University of California at Berkeley
College of Engineering
Department of Electrical Engineering and Computer Sciences

CS152/252A
Spring 2024

C. Fletcher
2/27/24

Midterm Exam 1

Name: _____

Student ID number: _____

You have 80 minutes to take the exam.

This is a *closed-book, closed-notes* exam, except for one two-sided cheat sheet. Also no calculators, phones, pads, or laptops allowed.

Each question is marked with its number of points (one point per expected minute of time). Start by answering the easier questions then move on to the more difficult ones. You can use the backs of the pages to work out your answers. Neatly copy your answer to the allocated places. **Neatness counts.** We will deduct points if we need to work hard to understand your answer.

Write the student ID numbers of the person to your left and to your right on this page. If you are sitting on an aisle, just indicate "aisle" for left or right.

Left student ID number: _____

Right student ID number: _____

Before you turn in your exam, write your student ID number on all pages.

Handout 1 and scratch paper are provided at the end of the exam booklet.

February 30, 2024 06:54

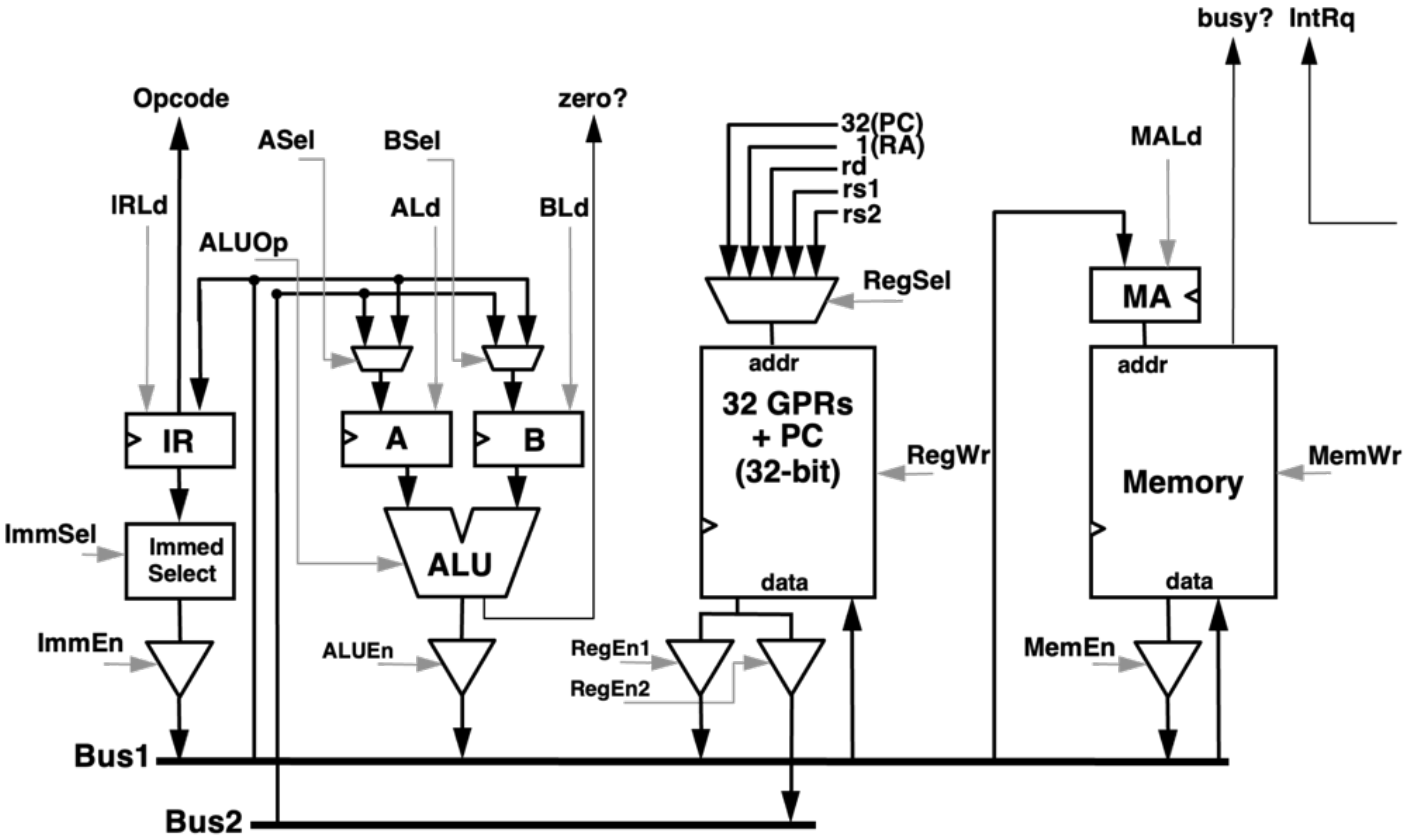
1 [20 points] Microcoding

One of the main issues with the single-bus datapath we've seen in class is that the bus frequently causes a structural hazard. To alleviate this issue, we will add a second bus to this design, called Bus2. However, to cut down on costs, we will only allow the RegFile write to this bus. Additionally, only the A and B registers will be able to read from this bus. Below is a description of the control signals we have to add:

RegEn1 and **RegEn2**: These signals replace RegEn from the original design. RegEn1 enables writing to Bus1, and RegEn2 enables writing to Bus2. These signals may be enabled simultaneously, allowing the RegFile to write the same data to both buses in the same cycle.

ASel and **BSel**: These signals determine which bus the A and B registers will load from during that cycle. They may be set to 1 or 2 to read from Bus1 or Bus2, respectively.

Below is a diagram of this double-bus approach, including the new control signals:



Your task in this question is to write microcode for this new design to implement the SetEqualMem instruction:

```
SetEqualMem rd, rs1, imm(rs2)
```

This instruction checks if the value in register rs1 is equal to the value stored in memory address rs2+imm. If they are equal, rd is set to 1. Otherwise, rd is set to 0. You may assume that reading from rs2+imm will not raise an exception. This instruction uses a new immediate type called X. Fill out the attached microcode table to implement this function. Your microcode should only use the optimal, minimal number of lines possible to implement SetEqualMem on the double-bus CPU. Additionally, your microcode is not allowed to modify either of the source registers (rs1 or rs2).

Some columns of the microcode table are gray. These columns will not be graded and you do not need to fill them out.

Focus first on making your implementation correct. Then, focus on making your correct implementation optimal. The majority of points will be awarded for your pseudocode.

State	Pseudocode	IR Ld	Reg Sel	Reg Wr	Reg En1	Reg En2	A Ld	B Ld	A Sel	B Sel	ALUOp	ALU En	MA Ld	Mem Wr	Mem En	Imm Sel	Imm En	μBr	Next State
FETCH0:	MA ← PC; A ← PC	*	PC	0	1	0	1	*	1	*	*	0	1	0	0	*	0	N	*
	IR ← Mem	1	*	0	0	0	0	*	*	*	*	0	0	0	1	*	0	S	*
	PC ← A+4	0	PC	1	0	0	0	*	*	*	INC_A_4	1	*	0	0	*	0	D	*
. . .																			
NOP0:	microbranch back to FETCH0	*	*	0	0	0	*	*	*	*	*	0	*	0	0	*	0	J	FETCH0
SetEqual Mem:																			

2 [20 points] Iron Law

For this problem, assume that we are working with the fully-bypassed five-stage pipelined processor shown in lecture.

