Problem M1.1: Self-Modifying Code (Spring 2015 Quiz 1, Part A)

In this problem we will use and extend the EDSACjr instruction set from Handout 1, shown in Table A-1.

Opcode	Description
ADD *n*	Accum ← Accum + M[*n*]
SUB *n*	Accum ← Accum - M[*n*]
LD *n*	Accum ← M[*n*]
ST *n*	M[*n*] ← Accum
CLEAR	Accum ← 0
OR *n*	Accum ← Accum | M[*n*]
AND *n*	Accum ← Accum & M[*n*]
SHIFTR *n*	Accum ← Accum shiftr *n*
SHIFTL *n*	Accum ← Accum shiftl *n*
BGE *n*	If Accum ≥ 0 then PC ← *n*
BLT *n*	If Accum < 0 then PC ← *n*
END	Halt machine

Table A-1. EDSACjr Instruction Set

Problem M1.1.A

Write a program that loops over an n-item array and replaces each item with its absolute value, as shown in the following pseudo-code:

```
for (i = 0; i < n; i++)
    A[i] = |A[i]|
```

Part of the program is already written for you, and to simplify your job you can assume the loop will be executed only once. The memory map on the next page shows the memory contents before the program starts. Array A is stored in memory in a contiguous manner, starting from location A. Memory locations N, I, and ONE hold the values of n, i, and 1, respectively. If you need to, you can use additional memory locations for your own variables. You should label each variable and define its initial value.

Memory:

	…
A	A[0]
	A[1]
	…
	A[n-1]
	…
ONE	1
N	n
I	0

Program:

```
loop:   LD      I
        SUB     N
        BGE     done




        LD      I
        ADD     ONE
        ST      I
        BGE     loop
done:   END
```

Problem M1.1.B

Tired of writing self-modifying code, Ben Bitdiddle decides to extend EDSACjr to support indirect addressing. However, because registers are expensive, Ben does not want to add an index register. Instead, he implements the indirect addressing instructions shown in Table A-2. To execute an indirect addressing instruction, the new architecture first reads the target address from memory and then loads/stores the data from/to memory.

Opcode	**Description**
ADDind *n*	Accum ← Accum + M[M[*n*]]
SUBind *n*	Accum ← Accum – M[M[*n*]]
LDind *n*	Accum ← M[M[*n*]]
STind *n*	M[M[*n*]]← Accum

Table A-2. Additional Indirect Addressing Instructions

Using the instructions in Table A-1 and Table A-2, rewrite the program from Question 1 without using self-modifying code. As before, you can use additional memory locations for your own variables. You should label each variable and define its initial value.

Memory:

	…
A	A[0]
	A[1]
	…
	A[n-1]
	…
ONE	1
N	n
I	0

Program:

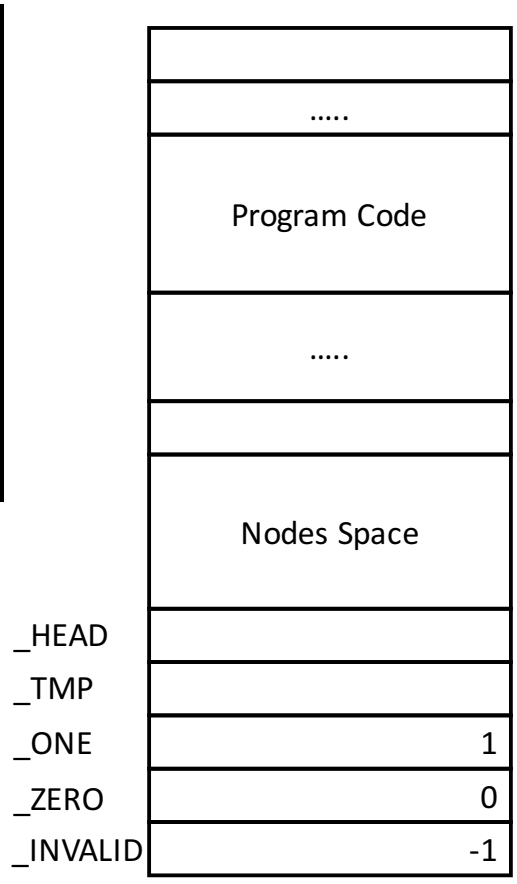
```
loop:   LD      I
        SUB     N
        BGE     done









        LD      I
        ADD     ONE
        ST      I
        BGE     loop
done:   END
```

Problem M1.2: Self-modifying Code (Spring 2017 Quiz 1, Part A)

In this question, you will implement linked-list operations using self-modifying code on an EDSACjr machine. The memory layout is shown in the figure on the right. You have access to the named memory locations as indicated. Linked-list nodes consist of two words: the first is an integer `value`, the second is an address pointing to the `next` node. `_HEAD` contains the address of the first node of the list (or `_INVALID` if it is empty). The `next` field of the last node is `_INVALID`. All valid addresses are positive. You may create new local and global labels as explained in the EDSACjr handout.

Table A-1 shows the EDSACjr instruction set.

Opcode	Description	Bit Representation
ADD *n*	Accum ← Accum + M[*n*]	00001 *n*
SUB *n*	Accum ← Accum - M[*n*]	10000 *n*
STORE *n*	M[n] ← Accum	00010 *n*
CLEAR	Accum ← 0	00011 00000000000
OR *n*	Accum ← Accum | M[*n*]	00000 *n*
AND *n*	Accum ← Accum & M[*n*]	00100 *n*
SHIFTR *n*	Accum ← Accum shiftr *n*	00101 *n*
SHIFTL *n*	Accum ← Accum shiftl *n*	00110 *n*
BGE *n*	If Accum ≥ 0 then PC ← *n*	00111 *n*
BLT *n*	If Accum < 0 then PC ← *n*	01000 *n*
END	Halt machine	01010 00000000000

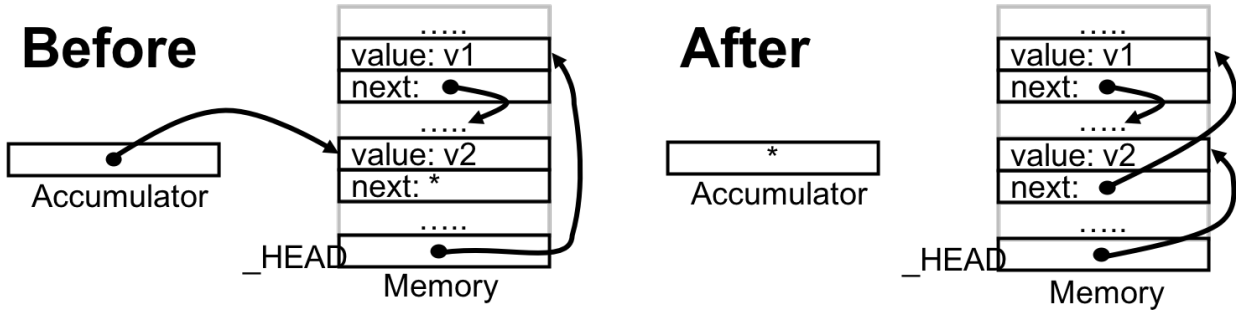


You may also use the following macros if required.

Macro	Description
STOREADR *n*	Replace the address field of location *n* with the contents of the accumulator
LOADADR *n*	Load the address field of location *n* into the accumulator

Problem M1.2.A

Write a macro for **LISTPUSH**, which pushes the node pointed to by the accumulator to the head of the list. **LISTPUSH** takes one argument, the memory address of the new node, which is available in the accumulator. As shown in the figure below, **LISTPUSH** stores the current `_HEAD` pointer in the new node's `next` field, and updates the `_HEAD` pointer to point to the new node. Implement the macro using the EDSACjr instruction set and macros provided above. Do not refer to "`value`" or "`next`"; they are for illustration only. You need not worry about memory allocation; the new node's address is provided in the accumulator.




```
.macro LISTPUSH
     STORE _TMP      ;; store accumulator (address of the new node)
```

```
.end
```

Problem M1.2.B

Write a macro for **LISTPOP**, which removes the node at the head of the list and stores its address in the accumulator, or stores `_INVALID` (-1) in the accumulator if the list is empty. Implement the macro using the EDSACjr instruction set and macros provided above.

```
.macro LISTPOP
     CLEAR           ;; accumulator is not an input









.end
```

Problem M1.2.C

Assume there exists a macro called **FREE** that takes an address as input in the accumulator and deallocates it (just like `free(void* ptr)` in C). Write a macro for **LISTCLEAR**, which uses the **FREE** macro and your **LISTPOP** macro to remove and deallocate all nodes in the list. Assume all valid node addresses are positive, or else a pointer is `_INVALID` (-1). Implement the macro using the EDSACjr instruction set and macros provided above.

```
.macro LISTCLEAR




.end
```

Problem M1.3: Self Modifying Code on the EDSACjr

This problem gives us a flavor of EDSAC-style programming and its limitations. Please read Handout #1 (EDSACjr) and Lecture 1 before answering the following questions (You may find local labels in Handout #1 useful for writing self-modifying code.)

Problem M1.3.A — Writing Macros For Indirection

With only absolute addressing instructions provided by the EDSACjr, writing self-modifying code becomes unavoidable for almost all non-trivial applications. It would be a disaster, for both you and us, if you put everything in a single program. As a starting point, therefore, you are expected to write *macros* using the EDSACjr instructions given in Table H1-1 (in Handout #1) to *emulate* indirect addressing instructions described in Table M1.1-1. Using macros may increase the total number of instructions that need to be executed because certain instruction level optimizations cannot be fully exploited. However, the code size *on paper* can be reduced dramatically when macros are appropriately used. This makes programming and debugging much easier.

Please use following global variables in your macros.

```
_orig_accum:    CLEAR          ; temp. storage for accum
_store_op:      STORE 0        ; STORE template
_bge_op:        BGE 0          ; BGE template
_blt_op:        BLT 0          ; BLT template
_add_op:        ADD 0          ; ADD template
```

These global variables are located somewhere in main memory and can be accessed using their labels. The `_orig_accum` location will be used to temporarily store the accumulator's value. The other locations will be used as "templates" for generating instructions.

Opcode	Description
ADDind *n*	Accum ← Accum + M[M[*n*]]
STOREind *n*	M[M[n]] ← Accum
BGEind *n*	If Accum ≥ 0 then PC ← M[n]
BLTind n	If Accum < 0 then PC ← M[n]

Table M1.1-1: Indirection Instructions

Problem M1.3.B **Subroutine Calling Conventions**

A possible subroutine calling convention for the EDSACjr is to place the arguments right after the subroutine call and pass the return address in the accumulator. The subroutine can then get its arguments by offset to the return address.

Describe how you would implement this calling convention for the special case of one argument and one return value using the EDSACjr instruction set. What do you need to do to the subroutine for your convention to work? What do you have to do around the calling point? How is your result returned? You may assume that your subroutines are in set places in memory and that subroutines cannot call other subroutines. You are allowed to use the original EDSACjr instruction set shown in Handout #1 (Table H1-1), as well as the indirection instructions listed in Table M1.1-1.

To illustrate your implementation of this convention, write a program for the EDSACjr to iteratively compute `fib(n),` where n is a non-negative integer. `fib(n)` returns the nth Fibonacci number (`fib(0)=0, fib(1)=1, fib(2)=1, fib(3)=2…`). Make `fib` a subroutine. (The C code is given below.) In few sentences, explain how could your convention be generalized for subroutines with an arbitrary number of arguments and return values?

The following program defines the iterative subroutine `fib` in C.

```
int fib(int n) {
  int i, x, y, z;
  x=0, y=1;
  if(n<2)
    return n;
  else{
    for(i=0; i<n-1; i++){
      z=x+y;
      x=y;
      y=z;
    }
    return z;
  }
}
```

Problem M1.3.C **Subroutine Calling Other Subroutines**

The following program defines a ***recursive*** version of the subroutine `fib` in C.

```
int fib_recursive (int n){
  if(n<2)
    return n;
  else{
    return(fib(n-1) + fib(n-2));
  }
}
```

In a few sentences, explain what happens if the subroutine calling convention you implemented in Problem M1.3.B is used for `fib_recursive`.

Problem M2.1: Execute Data Instructions (Spring 2014 Quiz 1, Part A)

One day, Ben Bitdiddle started an EDSACjr-based company. Ben wanted to leverage the speed of read-only memory and avoid the inherent hazards of the Princeton architecture, so he went with a Harvard architecture. Unfortunately, Ben's system didn't have any index registers, so he couldn't write self-modifying code. That meant there were a large number of programs he couldn't implement anymore. Ben decided to add an instruction to solve this problem. He called his new instruction `EXD` , for execute data. The `EXD` instruction treats the contents of the accumulator as a new instruction and executes whatever that instruction may be. If the accumulator does not contain a valid instruction, then `EXD` falls back on the processor's fault handling for bad instructions (which you needn't worry about).

For example, from Handout #1 the instruction `ADD 6` (which adds the contents of memory at address six to the accumulator) is encoded as: 0000 0100 0000 0110.

Therefore if the contents of the accumulator are 0000 0100 0000 0110, the `EXD` instruction will interpret the accumulator as an `ADD 6` instruction, and add the contents of memory at address six to the accumulator (now interpreted as the *number:* 0000 0100 0000 0110 = 1030). So if memory at address six holds the value one, then the accumulator will become 0000 0100 0000 0111. (Which can be interpreted either as the instruction `ADD 7` or the number 1031.)

To simplify writing assembly code, Ben Bitdiddle also augments the EDSACjr's instruction set with a load instruction, `LD n`. This load simply places the value in memory address n into the accumulator: `ACC` ← `Mem[n]`. LD is encoded as 01011 *n*; that is, the opcode is 01011.

Problem M2.1.A

When Ben shows his idea to Alyssa P. Hacker, she points out that `EXD` could cause an infinite loop. Provide a specific code sequence that illustrates Alyssa's point, using `EXD` to loop forever.

Problem M2.1.B

Ignoring Alyssa's observation, Ben decided to implement his `EXD` instruction for the EDSACjr, but he started having trouble figuring out how to use it. Help Ben by writing a series of EDSACjr instructions that will perform an indirect reduced add (that is, the instructions will take a vector of pointers, follow each pointer, and sum up the values stored at the locations in memory that the pointers specify). In C++, this might look something like:

```
int s=0;
for (int i=0; i < 10; i++){
    s += *A[i];
}
```

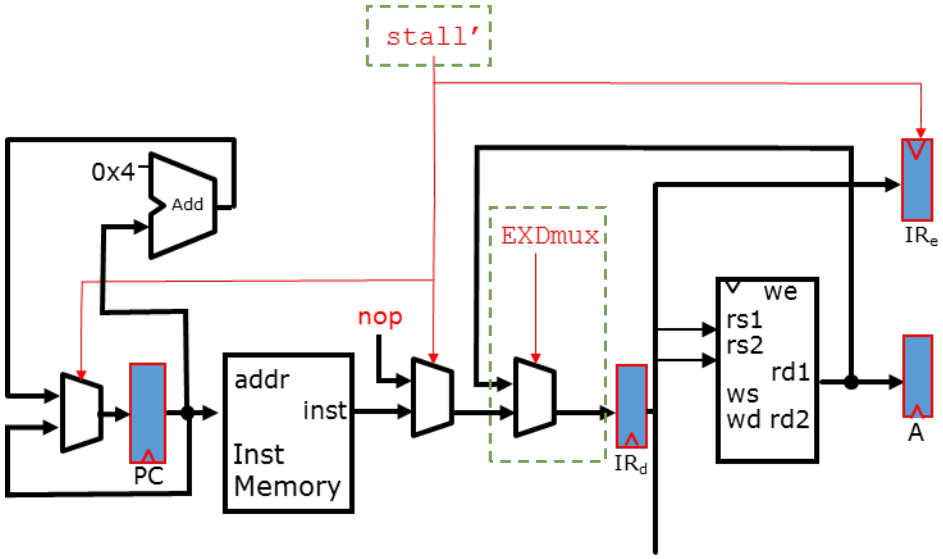
Fill in the template below with assembly code for this program on the Harvard EDSACjr. You can define memory contents for both the data and instruction memories.

```
Data Mem
Addr  Data
A:    120
      107
      122
      130
      151
      112
      132
      109
      140
      117

s:    0
i:    10

107:  40
109:  10
112:  24
117:  50
120:  5
122:  10
130:  20
132:  29
140:  22
151:  12

one:  1
```

```
Instr Mem
Addr  Data
Loop: LD i
      SUB one
      BLT Done
      STORE i




      CLEAR
      BGE Loop
Done: HLT
```

Problem M2.1.C

Unhappy with the performance, now Ben decided to implement a pipelined version of `EXD`. Ben realized that the `EXD` instruction was in and of itself a control hazard. Help Ben safeguard his pipeline. The diagram below shows the front end of the five-stage pipeline we used in class. A new datapath and mux have been added to move `rd1` into the instruction register of the decode stage.



Your task is to write the new stall signal (`stall'`) and fill in the missing signal, `EXDmux`. Write your signal in terms of signals (e.g., `PC` or `rd1` or IR_D) and feel free to use the old stall signal (`stall`).

`stall'` = _____

`EXDmux` = _____

Problem M2.2: CISC and RISC: Comparing ISAs

This problem requires the knowledge of Handout #2 (CISC ISA—x86jr), Handout #3 (RISC ISA—MIPS32), and Lectures 1 and 2. Please read these materials before answering the following questions.

Problem M2.2.A — CISC

Let us begin by considering the following C code.

```
int b;  //a global variable

void multiplyByB(int a){
  int i, result;
  for(i = 0; i<b; i++){
    result=result+a;
  }
}
```

Using gcc and objdump on a Pentium III, we see that the above loop compiles to the following x86 instruction sequence. (On entry to this code, register `%ecx` contains `i`, register `%edx` contains `result` and register `%eax` contains `a`. `b` is stored in memory at location `0x08047580`.) A brief explanation of each instruction in the code is given in Handout #2.

```
            xor    %edx,%edx
            xor    %ecx,%ecx
 loop:      cmp    0x08047580,%ecx
            jl     L1
            jmp    done
 L1:        add    %edx,%eax
            inc    %ecx
            jmp    loop
 done:    ...
```

How many bytes is the program? For the above x86 assembly code, how many bytes of instructions need to be fetched if b = 10? Assuming 32-bit data values, how many bytes of data memory need to be fetched? Stored?

Problem M2.2.B — RISC

Translate each of the x86 instructions in the following table into one or more MIPS32 instructions in Handout #3. Place the L1 and loop labels where appropriate. You should use the minimum number of instructions needed. Assume that upon entry R2 contains `a` and R3 contains `i`. R1 should

be loaded with the value of b from memory location 0x08047580, while R4 should receive `result`. If needed, use R5 to hold the condition value and R6, R7, etc., for temporaries. You should not need to use any floating point registers or instructions in your code.

x86 instruction	**label**	**MIPS32 instruction sequence**
`xor %edx,%edx`		
`xor %ecx,%ecx`		
`cmp 0x08049580,%ecx`		
`jl L1`		
`jmp done`		
`add %eax,%edx`		
`inc %ecx`		
`jmp loop`		
`...`	*done:*	`...`

How many bytes is the MIPS32 program using your direct translation? How many bytes of MIPS32 instructions need to be fetched for b = 10 using your direct translation? How many bytes of data memory need to be fetched? Stored?

Problem M2.2.C — Optimization

To get more practice with MIPS32, optimize the code from part **B** so that it can be expressed in fewer instructions. Your solution should contain commented assembly code, a paragraph which explains your optimizations and a short analysis of the savings you obtained.

Problem M2.3: Addressing Modes on MIPS ISA

Ben Bitdiddle is suspicious of the benefits of complex addressing modes. So he has decided to investigate it by incrementally removing the addressing modes from our MIPS ISA. Then he will write programs on the "crippled" MIPS ISAs to see what the programming on these ISAs is like.

Problem M2.3.A — Displacement addressing mode

As a first step, Ben has discontinued supporting the displacement (base+offset) addressing mode, that is, our MIPS ISA only supports register indirect addressing (without the offset).

Can you still write the same program as before? If so, please translate the following load instruction into an instruction sequence in the new ISA. If not, explain why.

`LW R1, 16(R2)` ➔

Problem M2.3.B — Register indirect addressing

Now he wants to take a bolder step by completely eliminating the register indirect addressing. The new load and store instructions will have the following format.

```
LW R1, imm16    ; R1 <- M[imm16]
SW R1, imm16    ; M[imm16] <- R1
```

6	5	5	16
Opcode	**Rs**		**Offset**

Can you still write the same program as before? If so, please translate the following load instruction into an instruction sequence in the new ISA. If not, explain why. (Don't worry about branches and jumps for this question.)

`LW R1, 16(R2)` ➔

Problem M2.3.C **Subroutine**

Ben is wondering whether we can implement a subroutine using only absolute addressing. He changes the original ISA such that all the branches and jumps take a 16-bit absolute address (the 2 lower orders bits are 0 for word accesses), and that `jr` and `jalr` are not supported any longer.

With the new ISA he decides to rewrite a piece of subroutine code from his old project. Here is the original C code he has written.

```
int b;  //a global variable

void multiplyByB(int a){
  int i, result;
  for(i=0; i<b; i++){
    result=result+a;
  }
}
```

The C code above is translated into the following instruction sequence on our original MIPS ISA. Assume that upon entry, R1 and R2 contain `b` and `a`, respectively. R3 is used for `i` and R4 for `result`. By a calling convention, the 16-bit word-aligned return address is passed in R31.

```
Subroutine: xor  R4, R4, R4   ; result = 0
            xor  R3, R3, R3   ; i = 0
loop:       slt  R5, R3, R1
            bnez R5, L1       ; if (i < b) goto L1
return:     jr   R31          ; return to the caller
L1:         add  R4, R4, R2   ; result += a
            addi R3, R3, #1   ; i++
            j    loop
```

If you can, please rewrite the assembly code so that the subroutine returns without using a `jr` instruction (which is a register indirect jump). If you cannot, explain why.

Problem M2.4: Write Effective Address Extensions (Spring 2014 Quiz 1, Part B)

You've noticed that many programs execute code similar to the following during loops:

```
LD  R1, 4(R2)
ADD R2, R2, 4
```

Or:

```
ST  R1, 4(R2)
ADD R2, R2, 4
```

You want to optimize your architecture for this common case. You are going to do so by adding "write effective address" variants of the load and store instructions, `LDWA` and `STWA`. The semantics of these instructions are that they will perform the normal memory operation (`LD` or `ST`) and then write the effective address in the register that indexed into memory (*not* the register whose contents are read/written to memory). Specifically these instructions do the following:

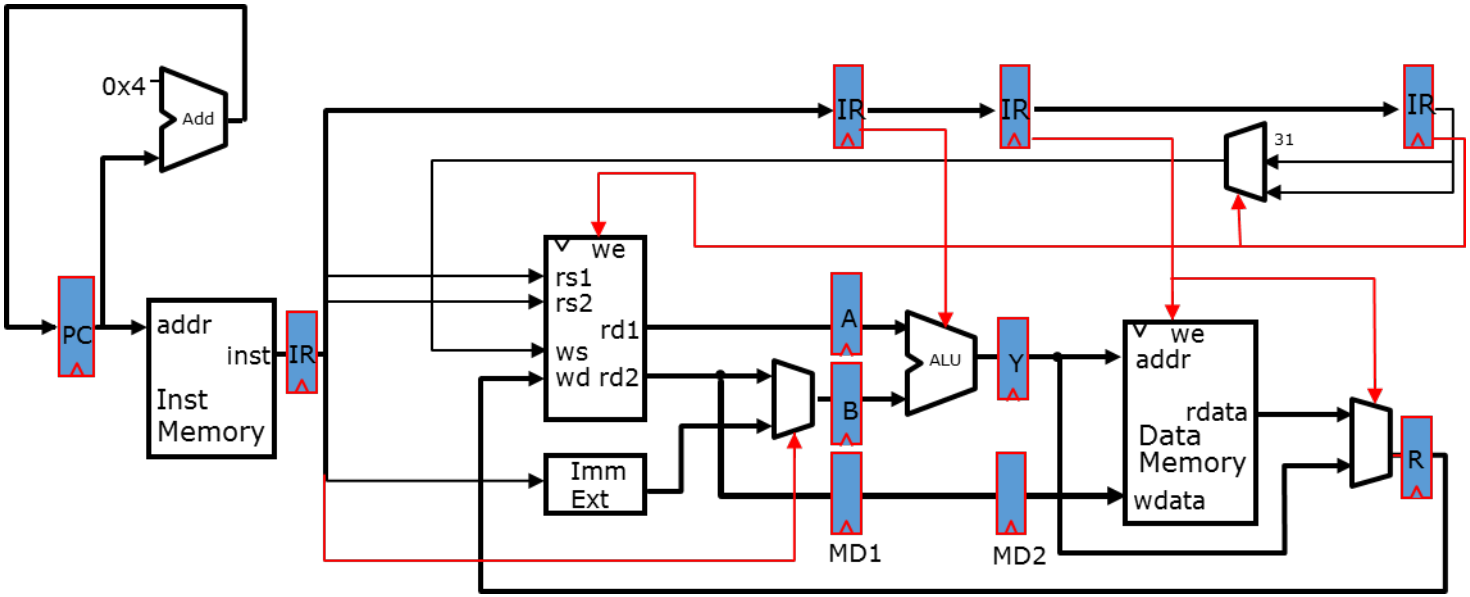
```
LDWA rs, rt, Imm:
     rs ← Memory[(rt) + Imm]
     rt ← (rt) + Imm

STWA rs, rt, Imm:
     Memory[(rt) + Imm] ← (rs)
     rt ← (rt) + Imm
```

These extensions allow us to rewrite the previous examples as:

```
LDWA R1, R2, 4
```

And:

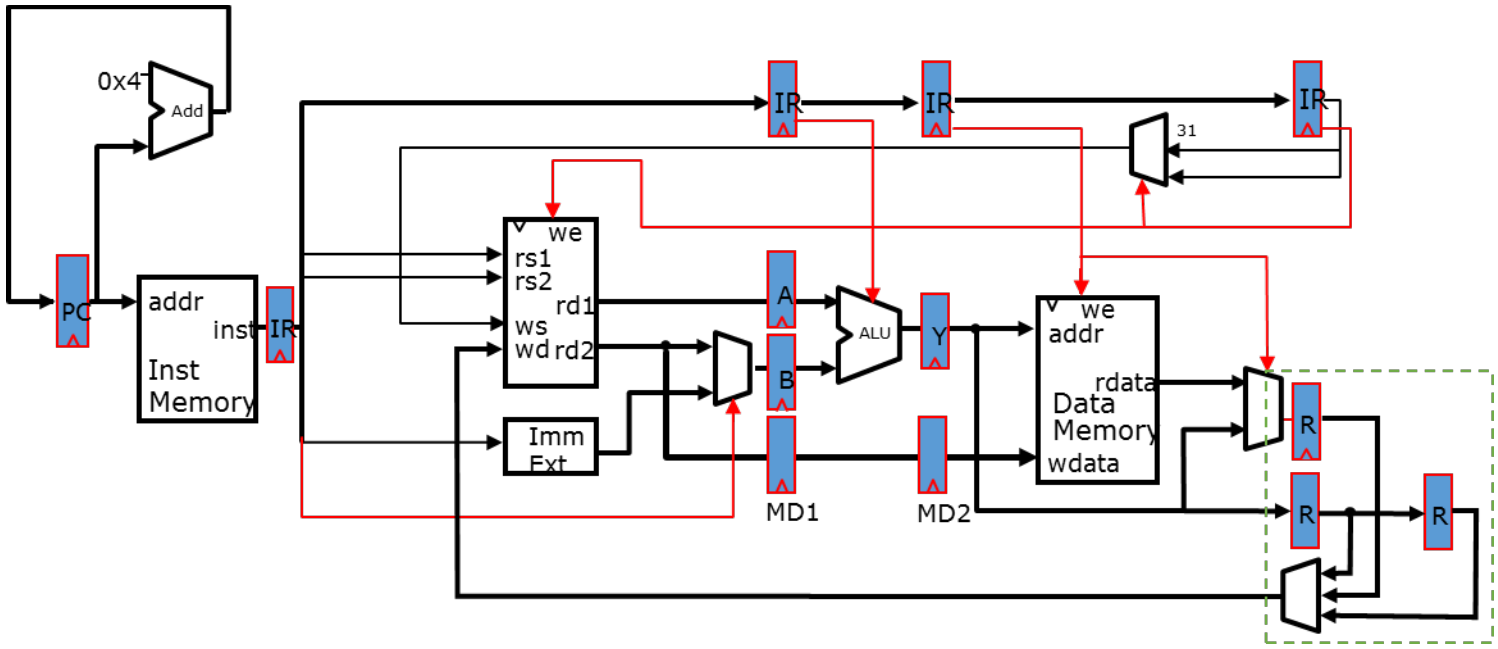
```
STWA R1, R2, 4
```



Problem M2.4.A

You start with implementing `STWA`. For the following sequence of instructions and the standard five-stage pipeline (shown above), indicate how each instruction will flow through the pipeline on the following page. Assume full bypassing and stall logic are implemented for your architecture. Use arrows to indicate forwarding and dashes for stalls, as illustrated.

Instruction	0	1	2	3	4	5	6	7	8	9	10	11	12
`LD R1, 0(R2)`	F	D	E	M	W								
`ADD R3, R1, 10`		F	D	-	E	M	W						
`LD R4, 0(R3)`													
`STWA, R4, R1, 4`													
`STWA R4, R1, 4`													
`ADD R2, R1, R3`													



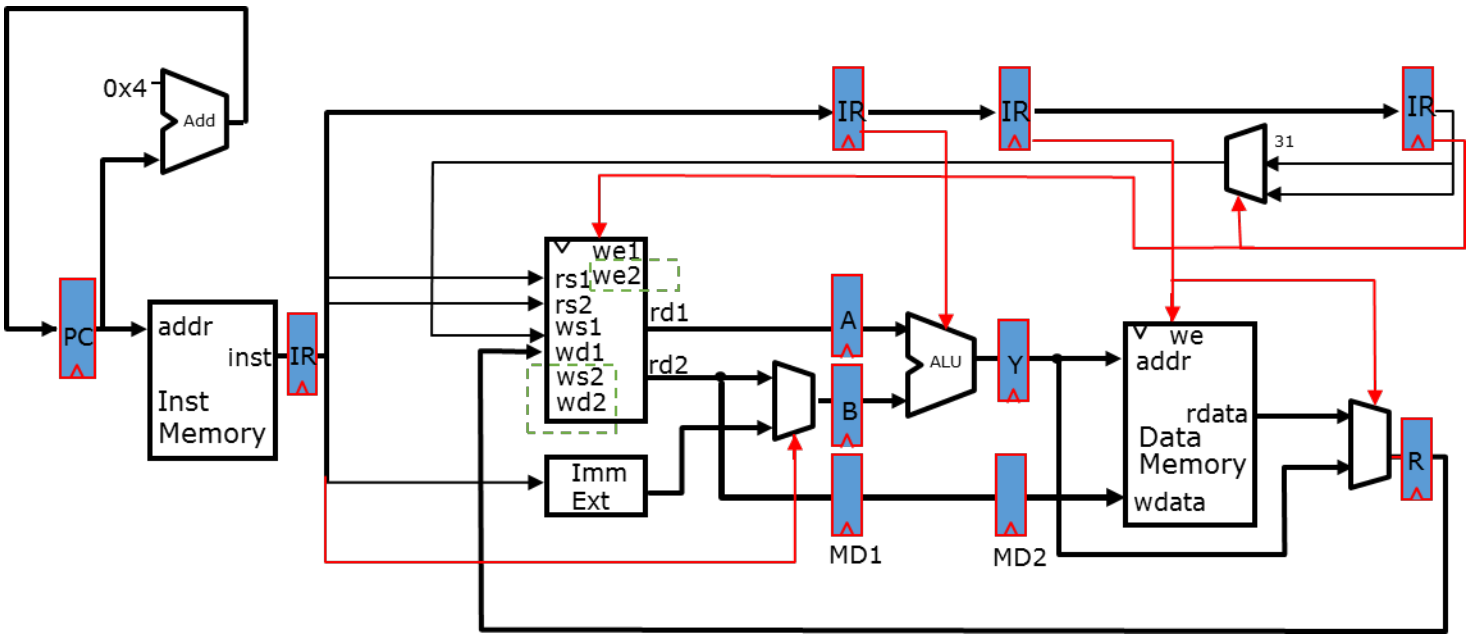
Problem M2.4.B

You next want to implement `LDWA`, and quickly realize that `LDWA` runs into a structural hazard on the register file. You decide to fix this by adding an extra writeback stage (W2) to your pipelined design as shown above. In one or two sentences, explain what the hazard is and why the additional stage fixes it (assume correct stall logic).

