit wide?

Inner misprediction: 2000

Outer misprediction: 2

2002

ii. (3 points) How many conditional branches would be mispredicted when PHT's entries are 2-bit saturating counters?

Inner misprediction: 1002

Outer misprediction: 3

2002

iii. (3 points) Based on the given code what is the best branch predictor and why? (mention the number of mispredicted branches of the branch predictor if it is different than a and b)

With a static taken branch predictor the mispredicted branches are: 1001

Question 5: MPs

Throughout the course of the MPs, you should have a fully functional and verified RV32I processor with caching (minus some instructions which aren't pertinent for non-machine mode operation). However, there are many extensions to RISC-V. For example, the current ports of Linux to RISC-V require the integer multiplication/ division (M), atomic memory operation (A), single precision floating point (F), double precision floating point (D), and compressed instruction (C) extensions. We will focus on the M and C extensions in this question. All applicable specifications are included in the appendix.

a. (3 points) The multiply extension mandates support for multiplication, division, and remainder operations. You have decided to first implement support for unsigned multiplication using the add-shift multiplier from MP0 (specifically the instructions `mul` and `mulhu`). If you do not remember the MP0 multiplier, the code has been provided for you in the appendix. Please modify the MP1 state diagram to support multiplication using the add-shift multiplier. Be sure to include both the states and transition conditions. (shown in the next page).

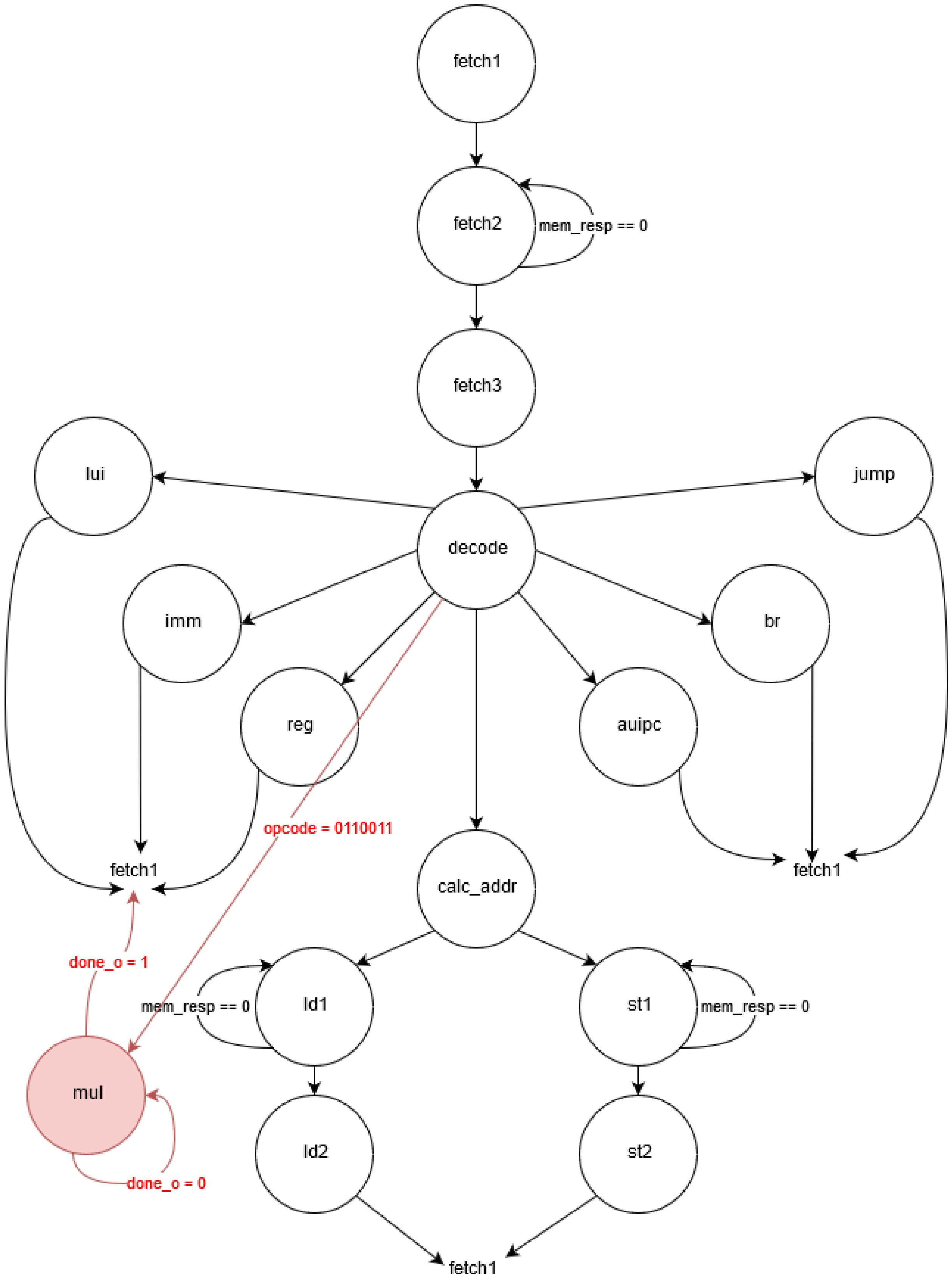
b. (3 points) Modify the MP1 datapath below in order to support the new multiplication operation. You do not need to preserve the value in the destination register file while the multiplication is ongoing. Make the minimum number of modifications necessary. If your solution requires no modifications, explain why.