(1) [1 pt each] Suppose that we include a hardware prefetcher. Explain how it would affect each term from the Iron Law.

(a) Instructions / Program:

(b) Cycles / Instruction:

(c) Time / Cycle:

(2) [1 pt each] Suppose we extended the RISC-V Base ISA with a conditional move instruction which copies `rs1` to `rd` if and only if `rs2` is nonzero. Explain how it would affect each term from the Iron Law.

(a) Instructions / Program:

(b) Cycles / Instruction:

(c) Time / Cycle:

(3) [1 pt each] Under certain settings, compilers will replace all instances where a function is called with the body of the function (a common optimization known as *inlining*). Explain how it would affect each term from the Iron Law.

(a) Instructions / Program:

(b) Cycles / Instruction:

(c) Time / Cycle:

(4) A very common task for computers is computation of dense matrix-matrix products. At the core of these routines is a microkernel, a highly optimized piece of code to compute a small matrix product. For this problem, assume we want to compute $C = AB$, where A, B, and C are all 2×2 matrices. Alice writes the following implementation:

```
int a0, a1;
for (int i = 0; i < 2; i += 1) {
    a0 = A[i][0];
    a1 = A[i][1];
    for (int j = 0; j < 2; j += 1) {
            C[i][j] = a0 * B[0][j] + a1 * B[1][j];
    }
}
```

On the other hand, Bob writes the following implementation:

```
int c00 = 0, c01 = 0;
int c10 = 0, c11 = 0;
int a0, a1, b0, b1;
for (int k = 0; k < 2; k += 1) {
    a0 = A[0][k];
    a1 = A[1][k];
    b0 = B[k][0];
    b1 = B[k][1];
    c00 += a0 * b0;
    c01 += a0 * b1;
    c10 += a1 * b0;
    c11 += a1 * b1;
}
C[0][0] = c00;
C[0][1] = c01;
C[1][0] = c10;
C[1][1] = c11;
```

(a) [6 pts] For this problem, we entirely ignore control flow instructions (jumps, branches) and loop variable increments. Using the RISC-V base ISA, what proportion of instructions are memory instructions? What proportion of instructions are arithmetic instructions? Assume that A, B, and C live in memory while other variables reside in registers (e.g. indexing into A requires a load or store while using `a0` once the value is loaded does not).

(b) [1 pt] Assume memory instructions take 5 cycles to execute on average, while arithmetic instructions only take 1 cycle to execute. Whose approach achieves a lower CPI?

(c) [4 pts] Assume that both Alice's microkernel and Bob's microkernel are scaled up to 6×6. Do your results from the previous part still hold? Justify your answer qualitatively. (Hint: how many registers are available to hold temporary values?)

Alice:

```
int a0, a1;
for (int i = 0; i < 6; i++) {
    a0 = A[i][0];
    a1 = A[i][1];
    a2 = A[i][2];
    a3 = A[i][3];
    a4 = A[i][4];
    a5 = A[i][5];
    for (int j = 0; j < 6; j++) {
            C[i][j] = a0 * B[0][j] + a1 * B[1][j] +
                      a2 * B[2][j] + a3 * B[3][j] +
                      a4 * B[4][j] + a5 * B[5][j];
    }
}
```

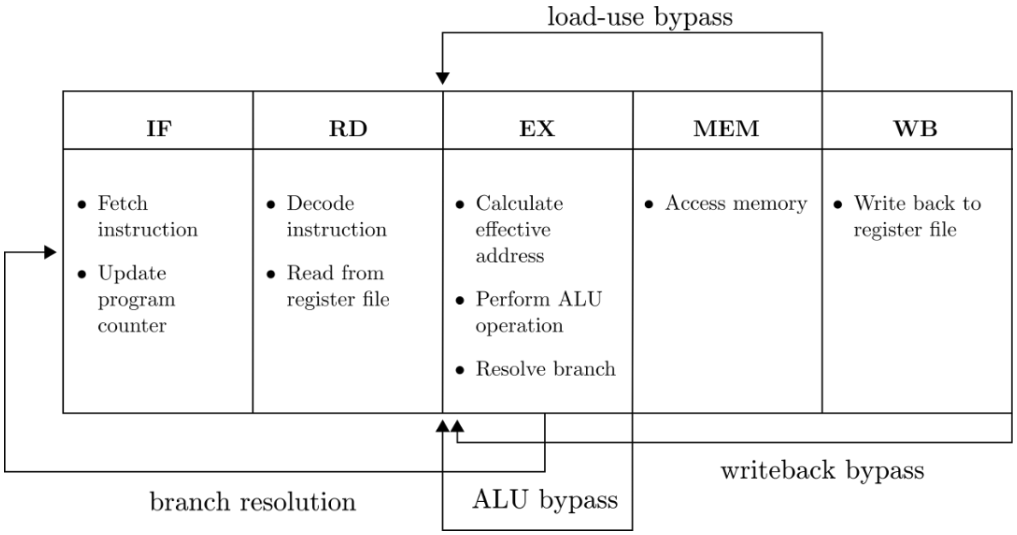
Bob:

```
int c00 = 0, c01 = 0, c02 = 0, c03 = 0, c04 = 0, c05 = 0;
...
int c50 = 0, c51 = 0, c52 = 0, c53 = 0, c54 = 0, c55 = 0;

int a0, a1, ... a5;
int b0, b1, ... b5;
for (int k = 0; k < 6; k++) {
    a0 = A[0][k];
    a1 = A[1][k];
    ...
    a5 = A[5][k];

    b0 = B[k][0];
    b1 = B[k][1];
    ...
    b5 = B[k][5];

    c00 += a0 * b0;
    c01 += a0 * b1;
    ...
    c55 += a5 * b5;
}
C[0][0] = c00;
C[0][1] = c01;
...
C[5][5] = c55;
```

3 [20 + 3 points] Pipelining Potpourri

For this question, consider the standard fully-bypassed RV32I 5-stage pipeline.

- Branches are resolved at the end of the Execute stage
- Bypass paths bypass into the operand registers before the Execute stage (the bypass select muxes are in the Decode stage)
- There are no illegal opcodes, integer overflows, or illegal memory accesses.



The subsections of this question will consider the execution of the following loop on this pipeline.

```
1  loop:  lw t0, 0(a0)
2         add x1, x1, t0
3         sw x1, 0(a0)
4         lw a0, 4(a0)
5         bne x0, a0, loop
6  end:
```

(1) **[4 pts] Hazards**

(a) What hazards do each of the following microarchitectural optimizations address? (Select all that apply.)

i. [1 pt] Bypass Paths

□ RAW □ WAR □ WAW □ RAR □ Control □ Structural

ii. [1 pt] Branch Prediction

□ RAW □ WAR □ WAW □ RAR □ Control □ Structural

(b) [2 pts] What hazards are unresolvable by bypassing on the above pipeline? (Select all that apply.)

□
```
lw t0, 0(a0)
add x1, x1, t0
```

□
```
add x1, x1, t0
sw x1, 0(a0)
```

□
```
sw x1, 0(a0)
lw a0, 4(a0)
```

□
```
lw a0, 4(a0)
bne x0, a0, loop
```

(2) **[5 pts] Pipeline Diagram** Analyze the above code sequence that runs on the fully-bypassed pipeline. Assume memory accesses take ***one cycle*** and the branch predictor predicts ***not taken***. For this question, fill in the pipeline diagram on the *next page* for the first iteration of this loop and the first instruction of the second iteration. **To reiterate, the pipeline diagram on the next page will be graded.**

(3) **[2 pts] CPI** What is the CPI (cycles per instruction) of 5 iterations of this loop?