Problem M2.4.C

Assume that the six-stage design above has full bypassing and correct stall logic. Fill in the pipeline for the instructions given below, using arrows and dashes as before.

Instruction	0	1	2	3	4	5	6	7	8	9	10	11	12
`LD R1, 0(R2)`	F	D	E	M	W1	W2							
`ADD R3, R1, 10`		F	D	-	E	M	W1	W2					
`LDWA R5, R3, 0`													
`ADD R1, R3, R4`													
`LDWA R5, R3, 0`													
`ADD R1, R5, R0`													



Problem M2.4.D

Adding a second writeback stage is only one way to fix this structural hazard. An alternative design is to add a second write port to the register file. Quickly sketch the datapath for this design in the diagram above. You do *not* need to write the stall logic. (Additional signals are: `we2`, `ws2`, `wd2`.)

Problem M2.4.E

In one or two sentences, explain the tradeoffs between adding an additional pipeline stage vs. adding a write port to the register file. What conditions might favor one or the other design?

Problem M3.1: Cache Access-Time & Performance

This problem requires the knowledge of Handout 4 (Cache Implementations) and Lecture 3 (Caches). Please, read these materials before answering the following questions.

Ben is trying to determine the best cache configuration for a new processor. He knows how to build two kinds of caches: direct-mapped caches and 4-way set-associative caches. The goal is to find the better cache configuration with the given building blocks. He wants to know how these two different configurations affect the clock speed and the cache miss-rate, and choose the one that provides better performance in terms of average latency for a load.

Problem M3.1.A — Access Time: Direct-Mapped

Now we want to compute the access time of a direct-mapped cache. We use the implementation shown in Figure H4-A in Handout #4. Assume a 128-KB cache with 8-word (32-byte) cache lines. The address is 32 bits, and the two least significant bits of the address are ignored since a cache access is word-aligned. The data output is also 32 bits, and the MUX selects one word out of the eight words in a cache line. Using the delay equations given in Table M3.1-1, fill in the column for the direct-mapped (DM) cache in the table. *In the equation for the data output driver, 'associativity' refers to the associativity of the cache (1 for direct-mapped caches, A for A-way set-associative caches).*

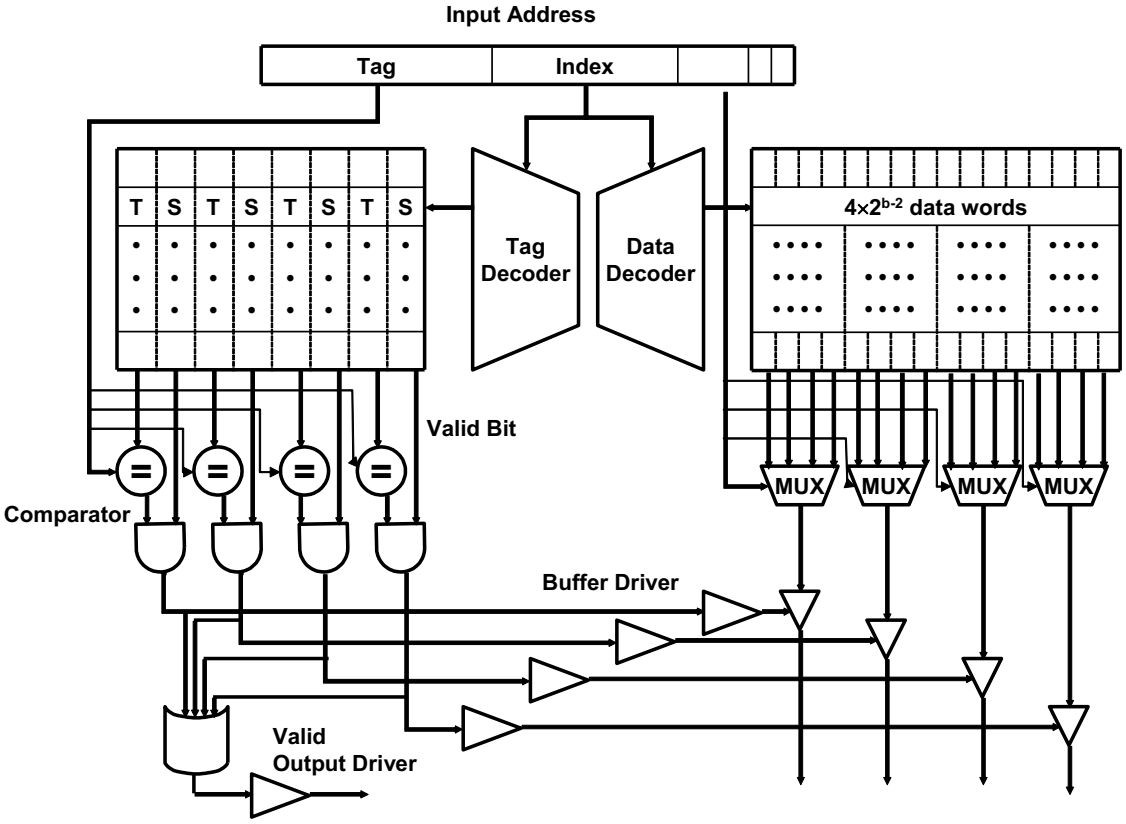
Component	Delay equation (ps)		DM (ps)	SA (ps)
Decoder	200×(# of index bits) + 1000	Tag		
		Data		
Memory array	200×$\log_2$ (# of rows) + 200×$\log_2$ (# of bits in a row) + 1000	Tag		
		Data		
Comparator	200×(# of tag bits) + 1000			
N-to-1 MUX	500×$\log_2$ N + 1000			
Buffer driver	2000			
Data output driver	500×(associativity) + 1000			
Valid output driver	1000			

Table M3.1-1: Delay of each Cache Component

What is the critical path of this direct-mapped cache for a cache read? What is the access time of the cache (the delay of the critical path)? To compute the access time, assume that a 2-input gate (AND, OR) delay is 500 ps. If the CPU clock is 150 MHz, how many CPU cycles does a cache access take?

We also want to investigate the access time of a set-associative cache using the 4-way set-associative cache in Figure H4-B in Handout #4. Assume the total cache size is still 128-KB (each way is 32-KB), a 4-input gate delay is 1000 ps, and all other parameters (such as the input address, cache line, etc.) are the same as part M3.1.A. Compute the delay of each component, and fill in the column for a 4-way set-associative cache in Table M3.1-1.

What is the critical path of the 4-way set-associative cache? What is the access time of the cache (the delay of the critical path)? What is the main reason that the 4-way set-associative cache is slower than the direct-mapped cache? If the CPU clock is 150 MHz, how many CPU cycles does a cache access take?



Problem M3.1.C **Miss-rate analysis**

Now Ben is studying the effect of set-associativity on the cache performance. Since he now knows the access time of each configuration, he wants to know the miss-rate of each one. For the miss-rate analysis, Ben is considering two small caches: a direct-mapped cache with 8 lines with 16 bytes/line, and a 4-way set-associative cache of the same size. For the set-associative cache, Ben tries out two replacement policies – least recently used (LRU) and round robin (FIFO).

Ben tests the cache by accessing the following sequence of hexadecimal byte addresses, starting with empty caches. For simplicity, assume that the addresses are only 12 bits. Complete the following tables for the direct-mapped cache and both types of 4-way set-associative caches showing the progression of cache contents as accesses occur (in the tables, 'inv' = invalid, and the column of a particular cache line contains the {tag,index} contents of that line). *You only need to fill in elements in the table when a value changes.*

D-map									
	line in cache								hit?
Address	L0	L1	L2	L3	L4	L5	L6	L7	
110	inv	11	inv	inv	inv	inv	inv	inv	no
136				13					no
202	20								no
1A3									
102									
361									
204									
114									
1A4									
177									
301									
206									
135									

	D-map
Total Misses	
Total Accesses	

4-way									LRU
Address	line in cache								hit?
	Set 0				Set 1				
	way0	**way1**	**Way2**	**way3**	**way0**	**way1**	**way2**	**way3**	
110	inv	Inv	Inv	inv	11	inv	inv	inv	no
136					11	13			no
202	20								no
1A3									
102									
361									
204									
114									
1A4									
177									
301									
206									
135									

	4-way LRU
Total Misses	
Total Accesses	

4-way									FIFO
Address	line in cache								hit?
	Set 0				Set 1				
	way0	**way1**	**way2**	**way3**	**way0**	**way1**	**way2**	**way3**	
110	inv	Inv	Inv	inv	11	inv	inv	inv	no
136						13			no
202	20								no
1A3									
102									
361									
204									
114									
1A4									
177									
301									
206									
135									

	4-way FIFO
Total Misses	
Total Accesses	

Problem M3.1.D **Average Latency**

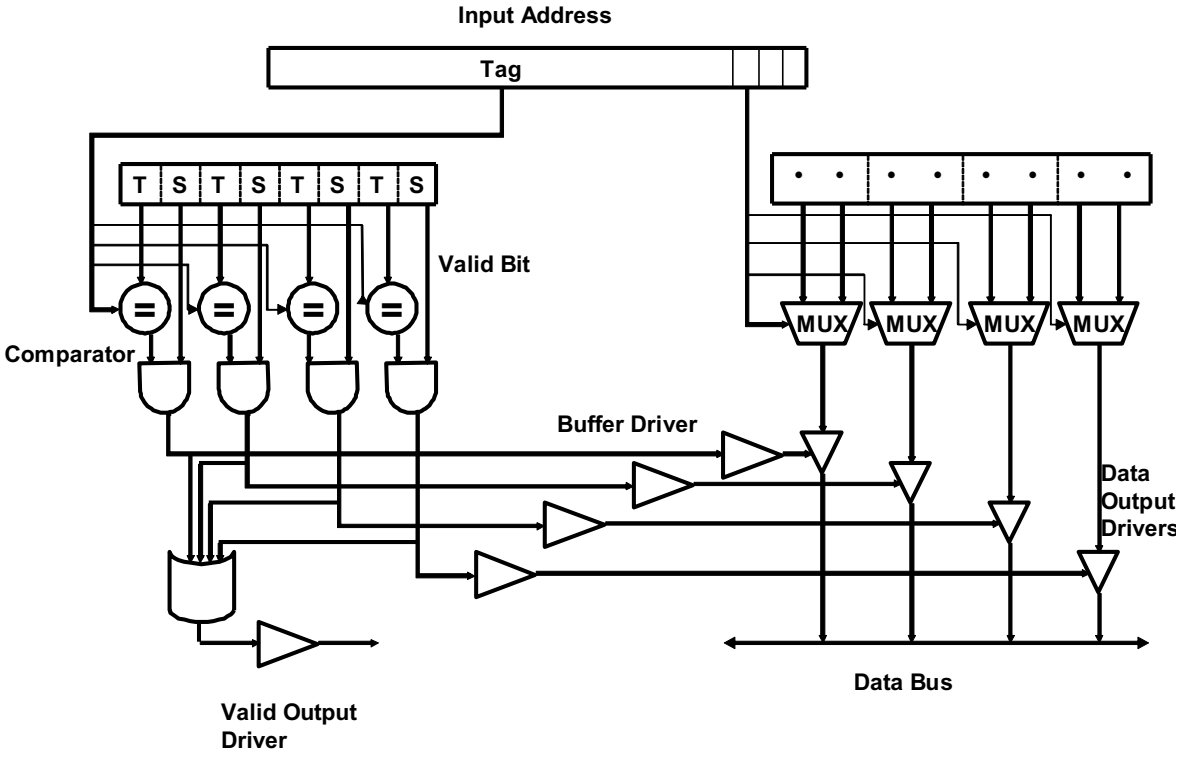
Assume that the results of the above analysis can represent the average miss-rates of the direct-mapped and the 4-way LRU 128-KB caches studied in M3.1.A and M3.1.B. What would be the average memory access latency in CPU cycles for each cache (assume that a cache miss takes 20 cycles)? Which one is better? For the different replacement policies for the set-associative cache, which one has a smaller cache miss rate for the address stream in M3.1.C? Explain why. Is that replacement policy always going to yield better miss rates? If not, give a counter example using an address stream.

Problem M3.2: Victim Cache Evaluation

This problem requires the knowledge of Handout #5 (Victim Cache) and Lecture 3. Please, read these materials before answering the following questions.

Problem M3.2.A — Baseline Cache Design

The diagram below shows a 32-Byte fully associative cache with four 8-Byte cache lines. Each line consists of two 4-Byte words and has an associated tag and two status bits (valid and dirty). The Input Address is 32-bits and the two least significant bits are assumed to be zero. The output of the cache is a 32-bit word.



Please complete Table M3.2-1 below with delays across each element of the cache. Using the data you compute in Table M3.2-1, calculate the critical path delay through this cache (from when the Input Address is set to when both Valid Output Driver and the appropriate Data Output Driver are outputting valid data).

Component	Delay equation (ps)		FA (ps)
Comparator	200×(# of tag bits) + 1000		
N-to-1 MUX	$500 \times \log_2 N + 1000$		
Buffer driver	2000		
AND gate	1000		
OR gate	500		
Data output driver	500×(associativity) + 1000		
Valid output driver	1000		

Table M3.2-1

Critical Path Cache Delay: _____

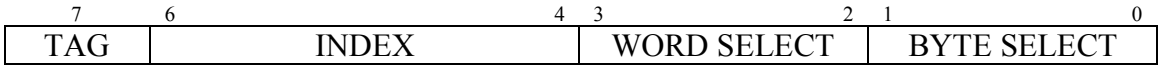
Problem M3.2.B — Victim Cache Behavior

Now we will study the impact of a victim cache on a cache hit rate. Our main L1 cache is a 128 byte, direct mapped cache with 16 bytes per cache line. The cache is word (4-bytes) addressable. The victim cache in Figure H5-A (in Handout #5) is a 32 byte fully associative cache with 16 bytes per cache line, and is also word-addressable. The victim cache uses the first in first out (FIFO) replacement policy.

Please complete Table M3.2-2 on the next page showing a trace of memory accesses. In the table, each entry contains the {tag,index} contents of that line, or "inv", if no data is present. You should only fill in elements in the table when a value changes. For simplicity, the addresses are only 8 bits.

The first 3 lines of the table have been filled in for you.

For your convenience, the address breakdown for access to the main cache is depicted below.



Problem M3.2.C — Average Memory Access Time

Assume **15%** of memory accesses are resolved in the victim cache. If retrieving data from the victim cache takes **5 cycles** and retrieving data from main memory takes **55 cycles**, by how many cycles does the victim cache improve the average memory access time?

Input Address	Main Cache									Victim Cache		
	L0	L1	L2	L3	L4	L5	L6	L7	Hit?	Way0	Way1	Hit?
	inv	inv	inv	inv	inv	inv	inv	inv	-	inv	inv	-
00	0								N			N
80	8								N	0		N
04	0								N	8		Y
A0												
10												
C0												
18												
20												
8C												
28												
AC												
38												
C4												
3C												
48												
0C												
24												

Table M3.2-2

Problem M3.3: Loop Ordering

This problem requires the knowledge of Lecture 3. Please, read it before answering the following questions.

This problem evaluates the cache performances for different loop orderings. You are asked to consider the following two loops, written in C, which calculate the sum of the entries in a 128 by 64 matrix of 32-bit integers:

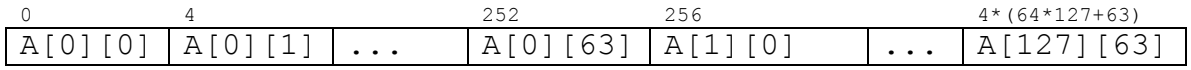
Loop A	***Loop B***

`sum = 0;`
`for (i = 0; i < 128; i++)`
`  for (j = 0; j < 64; j++)`
`  sum += A[i][j];`
	`sum = 0;`
	`for (j = 0; j < 64; j++)`
	`  for (i = 0; i < 128; i++)`
	`  sum += A[i][j];`

The matrix `A` is stored contiguously in memory in row-major order. Row major order means that elements in the same row of the matrix are adjacent in memory as shown in the following memory layout:

`A[i][j]` resides in memory location `[4*(64*i + j)]`

Memory Location:



For *Problem M3.3.A* to *Problem M3.3.C*, assume that the caches are initially empty. Also, assume that only accesses to matrix `A` cause memory references and all other necessary variables are stored in registers. Instructions are in a separate instruction cache.

Problem M3.3.A

Consider a 4KB direct-mapped data cache with 8-word (32-byte) cache lines.
Calculate the number of cache misses that will occur when running Loop A.
Calculate the number of cache misses that will occur when running Loop B.

The number of cache misses for Loop A:_____

The number of cache misses for Loop B:_____

Problem M3.3.B

Consider a direct-mapped data cache with 8-word (32-byte) cache lines. Calculate the minimum number of cache lines required for the data cache if Loop A is to run without any cache misses other than compulsory misses. Calculate the minimum number of cache lines required for the data cache if Loop B is to run without any cache misses other than compulsory misses.

Data-cache size required for Loop A: _____ cache line(s)

Data-cache size required for Loop B: _____ cache line(s)

Problem M3.3.C

Consider a 4KB fully-associative data cache with 8-word (32-byte) cache lines. This data cache uses a first-in/first-out (FIFO) replacement policy.
Calculate the number of cache misses that will occur when running Loop A.
Calculate the number of cache misses that will occur when running Loop B.

The number of cache misses for Loop A:_____

The number of cache misses for Loop B:_____

Problem M3.4: Cache Parameters

For each of the following statements about making a change to a cache design, circle **True** or **False** and provide a one sentence explanation of your choice. Assume all cache parameters (capacity, associativity, line size) remain fixed except for the single change described in each question. Please provide a one sentence explanation of your answer.

Problem M3.4.A

Doubling the line size halves the number of tags in the cache

True / False

Problem M3.4.B

Doubling the associativity doubles the number of tags in the cache.

True / False

Problem M3.4.C

Doubling cache capacity of a direct-mapped cache usually reduces conflict misses.

True / False

Problem M3.4.D

Doubling cache capacity of a direct-mapped cache usually reduces compulsory misses.

True / False

Problem M3.4.E

Doubling the line size usually reduces compulsory misses.

True / False

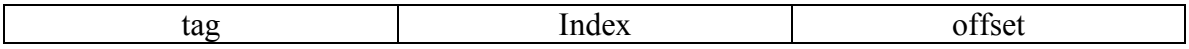
Problem M3.5: Microtags

Problem M3.5.A

Explain in one or two sentences why direct-mapped caches have much lower hit latency (as measured in picoseconds) than set-associative caches of the same capacity.

Problem M3.5.B

A 32-bit byte-addressed machine has an 8KB, 4-way set-associative data cache with 32-byte lines. The following figure shows how the address is divided into tag, index and offset fields. Give the number of bits in each field.



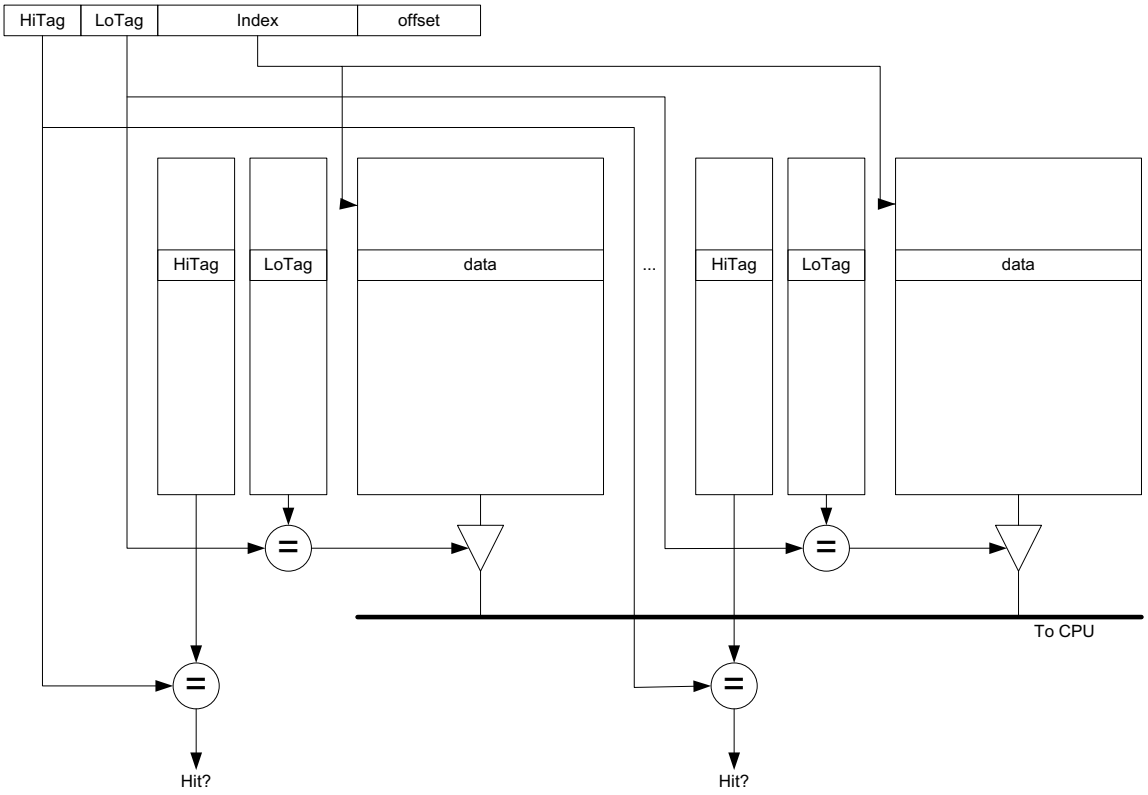
# of bits in the tag: _____

# of bits in the index: _____

# of bits in the offset: _____

Microtags (for questions M3.5.C – M3.5.H)

Several commercial processors (including the UltraSPARC-III and the Pentium-4) reduce the hit latency of a set-associative cache by using only a subset of the tag bits (a "microtag") to select the matching way before speculatively forwarding data to the CPU. The remaining tag bits are checked in a subsequent clock cycle to determine if the access was actually a hit. The figure below illustrates the structure of a cache using this scheme.



Problem M3.5.C

The tag field is sub-divided into a **loTag** field used to select a way and a **hiTag** field used for subsequent hit/miss checks, as shown below.

	tag		
hiTag	**loTag**	index	offset

The cache design requires that all lines within a set have unique loTag fields.
In one or two sentences, explain why this is necessary.

Problem M3.5.D

If the **loTag** field is exactly two bits long, will the cache have greater, fewer, or an equal number of conflict misses as a direct-mapped cache of the same capacity? State any assumptions made about replacement policy.

Problem M3.5.E

If the **loTag** field is greater than two bits long, are there any additional constraints on replacement policy beyond those in a conventional 4-way set-associative cache?

Problem M3.5.F

Does this scheme reduce the time required to complete a write to the cache? Explain in one or two sentences.

Problem M3.5.G

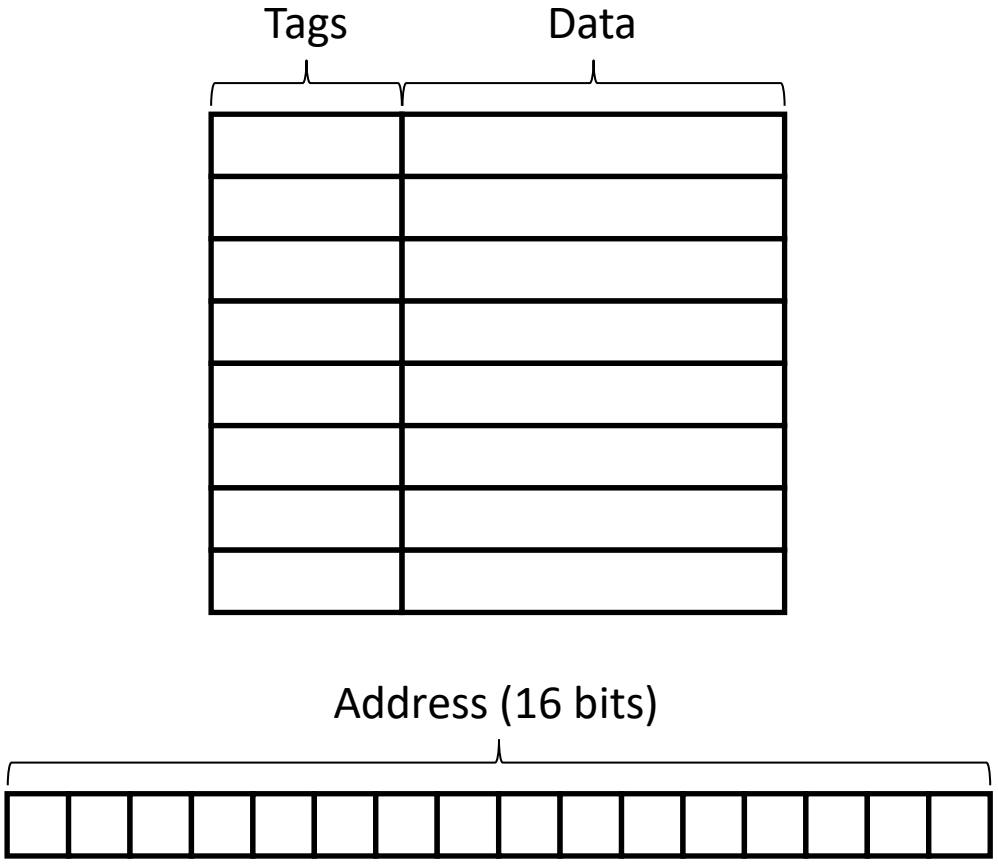
In practice, microtags hold virtual address bits to remove address translation from the critical path, while the full tag check is performed on translated physical addresses. If the **loTag** bits can only hold untranslated bits of the virtual address, what is the largest number of **loTag** bits possible if the machine has a 16KB virtual memory page size? (Assume 8KB 4-way set-associative cache as in Question M3.5.B)

Problem M3.5.H

Describe how microtags can be made much larger, to also include virtual address bits subject to address translation. Your design should not require address translation before speculatively forwarding data to the CPU. Your explanation should describe the replacement policy and any additional state the machine must maintain.

Problem M3.6: Caches (Spring 2014 Quiz 1, Part C)

Your processor has an 8-line level 1 data cache as illustrated below. Suppose that cache lines are 32 bytes (256 bits) and memory addresses are 16 bits, with byte-addressable memory. The cache is indexed by low bits without hashing.



Problem M3.6.A

We're first going to fill in the above diagram with more detail.

Divide the bits of the address according to how they are used to access the cache (tag, index, offset).

What exactly is contained in the cache tags? (Include all bits necessary for correct operation of the cache as discussed in lecture.)

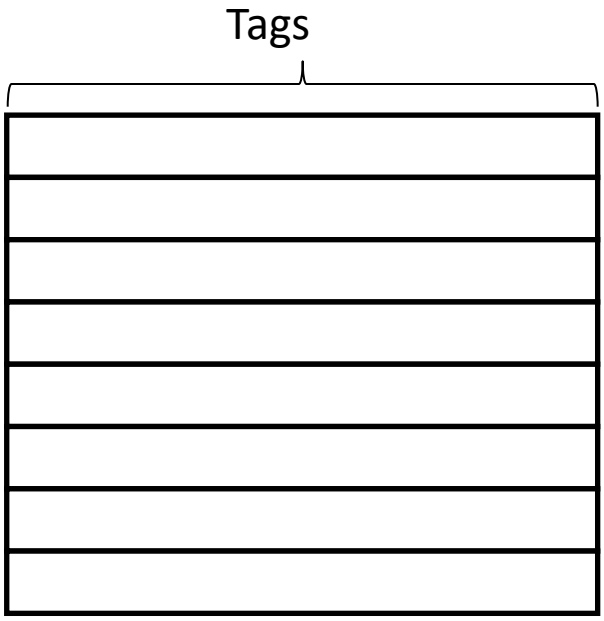
How many bits <u>in total</u> are needed to implement the level 1 data cache?

Problem M3.6.B

Suppose the processor accesses the following data addresses starting with an empty cache:

```
0x0028: 0000 0000 0010 1000
0x102A: 0001 0000 0010 1010
0x9435: 1001 0100 0011 0101
0xEFF4: 1110 1111 1111 0100
0xBEEF: 1011 1110 1110 1111
0x4359: 0100 0011 0101 1001
0x01DE: 0000 0001 1101 1110
0x8075: 1000 0000 0111 0101
0x9427: 1001 0100 0010 0111
```

What would the level 1 data cache tags look like after this sequence? How many hits would there be in the level 1 data cache? *(Don't worry about filling in the Data column – we didn't give you the data!)*



Problem M3.6.C

Suppose that the level 1 data cache has a hit rate of 40% on your application, an access time of a single cycle, and a miss penalty to memory of forty cycles. What is the average memory access time?

You aren't happy with your memory performance, so you decide to add a level two cache. Suppose the level two cache has a hit rate of 50%. What access time must the level two cache have for this to be a good design (ie, reduce AMAT)?

Problem M4.1: Virtual Memory Bits

This problem requires the knowledge of Handout #6 (Virtual Memory Implementation) and Lecture 4 and 5. Please, read these materials before answering the following questions.

In this problem we consider simple virtual memory enhancements.

Problem M4.1.A

Whenever a TLB entry is replaced we write the entire entry back to the page table. Ben thinks this is a waste of memory bandwidth. He thinks only a few of the bits need to be written back. For each of the bits explain why or why not they need to be written back to the page table.

With this in mind, we will see how we can minimize the number of bits we actually need in each TLB entry throughout the rest of the problem.

Problem M4.1.B

Ben does not like the TLB design. He thinks the TLB Entry Valid bit should be dropped and the kernel software should be changed to ensure that all TLB entries are always valid. Is this a good idea? Explain the advantages and disadvantages of such a design.

Problem M4.1.C

Alyssa got wind of Ben's idea and suggests a different scheme to eliminate one of the valid bits. She thinks the page table entry valid and TLB Entry Valid bits can be combined into a single bit.

On a refill this combined valid bit will take the value that the page table entry valid bit had. A TLB entry is invalidated by writing it back to the page table and setting the combined valid bit in the TLB entry to invalid.

How does the kernel software need to change to make such a scheme work? How do the exceptions that the TLB produces change?

Problem M4.1.D

Now, Bud Jet jumps into the game. He wants to keep the TLB Entry Valid bit. However, there is no way he is going to have two valid bits in each TLB entry (one for the TLB entry one for the page table entry). Thus, he decides to drop the page table entry valid bit from the TLB entry.

How does the kernel software need to change to make this work well? How do the exceptions that the TLB produces change?