c. (2 points) Will the MP0 implementation of an add-shift multiplier be faster or slower than the same algorithm implemented in assembly code? Explain your reasoning. If it works faster under some conditions and slower under others, list those conditions.

The MP0 add-shift multiplier will be faster than the **SAME ALGORITHM** implemented in software. Since each add, shift, and branch instruction would need to be fetched and decoded, having a single instruction with an MP0 multiplier avoids all of that overhead.

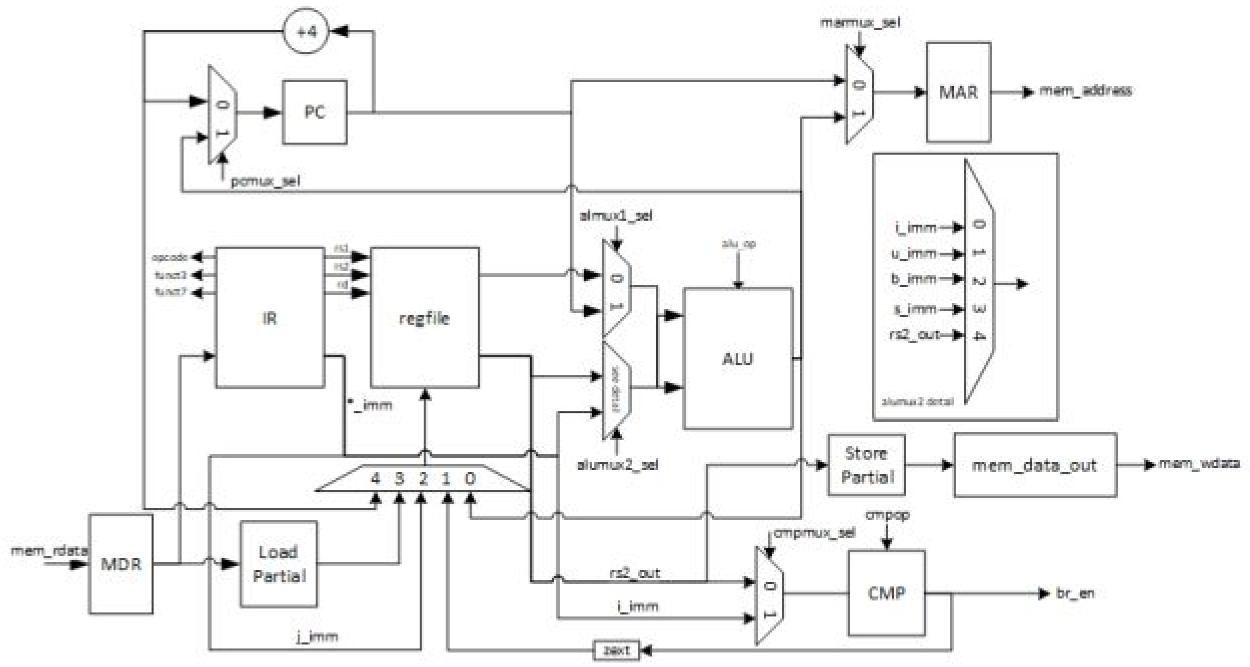
NetID:_________________________

NetID:__________________________



(A larger version is provided in Appendix A if that is easier for you to modify)

NetID:______________________



There were three parts to the solution to this problem. The first was drawing the multiplier with some kind of control signal(s). At minimum, we needed to see a start or reset signal and a done signal.