CPI:

(4) **[2 pts] Removing the Load-Use Bypass** Bypassing is expensive. Instead of keeping all of our bypass paths, we decide to remove the load-use bypass (the bypass path from the end of memory to the end of decode).

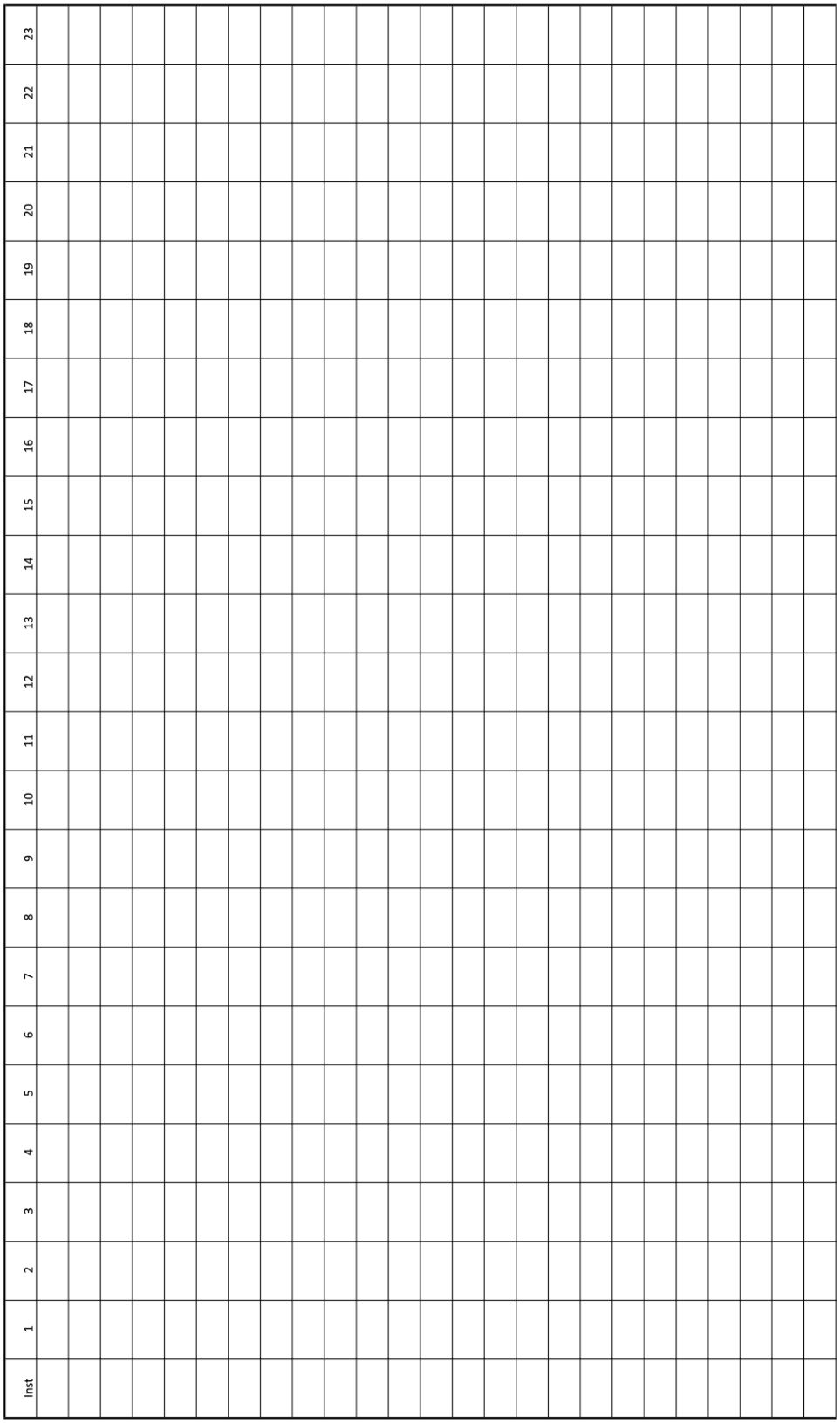
What is the CPI (cycles per instruction) of 5 iterations of this loop, with the load-use bypass removed? For your convenience, we have provided another pipeline diagram as scratch space, but it will *not be graded*.

CPI:

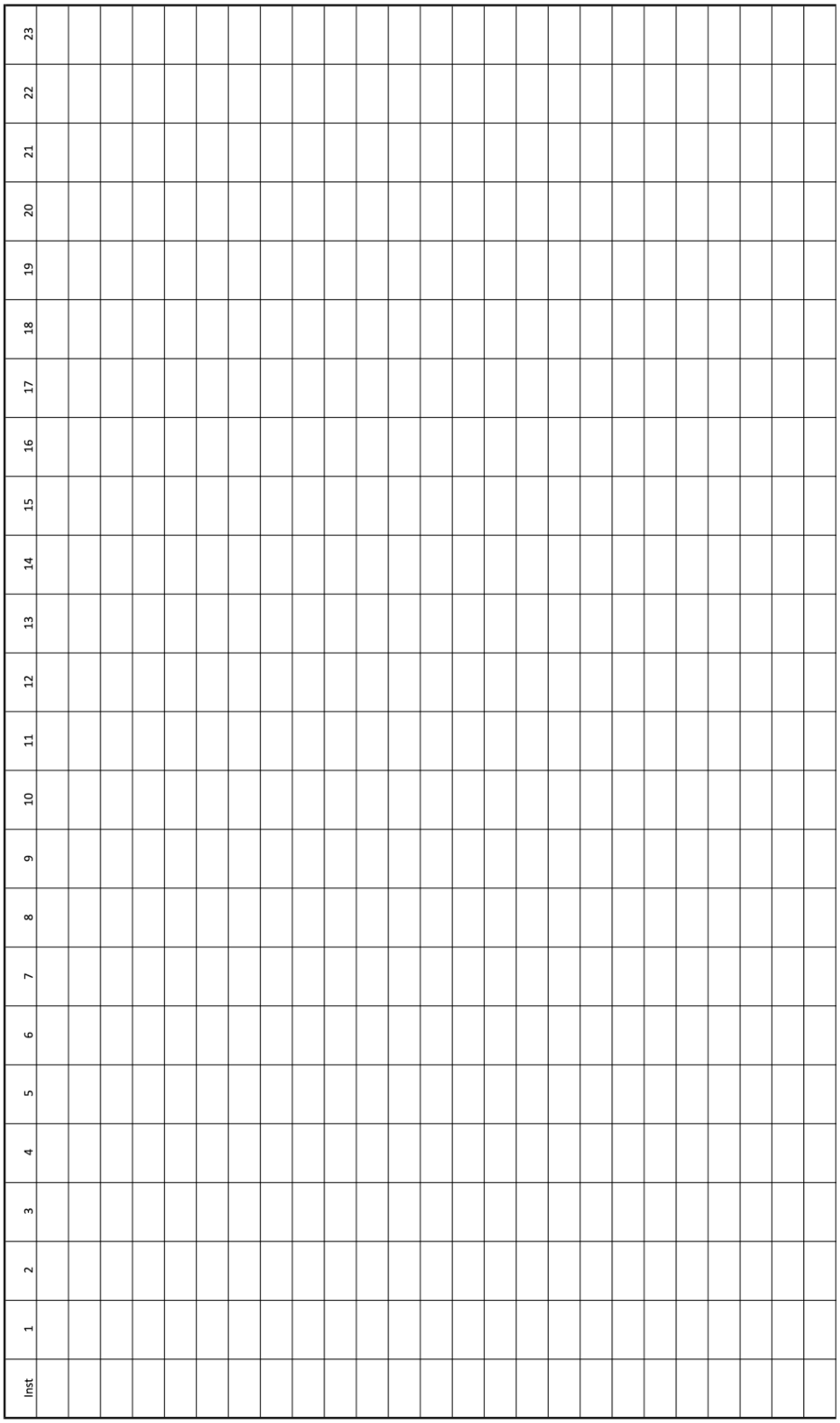
(5) **[3 pts] Modifications** Which of the following modifications, if any, would improve (decrease) CPI for the above code sequence relative to the original fully bypassed pipeline that has a not-taken branch prediction scheme? (Select all that apply.)