Problem M4.1.E

Compare your answers to Problem M4.1.C and M4.1.D. What scheme will lead to better performance?

Problem M4.1.F

How about the R bit? Can we remove them from the TLB entry without significantly impacting performance? Explain briefly.

Problem M4.1.G

The processor has a kernel (supervisor) mode bit. Whenever kernel software executes the bit is set. When user code executes the bit is not set. Parts of the user's virtual address space are only accessible to the kernel. The supervisor bit in the page table is used to protect this region—an exception is raised if the user tries to access a page that has the supervisor bit set.

Bud Jet is on a roll and he decides to eliminate the supervisor bit from each TLB entry. Explain how the kernel software needs to change so that we still have the protection mechanism and the kernel can still access these pages through the virtual memory system.

Problem M4.1.H

Alyssa P. Hacker thinks Ben and Bud are being a little picky about these bits, but has devised a scheme where the TLB entry does not need the M bit or the U bit. It works as follows. If a TLB miss occurs due to a load, then the page table entry is read from memory and placed in the TLB. However, in this case the W bit will always be set to 0. Provide the details of how the rest of the scheme works (what happens during a store, when do the entries need to be written back to memory, when are the U and M bits modified in the page table, etc.).

Problem M4.2: Page Size and TLBs (2005 Fall Part D)

This problem requires the knowledge of Handout #6 (Virtual Memory Implementation) and Lecture 5. Please, read these materials before answering the following questions.

Assume that we use a hierarchical page table described in Handout #6.

The processor has a data TLB with 64 entries, and each entry can map either a 4KB page or a 4MB page. After a TLB miss, a hardware engine walks the page table to reload the TLB. The TLB uses a first-in/first-out (FIFO) replacement policy.

We will evaluate the memory usage and execution of the following program which adds the elements from two 1MB arrays and stores the results in a third 1MB array (note that, 1MB = 1,048,576 Bytes):

```
byte A[1048576]; // 1MB array
byte B[1048576]; // 1MB array
byte C[1048576]; // 1MB array

for(int i=0; i<1048576; i++)
  C[i] = A[i] + B[i];
```

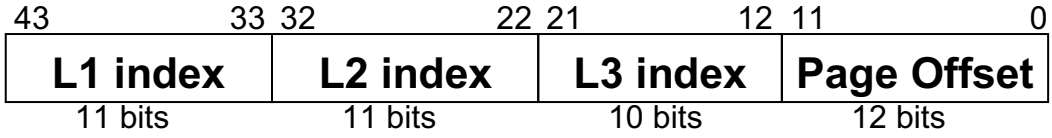
We assume the A, B, and C arrays are allocated in a contiguous 3MB region of physical memory. **We will consider two possible virtual memory mappings**:

- **4KB**: the arrays are mapped using 768 4KB pages (each array uses 256 pages).
- **4MB**: the arrays are mapped using a single 4MB page.

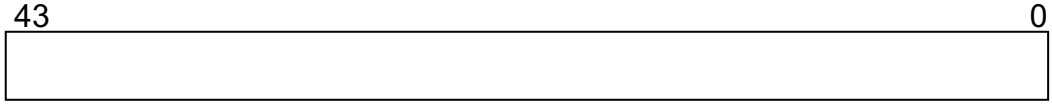
For the following questions, assume that the above program is the only process in the system, and ignore any instruction memory or operating system overheads. Assume that the arrays are aligned in memory to minimize the number of page table entries needed.

Problem M4.2.A

This is the breakdown of a virtual address which maps to a 4KB page:



Show the corresponding breakdown of a virtual address which maps to a 4MB page. Include the field names and bit ranges in your answer.



Problem M4.2.B **Page Table Overhead**

We define page table overhead (PTO) as:

$$\text{PTO} = \frac{\text{Physical memory that is allocated to page tables}}{\text{Physical memory that is allocated to data pages}}$$

For the given program, what is the PTO for each of the two mappings?

$$\text{PTO}_{4KB} = \frac{\quad}{\quad}$$

$$\text{PTO}_{4MB} = \frac{\quad}{\quad}$$

Problem M4.2.C **Page Fragmentation Overhead**

We define page fragmentation overhead (PFO) as:

$$\textbf{PFO} = \frac{\text{Physical memory that is allocated to data pages but is never accessed}}{\text{Physical memory that is allocated to data pages and is accessed}}$$

For the given program, what is the PFO for each of the two mappings?

PFO_{4KB} =	

PFO_{4MB} =	

Problem M4.2.D

Consider the execution of the given program, assuming that the data TLB is initially empty. For each of the two mappings, how many TLB misses occur, and how many page table memory references are required per miss to reload the TLB?

	Data TLB misses	Page table memory references (per miss)
4KB:		
4MB:		

Problem M4.2.E

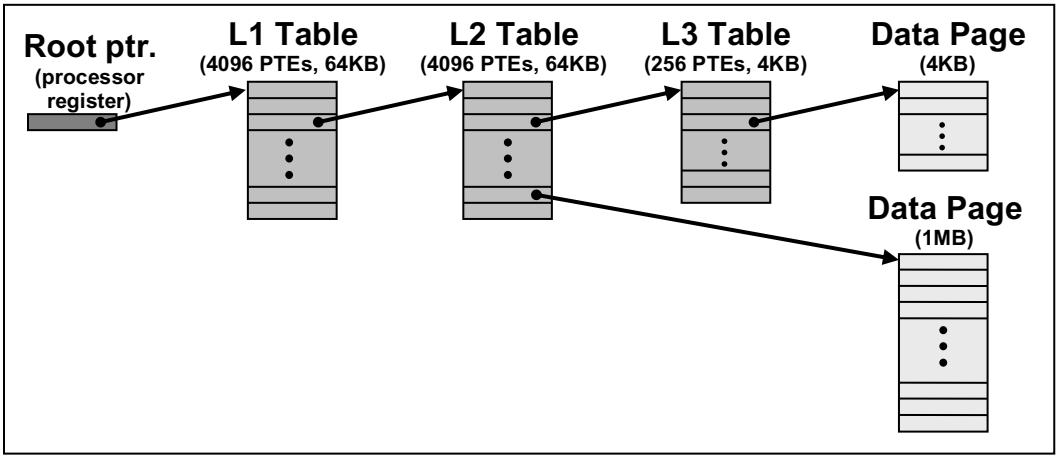
Which of the following is the best estimate for how much longer the program takes to execute with the 4KB page mapping compared to the 4MB page mapping?
Circle one choice and **briefly explain** your answer (about one sentence).

1.01×	**10×**	**1,000×**	**1,000,000×**

Problem M4.3: Page Size and TLBs

This problem requires the knowledge of Handout #6 (Virtual Memory Implementation) and Lecture 5. Please, read these materials before answering the following questions.

The configuration of the hierarchical page table in this problem is similar to the one in Handout #6, but we modify two parameters: 1) this problem evaluates a virtual memory system with two page sizes, 4KB and 1MB (instead of 4 MB), and 2) all PTEs are 16 Bytes (instead of 8 Bytes). The following figure summarizes the page table structure and indicates the sizes of the page tables and data pages (not drawn to scale):



The processor has a data TLB with 64 entries, and each entry can map either a 4KB page or a 1MB page. After a TLB miss, a hardware engine walks the page table to reload the TLB. The TLB uses a first-in/first-out (FIFO) replacement policy.

We will evaluate the execution of the following program which adds the elements from two 1MB arrays and stores the results in a third 1MB array (note that, 1MB = 1,048,576 Bytes, the starting address of the arrays are given below):

```
byte A[1048576]; // 1MB array 0x00001000000
byte B[1048576]; // 1MB array 0x00001100000
byte C[1048576]; // 1MB array 0x00001200000

for(int i=0; i<1048576; i++)
  C[i] = A[i] + B[i];
```

Assume that the above program is the only process in the system, and ignore any instruction memory or operating system overheads. The data TLB is initially empty.

Problem M4.3.A

Consider the execution of the program. There is no cache and each memory lookup has 100 cycle latency.

If all data pages are 4KB, compute the ratio of cycles for address translation to cycles for data access.

If all data pages are 1MB, compute the ratio of cycles for address translation to cycles for data access.

Problem M4.3.B

For this question, assume that in addition, we have a PTE cache with one cycle latency. A PTE cache contains page table entries. If this PTE cache has unlimited capacity, compute the ratio of cycles for address translation to cycles for data access for the 4KB data page case.

Problem M4.3.C

With the use of a PTE cache, is there any benefit to caching L3 PTE entries? Explain.

Problem M4.3.D

What is the minimum capacity (number of entries) needed in the PTE cache to get the same performance as an unlimited PTE cache? (Assume that the PTE cache does not cache L3 PTE entries and all data pages are 4KB)

Problem M4.4: 64-bit Virtual Memory

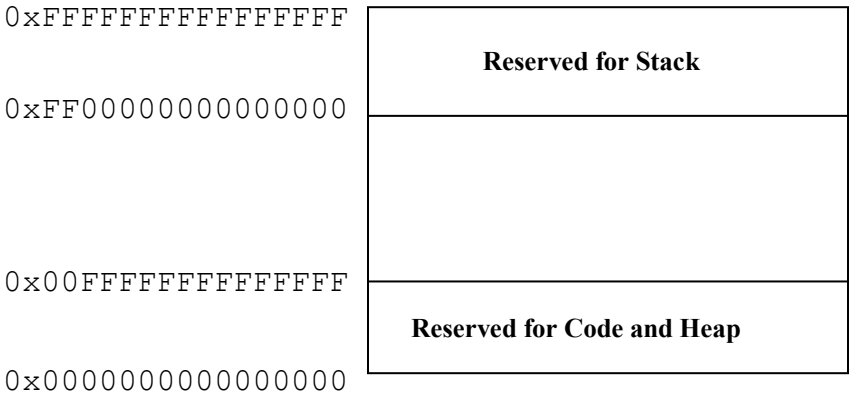
This problem examines page tables in the context of processors with 64-bit addressing.

Problem M4.4.A — Single level page tables

For a computer with 64-bit virtual addresses, how large is the page table if only a single-level page table is used? Assume that each page is 4KB, that each page table entry is 8 bytes, and that the processor is byte-addressable.

Problem M4.4.B — Let's be practical

Many current implementations of 64-bit ISAs implement only part of the large virtual address space. One way to do this is to segment the virtual address space into three parts as shown below: one used for stack, one used for code and heap data, and the third one unused.



A special circuit is used to detect whether the top eight bits of an address are all zeros or all ones before the address is sent to the virtual memory system. If they are not all equal, an invalid virtual memory address trap is raised. This scheme in effect removes the top seven bits from the virtual memory address, but retains a memory layout that will be compatible with future designs that implement a larger virtual address space.

The MIPS R10000 does something similar. Because a 64-bit address is unnecessarily large, only the low 44 address bits are translated. This also reduces the cost of TLB and cache tag arrays. The high two virtual address bits (bits 63:62) select between user, supervisor, and kernel address spaces. The intermediate address bits (61:44) must either be all zeros or all ones, depending on the address region.

How large is a single-level page table that would support MIPS R10000 addresses? Assume that each page is 4KB, that each page table entry is 8 bytes, and that the processor is byte-addressable.

Problem M4.4.C — Page table overhead

A three-level hierarchical page table can be used to reduce the page table size. Suppose we break up the 44-bit virtual address (VA) as follows:

VA[43:33]	VA[32:22]	VA[21:12]	VA[11:0]
1st level index	2nd level index	3rd level index	Page offset

If page table overhead is defined as (in bytes):

$$\frac{\text{PHYSICAL MEMORY USED BY PAGE TABLES FOR A USER PROCESS}}{\text{PHYSICAL MEMORY USED BY THE USER CODE, HEAP, AND STACK}}$$

Remember that a complete page table page (1024 or 2048 PTEs) is allocated even if only one PTE is used. Assume a large enough physical memory that no pages are ever swapped to disk. Use 64-bit PTEs. What is the smallest possible page table overhead for the three-level hierarchical scheme?

Assume that once a user page is allocated in memory, the whole page is considered to be useful. What is the largest possible page table overhead for the three-level hierarchical scheme?

Problem M4.4.D **PTE Overhead**

The MIPS R10000 uses a 40 bit physical address. The physical translation section of the TLB contains the physical page number (also known as PPN), one "valid," one "dirty," and three "cache status" bits.

What is the minimum size of a PTE assuming all pages are 4KB?

MIPS/Linux stores each PTE in a 64 bit word. How many bits are wasted if it uses the minimum size you have just calculated?

Problem M4.4.E **Page table implementation**

The following comment is from the source code of MIPS/Linux and, despite its cryptic terminology, describes a three-level page table.

```
/*
 * Each address space has 2 4K pages as its page directory, giving 1024
 * 8 byte pointers to pmd tables. Each pmd table is a pair of 4K pages,
 * giving 1024 8 byte pointers to page tables. Each (3rd level) page
 * table is a single 4K page, giving 512 8 byte ptes.
 * /
```

Assuming 4K pages, how long is each index?

Index	Length (bits)
Top-level ("page directory")	
2nd-level	
3rd-level	

Problem M4.4.F **Variable Page Sizes**

A TLB may have a *page mask* field that allows an entry to map a page size of any power of four between 4KB and 16MB. The page mask specifies which bits of the virtual address represent the page offset (and should therefore not be included in translation). What are the maximum and minimum reach of a 64-entry TLB using such a mask? The R10000 actually doubles this reach with little overhead by having each TLB entry map *two* physical pages, but don't worry about that here.

Problem M4.4.G **Virtual Memory and Caches**

Ben Bitdiddle is designing a 4-way set associative cache that is virtually indexed and virtually tagged. He realizes that such a cache suffers from a *homonym* aliasing problem. The *homonym* problem happens when two processes use the same virtual address to access different physical locations. Ben asks Alyssa P. Hacker for help with solving this problem. She suggests that Ben should add a PID (Process ID) to the virtual tag. Does this solve the *homonym* problem?

Another problem with virtually indexed and virtually tagged caches is called *synonym* problem. *Synonym* problem happens when distinct virtual addresses refer to the same physical location. Does Alyssa's idea solve this problem?

Ben thinks that a different way of solving *synonym* and *homonym* problems is to have a direct mapped cache, rather than a set associative cache. Is he right?

Problem M4.5: Cache Basics (2005 Fall Part A)

Questions in Part A are about the operations of virtual and physical address caches in two different configurations: direct-mapped and 2-way set-associative. The direct-mapped cache has 8 cache lines with 8 bytes/line (i.e. the total size is 64 bytes), and the 2-way set-associative cache is the same size (i.e. 32 bytes/way) with the same cache line size. The page size is 16 bytes.

Please answer the following questions.

Problem M4.5.A

We ask you to follow step-by-step operations of the virtually indexed, physically tagged, 2-way set-associative cache shown in the previous question (Figure B). You are given a snapshot of the cache and TLB states in the figure below. Assume that the smallest physical tags (i.e. no index part contained) are taken from the high order bits of an address, and that Least Recently Used (LRU) replacement policy is used.
(Only valid (V) bits and tags are shown for the cache; VPNs and PPNs for the TLB.)

Index	V	Tags (way0)	V	Tags (way1)
0	**1**	0x45	**0**	
1	**1**	0x3D	**0**	
2	**1**	0x1D	**0**	
3	**0**		**0**	

Initial cache tag states

VPN	PPN	VPN	PPN
0x0	0x0A	0x10	0x6A
0x1	0x1A	0x20	0x7A
0x2	0x2A	0x30	0x8A
0x3	0x3A	0x40	0x9A
0x5	0x4A	0x50	0xAA
0x7	0x5A	0x70	0xBA

TLB states

After accessing the address sequence (all in virtual address) given below, what will be the final cache states? Please fill out the table at the bottom of this page with the new cache states. You can write tags either in binary or in hexadecimal form.

Address sequence: 0x34 -> 0x38 -> 0x50 -> 0x54 -> 0x208 -> 0x20C -> 0x74 -> 0x54

Index	V	Tags (way0)	V	Tags (way1)

0				
1				
2				
3				

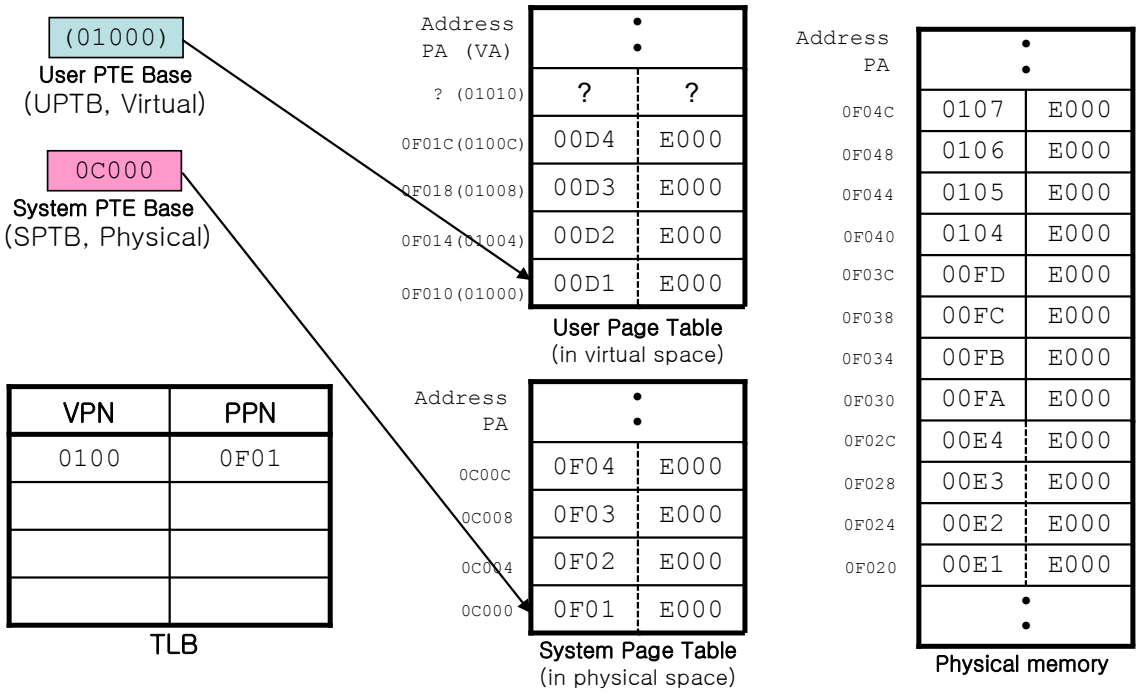
Final cache tag states

Problem M4.5.B

Assume that a cache hit takes one cycle and that a cache miss takes 16 cycles. What is the average memory access time for the address sequence of 8 words given in Question M4.5.A?

Problem M4.6: Handling TLB Misses (2005 Fall Part B)

In the following questions, we ask you about the procedure of handling TLB misses. The following figure shows the setup for this part and each component's initial states.



Notes 1. All numbers are in hexadecimal.
2. Virtual addresses are shown in parentheses, and physical addresses without parentheses.

For the rest of this part, we assume the following:

1) The system uses 20-bit virtual addresses and 20-bit physical addresses.
2) The page size is 16 bytes.
3) We use a linear (not hierarchical) page table with 4-byte page table entry (PTE). A PTE can be broken down into the following fields. (Don't worry about the status bits, PTE[15:0], for the rest of Part B.)

31 … 16	15	14	13	12	11	10	9 … 0
Physical Page Number (PPN)	V	R	W	U	M	S	0000000000

4) The TLB contains 4 entries and is fully associative.

On the next page, we show a pseudo code for the TLB refill algorithm.

```
// On a TLB miss, "MA" (Miss Address) contains the address of that
// miss.  Note that MA is a virtual address.

// UTOP is the top of user virtual memory in the virtual address
// space.  The user page table is mapped to this address and up.
#define UTOP 0x01000

// UPTB and SPTB stand for User PTE Base and System PTE Base,
// respectively.  See the figure in the previous page.

if (MA < UTOP) {
   // This TLB miss was caused by a normal user-level memory access

   // Note that another TLB miss can occur here while loading a PTE.
   LW Rtemp, UPTB+4*(MA>>4);   // load a PTE using a virtual address
}
else {
   // This TLB miss occurred while accessing system pages (e.g. page
    // tables)

   // TLB miss cannot happen here because we use a physical address.
   LW_physical Rtemp, SPTB+4*((MA-UTOP)>>4);  // load a PTE using a
                                              // physical address
}

(Protection check on Rtemp); // Don't worry about this step here
(Extract PPN from Rtemp and store it to the TLB with VPN);
(Restart the instruction that caused the TLB miss);
```

TLB refill algorithm

Problem M4.6.A

What will be the physical address corresponding to the virtual address 0x00030? Fill out the TLB states below after an access to the address 0x00030 is completed.

Virtual address 0x00030 -> Physical address (0x _____)

VPN	PPN
0x0100	0x0F01

TLB states

Problem M4.6.B

What will be the physical address corresponding to the virtual address 0x00050? Fill out the TLB states below after an access to the address 0x00050 is completed. (Start over from the initial system states, not from your system states after solving the previous question.)

Virtual address 0x00050 -> Physical address (0x _____)

VPN	PPN
0x0100	0x0F01

TLB states

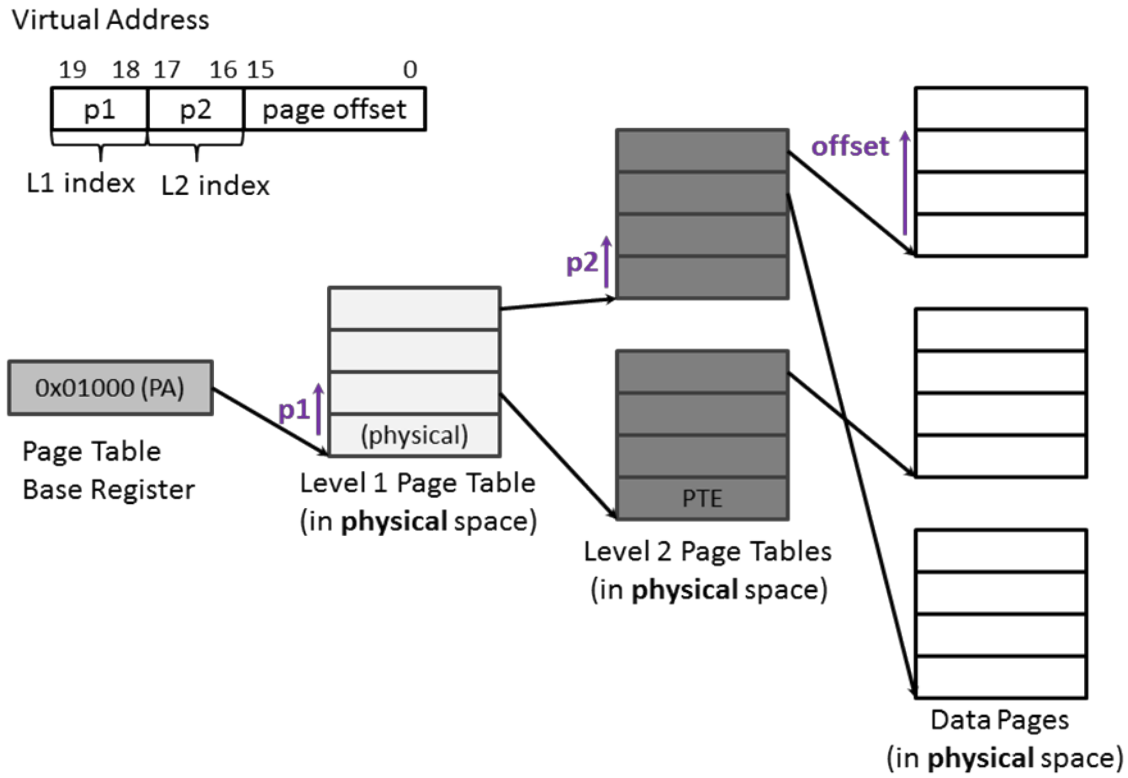
Problem M4.6.C

We integrate virtual memory support into our baseline 5-stage MIPS pipeline using the TLB miss handler. We assume that accessing the TLB does not incur an extra cycle in memory access in case of hits.

Without virtual memory support (i.e. we had only a single address space for the entire system), the average cycles per instruction (CPI) was 2 to run Program X. If the TLB misses 10 times for instructions and 20 times for data in every 1,000 instructions on average, and it takes 20 cycles to handle a TLB miss, what will be the new CPI (approximately)?

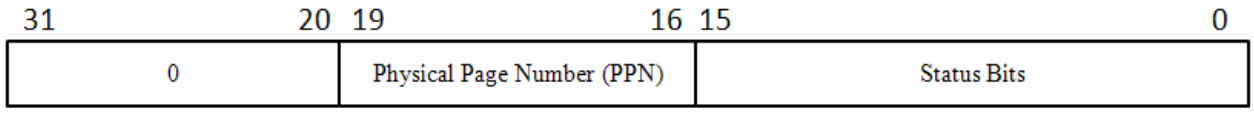
Problem M4.7: Hierarchical Page Table & TLB (Fall 2010 Part B)

Suppose there is a virtual memory system with 64KB page which has 2-level hierarchical page table. The **physical address** of the base of the level 1 page table (**0x01000**) is stored in a special register named Page Table Base Register. The system uses **20-bit** virtual address and **20-bit** physical address. The following figure summarizes the page table structure and shows the breakdown of a virtual address in this system. The size of both level 1 and level 2 page table entries is **4 bytes** and the memory is byte-addressed. Assume that all pages and all page tables are loaded in the main memory. Each entry of the level 1 page table contains the **physical address** of the base of each level 2 page tables, and each of the level 2 page table entries holds the **PTE** of the data page (the following diagram is not drawn to scale). As described in the following diagram, L1 index and L2 index are used as an index to locate the corresponding ***4-byte entry*** in Level 1 and Level 2 page tables.



2-level hierarchical page table

A PTE in level 2 page tables can be broken into the following fields (Don't worry about status bits for the entire part).



Problem M4.7.A

Assuming the TLB is initially at the state given below and the initial memory state is to the right, what will be the final TLB states after accessing the virtual address given below? Please fill out the table with the final TLB states. You only need to write VPN and PPN fields of the TLB. The TLB has 4 slots and is fully associative and if there are empty lines they are taken first for new entries. Also, translate the virtual address (VA) to the physical address (PA). ***For your convenience, we separated the page number from the rest with the colon ":".***

VPN	PPN
0x8	0x3

Initial TLB states

Address (PA)	
0x0:104C	0x7:1A02
0x0:1048	0x3:0044
0x0:1044	0x2:0560
0x0:1040	0xA:0FFF
0x0:103C	0xC:D031
0x0:1038	0xA:6213
0x0:1034	0x9:1997
0x0:1030	0xD:AB04
0x0:102C	0xF:A000
0x0:1028	0x6:0020
0x0:1024	0x5:1040
0x0:1020	0x4:AA40
0x0:101C	0x3:10EF
0x0:1018	0xB:EA46
0x0:1014	0x2:061B
0x0:1010	0x1:0040
0x0:100C	0x0:1020
0x0:1008	0x0:1048
0x0:1004	0x0:1010
0x0:1000	0x0:1038

The part of the memory (in physical space)

<u>Virtual Address</u>:

0xE:17B0 (1110:0001011110110000)

VPN	PPN
0x8	0x3

Final TLB states

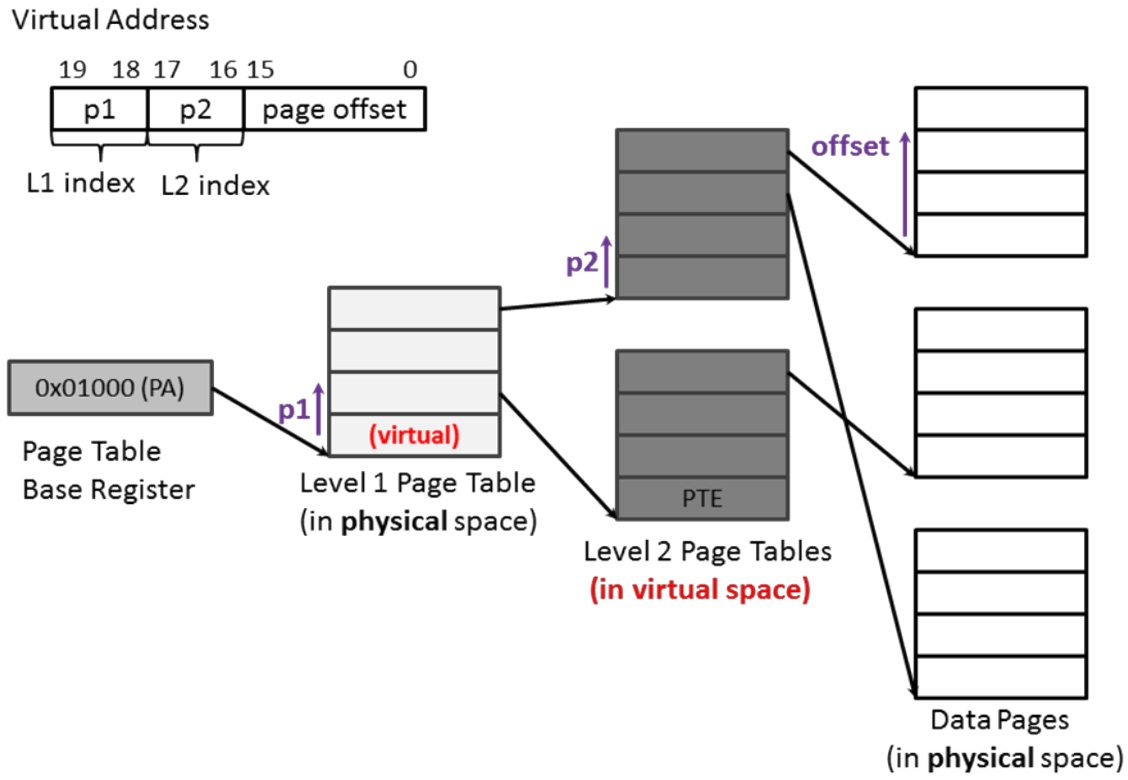
VA 0xE17B0 => PA _____

Problem M4.7.B

What is the total size of memory required to store both the level 1 and 2 page tables?

Problem M4.7.C

Ben Bitdiddle wanted to reduce the amount of physical memory required to store the page table, so he decided to only put the level 1 page table in the physical memory and use the virtual memory to store level 2 page tables. Now, each entry of the level 1 page table contains the **virtual address** of the base of each level 2 page tables, and each of the level 2 page table entries contains the **PTE** of the data page (the following diagram is not drawn to scale). Other system specifications remain the same. (The size of both level 1 and level 2 page table entries is **4 bytes.**)



Ben's design with 2-level hierarchical page table

Assuming the TLB is initially at the state given below and the initial memory state is to the right (**different** from M5.9.A), what will be the final TLB states after accessing the virtual address given below? Please fill out the table with the final TLB states. You only need to write VPN and PPN fields of the TLB. The TLB has 4 slots and it is fully associative and if there are empty lines they are taken first for new entries. Also, translate the virtual address to the physical address. ***Again, we separated the page number from the rest with the colon ":".***

VPN	PPN
0x8	0x1

Initial TLB states

Address (PA)	
.....	.....
0x1:1048	0x3:0044
0x1:1044	0x2:0560
0x1:1040	0x1:0FFF
0x1:103C	0x1:D031
0x1:1038	0xA:6213
0x1:1034	0x9:1997
.....	.....
0x1:0018	0xF:A000
0x1:0014	0x6:0020
0x1:0010	0x1:1040
0x1:000C	0x4:AA40
0x1:0008	0x3:10EF
0x1:0004	0xB:EA46
.....	.....
0x0:1010	0x1:0040
0x0:100C	0x0:1020
0x0:1008	0x2:0010
0x0:1004	0x8:0010
0x0:1000	0x8:1038

The part of the memory (in physical space)

Virtual Address:

0xA:0708 (1010:0000011100001000)

VPN	PPN
0x8	0x1

Final TLB states

VA 0xA0708 => PA _____

Problem M4.7.D

Alice P. Hacker examines Ben's design and points out that his scheme can result in infinite loops. Describe the scenario where the memory access falls into infinite loops.