The second component is the inputs to the multiplier. The spec only calls for rs1 and rs2, although coming from the ALU mux inputs would also be acceptable.

The final component is the output of the multiplier must go into the regfilemux. However, the output of the multiplier is 64 bits, and the registers are only 32 bits wide. So you need two inputs to the regfile mux ([63:32] and [31:0]) or a second mux to select between the two.

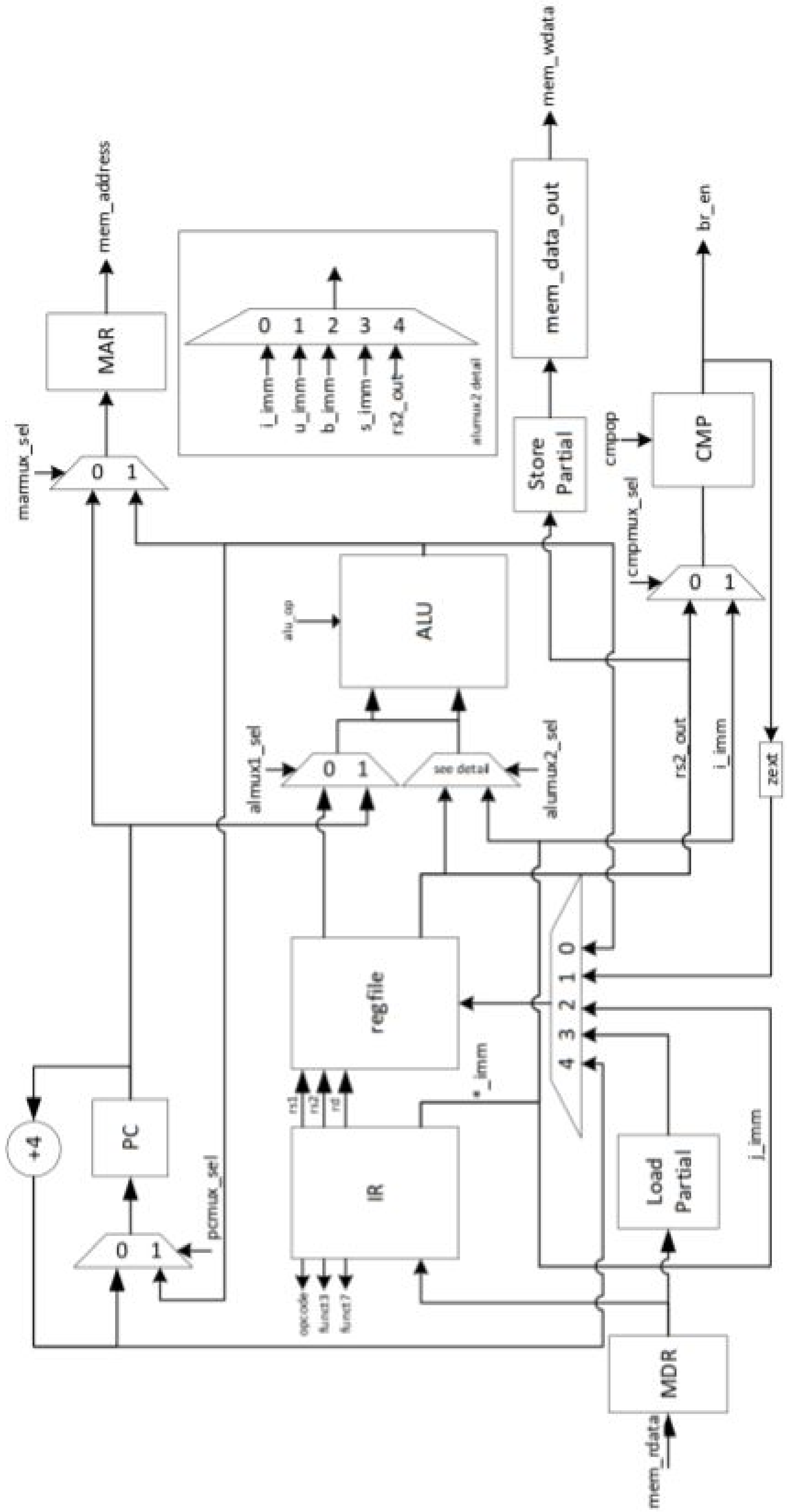
d. (2 points) In MP0 we asked that you verify the correctness of your multiplier by multiplying all possible combinations of inputs and verifying their correctness. Explain why that would not work for this multiplier, and then explain what a valid method of testing the multiplier would be.