- □ Adding a load delay slot
- □ Removing bypass paths
- □ Adding a branch delay slot
- □ Replacing the current branch prediction scheme with perfect branch prediction
- □ Splitting the memory stage into more stages, so memory access takes more cycles

[5 pts] Graded Pipeline Diagram: Fully Bypassed Pipeline



[0pts - Optional] Ungraded Pipeline Diagram: Load-Use Bypass Removed



(6) **[4 pts] Exceptions** Instead of directly computing a sum, we would like to apply a function where we scale the components. We want our RISCV pipeline, detailed at the beginning of this question, to support a new complex integer instruction, so we add a new ISA extension. We add a new instruction, SCALE:

```
SCALE rd, rs1, rs2
R[rd] = (R[rs1] * 5 + 5) / R[rs2]
```

We introduce an unpipelined arithmetic functional unit in parallel with the ALU that has a 3 cycle latency. Because the execute stage is of variable latency, a younger instruction may commit before an older complex integer instruction, assuming no data dependencies. Assume the only exceptions are illegal opcode exceptions and arithmetic exceptions (e.g. divide by zero and integer overflows). Illegal opcode exceptions are detected in the Decode stage and arithmetic exceptions are detected in the Execute stage.

(a) [1 pt] Does the pipeline, as described, support precise exceptions? Select one.

□ Yes □ No

(b) [3 pts] If yes, how are precise exceptions supported? If not, how do we add support for precise exceptions? Your answer should be a maximum total of **three sentences**.

(7) **[3 pts - Extra Credit] To Infinity and Beyond** For this part, we will start with a blank slate and not use the pipelines that we defined in the previous parts. Suppose you are given a combinational/single-cycle datapath where every instruction has the same latency of T ns. You are allowed to pipeline this datapath into P stages. Assume that for any P, there are a) no hazards of any type and b) no pipelining overheads due to digital logic-level effects. (In other words, you can assume that given P stages, the clock frequency can be increased by a factor of exactly P.)

Given any terminating program F, what is the Time / Program for F in the limit, as P $\rightarrow$ infinity, on this pipeline?

4 [20 points] Cache

Assume in this question that you have a 4KiB 2-way set associative L1 data cache with 64-byte cache lines, and that you are using 32-bit addresses.

(1) How many bits are required for the tag, index and offset? [2 pts]

(2) Now consider the following program. Assume that each long element is 8 bytes, and that the matrix elements are stored in row-major order. Assume that the first element in the matrix is aligned to the start of a cache line.

```
int sum = 0;
int matrix[64][64];
for (int i = 0; i < 64; i++) {
    for (int j = 0; j < 64; j++) {
        sum += matrix[i][j];
    }
}
```

(a) [2 pts] What's the number of cache misses?

(b) [2 pts each, 8 pts total] What's the number of occurrences each of the following events?

i. Hits:
ii. Compulsory misses:
iii. Conflict misses:
iv. Capacity misses:

(3) Suppose we add software prefetching. Assume that each iteration of the inner loop takes 30 cycles (when there is a L1 miss). Of those 30 cycles, 25 cycles is the L1 miss penalty while 5 cycles is taken by the addition operation. Ignore any overheads from executing other instructions, including the software prefetch instruction. The prefetcher only prefetches when **prefetch_cond** becomes true (which is set by instructions that are not shown, and that you likewise do not need to model).

```
long sum = 0;
long matrix[64][64];
for (int i = 0; i < 64; i++) {
    for (int j = 0; j < 64; j++) {
        if (prefetch_cond) prefetch(&matrix[i][j]+OFFSET);
        sum += matrix[i][j];
    }
}
```

(a) [2 pts] What should **OFFSET** be set to in order to prefetch the correct value? The address calculation works on the granularity of bytes.

(b) [2 pts] What should **prefetch_cond** be to minimize total number of cycles of execution?

(c) [2 pts] How many prefetch requests would be issued?

(d) [2 pts] With prefetching, how many cache misses are there?