Problem M4.8: Caches and Virtual Memory (Spring 2015 Quiz 1, Part B)

Problem M4.8.A

Consider a reference stream that repetitively **loops over four addresses, A, B, C, and D (ABCDABCDABCD….)**. We will study how different replacement policies perform on this reference stream, using a small, 2-entry, fully-associative cache.

1. Find out how the cache performs with LRU replacement. Fill the table below to show the cache contents over time, and note whether each access is a hit or a miss. Then, compute the long-term miss ratio (i.e., discounting cache warm-up).

Access	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Address	A	B	C	D	A	B	C	D	A	B	C	D
Entry 1	-	A	A									
Entry 2	-	-	B									
Hit?	N	N	N									

What is the long-term miss ratio under LRU? _____

2. Find out how the cache performs under optimal replacement. **This cache cannot bypass accesses, i.e., on every miss, it must replace an existing block and insert the new block.** Fill the time diagram below, and find the long-term hit rate.

Access	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Address	A	B	C	D	A	B	C	D	A	B	C	D
Entry 1	-	A	A									
Entry 2	-	-	B									
Hit?	N	N	N									

What is the long-term miss ratio under optimal replacement? _____

In the example, is there a simple policy that, without knowing the future, performs as well as the optimal one? If so, which one?

Problem M4.8.B

Consider a byte addressing system with **16-bit virtual and physical addresses**. The system has a cache with the following properties:

- 8 sets with 128 bytes per block
- 4-way set-associative organization
- Virtually-indexed, physically-tagged

1. Suppose we use **256-byte pages**. Where in the cache can **virtual address** 0xABCD live? Please use crosses (X) to mark its possible locations in the diagram below.
   (The binary representation of 0xABCD is 1010 1011 1100 1101.)

Index	**Cache Contents**			
	Way 0	Way 1	Way 2	Way 3
0				
1				
2				
3				
4				
5				
6				
7				

2. As before, suppose we use **256-byte pages**. Where in the cache can **physical address** 0xABCD live? Please use crosses (X) to mark its possible locations.

Index	**Cache Contents**			
	Way 0	Way 1	Way 2	Way 3
0				
1				
2				
3				
4				
5				
6				
7				

3. Suppose we use **1024-byte pages** instead. Where in the cache can **physical address** 0xABCD live? Please use crosses (X) to mark its possible locations.

Index	**Cache Contents**			
	Way 0	Way 1	Way 2	Way 3
0				
1				
2				
3				
4				
5				
6				
7				

Problem M4.8.C

We'd like our memory system to support **two page sizes**: **256-byte small pages** and **1024-byte large pages**. A common approach to support multiple page sizes is to use separate TLBs, one for each page size. Instead, to reduce area overheads, we will use a single TLB to cache translations of both small and large pages, shown in Figure B-1. The TLB has 8 sets and 2 ways. The L bit denotes whether the cached PTE is for a large page.

V = valid bit **L = large page bit (set to 1 when a large page is stored)**
PPN = physical page number

	Way 0				Way 1			
	V	L	Tag	PPN	V	L	Tag	PPN
0								
1								
2								
3								
4								
5								
6								
7								

Figure B-1. TLB for multiple page sizes.

Each TLB access consists of three steps. First, the TLB checks for a small-page match, using the tag and index bits shown in Figure B-2. Second, if it does not find a small-page match, it checks for a large-page match, using the tag and index bits in Figure B-3. Third, if the second lookup misses as well, it results in a TLB miss and a page table walk.

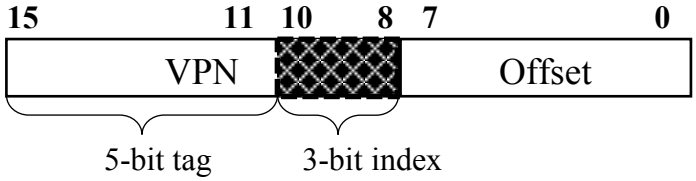


Figure B-2. Tag and index bits for small (256-byte) pages.

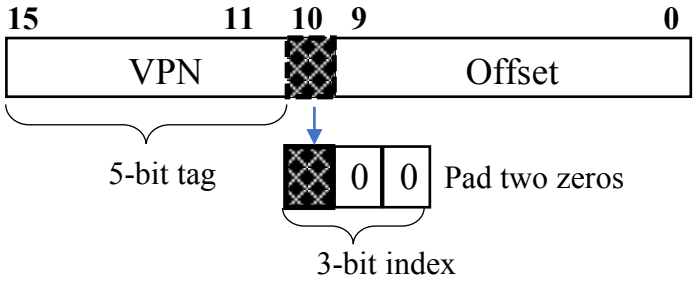


Figure B-3. Tag and index bits for large (1024-byte) pages.

Assume virtual address 0xABBA translates to physical address 0x47BA.

1. If virtual address 0xABBA **belongs to a small (256-byte) page**, fill in the fields of the TLB entry, and mark all possible TLB locations it can be in.

TLB entry

L	Tag	PPN

Possible locations

	Way 0	Way 1
0		
1		
2		
3		
4		
5		
6		
7		

2. If virtual address 0xABBA **belongs to a large (1024-byte) page**, fill in the fields of the TLB entry, and mark all possible TLB locations it can be in.

TLB entry

L	Tag	PPN

Possible locations

	Way 0	Way 1
0		
1		
2		
3		
4		
5		
6		
7		

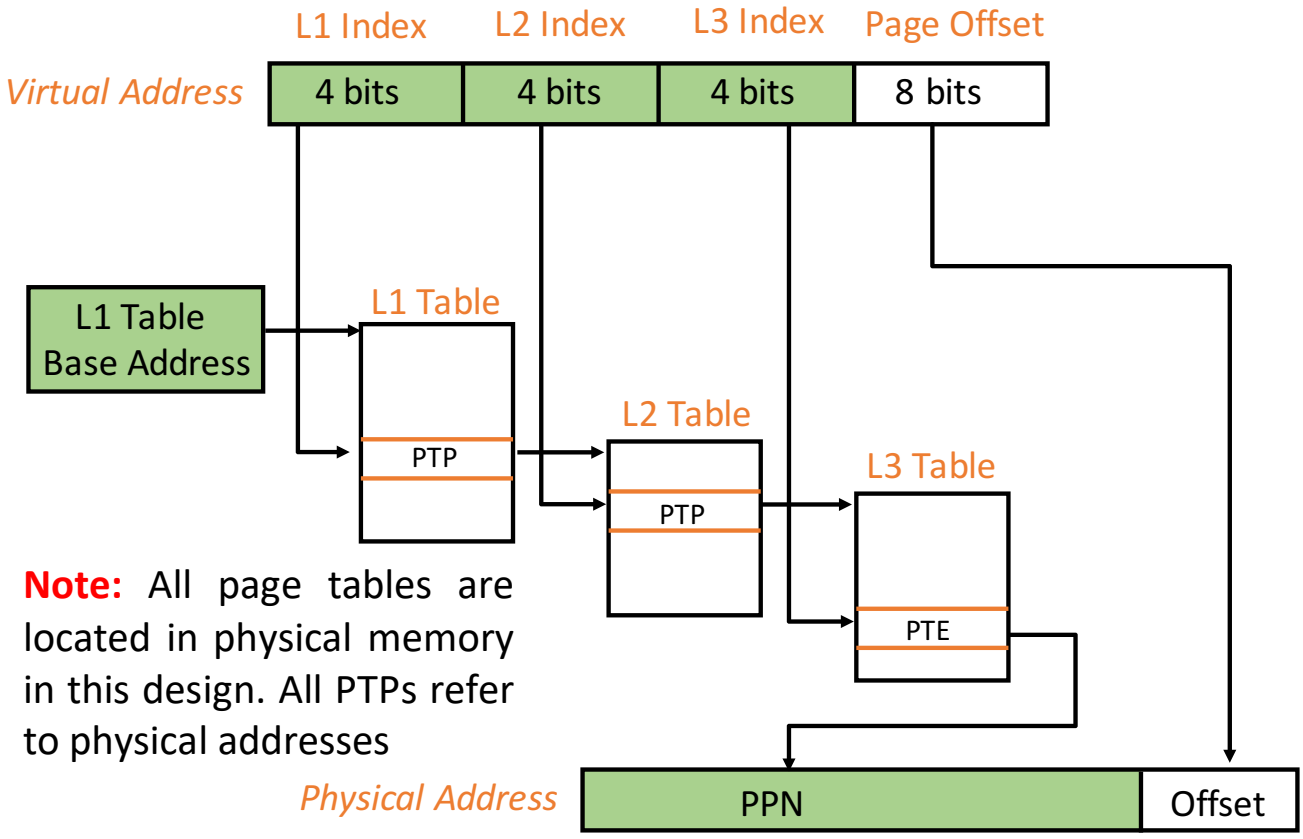
3. What is the reach of this TLB? (TLB reach = maximum amount of memory accessible without TLB misses)

4. This TLB has a utilization problem for large pages. Explain why it happens and how to solve it.

Problem M4.9: Caches and Virtual Memory (Spring 2016 Quiz 1, Part B)

Ben Bitdiddle purchases a new processor to run his 6.823 lab experiments. The processor manual informs Ben that the machine is byte-addressed with 20-bit virtual addresses and 16-bit physical addresses.

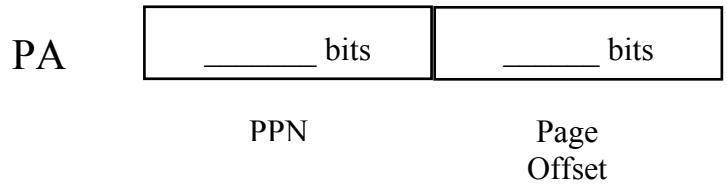
The processor manual only specifies that the machine uses a 3-level page table with the following virtual-address breakdown.



Problem M4.9.A

What is the page size of Ben's machine? _____

Demarcate the physical address into the following fields: Physical Page Number (PPN), Page Offset



Ben executes the following snippet of code on his new processor. Assume integers are 4-bytes long, and the array elements are mapped to virtual addresses `0x0000` through `0x1ffc`. Assume `array` and `sum` have been suitably initialized.

```
1     int array[2048];
2     while (1) {
3       for (int i = 0; i < 4; i++)
4           sum += array[i * 256];
5     }
```

The processor manual states this machine has a TLB with 4 entries. Assume that variables `i` and `sum` are stored in registers, and ignore address translation for instruction fetches; only accesses to `array` require address translation.

Problem M4.9.B

In steady state, how many misses from the TLB will Ben observe per iteration of the `while` loop (lines 3, 4) on average, if (state your reasoning):

a) the TLB is direct-mapped _____

b) the TLB is fully-associative _____
(assume LRU replacement policy)

Problem M4.9.C

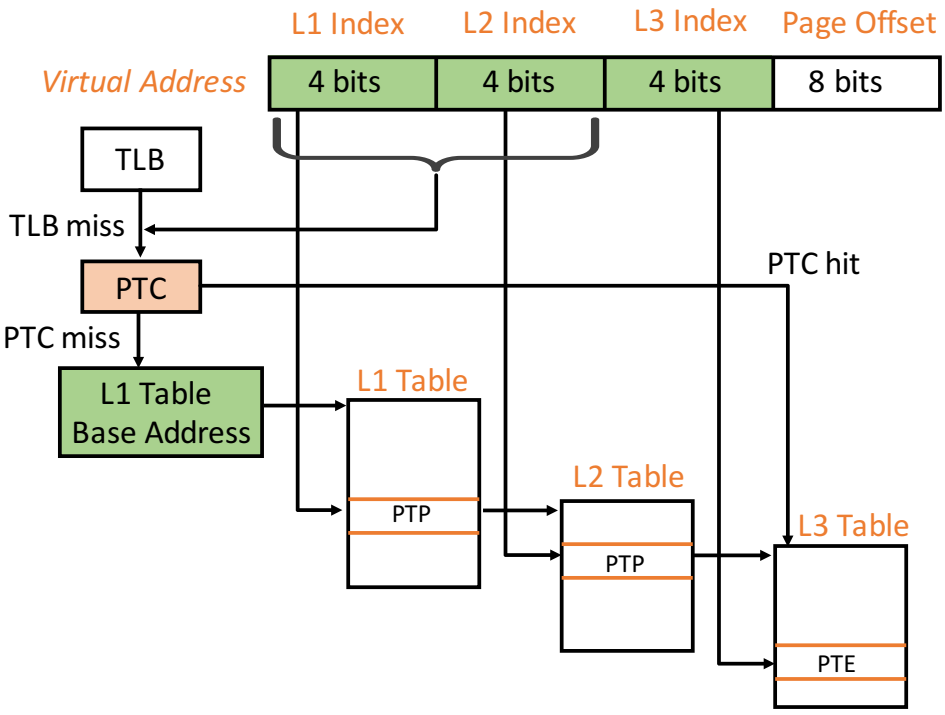
In steady state, how many total memory accesses will Ben observe per iteration of the `while` loop (lines 3, 4) on average, if (state your reasoning):

a) the TLB is direct-mapped _____

b) the TLB is fully-associative _____
(assume LRU replacement policy)

Problem M4.9.D

Ben wonders if he can reduce the number of memory accesses required to perform the address translations. His friend Alyssa P. Hacker suggests adding a *partial-translation cache (PTC)*, in addition to the TLB. The PTC stores a mapping of the higher-order bits of the virtual address to a L3 page table entry. If a translation misses in the TLB, but hits in the PTC, the MMU issues an access to the corresponding L3 page table directly, *skipping* the L1 and L2 page tables. On a TLB miss + PTC miss, the page walk returns the PPN and also installs a translation from VPN to L3 Page Table id in the PTC, and this incurs no additional cost.



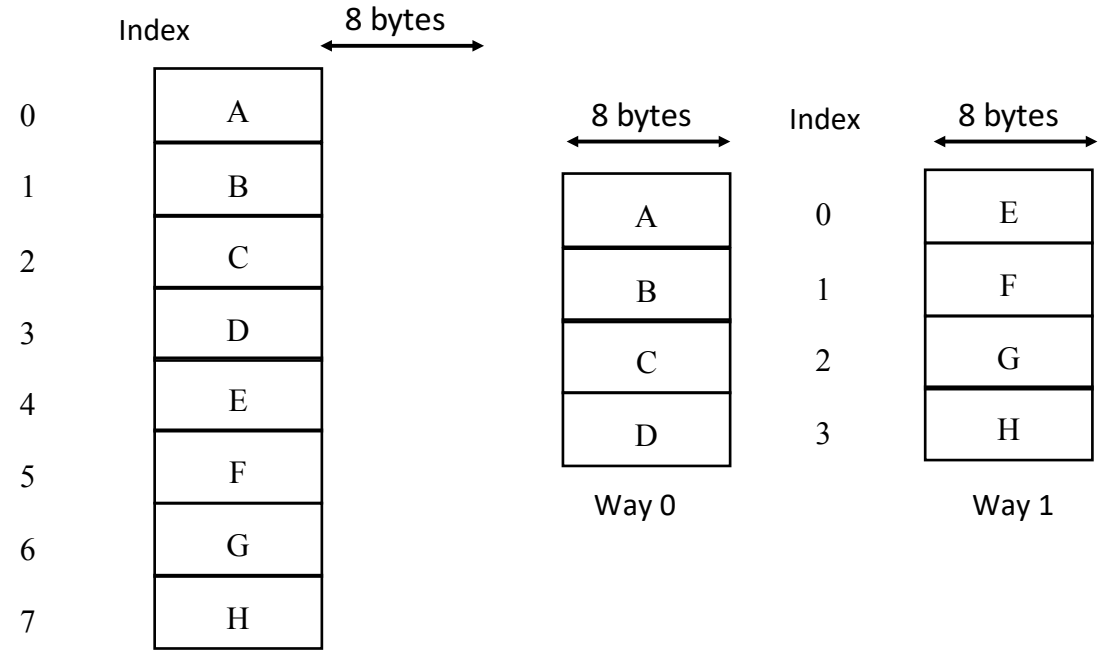
Alyssa proposes adding a PTC with 1 entry to the processor. Does this addition benefit the code snippet in Question 2? How many total memory accesses will Ben observe now, if:

a) the TLB is direct-mapped _____

b) the TLB is fully-associative _____
(assume LRU replacement policy)

Problem M4.9.E

Alyssa's processor contains a 64 byte L1 cache with eight **8-byte cache blocks**, denoted A—H in the figure below. For each configuration shown in the figure, which block(s) can virtual (byte) address `0x34` be mapped to? Assume a **page size of 16 bytes**. Fill out the table at the bottom of the page, indicating each of the possible blocks by its assigned letter (A—H).



(a) Direct-Mapped Cache

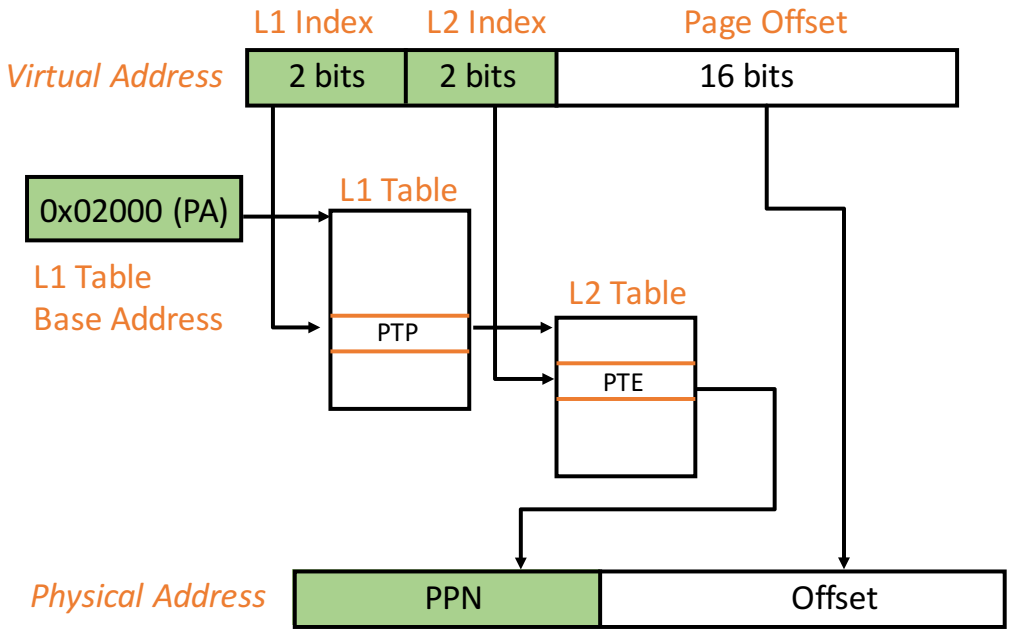
(b) Two-Way Set-Associative Cache

	Virtually Indexed	Physically Indexed
Direct-mapped (a)		
2-way set-associative (b)		

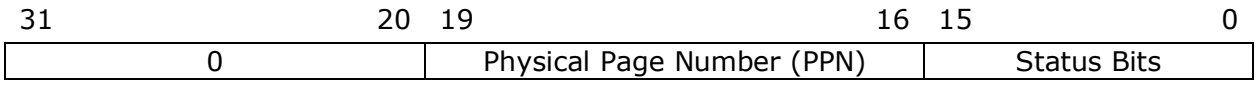
Problem M4.10: Nested paging (Spring 2017 Quiz 1, Part B)

This problem requires the knowledge of Handout #7 (Nested Paging) and Lecture 4 and 5. Please read these materials before answering the following questions.

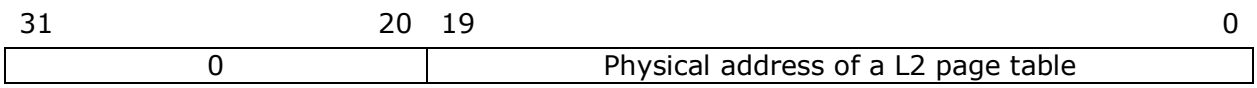
Ben Bitdiddle purchases a new processor to run his 6.823 labs. The processor manual indicates that the machine is byte-addressed with 20-bit virtual addresses and 20-bit physical addresses. The following figure summarizes the 2-level page table structure and shows the breakdown of a virtual address in this system. The physical address of the base of the Level 1 page table (0x02000) is stored in the L1 Table Base Address register. The L1 and L2 page tables are located in physical memory. The size of both L1 and L2 page table entries is 4 bytes. Each entry of the L1 page table contains the physical address of the base of each Level 2 page table (a PTP), and each of the L2 page table entries holds the PTE of the data page.



A PTE in L2 page tables can be broken into the following fields. (Don't worry about status bits).



A PTP in the L1 page table appears as follows.



Problem M4.10.A

Assuming the initial memory state is as shown to the right, what is the physical page number (PPN) of virtual address (VA) 0xB29A0? What is the physical address (PA)? Show and explain your work for full credit. **For your convenience, we separate the page number from the offset with a colon ":".**

Address (PA)	
0x0:2000	0x0:2048
0x0:2004	0x0:2010
0x0:2008	0x0:2038
0x0:200C	0x0:2028
0x0:2010	0x1:0084
0x0:2014	0x5:0DA8
0x0:2018	0x6:11A0
0x0:201C	0xB:9944
0x0:2020	0xC:7FFF
0x0:2024	0x4:B000
0x0:2028	0x7:30B1
0x0:202C	0xD:2E5C
0x0:2030	0x3:A000
0x0:2034	0x6:010C
0x0:2038	0xA:74C0
0x0:203C	0x8:A524
0x0:2040	0x9:FFEE
0x0:2044	0x2:93A4
0x0:2048	0xA:74D0
0x0:204C	0x3:FD40

Snapshot of physical memory

Virtual Address

0xB:29A0 = 0b 1011:0010100110100000

VPN 0xB => PPN _____

VA 0xB29A0 => PA _____

Unable to run Pin in his own environment, Ben's friend, Alyssa P. Hacker, refers him to the *Handout #7 (Nested Paging)* to learn how to run his labs in a virtual machine (much to the TA's dismay!) However, Ben is frustrated by the worst-case performance. Let's find out why.

Problem M4.10.B

Ben starts his foray into virtualization by thinking about `gPA=>hPA` translation.

a) Assuming Ben's host physical memory has the same snapshot as in Question 1, what is the host physical address (`hPA`) of guest physical address (`gPA`) `0xB29A4`? Explain.

`gPA 0xB29A4 => hPA` _____

b) Assuming no TLB, how many accesses to host physical memory are required to access the data associated with a `gPA` (i.e., perform the `gPA=>hPA` translation and fetch the data)? Explain.

Problem M4.10.C

Given a guest virtual address (gVA), the first step of a nested page table walk is to load the relevant guest L1 page table PTP. This provides the base gPA of the guest L2 table. Ben is shocked at how much work is required!

a) Assume host physical memory is initialized as in Question 1 and as shown to the right, the Guest Table Base Address register holds 0xB29A0 (a gPA), and the Host Table Base Address register holds 0x02000 (a hPA). During a nested page table walk of guest virtual address (gVA) 0x61EAC, what are the contents of the guest L1 page table PTP entry? **For your convenience, we separate the page number from the offset with a colon ":".**

Address (PA)	
0x0:2000	0x0:2048
0x0:2004	0x0:2010
0x0:2008	0x0:2038
0x0:200C	0x0:2028
0x0:2010	0x1:0084
...	...
0x2:2998	0xD:2E5C
0x2:299C	0x3:A000
0x2:29A0	0x6:010C
0x2:29A4	0xA:74CC
0x2:29A8	0x7:30B1
...	...
0x5:2994	0x6:11A0
0x5:2998	0xB:F149
0x5:299C	0xC:7BFF
0x5:29A0	0x4:B020
0x5:29A4	0xF:A120
...	...
0xA:299C	0x8:A624
0xA:29A0	0x9:FEED
0xA:29A4	0x2:93A4
0xA:29A8	0xA:7440
0xA:29AC	0x3:FD40

Snapshot of **host** physical memory

Guest Virtual Address (gVA)

0x6:1EAC = 0b 0110:0001111010101100

Guest L1 Table PTP = _____

b) Assume no TLB. Starting from some gVA, how many accesses to host physical memory are required to determine the guest L1 PTP entry of a guest virtual address (gVA)? Explain.

Problem M4.10.D

For the 2-level nested page table in the *Handout #7 (Nested Paging)*, assuming no TLB, how many accesses to host physical memory are required to perform a guest memory access? (i.e. given a gVA, find its corresponding hPA **and fetch the data**). Explain.

Problem M4.10.E

For an *M*-level hierarchical guest page table and an *N*-level hierarchical host page table, assuming no TLB, how many accesses to host physical memory are required to perform a guest memory access? (i.e. given a gVA, find its corresponding hPA **and fetch the data**). Explain.

Problem M5.1: Fully-Bypassed Simple 5-Stage Pipeline

We have reproduced the fully bypassed 5-stage MIPS processor pipeline from Lecture 7 in Figure M5.1-A. In this problem, we ask you to write equations to generate correct bypass and stall signals. Feel free to use any symbol introduced in the lecture.

Problem M5.1.A **Stall**

Do we still need to stall this pipeline? If so, explain why. (1) Write down the correct equation for the stall condition and (2) give an example instruction sequence which causes a stall.

Problem M5.1.B **Bypass Signal**

In Lecture 7, we gave you an example of bypass signal (ASrc) from EX stage to ID stage. In the fully bypassed pipeline, however, the mux control signals become more complex, because we have more inputs to the muxes in the ID stage.

Write down the bypass condition for each bypass path in Mux 1. Please indicate the priority of the signals; that is, if all bypass conditions are met, indicate which signals have the highest and the lowest priorities.

Bypass $_{EX->ID}$ ASrc = (rs_D=ws_E).we-bypass$_E$.re1$_D$ (given in Lecture 7)

Bypass $_{MEM->ID}$ =

Bypass $_{WB->ID}$ =

Priority:

Problem M5.1.C **Partial Bypassing**

While bypassing gives us a performance benefit, it may introduce extra logic in critical paths and may force us to lower the clock frequency. Suppose we can afford to have only one bypass in the datapath. How would you justify your choice? Argue in favor of one bypass path over another.

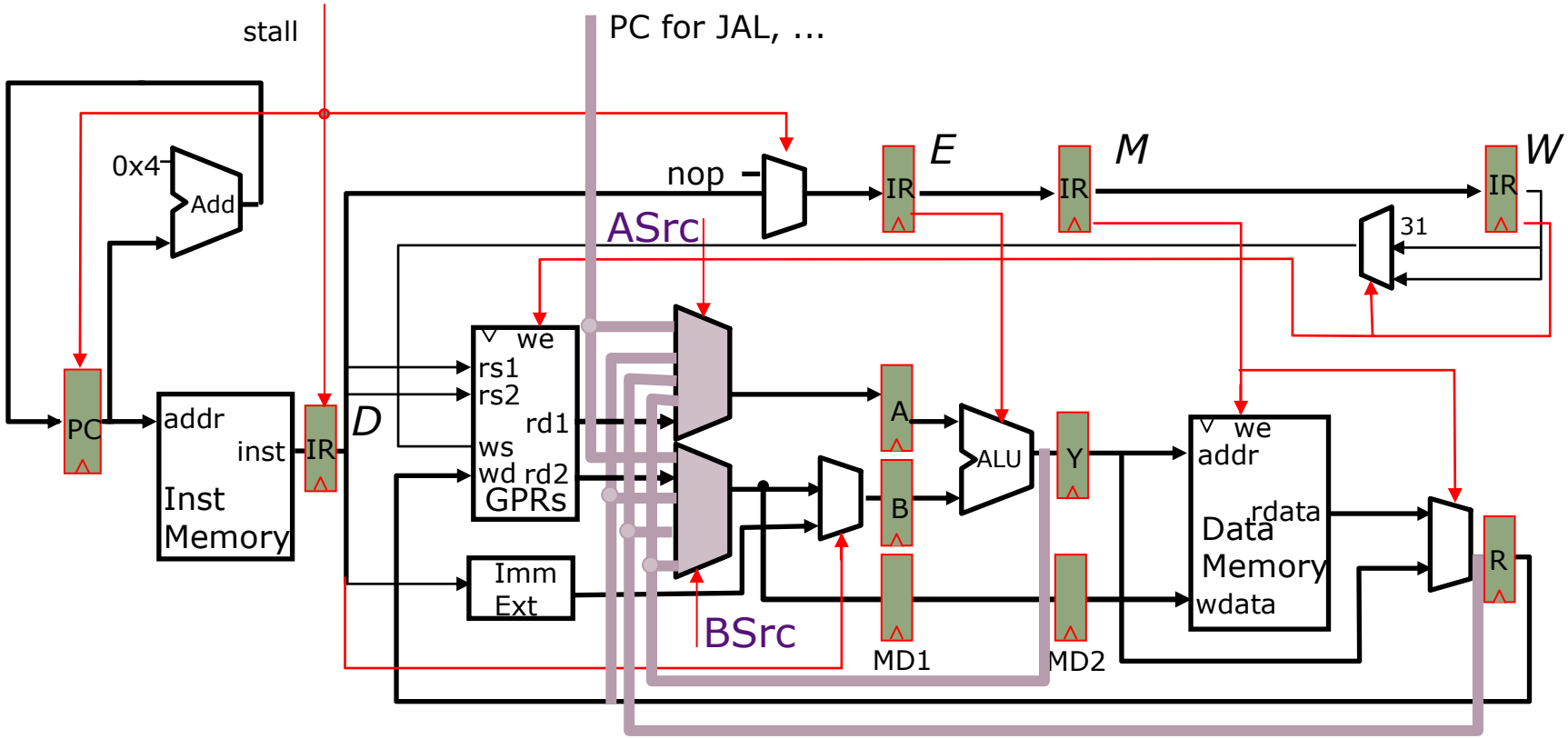


Figure M5.1-A: Fully-Bypassed MIPS Pipeline

Problem M5.2: Basic Pipelining

Unlike the Harvard-style (separate instruction and data memories) architectures, machines using the Princeton-style have a shared instruction and data memory. In order to reduce the memory cost, Ben Bitdiddle has proposed the following two-stage Princeton-style MIPS pipeline to replace a single-cycle Harvard-style pipeline from our lectures.

Every instruction takes exactly two cycles to execute (i.e., instruction fetch and execute) and there is no overlap between two sequential instructions; that is, fetching an instruction occurs in the cycle following the previous instruction's execution (no pipelining).

Assume that the new pipeline does not contain a branch delay slot. Also, don't worry about self-modifying code for now.