There are $2^{32} * 2^{32} = 2^{64}$ multiplications which would need to be tested. That is not a feasible amount to test. A better solution would be to test targeted edge cases and provide random stimuli to the multiplier to verify correctness.

e. (3 points) We will now consider the compressed extension. This extension allows for some commonly used instructions and registers to be compressed into 16 bits instead of requiring a full 32 bit word. Other instructions require the full 32 bit word and cannot be compressed. Will this instruction set extension work natively with the cache implemented in MP2? Why?

This ISA extension will **NOT** work natively with the cache implemented in MP2. The ISA extension allows for full 32-bit instruction words to be placed immediately after a 16 bit compressed instruction. This can cause a need to fetch an instruction across multiple cache lines, or across multiple 4 byte words, neither of which is natively supported by the MP2 cache.

Appendix A:


mem_address
MAR
marmux_sel
0 1
i_imm
u_imm
b_imm
s_imm
rs2_out
0 1 2 3 4
alumux2 detail
mem_wdata
mem_data_out
Store Partial
cmpop
CMP
br_en
cmpmux_sel
0 1
alu_op
ALU
alumux1_sel
0 1
see detail
alumux2_sel
rs2_out
i_imm
zext
regfile
rs1
rs2
rd
*_imm
0 1 2 3 4
IR
opcode
funct3
funct7
+4
PC
pcmux_sel
0 1
Load Partial
i_imm
MDR
mem_rdata

Appendix B:

31 27 26 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

NetID:________________________

RV32M Standard Extension

0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

MUL performs an XLEN-bit×XLEN-bit multiplication and places the lower XLEN bits in the destination register. MULH, MULHU, and MULHSU perform the same multiplication but return the upper XLEN bits of the full 2×XLEN-bit product, for signed×signed, unsigned×unsigned, and signed×unsigned multiplication respectively. If both the high and low bits of the same product are required, then the recommended code sequence is: MULH[[S]U] *rdh, rs1, rs2*; MUL *rdl, rs1, rs2* (source register specifiers must be in same order and *rdh* cannot be the same as *rs1* or *rs2*). Microarchitectures can then fuse these into a single multiply operation instead of performing two separate multiplies.