5 [20 points] Virtual Memory

Let us once again consider multiplying two matrices, this time using the classical inner product approach for multiplying 2 NxN square matrices A and B and storing the result in an NxN matrix C. We use the following C language program with N = 512 in this problem.

```
#define N 512
double A[N][N], B[N][N], C[N][N];
...
for (int i = 0; i < N; i += 1) {
    for (int k = 0; k < N; k += 1) {
        for (int j = 0; j < N; j+= 1) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Note:

- `sizeof(<type>)` provides the size of a data type in the C language.
- Arrays are stored contiguously in the C language. `A[x]` refers to row x of A and `A[x][y]` to the element at row x and column y.

(1) [2 pts] Page Usage by Matrix A (Select all that are true.)

- □ In the least optimal case, matrix A can use $\frac{N*N*sizeof(double)}{PageSize}$ + 1 pages when the start of A is not page aligned.
- □ In the most optimal case, matrix A can use $\frac{N*N*sizeof(double)}{PageSize}$ pages when the start of A is page aligned.
- □ Matrix A always uses $\frac{N*N*sizeof(double)}{PageSize}$ pages irrespective of page alignment.
- □ Matrix A always uses $\frac{N*N}{PageSize}$ pages irrespective of page alignment.

Your solution to questions (2) and (3) can use the following:

- Numerical Constants
- Arithmetic Operators: $+$, $-$, $*$, $/$, $\%$
- Parentheses: $(...)$
- Variables: `sizeof(<type>)`, $PageSize$, $TLBSize$, $PageTableSize$, $CacheSize$, $CacheLineSize$

(2) [4 pts] Page Faults in Matrix Multiply

Given that we enter the matrix multiply loops with arrays A, B, and C entirely in disk, how many page faults does the matrix multiply operation incur? Assume each matrix starts at the beginning of a page and we have enough main memory to fit all 3 matrices.

(3) [4 pts] TLB Usage in Matrix Multiply

What is the minimum number of entries we need in a fully-associative TLB to guarantee all TLB misses in the matrix multiply operation are only caused by page faults (i.e., there are only compulsory misses in the TLB)? Note the order of the three loops. Assume each matrix starts at the beginning of a page and we have enough main memory to fit all 3 matrices.

(4) [2 pts] Comparing Standard Pages and Hugepages

The standard page size provided by the Linux kernel is 4 KiB. The default hugepage size provided by the Linux kernel is 2 MiB. Which of the following is true for our matrix multiply operation, given the descriptions in (2) and (3)? (Select all that apply.)

- □ Hugepages incur more page faults than standard pages
- □ Standard pages incur more page faults than huge pages
- □ Hugepages lead to lesser TLB usage
- □ Standard pages lead to lesser TLB usage

(5) Broader Comparisons Between Standard Pages and Hugepages

The standard page size provided by the Linux kernel is 4 KiB. The default hugepage size provided by the Linux kernel is 2 MiB.

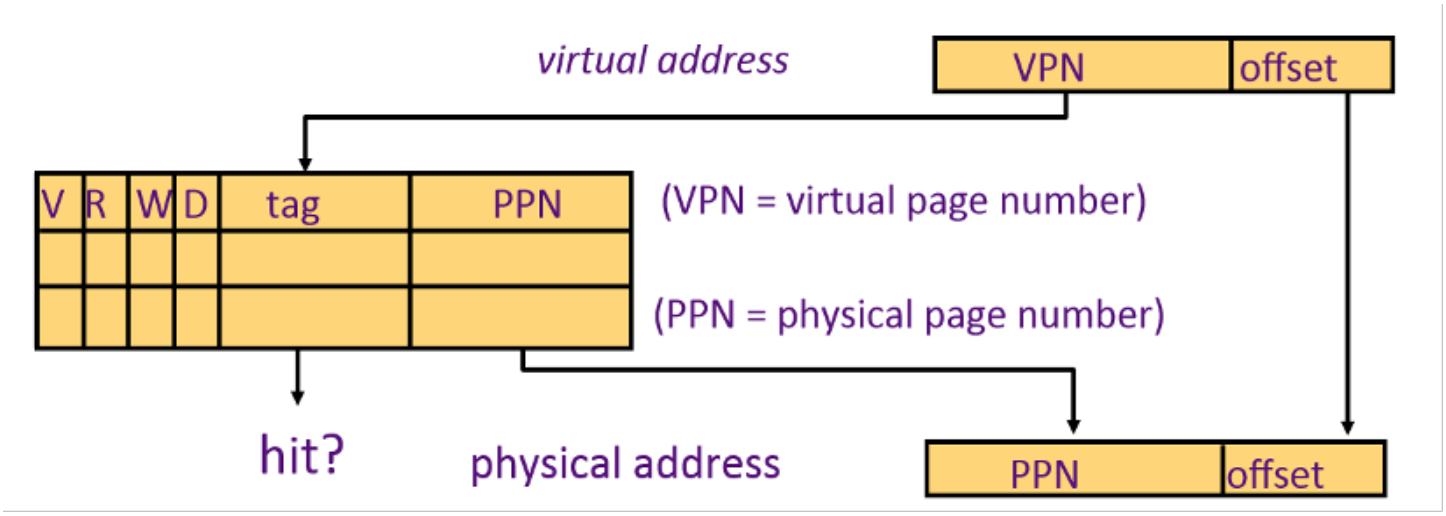
(a) [3 pts] What are the advantages of using hugepages in general? (Select all that apply.)

- □ Use fewer page table entries in general
- □ Greater TLB reach
- □ Lesser internal fragmentation
- □ Greater internal fragmentation
- □ Reduced TLB pressure
- □ Better performance for applications accessing large contiguous portions of memory

(b) [3 pts] What are the disadvantages of using hugepages in general? (Select all that apply.)

- □ Page faults can be much longer
- □ Lesser internal fragmentation
- □ Greater internal fragmentation
- □ Increased TLB pressure
- □ Inefficient at enforcing varying protection levels that change at intervals on the order of KiBs
- □ Expensive for sharing memory regions of size on the order of KiBs

(6) [2 pts] Does a TLB (as shown below) need any more information to support multiple page sizes at the same time? If so, why? If not, how can the existing TLB support multiple page sizes at once?



CS152 Computer Architecture and Engineering

Bus-Based RISC-V Implementation

General Overview

Figure H1-A shows a diagram of a bus-based implementation of the RISC-V architecture.[1] In this microarchitecture, the different components of the machine share a common 32-bit bus through which they communicate. Control signals determine how each of these components is used and which components get to use the bus during a given clock cycle. These components and their corresponding control signals are described below.

For this handout, we shall use a positive logic (active-high) convention. Thus, when we say signal X is "asserted", we mean that signal X is a logical 1, and that the wire carrying signal X is raised to the "HIGH" voltage level.

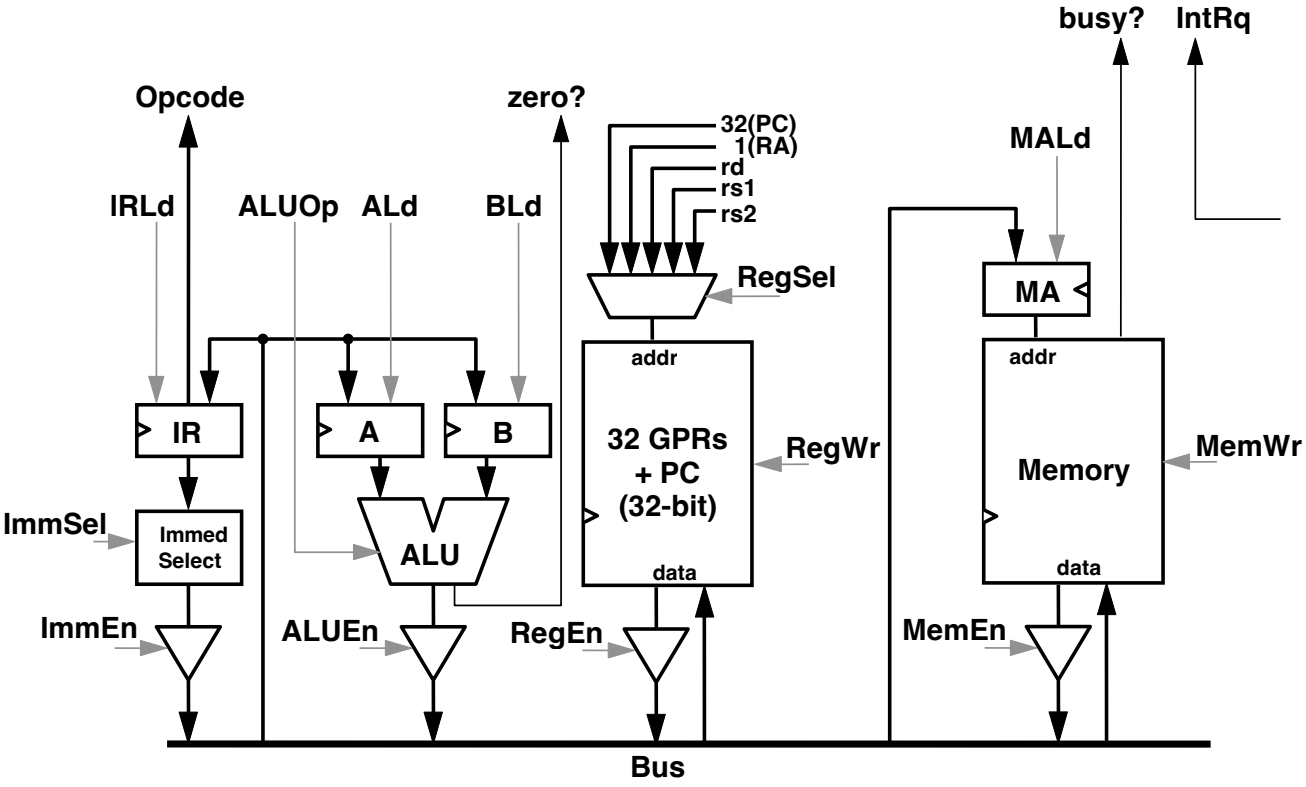


Figure H1-A: A single-bus datapath for RISC-V.

[1] This document discusses an implementation of RV32, the 32-bit address space variant of RISC-V. RV32 uses 32-bit wide general-purpose registers and a 32-bit wide PC register. An RV64 microcoded processor would appear identical to the one presented in this document, except for 64-bit registers, a 64-bit bus, and some additional ALU operations to support the new instructions added to RV64.