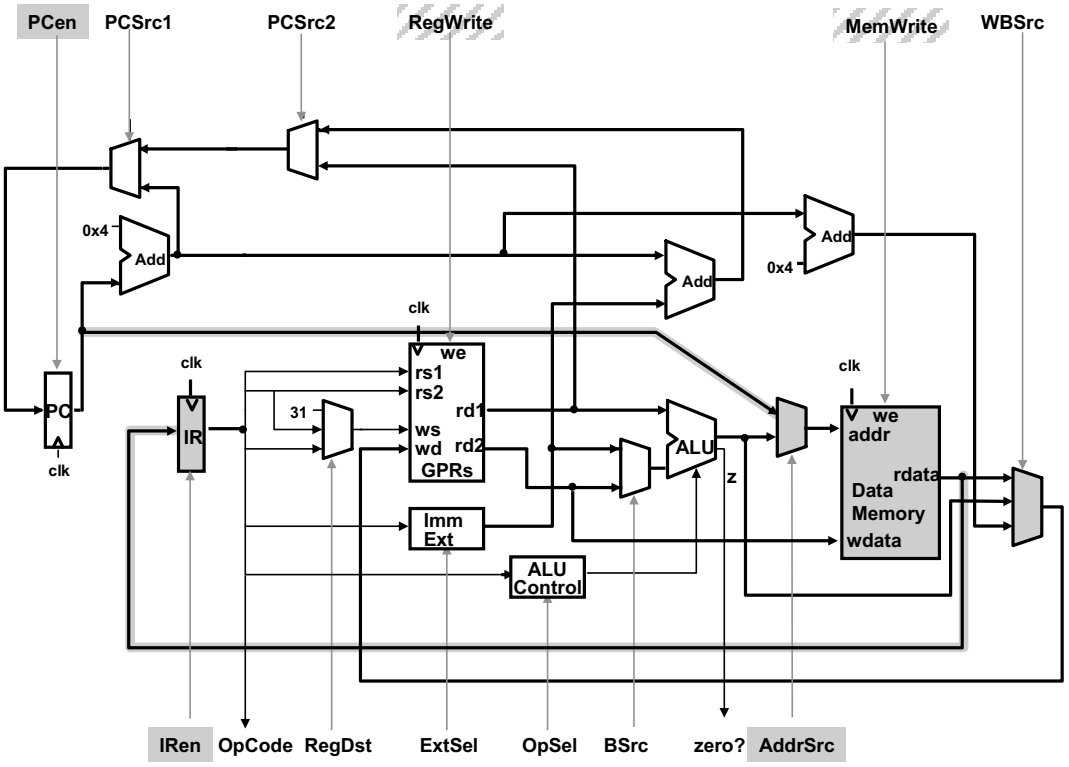


Figure M5.2-A: Two-stage pipeline, Princeton-style

Problem M5.2.A **Mux Control Signals (1)**

Please complete the following control signals. You are allowed to use any internal signals (e.g., OpCode, PC, IR, zero?, rd1, data, etc.) but not other control signals (ExtSel, IRSrc, PCSrc, etc.).

Example syntax: PCEn = (OpCode == ALUOp) or ((ALU.zero?) and (not (PC == 17)))

You may also use the variable S which indicates the pipeline's operation phase at a given time.

```
S := I-Fetch | Execute  (toggles every cycle)
```

PCEn =

IREn =

AddrSrc = Case _____

_____ => PC

_____ => ALU

Problem M5.2.B **Modified pipeline**

After having implemented his proposed architecture, Ben has observed that a lot of datapath is not in use because only one phase (either I-Fetch or Execute) is active at any given time. So he has decided to fetch the next instruction during the Execute phase of the previous instruction.

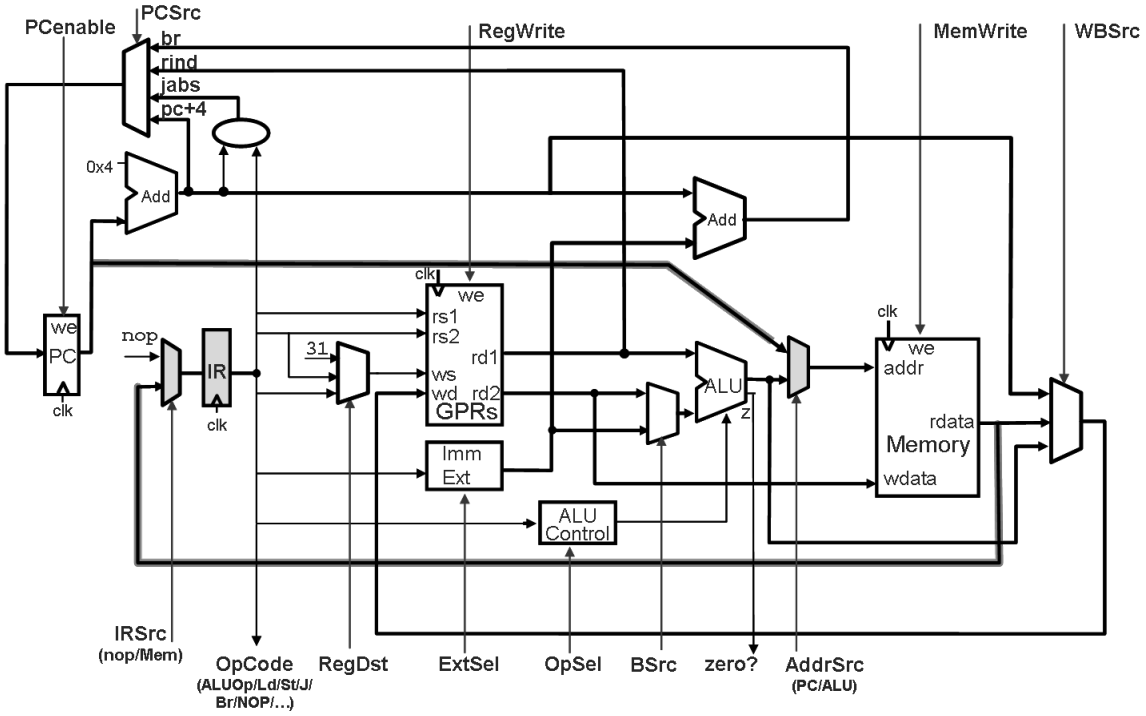


Figure M5.2-B: Modified Two-stage Princeton-style MIPS Pipeline

Do we need to stall this pipeline? If so, for each cause (1) write down the cause in one sentence and (2) give an example instruction sequence. If not, explain why. (Remember there is **no** delay slot.)

Problem M5.2.C **Mux Control Signals (2)**

Please complete the following control signals in the modified pipeline. As before, you are allowed to use any internal signals (e.g., OpCode, PC, IR, zero?, rd1, data, etc.) but not other control signals (ExtSel, IRSrc, PCSrc, etc.)

PCEnable =

AddrSrc = Case _____ _____ => PC _____ => ALU
IRSrc = Case _____ _____ => nop _____ => Mem

Problem M5.2.D

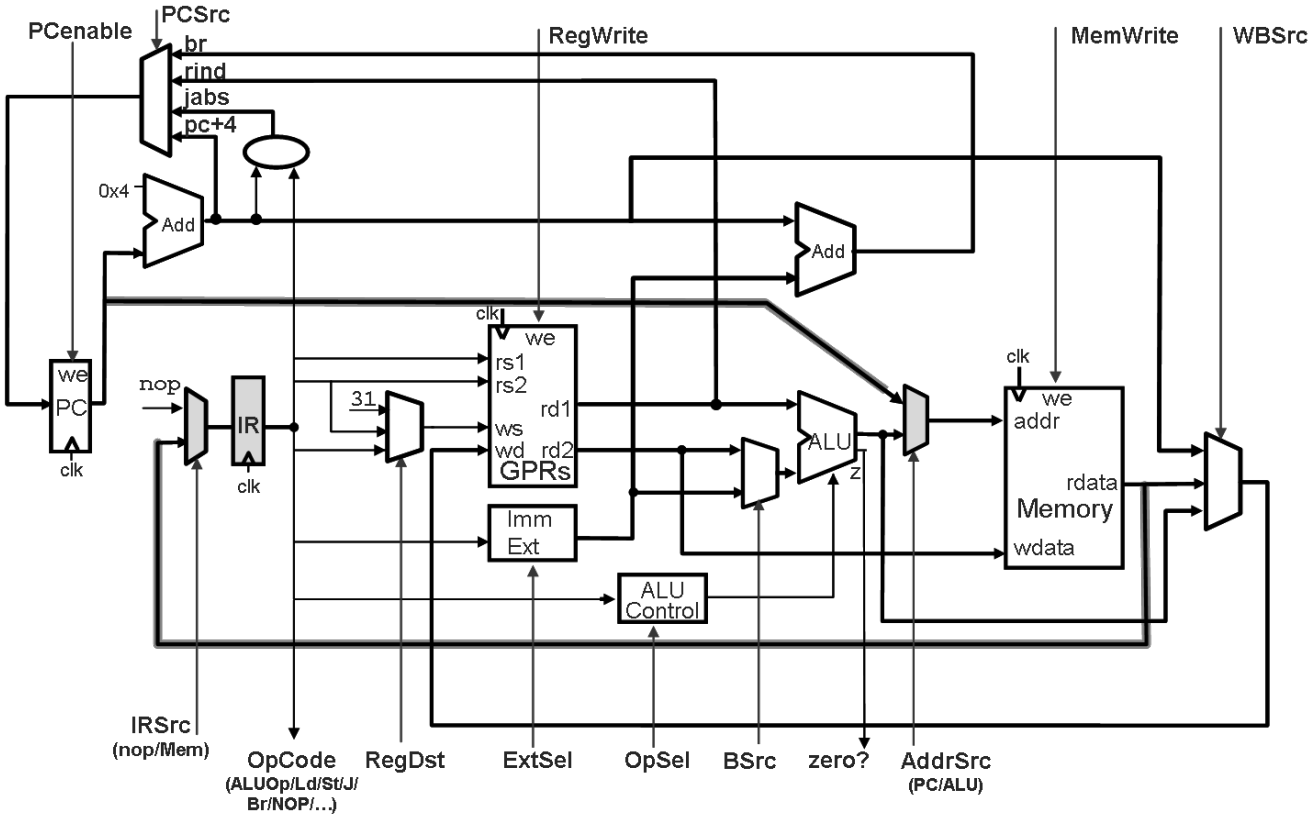
Now we are ready to put Ben's machine to the test. We would like to see a cycle-by-cycle animation of Ben's two-stage pipelined, Princeton-style MIPS machine when executing the instruction sequence below. In the following table, each row represents a snapshot of some control signals and the content of some special registers for a particular cycle. Ben has already finished the first two rows. Complete the remaining entries in the table. Use * for "don't care".

Label	Address	Instruction
I_1	100	ADD
I_2	104	LW
I_3	108	J I_7
I_4	112	LW
I_5	116	ADD
I_6	120	SUB
I_7	312	ADD
I_8	316	ADD

Time	PC	"IR"	PCenable	PCSrc1	AddrSrc	IRSrc
t_0	I_1:100	-	1	pc+4	PC	Mem
t_1	I_2:104	I_1	1	Pc+4	PC	Mem
t_2						
t_3						
t_4						
t_5						
t_6						

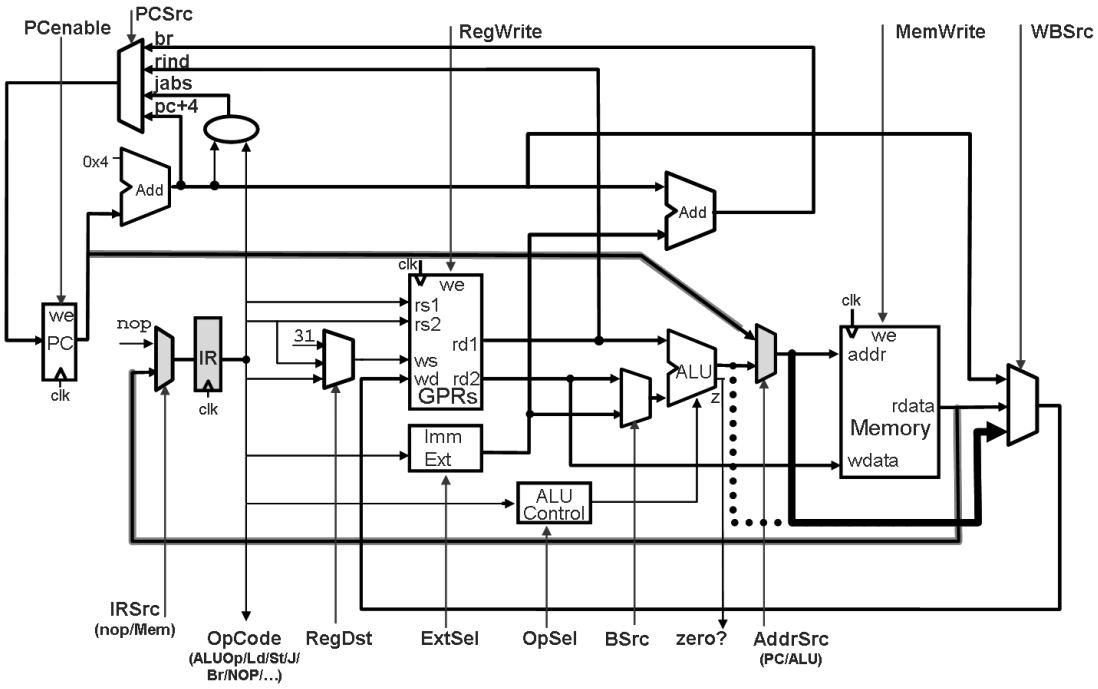
Problem M5.2.E **Self-Modifying Code**

Suppose we allow self-modifying code to execute, i.e., store instructions can write to the portion of memory that contains executable code. Does the two-stage Princeton pipeline need to be modified to support such self-modifying code? If so, please indicate how. You may use the diagram below to draw modifications to the datapath. If you think no modifications are required, explain why.



Problem M5.2.F

To solve a chip layout problem Ben decides to reroute the input of the WB mux to come from after the AddrSrc MUX rather than ahead of the AddrSrc MUX. (The new path is shown with a bold line, the old in a dotted line.) The rest of the design is unaltered.



How does this break the design? Provide a code sequence to illustrate the problem and explain in one sentence what goes wrong.

Problem M5.2.G **Architecture Comparison**

Give one advantage of the Princeton architecture over the Harvard architecture.

Give one advantage of the Harvard architecture over the Princeton architecture.

Problem M5.3: Processor Design (Short Yes/No Questions)

The following statements describe two variants of a processor which are otherwise identical. In each case, circle "**Yes**" if the variants might generate different results from the same compiled program, circle "**No**" otherwise. You must also briefly explain your reasoning. Ignore differences in the time that each machine takes to execute the program.

Problem M5.3.A **Interlock vs. Bypassing**

Pipelined processor A uses interlocks to resolve data hazards, while pipelined processor B has full bypassing.

Yes / No

Problem M5.3.B **Delay Slot**

Pipelined processor A uses branch delay slots to resolve control hazards, while pipelined processor B kills instructions following a taken branch.

Yes / No

Problem M5.3.C **Structural Hazard**

Pipelined processor A has a single memory port used to fetch instructions and data, while pipelined processor B has no structural hazards.

Yes / No

Problem M5.4: HAL 180 ISA and 6-Stage Pipelined Implementation (Spring 2015 Quiz 1, Part C)

Inspired by how the IBM 360 uses condition codes, Ben Bitdiddle designs the HAL 180 architecture, which features two flag registers. Table C-1 describes these flags.

Name	Description
Sign Flag (SF)	Stores 1 if the result of the *last arithmetic or comparison instruction* was negative, 0 if it was positive
Zero Flag (ZF)	Stores 1 if the result of the *last arithmetic, logical, or comparison instruction* was zero, and 0 if it was non-zero

Table C-1. HAL 180 status flags.

Table C-2 summarizes the different instruction types and the flags they read or write. The SF and ZF columns have an "R" when the instruction reads the status flag, a "W" if it writes the flag (and does not read it), or a blank if the instruction does not affect the status flag. For example, `JL` (jump if less than) reads SF; `ADD` writes all flags; and `JMP` (unconditional jump) does not affect any flag. Some instructions, like `CMP`, write the status flags but do not return any result.

Instruction	Description	SF	ZF
Arithmetic Instructions			
ADD *s1, s2*	*s1* ← *s1* + *s2*	W	W
SUB *s1, s2*	*s1* ← *s1* - *s2*	W	W
MUL *s1, s2*	*s1* ← *s1* × *s2*	W	W
Logical Instructions			
AND *s1, s2*	*s1* ← *s1* & *s2*		W
OR *s1, s2*	*s1* ← *s1* | *s2*		W
XOR *s1, s2*	*s1* ← *s1* ^ *s2*		W
Comparison Instructions			
CMP *s1, s2*	*temp* ← *s1* - *s2*	W	W
Jump Instructions			
JMP *target*	jump to the address specified by *target*		
JL *target*	jump to *target* if SF == 1	R	
JG *target*	jump to *target* if SF == 0 and ZF == 0	R	R
Memory Instructions			
LD *s1, s2*	*s1*← M[*s2*]		
ST *s1, s2*	M[*s1*] ← *s2*		

Table C-2. HAL 180 instruction set.

Ben also designs a 6-stage pipelined implementation of the HAL 180. In this pipeline, the ALU takes three pipeline stages (E1, E2, and E3), and status flags are updated in stage E3. Table C-3 describes each stage, and Figure C-4 shows the datapath of this 6-stage pipelined architecture, highlighting the differences with a conventional MIPS pipeline. **Note that this implementation does not have any data bypass paths.**

Stage	Description
Fetch and Decode Stage (FD)	Fetch an instruction from the instruction memory, decode the instruction, and fetch the register values from the register file. The status flag checking for conditional jumps is also done in this stage.
Execute Stage 1 (E1)	The first stage of the execution phase. Generate partial results and store them in the pipeline registers.
Execute Stage 2 (E2)	The second stage of the execution phase. Generate partial results and store them in the pipeline registers.
Execute Stage 3 (E3)	The final stage of the execution phase. Final results are generated and flag registers get updated if necessary.
Memory Stage (M)	Perform load/store from/to the data memory if necessary.
Writeback Stage (WB)	Write to the register file if necessary.

Table C-3. HAL 180 pipeline stages.

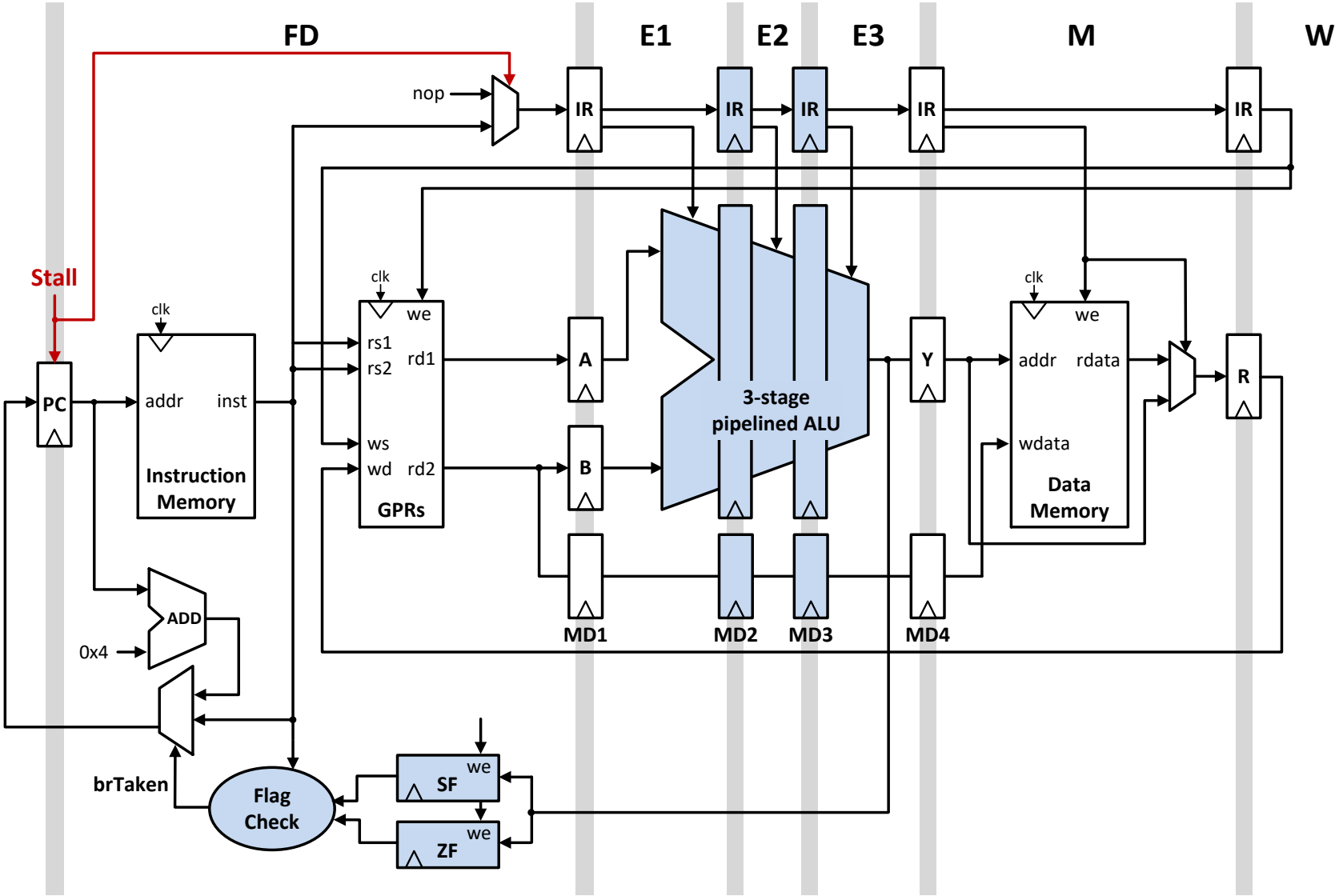


Figure M5.4-A. HAL 180 6-Stage pipelined implementation.

Problem M5.4.A

Write the HAL 180 assembly for the following program. For maximum credit, use the minimum number of comparison and jump instructions.

```
if (a < b) {
    c = c XOR b;
} else if (a > b) {
    c = c XOR a;
} else {
    c = 0;
}
a = 0;
b = 0;
```

Assume variables `a`, `b`, and `c` are stored in registers **R1, R2, and R3** respectively.

CMP R1, R2

Problem M5.4.B

Ben's HAL 180 6-stage pipeline (Figure M5.4-A) stalls to avoid data hazards through registers, but does not yet handle hazards due to status flags. To illustrate why this is problematic, consider the following instruction sequence:

```
I0:          ADD   R1, R2
I1:          JG    _L2
I2:          XOR   R1, R3
I3:          JL    _L2
I4:    _L1:  SUB   R1, R2
I5:    _L2:  ADD   R3, R1
```

Assume that when the program start, R1 = -1, R2 = -2, R3 = -3, and all the status flags are zero. Fill out the following instruction flow diagram to incur the minimum amount of stalls while maintaining correct operation (i.e., use stalls to respect both data and status flag dependences). Use "X"s to denote pipeline bubbles.

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9
FD	I0	I1								
E1		I0								
E2										
E3										
M										
W										

Problem M5.4.C

Let's fix Ben's implementation by extending the existing stall control signal, which already works for register hazards, to also stall on status flag hazards.

First, derive the stall conditions for the different jumps: JMP_{stall}, JL_{stall}, and JG_{stall}. Use $Opcode_X(Y)$ to indicate the condition when the instruction in X stage is Y. Y can be a specific instruction or an instruction class (see Table C-2). For example:

$Opcode_{FD}(JG)$:	if the instruction in the FD stage is a JG instruction.
$Opcode_{E1}(Logic)$:	if the instruction in the E1 stage belongs to the logical instruction class (e.g. OR).
$Opcode_{E2}(CMP\|Arith)$:	if the instruction in the E2 stage is a CMP instruction or belongs to the arithmetic instruction class.

JMP_{stall} =

JG_{stall} =

JL_{stall} =

Finally, write down the new stall signal (stall') by using the old stall signal (stall) and stall conditions you derive.

stall' =

Problem M5.4.D

Does this 6-stage pipeline add more challenges to precise exception handling? If so, please explain.

Problem M5.5: Pipelined Cache Access

This problem requires the knowledge of Lecture 3. Please, review it before answering the following questions. You may also want to take a look at pipeline lectures if you do not feel comfortable with the topic.

Problem M5.5.A

Ben Bitdiddle is designing a five-stage pipelined MIPS processor with separate 32 KB direct-mapped primary instruction and data caches. He runs simulations on his preliminary design, and he discovers that a cache access is on the critical path in his machine. After remembering that pipelining his processor helped to improve the machine's performance, he decides to try applying the same idea to caches. Ben breaks each cache access into three stages in order to reduce his cycle time. In the first stage the address is decoded. In the second stage the tag and data memory arrays are accessed; for cache reads, the data is available by the end of this stage. However, the tag still has to be checked—this is done in the third stage.

After pipelining the instruction and data caches, Ben's datapath design looks as follows:

I-Cache Address Decode	I-Cache Array Access	I-Cache Tag Check	Instruction Decode & Register Fetch	Execute	D-Cache Address Decode	D-Cache Array Access	D-Cache Tag Check	Write-back

Alyssa P. Hacker examines Ben's design and points out that the third and fourth stages can be combined, so that the instruction cache tag check occurs in parallel with instruction decoding and register file read access. If Ben implements her suggestion, what must the processor do in the event of an instruction cache tag mismatch? Can Ben do the same thing with load instructions by combining the data cache tag check stage with the write-back stage? Why or why not?

Problem M5.5.B

Alyssa also notes that Ben's current design is flawed, as using three stages for a data cache access won't allow writes to memory to be handled correctly. She argues that Ben either needs to add a fourth stage or figure out another way to handle writes. What problem would be encountered on a data write? What can Ben do to keep a three-stage pipeline for the data cache?

Problem M5.5.C

With help from Alyssa, Ben streamlines his design to consist of eight stages (the handling of data writes is not shown):

I-Cache Address Decode	I-Cache Array Access	I-Cache Tag Check, Instruction Decode & Register Fetch	Execute	D-Cache Address Decode	D-Cache Array Access	D-Cache Tag Check	Write-Back

Both the instruction and data caches are still direct-mapped. Would this scheme still work with a set-associative instruction cache? Why or why not? Would it work with a set-associative data cache? Why or why not?

Problem M5.5.D

After running additional simulations, Ben realizes that pipelining the caches was not entirely beneficial, as now the cache access latency has increased. If conditional branch instructions resolve in the Execute stage, how many cycles is the processor's branch delay?

Problem M5.5.E

Assume that Ben's datapath is fully-bypassed. When a load is executed, the data becomes available at the end of the D-cache Array Access stage. However, the tag has not yet been checked, so it is unknown whether the data is correct. If the load data is bypassed immediately, before the tag check occurs, then the instruction that depends on the load may execute with incorrect data. How can an interlock in the Instruction Decode stage solve this problem? How many cycles is the load delay using this scheme (assuming a cache hit)?

Problem M5.5.F

Alyssa proposes an alternative to using an interlock. She tells Ben to allow the load data to be bypassed from the end of the D-Cache Array Access stage, so that the dependent instruction can execute while the tag check is being performed. If there is a tag mismatch, the processor will wait for the correct data to be brought into the cache; then it will re-execute the load and all of the instructions behind it in the pipeline before continuing with the rest of the program. What processor state needs to be saved in order to implement this scheme? What additional steps need to be taken in the pipeline? Assume that a **DataReady** signal is asserted when the load data is available in the cache, and is set to 0 when the processor restarts its execution (you don't have to worry about the control logic details of this signal). How many cycles is the load delay using this scheme (assuming a cache hit)?

Problem M5.5.G

Ben is worried about the increased latency of the caches, particularly the data cache, so Alyssa suggests that he add a small, unpipelined cache in parallel with the D-cache. This "fast-path" cache can be considered as another level in the memory hierarchy, with the exception that it will be accessed simultaneously with the "slow-path" three-stage pipelined cache. Thus, the slow-path cache will contain a superset of the data found in the fast-path cache. A read hit in the fast-path cache will result in the requested data being available after one cycle. In this situation, the simultaneous read request to the slow-path cache will be ignored. A write hit in the fast-path cache will result in the data being written in one cycle. The simultaneous write to the slow-path cache will proceed as normal, so that the data will be written to both caches. If a read miss occurs in the fast-path cache, then the simultaneous read request to the slow-path cache will continue to be processed—if a read miss occurs in the slow-path cache, then the next level of the memory hierarchy will be accessed. The requested data will be placed in both the fast-path and slow-path caches. If a write miss occurs in the fast-path cache, then the simultaneous write to the slow-path cache will continue to be processed as normal. The fast-path cache uses a no-write allocate policy, meaning that on a write miss, the cache will remain unchanged—only the slow-path cache will be modified.

Ben's new pipeline design looks as follows after implementing Alyssa's suggestion:

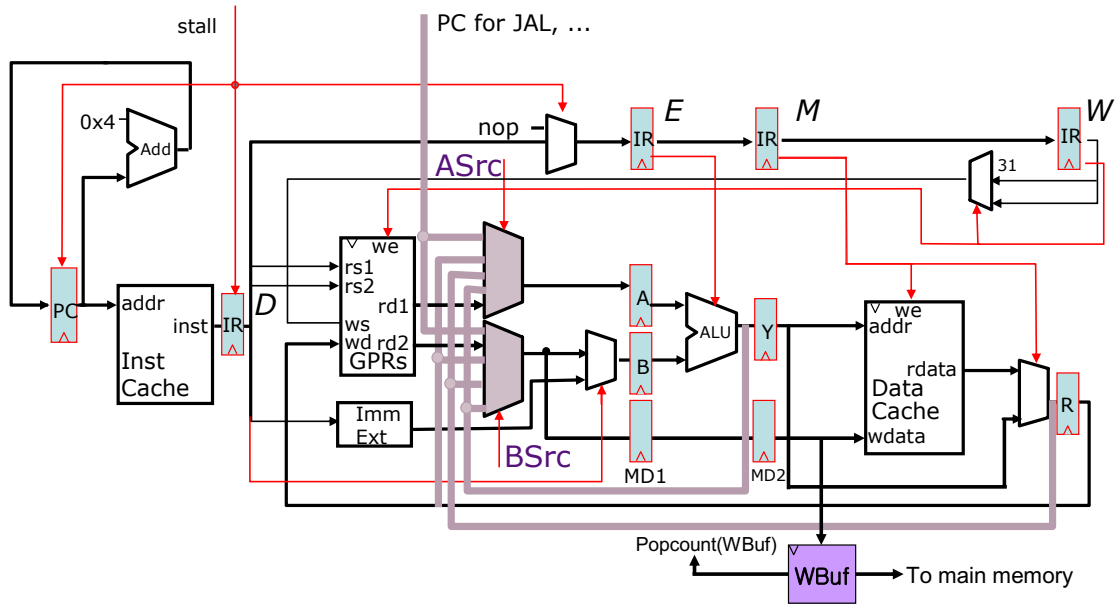
I-Cache Address Decode	I-Cache Array Access	I-Cache Tag Check, Instruction Decode & Register Fetch	Execute	Fast-Path D-Cache Access and Tag Check & Slow Path D-Cache Address Decode	Slow-Path D-Cache Array Access	Slow-Path D-Cache Tag Check	Write-Back

The number of processor pipeline stages is still eight, even with the addition of the fast-path cache. Since the processor pipeline is still eight stages, what is the benefit of using a fast-path cache? Give an example of an instruction sequence and state how many cycles are saved if the fast-path cache always hits.

Problem M5.6: Write Buffer for Data Cache (2005 Fall Part C)

In order to boost the performance of memory writes, Ben Bitdiddle has proposed to add a write buffer to our 5-stage fully-bypassed MIPS pipeline as shown below. Assuming a write-through/write no-allocate cache, every memory write request will be queued in the write buffer in the MEM stage, and the pipeline will continue execution without waiting for writes to be completed. A queued entry in the write buffer gets cleared only after the write operation completes, so the maximum number of outstanding memory writes is limited by the size of the write buffer.