RV32C Standard Extension

RVC uses a simple compression scheme that offers shorter 16-bit versions of common 32-bit RISC-V instructions when:

- the immediate or address offset is small, or
- one of the registers is the zero register (`x0`), the ABI link register (`x1`), or the ABI stack pointer (`x2`), or
- the destination register and the first source register are identical, or
- the registers used are the 8 most popular ones.

The C extension is compatible with all other standard instruction extensions. The C extension allows 16-bit instructions to be freely intermixed with 32-bit instructions, with the latter now able to start on any 16-bit boundary. With the addition of the C extension, JAL and JALR instructions will no longer raise an instruction misaligned exception.

Appendix C:

```
module add_shift_multiplier
(
    input logic clk_i,
    input logic reset_n_i,
    input operand_t multiplicand_i,
    input operand_t multiplier_i,
    input logic start_i,
    output logic ready_o,
    output result_t product_o,
    output logic done_o
);
/**************************** Declarations *****************************/
mstate_s ms;
mstate_s ms_reset;
mstate_s ms_init;
mstate_s ms_add;
mstate_s ms_shift;
logic update_state;
/***************************** Assignments ******************************/
assign ready_o = ms.ready;
assign done_o = ms.done;
assign product_o = {ms.A, ms.Q};
/********************** Behavioral Descriptions *************************/
// Describes reset state
function void reset(output mstate_s ms_next);
    ms_next = 0;
    ms_next.ready = 1'b1;
endfunction
// Describes multiplication initialization state
function void init(input logic[width_p-1:0] multiplicand,
                   input logic[width_p-1:0] multiplier,
                   output mstate_s ms_next);
    ms_next.ready = 1'b0;
    ms_next.done = 1'b0;
    ms_next.iteration = 0;
    ms_next.op = ADD;

    ms_next.M = multiplicand;
    ms_next.C = 1'b0;
    ms_next.A = 0;
    ms_next.Q = multiplier;
endfunction
// Describes state after add occurs
function void add(input mstate_s cur, output mstate_s next);
    next = cur;
    next.op = SHIFT;
    if (cur.Q[0])
        {next.C, next.A} = cur.A + cur.M;
    else
        next.C = 1'b0;
endfunction
```

```
// Describes state after shift occurs
function void shift(input mstate_s cur, output mstate_s next);
      next = cur;
      {next.A, next.Q} = {cur.C, cur.A, cur.Q[width_p-1:1]};
      next.op = ADD;
      next.iteration += 1;
      if (next.iteration == width_p) begin
            next.op = DONE;
            next.done = 1'b1;
            next.ready = 1'b1;
      end
endfunction

always_comb begin
    update_state = 1'b0;
    if ((~reset_n_i) | (start_i) | (ms.op == ADD) || (ms.op == SHIFT))
        update_state = 1'b1;
    reset(ms_reset);
    init(multiplicand_i, multiplier_i, ms_init);
    add(ms, ms_add);
    shift(ms, ms_shift);
end
/********************** Non-Blocking Assignments ***********************/
always_ff @(posedge clk_i) begin
    if (~reset_n_i)
            ms <= ms_reset;
    else if (update_state) begin
        if (start_i & ready_o) begin
            ms <= ms_init;
        end
        else begin
            case (ms.op)
                ADD: ms <= ms_add;
                SHIFT: ms <= ms_shift;
                default: ms <= ms_reset;
            endcase
        end
    end
end

endmodule : add_shift_multiplier

typedef enum bit[2:0] {
      NONE=3'b0, ADD=3'b101, SHIFT=3'b110, DONE=3'b011
} op_e;

typedef struct packed {
      logic ready; logic done; int iteration; op_e op;

      logic[width_p-1:0] M; logic C; logic[width_p-1:0] A;
      logic[width_p-1:0] Q;
} mstate_s;
```