The Enable Signals

Since the bus is shared by different components, there is a need to make sure that only one component is driving ("writing") the bus at any point in time. We do this by using tri-state buffers at the output of each component that can write to the bus. A tri-state buffer is a simple combinational logic buffer with enable control. When enable is 1, then the output of the buffer simply follows its input. When enable is 0, then the output of the buffer "floats" – i.e., regardless of its input, the tri-state buffer will not try to drive any voltage on the bus. Floating the buffer's output allows some other component to drive the bus.[2]

In this bus-based RISC-V implementation, there are four enable signals – ImmEn, ALUEn, RegEn, and MemEn – each of which are directly connected to the enable of a tri-state buffer. When setting these signals, it is important to ensure that *at most one* device is driving the bus at any time. It is possible not to assert any of the four signals, in which this case the bus will float and have an undefined value.

Special-Purpose Registers and Load Signals

In addition to the entries in the register file, this bus-based implementation contains four other special-purpose internal registers: IR (instruction register), A, B, and MA (memory address). These 32-bit registers are *positive edge-triggered* with load enable control. As shown, these registers take their data inputs from the bus. If a register's enable is asserted during a given cycle, then the value on the bus during that cycle will be copied into the register (sampled) at the *next* rising clock edge. If the enable control is 0, then register's value remains changed.

We refer to these register enable signals as *load* signals denoted by the suffix "Ld": IRLd, ALd, BLd, and MALd. In addition, the RegWr and MemWr signals are also load signals, but their exact functionality will be discussed later. It is possible to assert more than one load signal at a time, in which case the value on the bus will be loaded into all registers whose load signals are asserted.

The Instruction Register

The instruction register (IR) is used to hold the current 32-bit instruction word. The opcode and function fields (see the base instruction formats described by the RISC-V user-level ISA specification) are used by the microcode control logic to identify the instruction and dispatch to the appropriate microcode sequence. As shown in Figure H1-A, the immediate field is connected to an immediate selector and sign extender and then to the bus. Finally, as described below, the register specifier fields connect to a multiplexer that drives the register file address input.

The Immediate Selector & Sign Extender

The module labeled "Immed Select" is an immediate selector and sign extender that extracts the immediate from the IR and sign-extends the immediate to a 32-bit value. Five different immediates can be produced: ImmI, ImmU, ImmS, ImmJ, and ImmB. ImmI selects the immediate for I-type instructions; ImmU selects for the U-type format used with LUI/AUIPC instructions; ImmS selects

[2] You can also think of a tri-state buffer as an electronically controlled switch. If enable is 1, then the switch connects the input and the output as if they were connected by a wire. If enable is 0, then the input is electrically disconnected from the output. Note that the tri-state buffer is a memoryless device and is *not* the same as a latch or a flip-flop.

for the S-type format used with stores; ImmJ selects for the J-type format used with unconditional jumps; and ImmB selects for the B-type format used with conditional branches. All immediates are signed-extended.

The Arithmetic Logic Unit

The ALU takes 3 inputs: two 32-bit operand inputs, connected to the A and B registers, and an ALUOp input. ALUOp selects the operation to be performed on the operands. Assume that the ALU can perform the following operations by default, if not explicitly stated otherwise:

ALUOp	ALU Result Output
COPY_A	A
COPY_B	B
INC_A_1	A+1
DEC_A_1	A-1
INC_A_4	A+4
DEC_A_4	A-4
ADD	A+B
SUB	A-B
SLT	Signed(A) < Signed(B)
SLTU	A < B
AND	A&B

Table H1-1: ALU Operations for Handout #5.

In order to implement the entire RISC-V ISA, we will need a few more ALU operations.

The ALU is purely combinational logic. It has two outputs, a 32-bit main result output and 1-bit zero flag output, *zero*. The result output is computed as in Table H1-1. The zero flag simply indicates if the ALU result output equals zero. If the result is 0 then *zero* is 1, otherwise *zero* is 0. For example, if A=2, B=2, and ALUOp=SUB, then the ALU result will be 0, and *zero* will be 1. This flag is used to resolve conditional branches in microcode.

The Register File

The register file contains the 32 general-purpose registers (GPRs) and the PC. The register file itself has a 6-bit address input (addr) and a single 32-bit bidirectional data port.