Please answer the following questions.



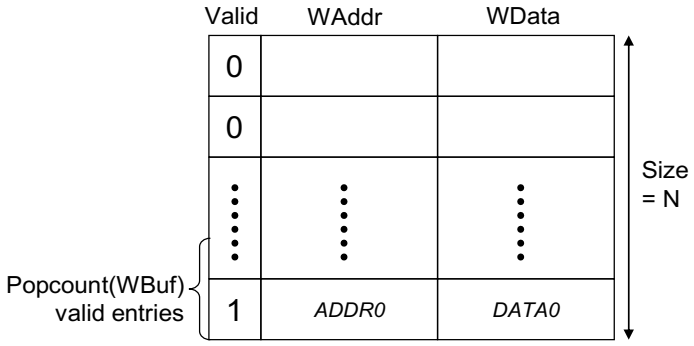
Problem M5.6.A

Ben wants to determine the size of the write buffer, so he runs benchmark X to get the observation below. What will be the average number of writes in flight (=the number of valid entries in the write buffer on average)?

1) The CPI of the benchmark is 2.
2) On average, one of every 20 instructions is a memory write.
3) Memory has a latency of 100 cycles, and is fully pipelined.

Problem M5.6.B

Based on the experiment in the previous question, Ben has added the write buffer with N entries to the pipeline. (Do not use your answer in M5.6A to replace N.) Now he wants to design a stall logic to prevent a write buffer overflow. The structure of the write buffer is shown in the figure below. `Popcount(WBuf)` gives the number of valid entries in the write buffer at any given moment.

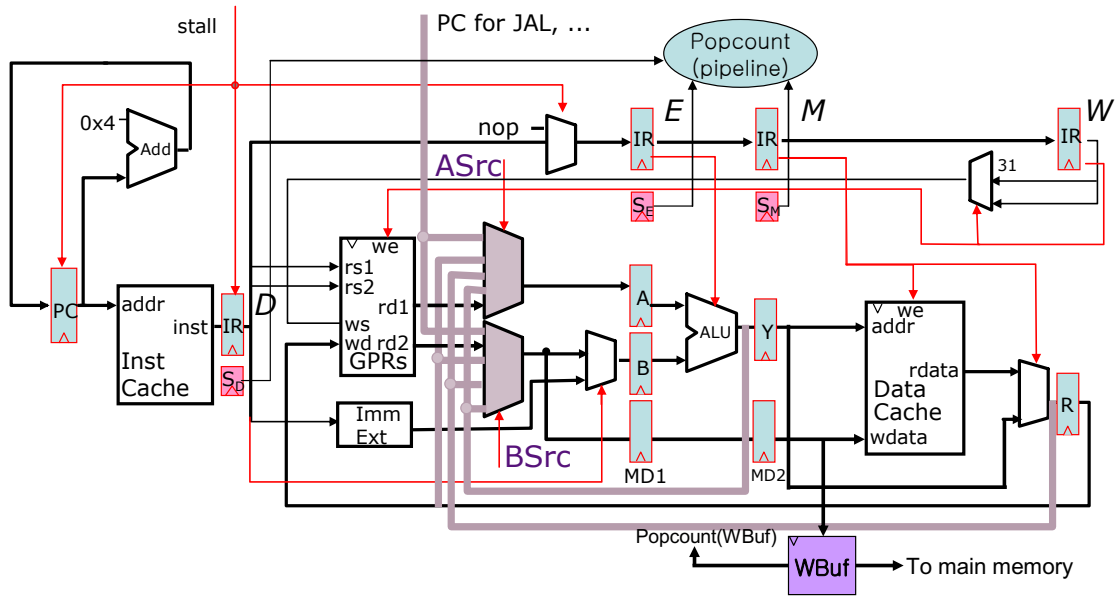


Please write down the stall condition to prevent write buffer overflows. You should derive the condition without assuming any modification of the given pipeline. You can use Boolean and arithmetic operations in your stall condition.

Stall =

Problem M5.6.C

In order to optimize the stall logic, Ben has decided to add a predecode bit to detect store instructions in the instruction cache (I-Cache). That is, now every entry in the I-Cache has a store bit associated with it, and it propagates through the pipeline with an S_{stage} bit added to each pipeline register (except the one between MEM and WB stages) as shown below. `Popcount(Pipeline)` gives the number of store instructions that are in flight (= number of S_{stage} bits set to 1).



How will this optimization change the stall condition, if at all?

Stall =

Problem M5.7: Instruction Pipelining (Spring 2016 Quiz 1, Part C)

This problem requires the knowledge of Handout #8 (LMIPS) and Lecture 6 and 7. Please, read these materials before answering the following questions.

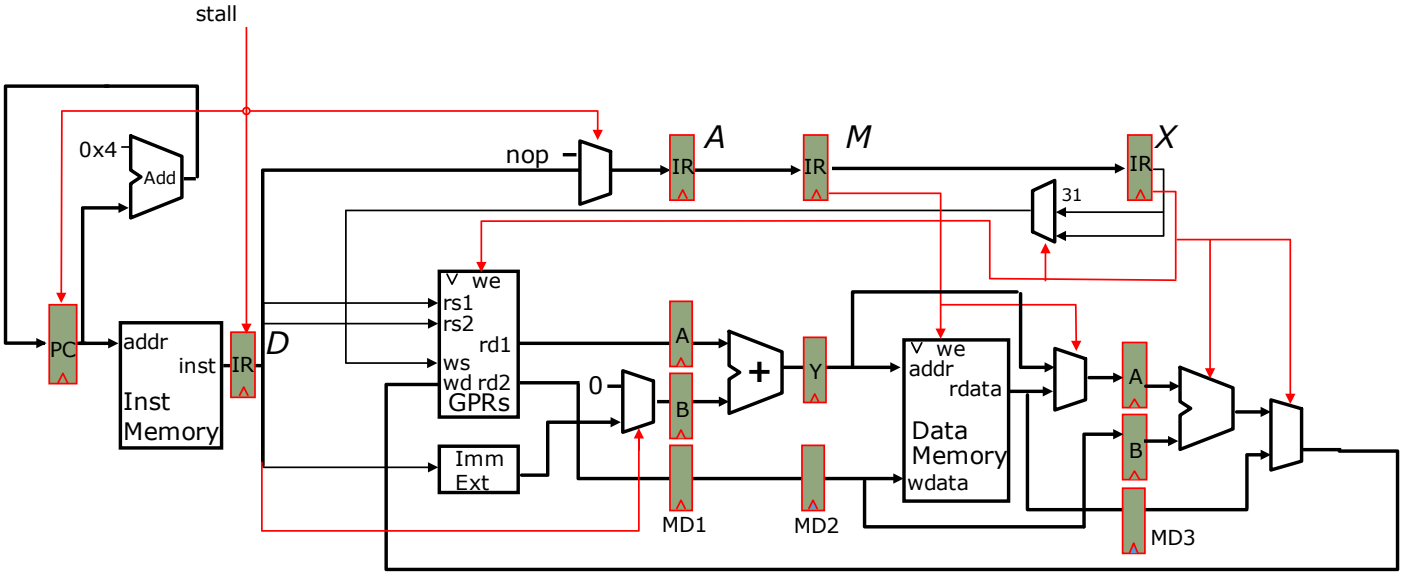
Consider the following MIPS code sequence:

```
I1    LW    R1, 0(R3)
I2    XOR   R1, R1, R4
I3    MUL   R2, R1, R4
I4    LW    R4, 5(R2)
I5    XOR   R4, R4, R5
I6    SW    R2, 0(R3)
```

Problem M5.7.A

Assume the classic 5-stage MIPS pipeline as discussed in lecture, with **full bypassing** and correct stall logic. Which instructions in the above sequence would have to stall?

Ben is unhappy with the performance of the classic 5-stage MIPS pipeline discussed in 6.823 lectures. Ben uses the L-MIPS ISA, presented in the L-MIPS handout, and pipelines the single-cycle L-MIPS datapath in the handout as shown in the figure below. This is also a 5-stage pipeline, with the following stages: instruction fetch (F), instruction decode and register file fetch (D), address generation (A), memory access (M), and execute + write-back (X) stages. **We will ignore branches and jumps for all following questions**.



Problem M5.7.B

Using the new class of Load-ALU instructions available in L-MIPS, rewrite the assembly sequence to produce a code sequence with minimum number of instructions. Do not change the order of any operations as you do this.

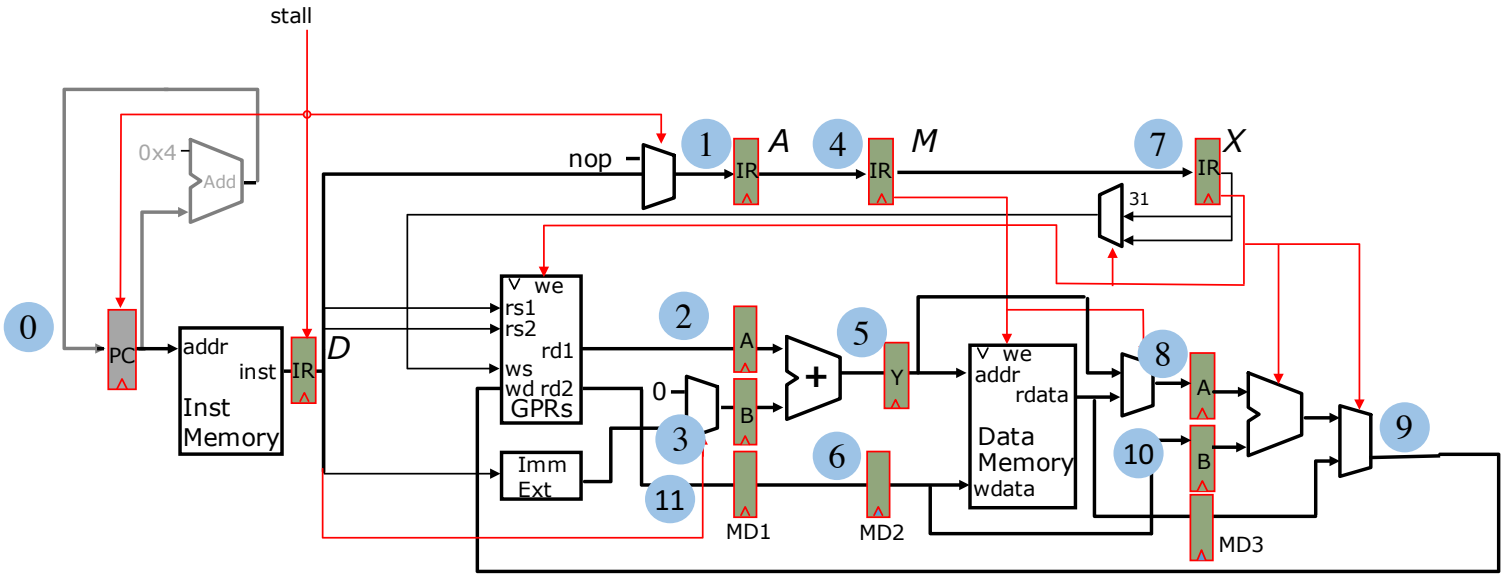
Problem M5.7.C

Complete the instruction flow diagram for the new sequence of instructions for Ben's pipelined L-MIPS processor. **Assume no bypassing** and correct stall logic. (In case you need it, page 18 has an extra/scratch instruction flow diagram.)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
I1	F	D	A	M	X														
I2																			
I3																			
I4																			
I5																			
I6																			
I7																			
I8																			

Problem M5.7.D

Ben wants to improve performance by adding bypass paths to his pipeline. Help Ben by indicating which locations he needs to insert bypass multiplexers. **Ignore any bypasses needed for control-flow instructions**.



From	To
9	8

From	To

Problem M5.7.E

Complete the instruction flow diagram for the new sequence of instructions for the L-MIPS pipeline. **Assume full bypassing** and correct stall logic this time. Use arrows to show forwarding of values from one stage to another. (In case you need it, page 18 has an extra/scratch instruction flow diagram.)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
I1	F	D	A	M	X														
I2																			
I3																			
I4																			
I5																			
I6																			
I7																			
I8																			

Problem M5.7.F

Is it possible to reorder the instructions in your code sequence (without affecting correctness) to improve performance in the fully-bypassed L-MIPS pipeline? If so, give the reordered code sequence and explain why. Otherwise, briefly explain why this is not possible.

Problem M6.1: Complex Pipelining Dependencies

Consider the following instruction sequence. An equivalent sequence of C-like pseudocode is also provided.

```
I1:  L.D     F1, 0 (R1)        ;    F1 = *r1;
I2:  MUL.D   F2, F0, F2        ;    F2 = F0*F2;
I3:  ADD.D   F1, F2, F2        ;    F1 = F2 + F2;
I4:  L.D     F2, 0 (R2)        ;    F2 = *r2;
I5:  ADD.D   F3, F1, F2        ;    F3 = F1 + F2;
I6:  S.D     F3, 0 (R3)        ;    *r3 = F3;
……
```

Fill out the table below to identify all Read-After-Write (RAW), Write-After-Read (WAR), and Write-After-Write (WAW) dependencies in the above sequence. Do not worry about memory dependencies for this question. The dependency between I2 and I3 is already filled in for you.

Current Instruction \ **Earlier (Older) Instruction**	I1	I2	I3	I4	I5	I6
I1	-					
I2		-				
I3		RAW	-			
I4				-		
I5					-	
I6						-

Problem M6.2: Out-of-order Scheduling

Ben Bitdiddle is adding a floating-point unit to the basic MIPS pipeline. He has patterned the design after the IBM 360/91's floating-point unit. His FPU has one adder, one multiplier, and one load/store unit. The *adder* has a four-cycle latency and is fully pipelined. The *multiplier* has a fifteen-cycle latency and is fully pipelined. Assume that loads and stores take 1 cycle (plus one cycle for the write-back stage for loads) and that we have perfect branch prediction.

There are 4 floating-point registers, **`F0-F3`**. These are separate from the integer registers. There is a single write-back port to each register file. In the case of a write-back conflict, the older instruction writes back first. Floating-point instructions (and loads writing floating point registers) must spend one cycle in the write-back stage before their result can be used. Integer results are available for bypass the next cycle after issue.

Ben is now deciding whether to go with (a) in-order issue using a scoreboard, (b) out-of-order issue, or (c) out-of-order issue with register renaming. His favorite benchmark is this DAXPY loop central to Gaussian elimination (Hennessy and Patterson, 291). The following code implements the operation Y=aX+Y for a vector of length 100. Initially R1 contains the base address for X, R2 contains the base address for Y, and F0 contains `a`. Your job is to evaluate the performance of the three scheduling alternatives on this loop.

```
   loop:
I1         L.D     F2, 0(R1)    ;load X(i)
I2         MUL.D   F1, F2, F0   ;multiply a*X(i)
I3         L.D     F3, 0(R2)    ;load Y(i)
I4         ADD.D   F3, F1, F3   ;add a*X(i)+Y(i)
I5         S.D     F3, 0(R2)    ;store Y(i)
I6         DADDUI  R1, R1, 8    ;increment X index
I7         DADDUI  R2, R2, 8    ;increment Y index
I8         DSGTUI  R3, R1, 800  ;test if done
I9         BEQZ    R3, loop     ;loop if not done
```

Problem M6.2.A **In-order using a scoreboard**

Fill in the scoreboard in table M6.2-1 to simulate the execution of one iteration of the loop for in-order issue using a scoreboard. Keep in mind that, in this scheme, no instruction is issued that has a WAW hazard with any previous instruction that has not written back (as mentioned in the lecture slides). Recall the WB stage is only relevant for FP instructions (integer instructions can forward results). You may use ellipses in the table to represent the passage of time (to compress repetitive lines).

In steady state, how many cycles does each iteration of the loop take? What is the bottleneck?

Problem M6.2.B Out-of-order

Now consider a single-issue out-of-order implementation. In this scheme, the issue stage buffer holds multiple instructions waiting to issue. The decode stage can add up to one instruction per cycle to the issue buffer. The decode stage adds an instruction to the issue buffer if there is space and if the instruction does not have a WAR hazard with any previous instruction that has not issued or a WAW hazard with any previous instruction that has not written back. Assume you have an infinitely large issue buffer. Assume only one instruction can be dispatched from the issue buffer at a time.

Table M6.2-2 represents the execution of one iteration of the loop ***in steady state***. Fill in the cycle numbers for the cycles at which each instruction issues and writes back. The first row has been filled out for you already; please complete the rest of the table. Note that the order of instructions listed is not necessarily the issue order. We define cycle 0 as the time at which instruction I_1 is issued.

Draw arrows for the RAW, WAR, and WAW dependencies that are involved in the *critical path* of the loop in table M2.1-2. In steady state, how many cycles does each iteration of the loop take?

Problem M6.2.C Register Renaming

The number of registers specified in an ISA limits the maximum number of instructions that can be in the pipeline. This question studies register renaming to solve this problem. In this question, we will consider an ideal case where we have unlimited hardware resources for renaming registers.

Table M6.2-3 shows instructions from our benchmark for two iterations using the same format as Table M6.2-2. First, fill in the new register names for each instruction, where applicable. Since we have an infinite supply of register names, you should use a new register name each time a register is written (T0, T1, T2, etc). Keep in mind that after a register has been renamed, subsequent instructions that refer to that register need to refer instead to the new register name. You may find it helpful to create a rename table. Rename both integer and floating-point instructions.

Next, fill in the cycle numbers for the cycles at which each instruction issues and writes back. The decode stage can add up to one instruction per cycle to the re-order buffer (ROB). Assume that instruction I_2 was decoded in cycle 0, and cannot be issued until cycle 2. Also assume that you have an infinitely large ROB.

In steady state, how many cycles does each iteration of the loop take? What is the performance bottleneck?

Instr. Issued	Time (cycles)	Functional Unit Status					Floating Point Registers Reserved for Writes
		Int	Load (1)	Adder (4)	Multiplier (15)	WB	
I_1	0		F2				F2
	1					F2	F2
I_2	2				F1		F1

Table M6.2-1

	Time			Op	Dest	Src1	Src2
	Decode → Issue	Issued	WB				
I_1	-1	0	1	L.D	F2	R1	
I_2				MUL.D	F1	F2	F0
I_3				L.D	F3	R2	
I_4				ADD.D	F3	F1	F3
I_5				S.D		R2	F3
I_6				DADDUI	R1	R1	
I_7				DADDUI	R2	R2	
I_8				DSGTUI	R3	R1	
I_9				BEQZ		R3	

Table M6.2-2

	Time			Op	Dest	Src1	Src2
	Decode → Issue	Issued	WB				
I_1	-1	0	1	L.D	T0	R1	
I_2				MUL.D	T1	t0	F0
I_3				L.D	T2	R2	
I_4				ADD.D	T3		
I_5				S.D			
I_6				DADDUI			
I_7				DADDUI			
I_8				DSGTUI			
I_9				BEQZ			
I_1				L.D			
I_2				MUL.D			
I_3				L.D			
I_4				ADD.D			
I_5				S.D			
I_6				DADDUI			
I_7				DADDUI			
I_8				DSGTUI			
I_9				BEQZ			

Table M6.2-3

Problem M6.3: Out-of-Order Scheduling

This problem deals with an out-of-order single-issue processor that is based on the basic MIPS pipeline and has floating-point units. The FPU has one adder, one multiplier, and one load/store unit. The adder has a two-cycle latency and is fully pipelined. The multiplier has a ten-cycle latency and is fully pipelined. Assume that loads and stores take 1 cycle (plus one cycle for write-back for loads).

There are 4 floating-point registers, `F0-F3`. These are separate from the integer registers. There is a single write-back port to each register file. In the case of a write-back conflict, the older instruction writes back first. Floating-point instructions (including loads writing floating point registers) must spend one cycle in the write-back stage before their result can be used. Integer results are available for bypass the next cycle after issue.

To maximize number of instructions that can be in the pipeline, register renaming is used. The decode stage can add up to one instruction per cycle to the re-order buffer (ROB).

The instructions are committed in order and only one instruction may be committed per cycle. The earliest time an instruction can be committed is one cycle after write back.

For the following questions, we will evaluate the performance of the code segment in Figure M6.3-A.

I_1	`L.D`	`F1, 5(R2)`
I_2	`MUL.D`	`F2, F1, F0`
I_3	`ADD.D`	`F3, F2, F0`
I_4	`ADDI`	`R2, R2, 8`
I_5	`L.D`	`F1, 5(R2)`
I_6	`MUL.D`	`F2, F1, F1`
I_7	`ADD.D`	`F2, F2, F3`

Figure M6.3-A

Problem M6.3.A

For this question, we will consider an ideal case where we have unlimited hardware resources for renaming registers. Assume that you have an **infinitely large ROB**.

Your job is to complete Table M6.3-1. Fill in the cycle numbers when each instruction enters the ROB, issues, writes back, and commits. Also fill in the new register names for each instruction, where applicable. Since we have an infinite supply of register names, you should use a new register name each time a register is written (T0, T1, T2, … etc). Keep in mind that after a register has been renamed, subsequent instructions that refer to that register need to refer instead to the new register name.

	Time				OP	Dest	Src1	Src2
	Decode → ROB	Issued	WB	Committed				
I_1	-1	0	1	2	L.D	T0	R2	-
I_2	0	2	12	13	MUL.D	T1	T0	F0
I_3	1				ADD.D			
I_4					ADDI			-
I_5					L.D			-
I_6					MUL.D			
I_7					ADD.D			

Table M6.3-1

Problem M6.3.B

For this question, assume that you have a **two-entry ROB**. An ROB entry can be reused **one cycle** after the instruction using it commits.

Your job is to complete Table M6.3-2. Fill in the cycle numbers when each instruction enters the ROB, issues, writes back, and commits. Also fill in the new register names for each instruction, where applicable.

	Time				OP	Dest	Src1	Src2
	Decode → ROB	Issued	WB	Committed				
I_1	-1	0	1	2	L.D	T0	R2	-
I_2	0	2	12	13	MUL.D	T1	T0	F0
I_3	3				ADD.D			
I_4					ADDI			-
I_5					L.D			-
I_6					MUL.D			
I_7					ADD.D			

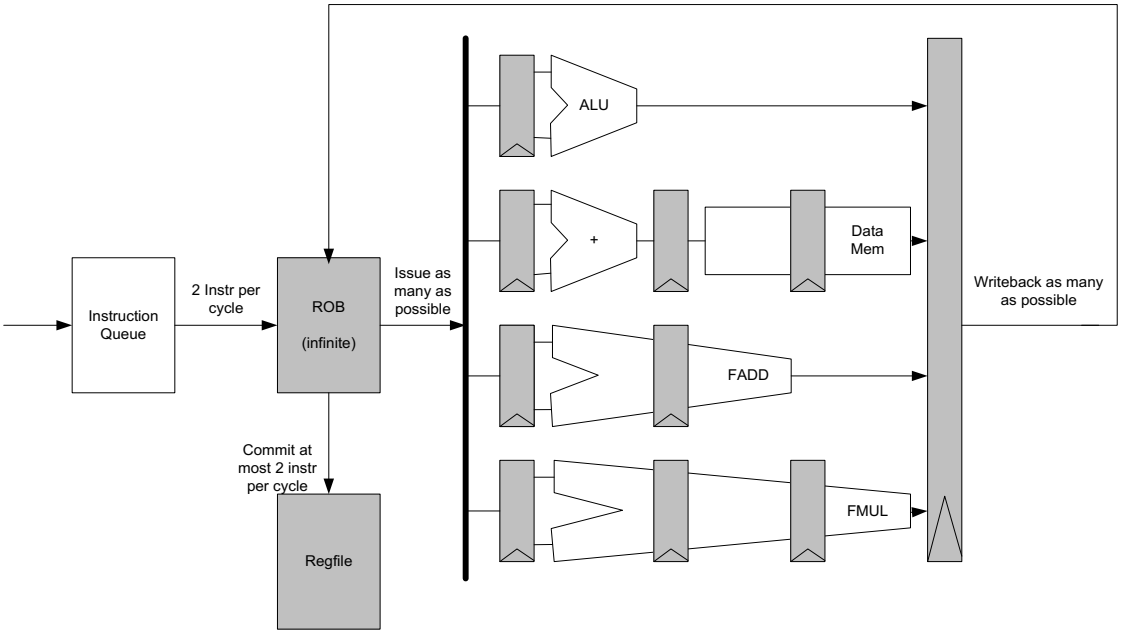
Table M6.3-2

Problem M6.4: Superscalar Processor

Consider the out-of-order, superscalar CPU shown in the diagram. It has the following features.

- Four fully-pipelined functional units: ALU, MEM, FADD, FMUL
- Instruction Fetch and Decode Unit that renames and sends 2 instructions per cycle to the ROB (assume perfect branch prediction and no cache misses)
- An unbounded length Reorder Buffer that can perform the following operations on every cycle.
  - Accept two instructions from the Instruction Fetch and Decode Unit
  - Dispatch an instruction to each functional unit including Data Memory
  - Let Write-back update an unlimited number of entries
  - Commit up to 2 instructions in-order
- There is no bypassing or short circuiting. For example, data entering the ROB cannot be passed on to the functional units or committed in the same cycle.

.



Now consider the execution of the following program on this machine using:

```
I1        loop:   LD F2, 0(R2)
I2                LD F3, 0(R3)
I3                FMUL F4, F2, F3
I4                LD F2, 4(R2)
I5                LD F3, 4(R3)
I6                FMUL F5, F2, F3
I7                FMUL F6, F4, F5
I8                FADD F4, F4, F5
I9                FMUL F6, F4, F5
I10               FADD F1, F1, F6
I11               ADD R2, R2, 8
I12               ADD R3, R3, 8
I13               ADD R4, R4, -1
I14               BNEZ R4, loop
```

Problem M6.4.A

Fill in the renaming tags in the following two tables for the execution of instructions I1 to I10. *Tags should not be reused.*

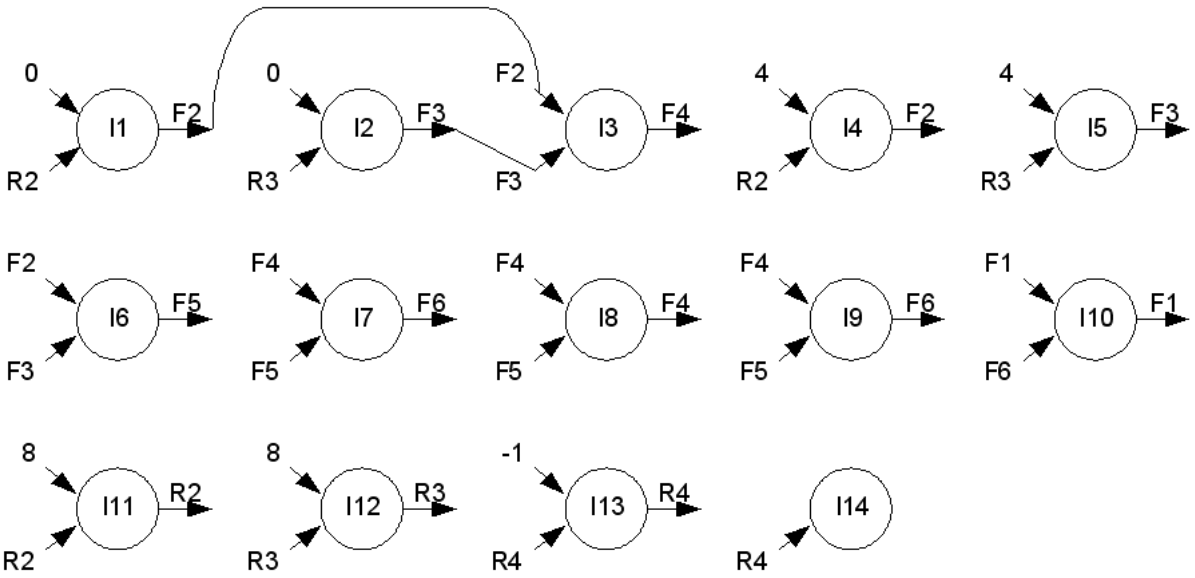
Instr #	**Instruction**	**Dest**	**Src1**	**Src2**
I1	LD F2, 0(R2)	T1	R2	0
I2	LD F3, 0(R3)	T2	R3	0
I3	FMUL F4, F2, F3			
I4	LD F2, 4(R2)		R2	4
I5	LD F3, 4(R3)		R3	4
I6	FMUL F5, F2, F3			
I7	FMUL F6, F4, F5			
I8	FADD F4, F4, F5			
I9	FMUL F6, F4, F5			
I10	FADD F1, F1, F6		F1	

Renaming table

	I1	I2	I3	I4	I5	I6	I7	I8	I9	I10
R2										
R3										
F1										
F2	T1									
F3		T2								
F4										
F5										
F6										

Problem M6.4.B

Consider the execution of ***one*** iteration of the loop (I1 to I14). In the following diagram draw the data dependencies between the instructions after register renaming



Problem M6.4.C

The attached table is a data structure to record the times when some activity takes place in the ROB. For example, one column records the time when an instruction enters ROB, while the last two columns record, respectively, the time when an instruction is dispatched to the FU's and the time when results are written back to the ROB. This data structure has been designed to test your understanding of how a Superscalar machine functions.

Fill in the blanks in the last two columns up to slot T13 (you may use the source columns for book keeping).

Slot	Instruction	Cycle instruction entered ROB	Argument 1		Argument 2		dst	***Cycle dispatched***	***Cycle written back to ROB***
			src1	cycle available	Src2	cycle available	dst reg		
T1	**LD F2, 0(R2)**	1	C	1	R2	1	F2	2	6
T2	**LD F3, 0(R3)**	1	C	1	R3	1	F3	3	7
T3	**FMUL F4, F2, F3**	2			F3	7	F4		
T4	**LD F2, 4(R2)**	2	C	2	R2		F2		
T5	**LD F3, 4(R3)**	3	C	3	R3		F3		
T6	**FMUL F5, F2, F3**	3					F5		
T7	**FMUL F6, F4, F5**	4					F6		
T8	**FADD F4, F4, F5**	4					F4		
T9	**FMUL F6, F4, F5**	5					F6		
T10	**FADD F1, F1, F6**	5					F1		
T11	**ADD R2, R2, 8**	6	R2	6	C	6	R2		
T12	**ADD R3, R3, 8**	6	R3	6	C	6	R3		
T13	**ADD R4, R4, -1**	7	R4	7	C	7	R4		
T14	**BNEZ R4, loop**	7			C	Loop			
T15	**LD F2, 0(R2)**	8	C	8			F2	10	14
T16	**LD F3, 0(R3)**	8	C	8			F3	11	15
T17	**FMUL F4, F2, F3**	9					F4		
T18	**LD F2, 4(R2)**	9	C	9			F2		
T19	**LD F3, 4(R3)**	10	C	10			F3		
T20	**FMUL F5, F2, F3**	10					F5		
T21	**FMUL F6, F4, F5**	11					F6		
T22	**FADD F4, F4, F5**	11					F4		
T23	**FMUL F6, F4, F5**	12					F6		
T24	**FADD F1, F1, F6**	12					F1		
T25	**ADD R2, R2, 8**	13			C	13	R2		
T26	**ADD R3, R3, 8**	13			C	13	R3		
T27	**ADD R4, R4, -1**	14			C	14	R4		
T28	**BNEZ R4, loop**	14			C	Loop			

Problem M6.4.D

Identify the instructions along the longest latency path in completing this iteration of the loop (up to instruction 13). Suppose we consider an instruction to have executed when its result is available in the ROB. How many cycles does this iteration take to execute?

Problem M6.4.E

Do you expect the same behavior, i.e., the same dependencies and the same number of cycles, for the next iteration? (You may use the slots from T15 onwards in the attached diagram for bookkeeping to answer this question). Please give a simple reason why the behavior may repeat, or identify a resource bottleneck or dependency that may preclude the repetition of the behavior.

Problem M6.4.F

Can you improve the performance by adding at most one additional memory port and an FP Multiplier? Explain briefly.

Yes / No

Problem M6.4.G

What is the minimum number of cycles needed to execute a typical iteration of this loop if we keep the same latencies for all the units but are allowed to use as many FUs and memory ports and are allowed to fetch and commit as many instructions as we want.

Problem M6.5: Register Renaming and Static vs. Dynamic Scheduling

The following MIPS code calculates the floating-point expression E = A * B + C * D, where the addresses of A, B, C, D, and E are stored in R1, R2, R3, R4, and R5, respectively:

```
L.S     F0, 0(R1)
L.S     F1, 0(R2)
MUL.S   F0, F0, F1
L.S     F2, 0(R3)
L.S     F3, 0(R4)
MUL.S   F2, F2, F3
ADD.S   F0, F0, F2
S.S     F0, 0(R5)
```

Problem M6.5.A — Simple Pipeline

Calculate the number of cycles this code sequence would take to execute (i.e., the number of cycles between the issue of the first load instruction and the issue of the final store, inclusive) on a simple in-order pipelined machine that has no bypassing. The datapath includes a load/store unit, a floating-point adder, and a floating-point multiplier. Assume that loads have a two-cycle latency, floating-point multiplication has a four-cycle latency and floating-point addition has a two-cycle latency. Write-back for floating-point registers takes one cycle. Also assume that all functional units are fully pipelined and ignore any write-back conflicts. Give the number of cycles between the issue of the first load instruction and the issue of the final store, inclusive.

Problem M6.5.B — Static Scheduling

Reorder the instructions in the code sequence to minimize the execution time. Show the new instruction sequence and give the number of cycles this sequence takes to execute on the simple in-order pipeline.

Problem M6.5.C — Fewer Registers

Rewrite the code sequence, but now using only two floating-point registers. Optimize for minimum run-time. You may need to use temporary memory locations to hold intermediate values (this process is called register-spilling when done by a compiler). List the code sequence and give the number of cycles it takes to execute.

Problem M6.5.D **Register renaming and dynamic scheduling**

Simulate the effect of running the original code on a single-issue machine with register renaming and out-of-order issue. Ignore structural hazards apart from the single instruction decode per cycle. Show how the code is executed and give the number of cycles required. Compare it with results from the optimized execution in M2.4.B.

Problem M6.5.E **Effect of Register Spills**

Now simulate the effect of running the code you wrote in M2.4.C on the single-issue machine with register renaming and out-of-order issue from M2.4.D. Compare the number of cycles required to execute the program. What are the differences in the program and/or architecture that change the number of cycles required to execute the program? You should assume that all load instructions before a store must issue before the store is issued, and load instructions after a store must wait for the store to issue.

Problem M6.6: Register Renaming Schemes

This problem requires the knowledge of Handout Out-of-Order Execution with ROB and Lectures 8 and 9. Please, read these materials before answering the following questions.

Future File Scheme

In order to eliminate the step of reading operands from the reorder buffer in the decode stage, we can insert a second register file into the processor shown in Figure M6.6-A, called the *future file*. The future file contains the most up-to-date speculatively-executed value for a register, while the primary register file contains committed values. Each entry in the future file has a valid bit. A summary of the operations is given below.

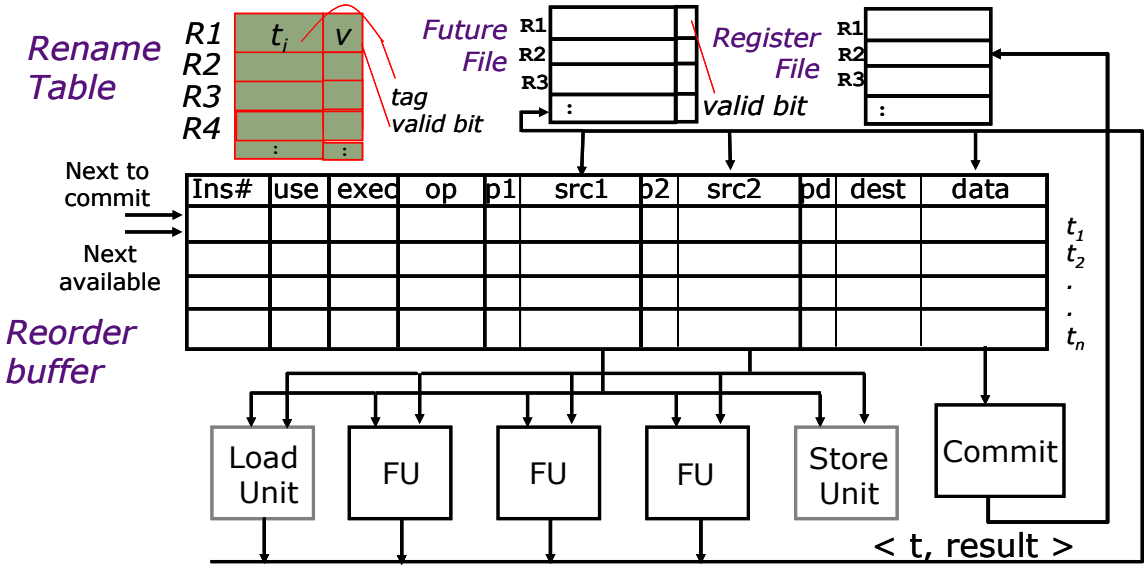


Figure M6.6-A

Only the decode and write-back stages have to change from the baseline implementation in Handout *Out-of-Order Execution with ROB* to implement the future file, as described below.

Decode: The Rename table, the register file, and the future file are read simultaneously. If the rename table has the valid bit set for an operand, then the value has not yet been produced and the tag will be used. Otherwise, if the future file has a valid bit set for its entry, then use the future file value. Otherwise, use the register file value. The instruction is assigned a slot in the ROB (the ROB index is this instruction's tag). If the instruction writes a register, its tag is written to the destination register entry in the rename table.

Write-Back: When an instruction completes execution, the result, if any, will be written back to the *data* field in the reorder buffer and the *pd* bit will be set. Additionally, any dependent instructions in the reorder buffer will receive the value. If the tag in the rename table for this register matches the tag of the result, the future file is written with the value and the valid bit on the rename table entry is cleared.

Problem M6.6.A **Finding Operands: Original ROB scheme**

Consider the original ROB scheme in Handout *Out-of-Order Execution with ROB*, and suppose the processor state is as given in Figure H4-A. Assume that the following three instructions enter the ROB simultaneously in a single cycle, and that no instruction commits or completes execution in this cycle. In the table below, write the contents of each instruction's source operand entries (either a register value or a tag t_1, t_2, etc., for both Src1 and Src2) and whether that entry came from the register file, the reorder buffer, the rename table or the instruction itself.

Instruction	Src1 value	Regfile, ROB, rename table, or instruction?	Src2 value	Regfile, ROB, rename table, or instruction?
`sub r5,r1,r3`				
`addi r6,r2,4`				
`andi r7,r4,3`				

Problem M6.6.B **Finding Operands: Future File Scheme**

In the future file scheme, explain why an instruction entering the ROB will never need to fetch either of its operands from the ROB.

Problem M6.6.C **Future File Operation**

Describe a situation in which an instruction result is written to the ROB but might not be written to the future file. Provide a simple code sequence to illustrate your answer.

Problem M7.1: Branch Prediction

This problem will investigate the effects of adding global history bits to a standard branch prediction mechanism. **In this problem assume that the MIPS ISA has no delay slots.**

Throughout this problem we will be working with the following program.

```
loop:
     LW   R4, 0(R3)
     ADDI R3, R3, 4
     SUBI R1, R1, 1
b1:
     BEQZ R4, b2
     ADDI R2, R2, 1
b2:
     BNEZ R1, loop
```

Assume the initial value of R1 is n (n>0).
Assume the initial value of R2 is 0 (R2 holds the result of the program).
Assume the initial value of R3 is p (a pointer to the beginning of an array of 32-bit integers).

All branch prediction schemes in this problem will be based on those covered in the lecture. We will be using a 2-bit predictor state machine, as shown below.

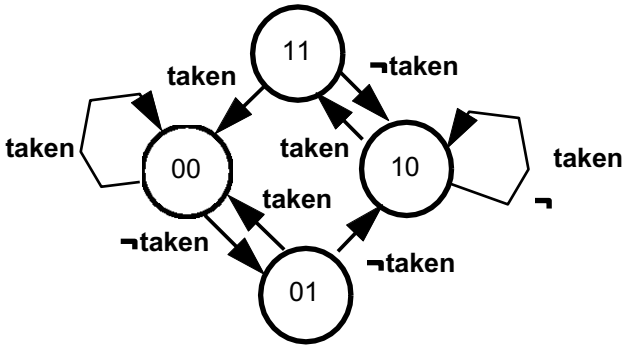


Figure M7.1-A: BP bits state diagram

In state 1X we will guess not taken. In state 0X we will guess taken.

Assume that b1 and b2 do not conflict in the BHT.

Problem M7.1.A **Program**

What does the program compute? That is, what does R2 contain when we exit the loop?

Problem M7.1.B **2-bit branch prediction**

Now we will investigate how well our standard 2-bit branch predictor performs. Assume the inputs to the program are n=8 and p[0] = 1, p[1] = 0, p[2] = 1, p[3] = 0,… etc. That is the array elements exhibit an alternating pattern of 1's and 0's. Fill out Table M7.1-1 (note that the first few lines are filled out for you). What is the number of mispredicts?

Table M7.1-1 contains an entry for every branch (either b1 or b2) that is executed. The Branch Predictor (BP) bits in the table are the bits from the BHT. For each branch, check the corresponding BP bits (indicated by the bold entries in the examples) to make a prediction, then update the BP bits in the following entry (indicated by the italic entries in the examples).

Problem M7.1.C **Branch prediction with one global history bit**

Now we add a global history bit to the branch predictor, as described in the lecture. Fill out Table M7.1-2, and again give the total number of mispredicts you get when running the program with the same inputs.

Problem M7.1.D **Branch prediction with two global history bits**

Now we add a second global history bit. Fill out Table M7.1-3. Again, compute the number of mispredicts you get for the same input.

Problem M7.1.E **Analysis**

Compare your results from problems M7.1.B, M7.1.C and M7.1.D. When do most of the mispredicts occur in each case (at the beginning, periodically, at the end, etc.)? What does this tell you about global history bits in general? For a large n, what prediction scheme will work best? Explain briefly.

Problem M7.1.F **Analysis II**

The input we worked with in this problem is quite regular. How would you expect things to change if the inputs were random (each array element were equally probable to be 0 or 1). Of the three branch predictors we looked at in this problem, which one will perform best for this type of input? Is your answer the same for large and small n?

What does this tell you about additional history bits: when are they useful and when do they hurt you?

System State		Branch Predictor		Branch Behavior	
PC	R3/R4	b1 bits	b2 bits	Predicted	Actual
b1	4/1	**10**	10	**N**	**N**
b2	4/1	*10*	**10**	**N**	**T**
b1	8/0	**10**	*11*	**N**	**T**
b2	8/0	*11*	**11**	**N**	**T**
b1	12/1	**11**	*00*		
b2	12/1				
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					

Table M7.1-1

System State			Branch Predictor				Behavior	
PC	R3/R4	history bit	b1 bits		b2 bits		Predicted	Actual
			set 0	set 1	set 0	set 1		
b1	4/1	**1**	10	**10**	10	10	**N**	**N**
b2	4/1	**0**	10	*10*	**10**	10	**N**	**T**
b1	8/0	**1**	10	**10**	*11*	10		
b2	8/0							
b1	12/1							
b2	12/1							
b1								
b2								
b1								
b2								
b1								
b2								
b1								
b2								
b1								
b2								
b1								
b2								
b1								
b2								
b1								
b2								
b1								
b2								

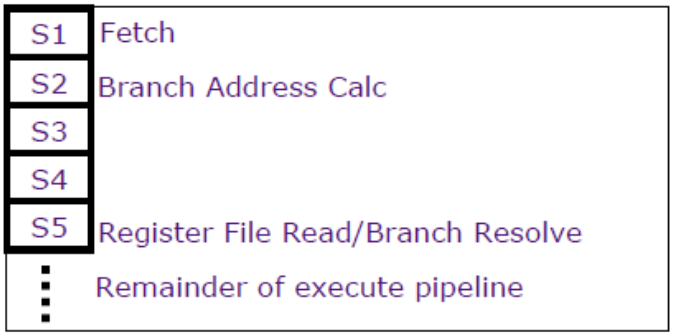
Table M7.1-2

System State			Branch Predictor								Behavior	
PC	R3/R4	history bits	b1 bits				b2 bits				Predicted	Actual
			set 00	set 01	set 10	set 11	set 00	set 01	set 10	set 11		
b1	4/1	**11**	10	10	10	**10**	10	10	10	10	**N**	**N**
b2	4/1	**01**	10	10	10	*10*	10	**10**	10	10	**N**	**T**
b1	8/0	**10**	10	10	**10**	10	10	*11*	10	10		
b2	8/0											
b1	12/1											
b2	12/1											
b1												
b2												
b1												
b2												
b1												
b2												
b1												
b2												
b1												
b2												
b1												
b2												
b1												
b2												
b1												
b2												
b1												
b2												

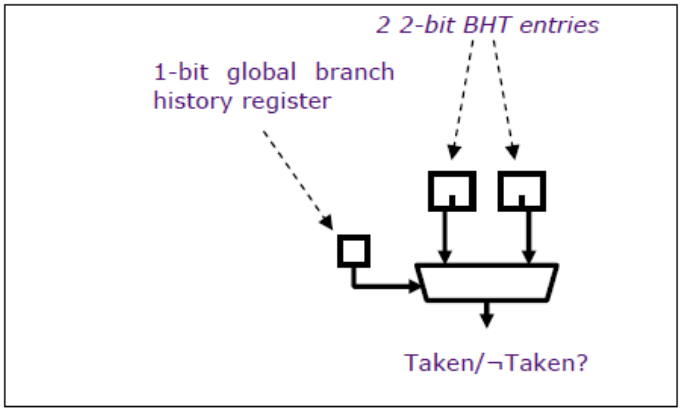
Table M7.1-3

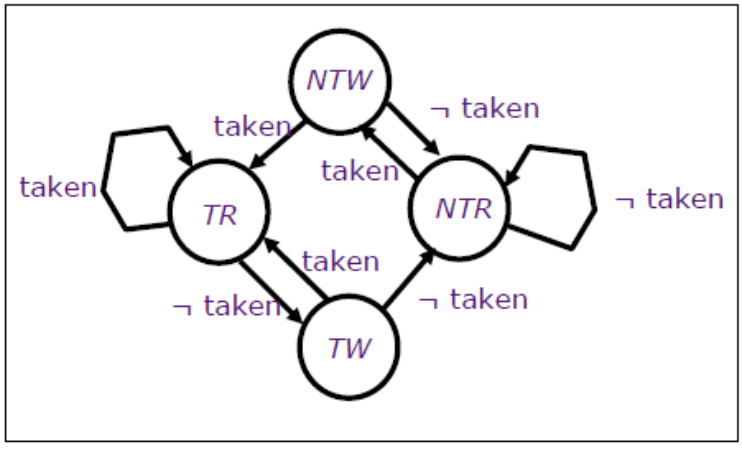
Problem M7.2: Branch Prediction

Consider a CPU with a pipeline pictured on the right. The first stage of the pipeline fetches the instruction. The **second stage** of the pipeline recognizes branch instructions and **performs branch prediction** using a BHT. If the branch is predicted to be taken, it forwards the decoded target of the branch to the first stage, and kills the instruction in the first stage. The **fifth stage** of the pipeline reads the registers and **resolves the correct target of the branch**. If the branch target was mispredicted, the correct target is forwarded to the first stage, and all instructions in between are killed. The remaining stages finish the computation of the instruction.



The processor uses a **single global history bit** to remember whether the last branch was taken or not. There is only **one line in the BHT**, so the address of the branch instruction is not used for selecting the proper table entry. Each entry in the table is labeled as TW for Take Wrong, TR for Take Right, NTW for do Not Take Wrong and NTR for do Not Take Right, as pictured below. The setup of the BHT predictor is illustrated on the right.





In this question we will study execution of the following loop. This processor has **no** branch delay slots. You should assume that branch at address 1 is **never** taken, and that the branch at address 5 is **always** taken.

Instruction Label	Address	Instruction
LOOP	1	BEQ R2, R5, NEXT
	2	ADD R4, R4, 1
	3	MULT R3, R3, R4
NEXT	4	MULT R2, R2, 3847
	5	BNEZ R4, LOOP
	6	NOP
	7	NOP
	8	NOP
	9	NOP
	10	NOP

You should also **disregard any possible structural hazards**. The processor always runs at full speed, and there are **no pipeline bubbles** (except for those created by the branches).

Problem M7.2.A

Now we study how well the history bit works, when it is being **updated by the fifth stage** of the processor. The fifth stage also updates the BHT based on the result of a branch. The same BHT entry that was used to make the original prediction is updated.

Please fill in the table below.

You should fetch a new instruction every cycle. You should fill in the *Branch Prediction* and the *Prediction Correct?* columns for branch instructions only (note that the branch prediction actually happens one cycle **after** the instruction is fetched). You should fill in the *Branch Predictor State* columns whenever they are updated. Please **circle the instructions which will be committed**.

The first three committing instructions fetched have been filled in for you. You should enter enough instructions to **add 8 more committing instructions**. You may not need all the rows in the table.

Cycle	Instruction Fetched	Branch Prediction	Prediction Correct?	Branch Predictor State		
				Branch History	Last Branch Taken Predictor	Last Branch Not Taken Predictor
0	-	-		T	TW	TW
1	1	T	N			
2	2					
3	4					
4	5	T	Y			
5	6			NT	NTR	
6	2					
7	3					
8						
9						
10						
11						
12						
13						
14						
15						
16						
17						
18						
19						
20						
21						
22						
23						
24						

Problem M7.2.B

Now we study how well the branch **history bit** works, when it **is being updated speculatively** by the second stage of the processor. If the branch is mispredicted, the fifth stage sets the branch history bit to the correct value. Finally, the fifth stage also updates the BHT based on the result of a branch. The same BHT entry that was used to make the original prediction is updated.

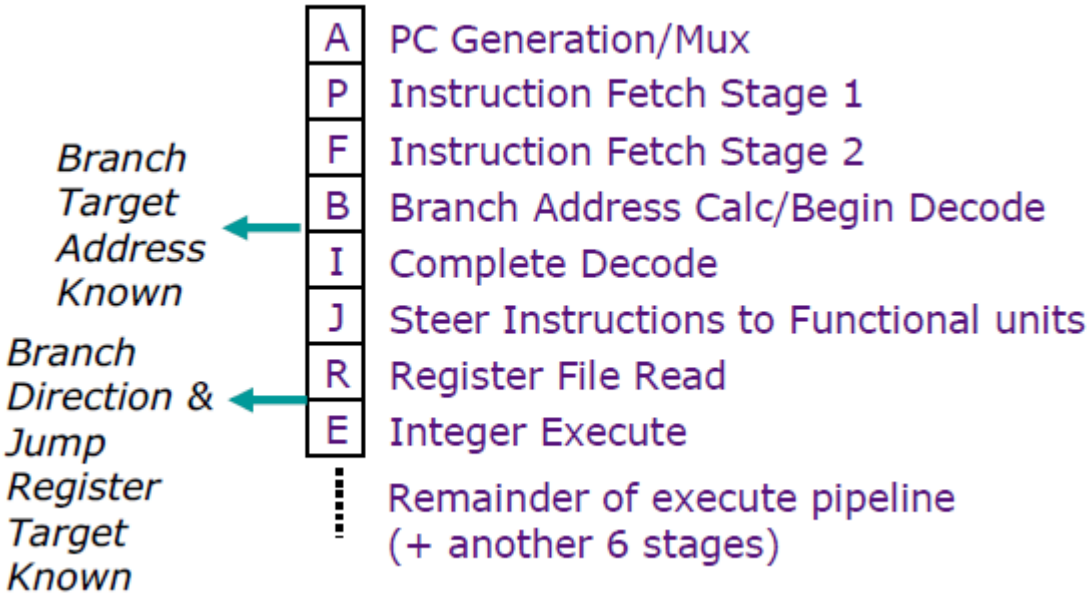
Please fill in the table below. The notation in the table should be same as in ***M7.2.A***.

The first three committing instructions fetched have been filled in for you. You should enter enough instructions to **add 8 more committing instructions**. You may not need all the rows in the table.

Cycle	Instruction Fetched	Branch Prediction	Prediction Correct?	Branch Predictor State		
				Branch History	Last Branch Taken Predictor	Last Branch Not Taken Predictor
0	-	-		T	TW	TW
1	1	T	N			
2	2			T		
3	4					
4	5	T	Y			
5	6			NT	NTR	
6	2					
7	3					
8						
9						
10						
11						
12						
13						
14						
15						
16						
17						
18						
19						
20						
21						
22						
23						
24						

Problem M7.3: Branch Prediction

Consider the fetch pipeline of UltraSparc-III processor. In this part, we evaluate the impact of branch prediction on the processor's performance. There are no branch delay slots.



Here is a table to clarify when the direction and the target of a branch/jump is known.

Instruction	Taken known? (At the end of)	Target known? (At the end of)
BEQZ/BNEZ	R	B
J	B (always taken)	B
JR	B (always taken)	R

Problem M7.3.A

As a first step, we add a branch history table (BHT) in the fetch pipeline as shown on the next page. In the B stage (Branch Address Calc/Begin Decode), a conditional branch instruction (BEQZ/BNEZ) looks up the BHT, but an unconditional jump does not. If a branch is predicted to be taken, some of the instructions are flushed and the PC is redirected to the calculated branch target address. The instruction at PC+4 is fetched by default unless PC is redirected by an older instruction.

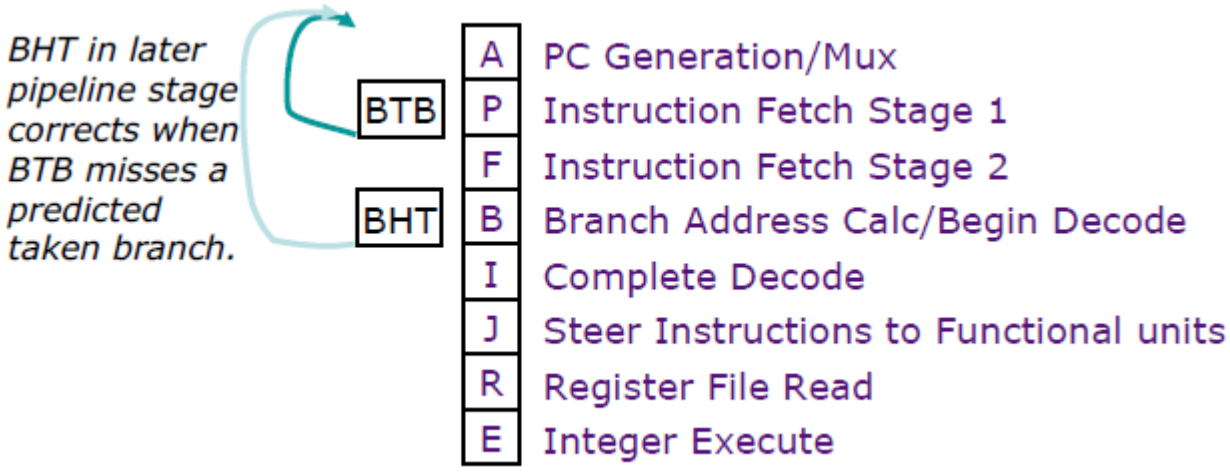
For each of the following cases, write down the number of pipeline bubbles caused by a branch or jump. If there is no bubble, you can simply put 0. (Y = yes, N= no)

	Predicted Taken?	**Actually Taken?**	**Pipeline bubbles**
BEQZ/ BNEZ	Y	Y	
	Y	N	
	N	Y	
	N	N	
J	Always taken (No lookup)	Y	
JR	Always taken (No lookup)	Y	

Problem M7.3.B

To improve the branch performance further, we decide to add a branch target buffer (BTB) as well. Here is a description for the operation of the BTB.

1. The BTB holds entry_PC, target_PC pairs for jumps and branches predicted to be taken. Assume that the target_PC predicted by the BTB is always correct for this question. (Yet the direction still might be wrong.)
2. The BTB is looked up every cycle. If there is a match with the current PC, PC is redirected to the target_PC predicted by the BTB (unless PC is redirected by an older instruction); if not, it is set to PC+4.



Fill out the following table of the number of pipeline bubbles (only for conditional branches).

	BTB Hit?	(BHT) Predicted Taken?	Actually Taken?	Pipeline bubbles
Conditional Branches	Y	Y	Y	
	Y	Y	N	
	Y	N	Y	Cannot occur
	Y	N	N	Cannot occur
	N	Y	Y	
	N	Y	N	
	N	N	Y	
	N	N	N	

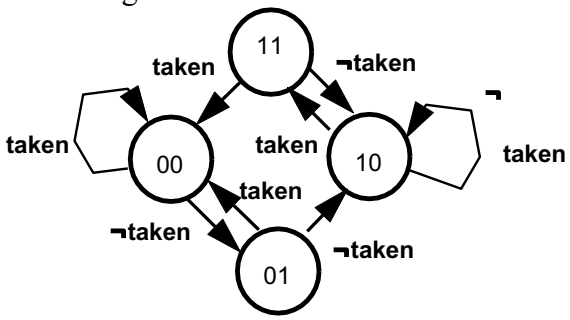
Problem M7.3.C

We will be working on the following program:

```
ADDRESS              INSTRUCTION
 0x1000        BR1:  BEQZ R5, NEXT     ; always taken
 0x1004              ADDI R4, R4, #4
 0x1008              MULT R3, R5, R3
 0x100C              ST   R3, 0(R4)
 0x1010              SUBI R5, R5, #1
 0x1014       NEXT:  ADDI R1, R1, #1
 0x1018              SLTI R2, R1, 100  ; repeat 100 times
 0x101C        BR2:  BNEZ R2, BR1
 0x1020              NOP
 0x1024              NOP
 0x1028              NOP
```

Given a snapshot of the BTB and the BHT states on entry to the loop, fill in the timing diagram for one iteration (plus two instructions) on the next page. (Don't worry about the stages beyond the E stage.) We assume the following for this question.

1. The initial values of R5 and R1 are zero, so `BR1` is always taken.
2. We disregard any possible structural hazards. There are no pipeline bubbles (except for those created by branches.)
3. We fetch only one instruction per cycle.
4. We use a two-bit predictor whose state diagram is shown below. In state 1X we will guess not taken; in state 0X we will guess taken. `BR1` and `BR2` do not conflict in the BHT.



5. We use a two-entry fully-associative BTB with the LRU replacement policy.

Initial Snapshot

(Valid) V	Entry PC	Predicted Target PC
1	0x101C	0x1000

BTB

	...
BR1	1 1
	...
BR2	0 0
	...

BHT

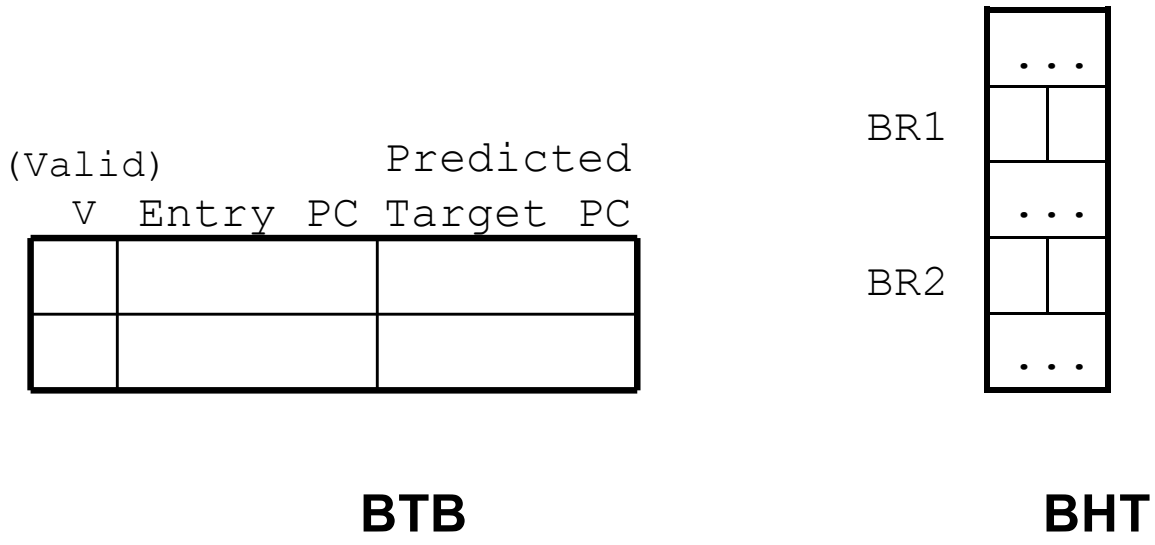
		TIME																						
Address	Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0x1000	BEQZ R5, NEXT	A	P	F	B	I	J	R	E															
0x1014	ADDI R1, R1, #1																							
0x1018	SLTI R2, R1, 100																							
0x101C	BNEZ R2, LOOP																							
0x1000	BEQZ R5, NEXT																							
0x1014	ADDI R1, R1, #1																							

Timing diagram for M3.4.C

Last updated:
3/30/2021

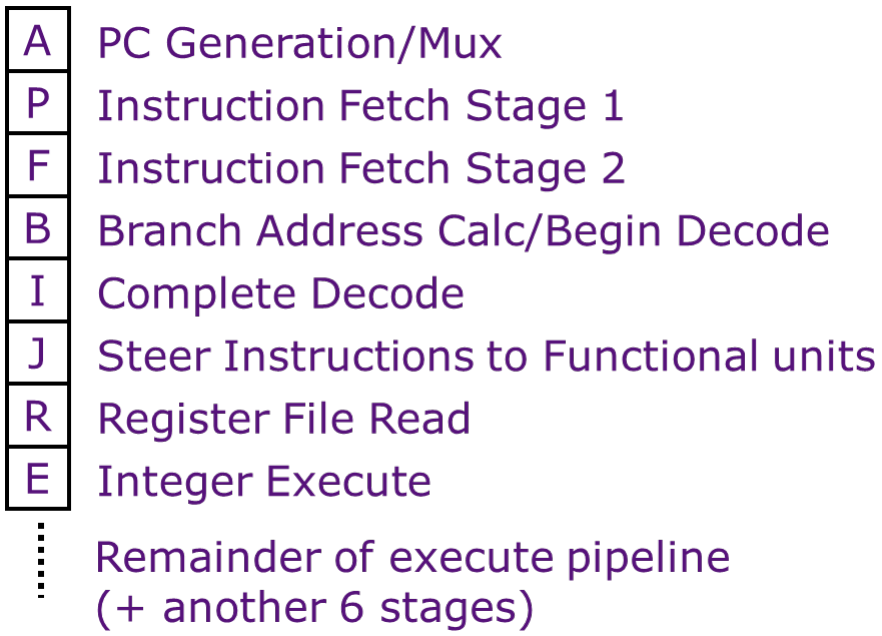
Problem M7.3.D

What will be the BTB and BHT states right after the 6 instructions in Question 9 have updated the branch predictors' states? Fill in (1) the BTB and (2) the entries corresponding to BR1 and BR2 in the BHT.