Two control signals determine how the register file is used during a certain cycle: RegWr and RegEn. RegWr determines whether a write operation is performed. RegEn is the enable signal for the tri-state buffer connecting the register file to the bus. If RegEn is 1, then the data from a read operation is driven onto the bus. Note that RegWr and RegEn are mutually exclusive; while it is possible for both signals to be 0 at the same time, indicating that the register file is unused that cycle, RegWr and RegEn should never be asserted simultaneously.

Figure H1-B shows how RegEn and RegWr is connected to the peripheral control circuity of the register file. The address input (addr) determines which entry in the register file is used. Read operations are assumed to be combinational – if you change the value at the addr input, then the value at the data output will change appropriately (after some delay), *even if* no clock edge occurs. Write operations, on the other hand, are positive edge-triggered, such that data from the register's data input is only stored to the selected address at the next rising clock edge.

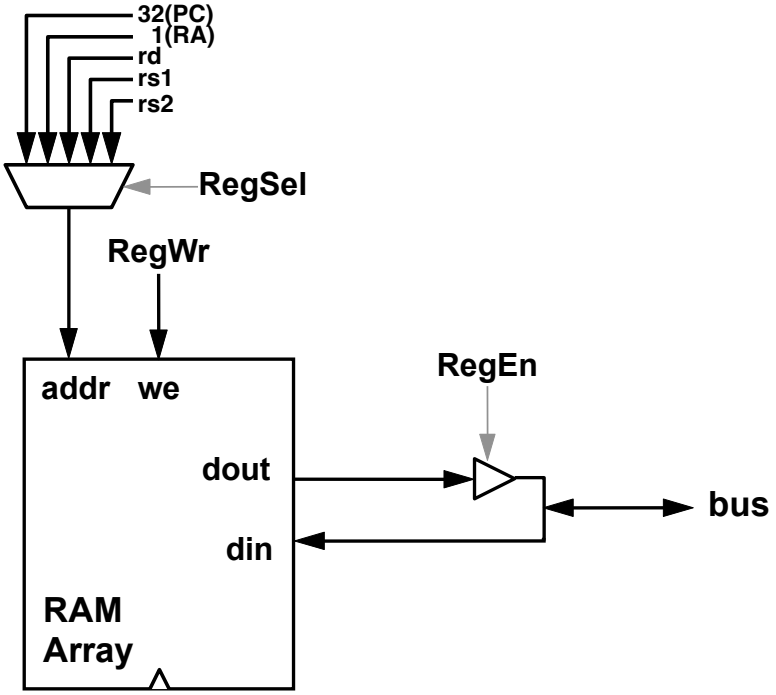


Figure H1-B: Control signals and bus connections for the register file and the memory.

While the addr input selects which register is used, the RegSel control signal selects what value is used as the addr input. As shown, RegSel controls a multiplexer that chooses between (at least) five possible values. The first two are the hardwired values 32 and 1, corresponding to the PC and the return address (RA) register, respectively. The next three (rs1, rs2, and rd) are taken from the register specifier fields of the IR. The following figure shows the corresponding locations of the rs1, rs2, and rd fields:

…	rs2	rs1	…	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

The exact meaning of these fields depends on the instruction and instruction type. Please refer to the RISC-V ISA manual and the lecture notes. Note that these fields are 5 bits wide, while addr is 6 bits wide since the register file here contains more than just the 32 GPRs. The 5-bit fields are widened to a 6-bit address by padding the MSB with a 0. When specifying the value of RegSel, you should use the symbols PC, RA, rs1, rs2, and rd.

The Memory

The memory module is very similar to the register file except that the address input is taken from an edge-triggered register, MA (memory address). Thus, it takes two steps to access a particular memory location. First, the MA is loaded with the desired memory address, and then the desired memory operation is performed on the location specified by MA. The MemWr and MemEn control signals work identically to RegWr and RegEn.

The main difference between memory module and the register file is the busy signal. Unlike the register file, the memory may take a variable number of cycles to perform a read or write (e.g., if a cache miss occurs). The busy signal indicates that the memory operation has not yet completed. The microcode can then respond to the busy signal appropriately by stalling. Note that the value of MA, as well as the data input for stores, should remain unchanged while busy is asserted.

The Clock Period and Timing Issues

We will assume that the clock period is long enough to guarantee that the results of all *combinational* logic paths are valid and stable before the setup time of any edge-triggered components attempting to latch these results.

Remember that these combinational paths include not only computational elements like the sign extender, and the ALU, but also the register file during *read* operations. Specifically, the path from addr to the data output in the register file is purely combinational. As mentioned above, if you change the value at the addr input, then the value at the data output will change appropriately (after some delay), *even if* no clock edge occurs.

Also remember that the path from data *input* to data *output* is *not* combinational. This applies not only to the register file and memory but also to other edge-triggered registers (i.e., IR, A, B, and MA). When performing a write to any of these components, the value at the data input is not sampled until the *next* rising clock edge and can thus only be read during the next clock cycle.

Finally, we will assume that there are no hold time violations.

Microprogramming on the Bus-Based RISC-V Implementation

In the past, simple CISC machines employed a bus-based architecture wherein the different components of the machine (ALU, memory, etc.) communicated through a common bus. This type of architecture is easy and inexpensive to implement but has the disadvantage of requiring all data movement between components to share the bus. Because of this bottleneck, an instruction on such an architecture typically took several cycles to execute.

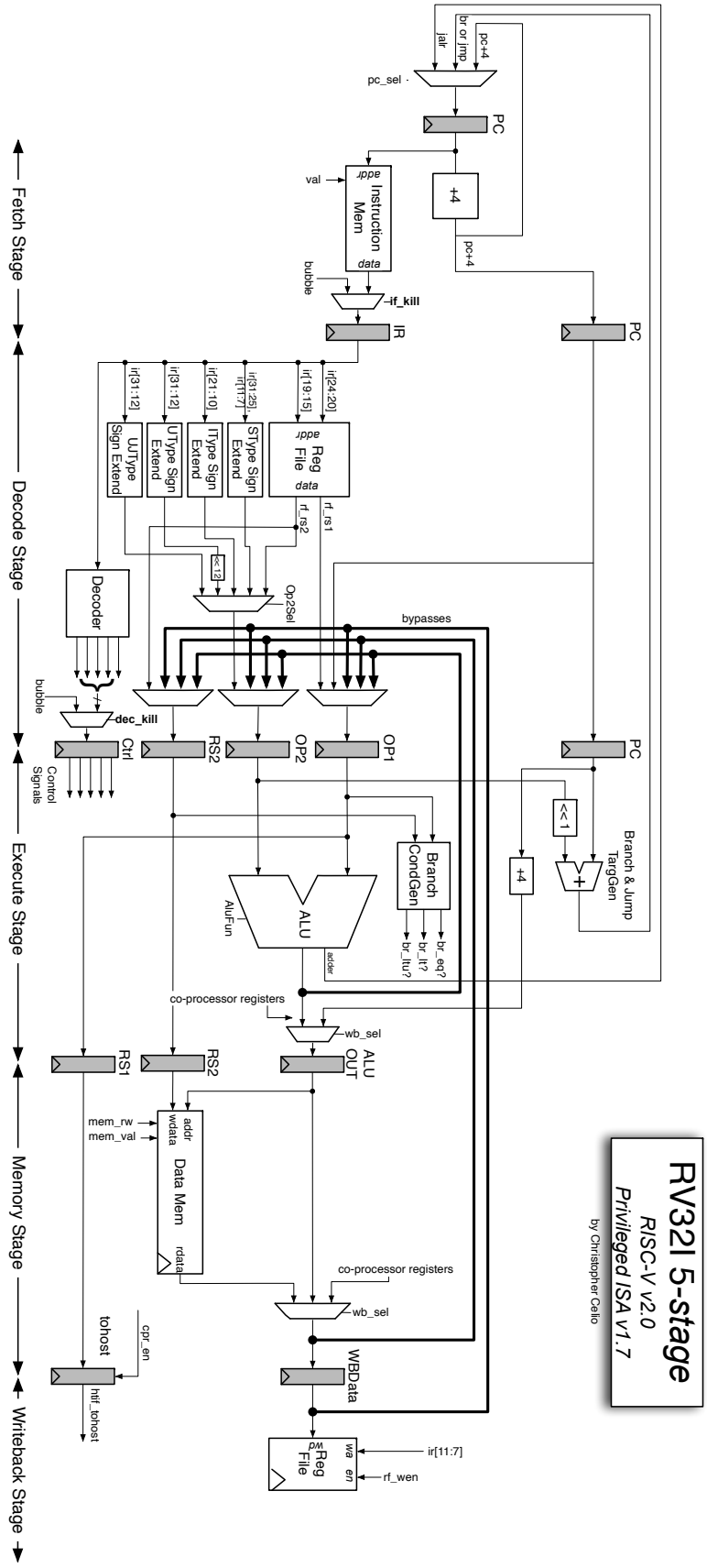
Microprogramming makes it easy to generate the control signals for such multi-cycle instructions. A microcode program is basically a finite state machine description. Each line of microcode corresponds to some state of the machine and contains a "microinstruction" which specifies the outputs for that state and the next state. The outputs are used to control the different components in the datapath. The next state depends on the current state and possibly on certain other signals (e.g., condition flags). As we shall see, this simple FSM model proves to be very powerful, allowing complex operations like conditional branches and loops to be performed.

Table H1-3 in Appendix B shows a microcode table for the bus-based RISC-V implementation. Some lines have been filled in as examples. The first column in the microcode table contains the state label of each microinstruction. For brevity, we assume that the states are listed in increasing numerical order, and we only label important states. For example, we label state FETCH0, and assume that the unlabeled line immediately following it corresponds to state (FETCH1). The second column contains a pseudo-code explanation of the microinstruction in RTL-like notation. The rest of the columns, except the last two, represent control signals that are asserted during the current cycle. 'Don't care' entries are marked with a '*'.

The last two columns specify the next state. The μBr (*micro*branch) column represents a 3-bit field with six possible values: N, J, EZ, NZ, D and S. If μBr is N (next), then the next state is simply (*current state* + 1). If it is J (jump), then the next state is *unconditionally* the state specified in the Next State column (i.e., an unconditional microbranch). If it is EZ (branch-if-equal-zero), then the next state depends on the value of the ALU's *zero* output signal (i.e., a conditional microbranch). If *zero* is asserted (== 1), then the next state is that specified in the Next State column, otherwise, it is (*current state* + 1). NZ (branch-if-not-zero) behaves exactly like EZ, except NZ branches to the next state is *zero* is not asserted (!= 1). If μBr is D (dispatch), then the FSM looks at the opcode and function fields in the IR and goes into the corresponding state. In this implementation, we assume that the dispatch goes to the state labeled (RISC-V-instruction-name + "0"). For example, if the instruction in the IR is SW, then the dispatch will go to state SW0. If μBr is S, then the next state depends on the value of the *busy?* signal. If *busy?* is asserted, then the next state is the same as the *current state* (the μPC "spins" on the same microaddress), otherwise, the next state is (*current state* + 1).

The first three lines in the table (starting at FETCH0) form the *instruction fetch* stage. These lines are responsible for fetching the next instruction opcode, incrementing the PC, and then jumping to the appropriate line of microcode based on the opcode fetched. The instruction fetch stage is performed for every instruction, and every instruction's microcode should always end by executing an instruction fetch for the next instruction. The easiest way to do this is by having the last microinstruction of an instruction's microcode include a microbranch to FETCH0. The microcode for NOP provides a simple example. For the rest of the instructions, their microcode sequences intentionally omitted; we expect students to fill out the table themselves.

RV32I 5-Stage Pipeline Diagram



Appendix A. A Cheat Sheet for the Bus-based RISC-V Implementation

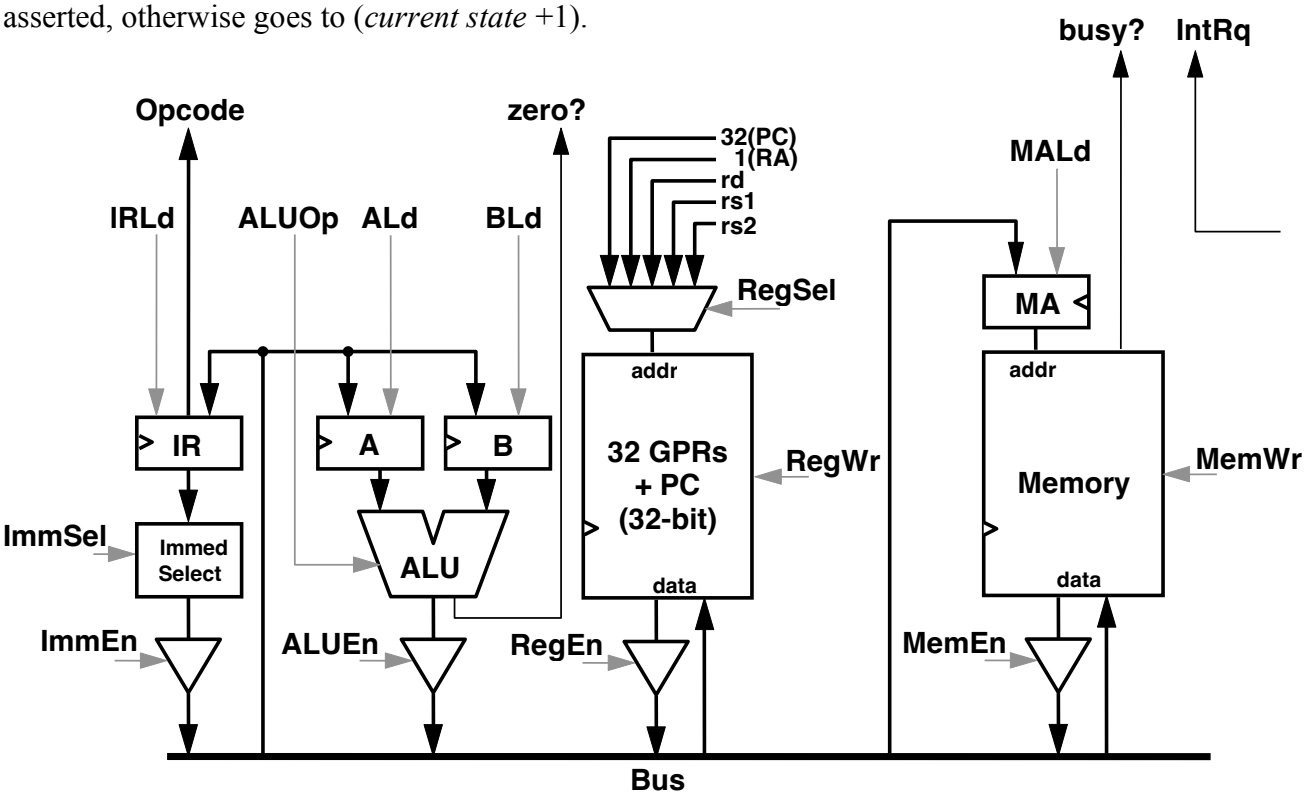
For your reference, we have reproduced the bus-based datapath diagram as well as a summary of some important information about microprogramming in the bus-based architecture.

Remember that you can use the following ALU operations:

ALUOp	ALU Result Output
COPY_A	A
COPY_B	B
INC_A_1	A+1
DEC_A_1	A-1
INC_A_4	A+4
DEC_A_4	A-4
ADD	A+B
SUB	A-B
SLT	Signed(A) < Signed(B)
SLTU	A < B
AND	A&B

Table H1-2: Available ALU operations

Also remember that μBr (*micro*branch) column in Table H1-3 represents a 3-bit field with six possible values: N, J, EZ, NZ, D, and S. If μBr is N (next), then the next state is simply (*current state* + 1). If it is J (jump), then the next state is *unconditionally* the state specified in the Next State column (i.e., an unconditional microbranch). If it is EZ (branch-if-equal-zero), then the next state depends on the value of the ALU's *zero* output signal (i.e., a conditional microbranch). If *zero* is asserted (== 1), then the next state is that specified in the Next State column, otherwise, it is (*current state* + 1). NZ (branch-if-not-zero) behaves exactly like EZ, but instead performs a microbranch if *zero* is not asserted (!= 1). If μBr is D (dispatch), then the FSM looks at the opcode and function fields in the IR and goes into the corresponding state. If S, the μPC spins if *busy?* is asserted, otherwise goes to (*current state* +1).



Appendix B. Microcode Table for the Bus-based RISC-V Implementation

State	Pseudocode	IR Ld	Reg Sel	Reg Wr	Reg En	A Ld	B Ld	ALUOp	ALU En	MA Ld	Mem Wr	Mem En	Imm Sel	Imm En	μBr	Next State
FETCH0:	MA ← PC; A ← PC	*	PC	0	1	1	*	*	0	1	0	0	*	0	N	*
	IR ← Mem	1	*	0	0	0	*	*	0	0	0	1	*	0	S	*
	PC ← A+4	0	PC	1	0	0	*	INC_A_4	1	*	0	0	*	0	D	*
. . .																
NOP0:	microbranch back to FETCH0	*	*	0	0	*	*	*	0	*	0	0	*	0	J	FETCH0

Table H1-3 (Worksheet 1)

Scratch Paper

Scratch Paper