BTB **BHT**

Problem M7.4: Complex Pipelining (Spring 2014 Quiz 2, Part B)



You are designing a processor with the complex pipeline illustrated above. For this problem assume there are no unconditional jumps or jump register—*only* conditional branches.

Suppose the following:

- Each stage takes a single cycle.
- Branch addresses are known after stage Branch Address Calc/Begin Decode.
- Branch conditions (taken/not taken) are known after Register File Read.
- Initially, the processor *always* speculates that the next instruction is at PC+4, without any specialized branch prediction hardware.
- Branches always go through the pipeline without any stalls or queuing delays.

Problem M7.4.A

How much work is lost (in cycles) on a branch misprediction in this pipeline?

Problem M7.4.B

If one quarter of instructions are branches, and half of these are taken, then how much should we expect branches to increase the processor's CPI (cycles per instruction)?

Problem M7.4.C

You are unsatisfied with this performance and want to reduce the work lost on branches. Given your hardware budget, you can add only one of the following:

- A branch predictor to your pipeline that resolves after Instruction Fetch Stage 1.
- Or a branch target buffer (BTB) that resolves after Instruction Fetch Stage 2.

If each make the same predictions, which do you prefer? In one or two sentences, why?

Problem M7.4.D

You decide to add the BTB (not the branch predictor). Your BTB is a fully tagged structure, so if it predicts an address other than PC+4 then it always predicts the branch address of a conditional branch (but not the condition!) correctly. **For partial credit, show your work.**

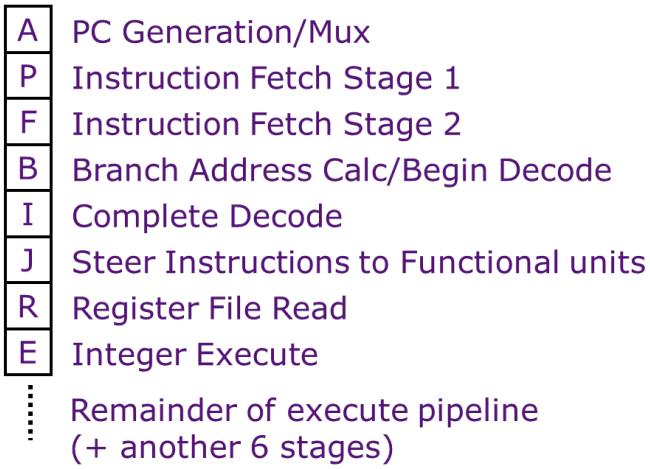
If the BTB correctly predicts a next PC other than PC+4, what is the effect on the pipeline?

If the BTB predicts the next PC incorrectly, what is the effect on the pipeline?

Assume the BTB predicts PC+4 90% of the time. When the BTB predicts PC+4 it is accurate 90% of the time. Otherwise it is accurate 80% of the time. How much should we expect branches to increase the CPI of the BTB design? *(Don't bother trying to compute exact decimal values.)*

Problem M7.5: Branch Prediction (Spring 2015 Quiz 2, Part A)

Ben Bitdiddle is designing a processor with the complex pipeline illustrated below:



The processor has the following characteristics:

- Issues at most one instruction per cycle.
- Branch addresses are known at the end of the B stage (Branch Address Calc/Begin Decode).
- Branch conditions (taken/not taken) are known at the end of the R stage (Register File Read).
- Branches always go through the pipeline without any stalls or queuing delays.

Ben's target program is shown below:

```
for(int i = 0; i <= 1000000; i++)
{
    if(i % 2 == 0) //Branch B1
    { //Not taken
       (Do something A)
    }
    if(i % 4 == 0) //Branch B2
    { //Not taken
       (Do something B)
    }
}     //Branch LP
```

```
      ANDi R1 0
LOOP:MODi R2 R1 2
      BNE  R2 M4      // B1
      (Do something A)
      … …
M4:   MODi R3 R1 4
      BNE  R3 END     // B2
      (Do something B)
      … …
END:  SUBi R4 R1 1000000
      BNE  R4 LOOP    // LP
      … …
```

The MODi (modulo-immediate) instruction is defined as follows:
`MODi Rd Rs imm: Rd <- Rs Mod imm`

Problem M7.5.A

In steady state, what is the probability for each branch in the code to be taken/not taken on average? Fill in the table below.

Branch	Probability to be **taken**	Probability to be **not taken**
B1		
B2		
LP		

Problem M7.5.B

In steady state, how many cycles per iteration are lost on average if the processor always speculates that every branch is not taken (i.e., next PC is PC+4)?

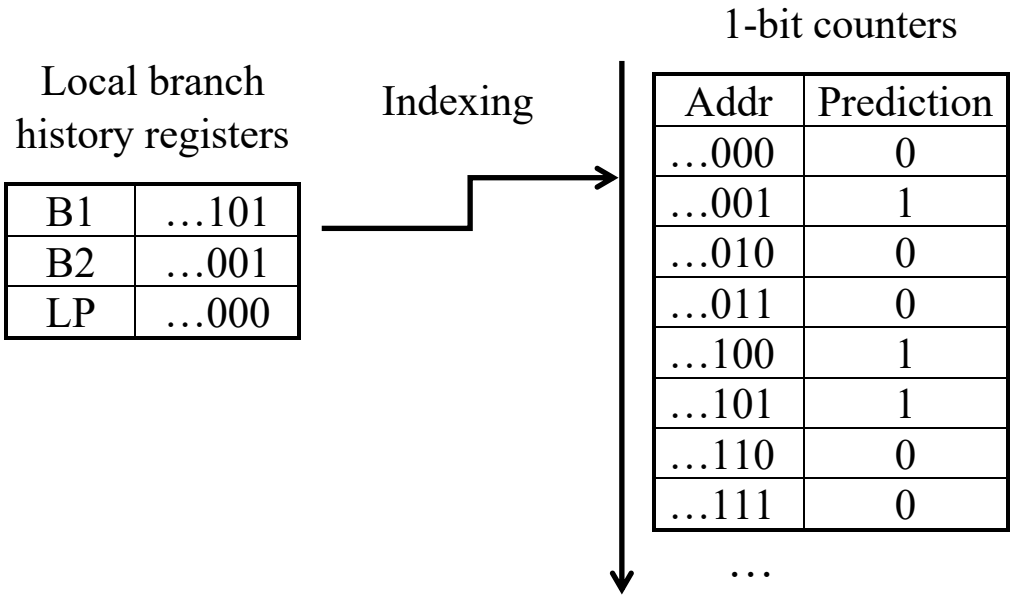
Problem M7.5.C

Ben designs a **static branch predictor** to improve performance. This predictor always predicts **not taken for forward jumps** and **taken for backward jumps**. The prediction is available at the end of the **B** stage. In steady state, how many cycles per iteration are lost on average?

Problem M7.5.D

To improve performance further, Ben designs a **dynamic branch predictor with local branch history registers and 1-bit counters**.

Each local branch history registers store the last several outcomes of a single branch (branches B1, B2 and LP in our case). By convention, the most recent branch outcome is the least significant bit, and so on. The predictor uses the local history of the branch to index a table of 1-bit counters. It predicts not taken if the corresponding 1-bit counter is 0, and taken if it is 1. Assume local branch history registers are always correct.



How many bits per branch history register do we need to perform perfect prediction in steady state?

Problem M7.5.E

The local-history predictor itself is a speculative structure. That is, for subsequent predictions to be accurate, the predictor has to be updated speculatively.

Explain what guess the local history update function should use.

Problem M7.5.F

Ben wants to design the data management policy (i.e., how to manage the speculative data in different structures of the predictor) for the local-history branch predictor to work well. Use a couple of sentences to answer the following questions.

1) What data management policies should be applied to each structure?

2) For your selected data management policies, is there any challenge for the recovery mechanism when there is misspeculation? If so, what are the challenges?

Problem M8.1: Fetch Pipelines

Ben is designing a deeply-pipelined, single-issue, in-order MIPS processor. The first half of his pipeline is as follows:

PC	PC Generation
F1	ICache Access
F2	
D1	Instruction Decode
D2	
RN	Rename/Reorder
RF	Register File Read
EX	Integer Execute

There are no branch delay slots and currently there is **no** branch prediction hardware (instructions are fetched sequentially unless the PC is redirected by a later pipeline stage). Subroutine calls use **`JAL/JALR`** (jump and link). These instructions write the return address (PC+4) into the link register (r31). Subroutine returns use **`JR r31`**. Assume that PC Generation takes a whole cycle and that you cannot bypass anything into the end of the PC Generation phase.

Problem M8.1.A — Pipelining Subroutine Returns

Immediately after what pipeline stage does the processor know that it is executing a subroutine return instruction? Immediately after what pipeline stage does the processor know the subroutine return address? How many pipeline bubbles are required when executing a subroutine return?

Problem M8.1.B — Adding a BTB

Louis Reasoner suggests adding a BTB to speed up subroutine returns. Why doesn't a standard BTB work well for predicting subroutine returns?

Problem M8.1.C — Adding a Return Stack

Instead of a BTB, Ben decides to add a return stack to his processor pipeline. This return stack records the return addresses of the *N* most recent subroutine calls. This return stack takes no time to access (it is always presenting a return address).
Explain how this return stack can speed up subroutine returns. Describe when and in which pipeline stages return addresses are pushed on and popped off the stack.

Problem M8.1.D — Return Stack Operation

Fill in the pipeline diagram below corresponding to the execution of the following code on the return stack machine:

```
A: JAL B
A+1:  A+2:
…

B: JR r31
B+1:   B+2:
…
```

Make sure to indicate the instruction that is being executed. The first two instructions are illustrated below. The crossed out stages indicate that the instruction was killed during those cycles.

instruction	time→																		
A	PC	F1	F2	D1	D2	RN	RF	EX											
A+1		PC	F1	F2	D1	~~D2~~	~~RN~~	~~RF~~	~~EX~~										

Problem M8.1.E **Handling Return Address Mispredicts**

If the return address prediction is wrong, how is this detected? How does the processor recover, and how many cycles are lost (relative to a correct prediction)?

Problem M8.1.F **Further Improving Performance**

Describe a hardware structure that Ben could add, in addition to the return stack, to improve the performance of return instructions so that there is usually only a one-cycle pipeline bubble when executing subroutine returns (assume that the structure takes a full cycle to access).

Problem M8.2: Managing Out-of-order Execution

This problem investigates the operation of a superscalar processor with branch prediction, register renaming, and out-of-order execution. The processor holds all data values in a **physical register file**, and uses a **rename table** to map from architectural to physical register names. A **free list** is used to track which physical registers are available for use. A **reorder buffer (ROB)** contains the bookkeeping information for managing the out-of-order execution (but, it does not contain any register data values).

When a branch instruction is encountered, the processor predicts the outcome and takes a snapshot of the rename table. If a misprediction is detected when the branch instruction later executes, the processor recovers by flushing the incorrect instructions from the ROB, rolling back the "next available" pointer, updating the free list, and restoring the earlier rename table snapshot.

We will investigate the execution of the following code sequence (assume that there is **no** branch-delay slot):

```
loop:  lw    r1, 0(r2)     # load r1 from address in r2
       addi  r2, r2, 4     # increment r2 pointer
       beqz  r1, skip      # branch to "skip" if r1 is 0
       addi  r3, r3, 1     # increment r3
skip:  bne   r2, r4, loop # loop until r2 equals r4
```

The diagram for Question M3.5.A on the next page shows the state of the processor during the execution of the given code sequence. An instance of each instruction in the loop has been issued into the ROB (the beqz instruction has been predicted not-taken), but none of the instructions have begun execution. In the diagram, old values which are no longer valid are shown in the following format: ~~P4~~. The rename table snapshots and other bookkeeping information for branch misprediction recovery are not shown.

Problem M8.2.A

Assume that the following events occur in order (though not necessarily in a single cycle):

Step 1. The first three instructions from the next loop iteration (lw, addi, beqz) are written into the ROB (note that the bne instruction has been predicted taken).

Step 2. All instructions which are ready after Step 1 execute, write their result to the physical register file, and update the ROB. Note that this step only occurs **once**.

Step 3. As many instructions as possible commit.

Update the diagram below to reflect the processor state after these events have occurred. **Cross out** any entries which are no longer valid. Note that the "**ex**" field should be **marked** when an instruction executes, and the "**use**" field should be **cleared** when it commits. Be sure to update the "**next to commit**" and "**next available**" pointers. If the **load** executes, assume that the data value it retrieves is **0**.

Rename Table

R1	~~P1~~	P4	
R2	~~P2~~	P5	
R3	~~P3~~	P6	
R4	P0		

Physical Regs

P0	8016	p
P1	6823	p
P2	8000	p
P3	7	p
P4		
P5		
P6		
P7		
P8		
P9		

Free List

~~P4~~
~~P5~~
~~P6~~
P7
P8
P9

Reorder Buffer (ROB)

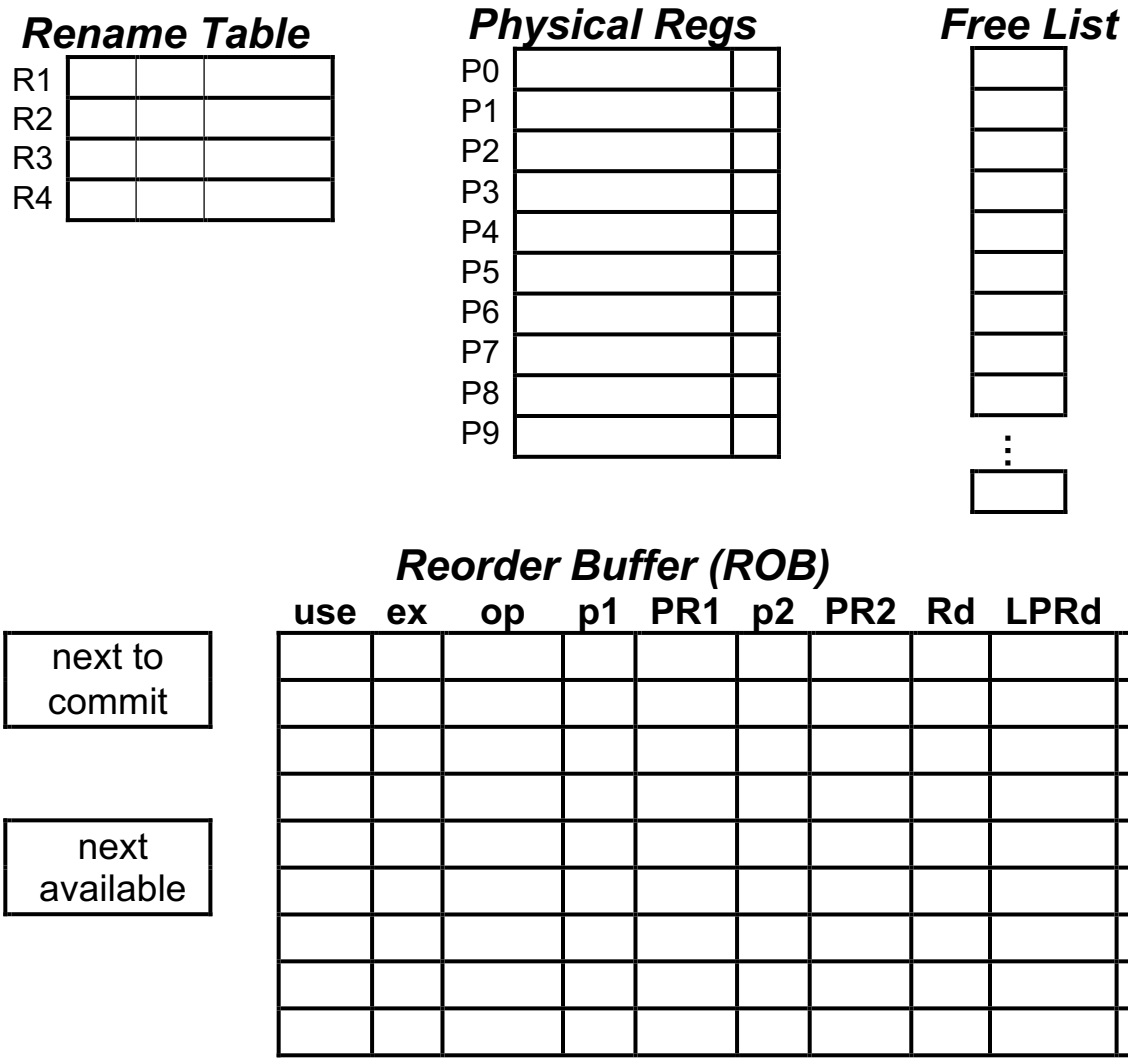
	use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
next to commit →	x		lw	p	P2			r1	P1	P4
	x		addi	p	P2			r2	P2	P5
	x		beqz		P4					
	x		addi	p	P3			r3	P3	P6
	x		bne		P5	p	P0			
next available →										

Problem M8.2.B

Assume that after the events from Question M3.6.A have occurred, the following events occur in order:

Step 1. The processor detects that the beqz instruction has mispredicted the branch outcome, and recovery action is taken to repair the processor state.
Step 2. The beqz instruction commits.
Step 3. The correct next instruction is fetched and is written into the ROB.

Fill in the diagram below to reflect the processor state after these events have occurred. Although you are not given the rename table snapshot, you should be able to deduce the necessary information from the diagram from Question M3.6.A. You do not need to show invalid entries in the diagram, but be sure to **fill in all the fields** which have valid data, and update the "**next to commit**" and "**next available**" pointers. Also make sure that the **free list** contains all available registers.



Problem M8.2.C

Consider (1) a single-issue, in-order processor with no branch prediction and (2) a multiple-issue, out-of-order processor with branch prediction. Assume that both processors have the same clock frequency. Consider how fast the given loop executes on each processor, assuming that it executes for many iterations.

Under what conditions, if any, might the loop execute at a faster rate on the in-order processor compared to the out-of-order processor?

Under what conditions, if any, might the loop execute at a faster rate on the out-of-order processor compared to the in-order processor?

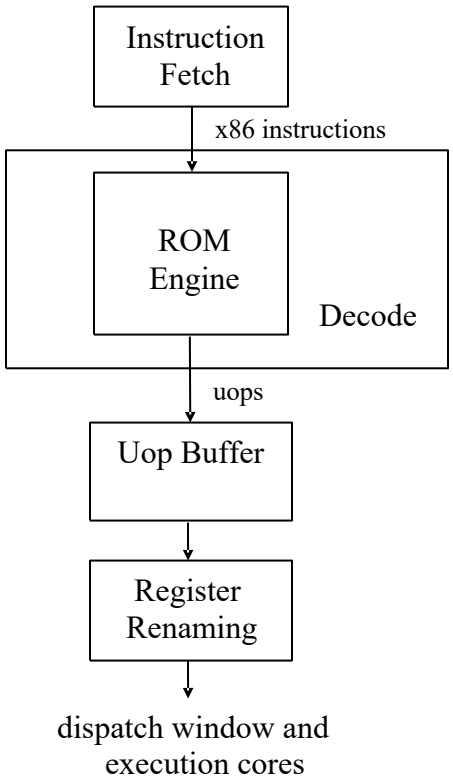
Problem M8.3: Exceptions and Register Renaming

Ben Bitdiddle has decided to start Bentel Corporation, a company specializing in high-end x86 processors to compete with Intel. His latest project is the Bentium 4, a superscalar, out-of-order processor with register renaming and speculative execution.

The Bentium 4 has 8 architectural registers (EAX, EBX, ECX, EDX, ESP, EBP, EDI, and ESI). In addition, the processor provides 8 internal registers T0-T7 not visible to the ISA that can be used to hold intermediary values used by micro-operations (µops) generated by the microcode engine. The microcode engine is the decode unit and is used to generate µops for all the x86 instructions. For example, the following register-memory x86 instruction might be translated into the following RISC-like µops:

$$\text{ADD } R_d, R_a, \text{offset}(R_b) \rightarrow \begin{array}{l} \text{LW} \quad T0, \text{offset}(R_b) \\ \text{ADD } R_d, R_a, T0 \end{array}$$

All 16 µop-visible registers are renamed by the register allocation table (RAT) into a set of physical registers (P0-Pn). There is a separate shadow map structure that takes a snapshot of the RAT on a speculative branch in case of a misprediction. The block diagram for the front-end of the Bentium 4 is shown below:



Note: The decode block is actually replicated in the Bentium 4 in order to decode multiple instructions per cycle (not shown in the diagram).

Problem M8.3.A — Recovering from Exceptions

For the Bentium 4, if an x86 instruction takes an exception before it is committed, the machine state is reset back to the precise state that existed right before the excepting instruction started executing. This instruction is then re-executed after the exception is handled. Ben proposes that the shadow map structure used for speculative branches can also be used to recover a precise state in the event of an exception. Specify a strategy that can be implemented for taking the least number of snapshots of the RAT that would still allow the Bentium 4 to implement precise exception handling.

Problem M8.3.B — Minimizing Snapshots

Ben further states that the shadow map structure does not need to take a snapshot of all the registers in the Bentium 4 to be able to recover from an exception. Is Ben correct or not? If so, state which registers do not need to be recorded and explain why they are not necessary, or explain why all the registers are necessary in the snapshot.

Problem M8.3.C — Renaming Registers

Assume that the Bentium 4 has the same register renaming scheme as the Pentium 4. What is the minimum number of physical registers (P) that the Bentium 4 must have to allow register renaming to work? Explain your answer.

Problem M8.4: Out-of-order Execution (Spring 2014 Quiz 2, Part C)

In this problem, we are going to update the state of the processor when different events happen. You are given an out-of-order processor in some initial state, as described by the registers (renaming table, physical registers, and free list), one-bit branch predictor, and re-order buffer. Your job is to show the changes that occur when some event occurs, starting from the same initial state except where noted. For partial credit, briefly describe what changes occur.

Problem M8.4.A

Show the state of the processor if the first load completes (but does not commit).

```
    00: LD R1, 0(R2)
    04: ADD R3, R1, R4
    08: ADD R2, R1, R2
    0c: BGEZ R4, A
    10: LD R3, 0(R2)
A:  14: SUB R1, R3, R2
    18: ADD R4, R3, R1
```

INSTRUCTIONS

BRANCH PREDICTOR	
00	1
01	0
10	1
11	0

RENAMING TABLE	
R1	P4
R2	P6
R3	P5
R4	P3

PHYS. REG. FILE		
P0	(R1)	p
P1	(R2)	p
P2	(R3)	p
P3	(R4)	p
P4		
P5		
P6		
P7		

FREE LIST
P7

RE-ORDER BUFFER (ROB)

	Use?	Ex	Op	P1	PR1	P2	PR2	Rd	LPRd	PRd
Next to commit →	X		LD	p	P1			R1	P0	P4
	X		ADD		P4	p	P3	R3	P2	P5
	X		ADD		P4	p	P1	R2	P1	P6
	X		BGEZ	p	P3					
Next available →										

Problem M8.4.B

Show the state of the processor after the next instruction is issued.

```
    00: LD R1, 0(R2)
    04: ADD R3, R1, R4
    08: ADD R2, R1, R2
    0c: BGEZ R4, A
    10: LD R3, 0(R2)
A:  14: SUB R1, R3, R2
    18: ADD R4, R3, R1
```

INSTRUCTIONS

BRANCH PREDICTOR	
00	1
01	0
10	1
11	0

RENAMING TABLE	
R1	P4
R2	P6
R3	P5
R4	P3

PHYS. REG. FILE		
P0	(R1)	p
P1	(R2)	p
P2	(R3)	p
P3	(R4)	p
P4		
P5		
P6		
P7		

FREE LIST
P7

RE-ORDER BUFFER (ROB)

	Use?	Ex	Op	P1	PR1	P2	PR2	Rd	LPRd	PRd
Next to commit →	X		LD	p	P1			R1	P0	P4
	X		ADD		P4	p	P3	R3	P2	P5
	X		ADD		P4	p	P1	R2	P1	P6
	X		BGEZ	p	P3					
Next available →										

Problem M8.4.C

From the state at the end of Question 2, as the next action can the processor issue (not execute) another instruction?

In one or two sentences, what does this say about our design? How can we improve it?

Problem M8.4.D

Show the state of the processor if the first LD triggers a page fault and after abort finishes.

```
    00: LD R1, 0(R2)
    04: ADD R3, R1, R4
    08: ADD R2, R1, R2
    0c: BGEZ R4, A
    10: LD R3, 0(R2)
A:  14: SUB R1, R3, R2
    18: ADD R4, R3, R1
```

INSTRUCTIONS

BRANCH PREDICTOR	
00	1
01	0
10	1
11	0

RENAMING TABLE	
R1	P4
R2	P6
R3	P5
R4	P3

PHYS. REG. FILE		
P0	(R1)	p
P1	(R2)	p
P2	(R3)	p
P3	(R4)	p
P4		
P5		
P6		
P7		

FREE LIST
P7

	RE-ORDER BUFFER (ROB)									
	Use?	Ex	Op	P1	PR1	P2	PR2	Rd	LPRd	PRd
Next to commit →	X		LD	p	P1			R1	P0	P4
	X		ADD		P4	p	P3	R3	P2	P5
	X		ADD		P4	p	P1	R2	P1	P6
	X		BGEZ	p	P3					
Next available →										

Problem M8.5 (Spring 2015 Quiz 2, Part B)

You are given an out-of-order processor that

- Issues at most one instruction per cycle
- Commits at most one instruction per cycle
- Uses an unified physical register file

Problem M8.5.A

Consider the following code sequence:

```
        Addr
I0   (0x24)       lw     r2, (r4), #0
I1   (0x28)       addi   r2, r2,   #16
I2   (0x2C)       lw     r3, (r4), #4
I3   (0x30)       blez   r3, L1
I4   (0x34)       addi   r4, r2, #8
I5   (0x38)       mul    r1, r2, r3
I6   (0x3C)       addi   r3, r2, #8
I7   (0x40)  L1:  add    r2, r1, r3
```

Assume the branch instruction (blez) is not taken. Fill out the table below to identify all Read-After-Write (RAW), Write-After-Read (WAR), and Write-After-Write (WAW) dependencies in the above sequence.

Younger Instruction \ Older Instruction	I0	I1	I2	I3	I4	I5	I6	I7
I0	-							
I1	WAW RAW	-						
I2			-					
I3				-				
I4					-			
I5						-		
I6							-	
I7								-

In Problems M8.5.B to M8.5.D, you should update the state of the processor when different events happen. The starting state in each question is the same, and the event specified in each question is the ONLY event that takes place for that question. The starting state is shown in the different structures: renaming table, physical registers, free list, two-bit branch predictor, global history buffer, and reorder buffer (ROB).

Note the following conventions:

- The valid bit for any entry is represented by "1".
- The valid bit can be cleared by crossing it out.
- In the ROB, the "ex" field should be marked with "1" when an instruction starts execution, and the "use" field should be cleared when it commits. Be sure to update the "next to commit" and "next available" pointers, if necessary.
- Fill out the "after" fields in all the tables. Write new values in these boxes if the values change due to the event specified in the question. You do not have to repeat the values if they do not change due to the event.

In Questions 2 through 4, we will use the same code sequence as in Question 1:

```
      Addr
I0   (0x24)       lw     r2, (r4), #0
I1   (0x28)       addi   r2, r2,   #16
I2   (0x2C)       lw     r3, (r4), #4
I3   (0x30)       blez   r3, L1
I4   (0x34)       addi   r4, r2, #8
I5   (0x38)       mul    r1, r2, r3
I6   (0x3C)       addi   r3, r2, #8
I7   (0x40)  L1:  add    r2, r1, r3
```

The starting state of the processor is as follows:

- Instructions I0-I4 are already in the ROB.
- I0 (lw) has already finished execution.
- I1 (addi) and I2 (lw) have started executing but have not finished yet.
- I3 (blez) has been predicted to be Not-Taken by the branch predictor.
- I5 (mul) has completed the decode stage.
- I6 (addi) has completed the Fetch Stage.
- The next PC is set to 0x40, which is the PC of I7 (add).

Problem M8.5.B

The following figure shows the starting state of the processor. Suppose the decoded instruction I5 (mul) is now inserted into the ROB. Update the diagram to reflect the processor state after this event has occurred.

Prediction Counter		
Index	**Before**	**After**
000	11	
001	00	
010	11	
011	01	
100	10	
101	11	
110	01	
111	00	

Rename Table (Latest)		
Name	**Before**	**After**
R1	P0	
R2	P5	
R3	P6	
R4	P7	

Rename Table (Snapshot 1)		Valid
		1
Name	**Before**	**After**
R1	P0	
R2	P5	
R3	P6	
R4	P3	

Fetched Inst. Queue	
PC	**Inst.**
0x3C	I6 (addi)

Branch Global History	
Before	**After**
0010110	

Physical Registers		
Name	**Value**	**Valid**
P0	45	1
P1	2	1
P2	-3	1
P3	100	1
P4	20	1
P5		
P6		
P7		
P8		
P9		
P10		

Decoded Inst. Queue
Inst.
I5 (mul)

Next PC to be fetched	
Before	**After**
0x40	

Free List
P8
P9
P10

Reorder Buffer (ROB)									
use	**ex**	**op**	**p1**	**PR1**	**p2**	**PR2**	**Rd**	**LPRd**	**PRd**
1	1	lw	1	P3			r2	P1	P4
1	1	addi	1	P4			r2	P4	P5
1	1	lw	1	P3			r3	P2	P6
1		blez		P6					
1		addi		P5			r4	P3	P7

Next to commit

Next available

Problem M8.5.C

Start from the same processor state, shown below. Suppose now I1 (addi) has completed execution. Commit as many instructions as possible. Update the diagram to reflect the processor state after I1 execution completes and as many instructions as possible have committed. Again, assume no other events take place.

Prediction Counter		
Index	**Before**	**After**
000	11	
001	00	
010	11	
011	01	
100	10	
101	11	
110	01	
111	00	

Fetched Inst. Queue	
PC	**Inst.**
0x3C	I6 (addi)

Decoded Inst. Queue
Inst.
I5 (mul)

Branch Global History	
Before	**After**
0010110	

Next PC to be fetched	
Before	**After**
0x40	

Rename Table (Latest)		
Name	**Before**	**After**
R1	P0	
R2	P5	
R3	P6	
R4	P7	

Rename Table (Snapshot 1)		Valid
		1
Name	**Before**	**After**
R1	P0	
R2	P5	
R3	P6	
R4	P3	

Physical Registers		
Name	**Value**	**Valid**
P0	45	1
P1	2	1
P2	-3	1
P3	100	1
P4	20	1
P5		
P6		
P7		
P8		
P9		
P10		

Free List
P8
P9
P10

Reorder Buffer (ROB)									
use	**ex**	**op**	**p1**	**PR1**	**p2**	**PR2**	**Rd**	**LPRd**	**PRd**
1	1	lw	1	P3			r2	P1	P4
1	1	addi	1	P4			r2	P4	P5
1	1	lw	1	P3			r3	P2	P6
1		blez		P6					
1		addi		P5			r4	P3	P7

Next to commit

Next available

Problem M8.5.D

Start from the same processor state, shown below. Suppose instruction I2 (lw) triggers an ALU overflow exception. Restore the architectural and microarchitectural state to recover from misspeculation. The exception handler for the processor is at address 0x8C (control is transferred to the exception handler after recovery). You do not need to worry about the number of cycles taken by recovery. Show the processor state after recovery.

Prediction Counter		
Index	**Before**	**After**
000	11	
001	00	
010	11	
011	01	
100	10	
101	11	
110	01	
111	00	

Fetched Inst. Queue	
PC	**Inst.**
0x3C	I6 (addi)

Decoded Inst. Queue
Inst.
I5 (mul)

Branch Global History	
Before	**After**
0010110	

Next PC to be fetched	
Before	**After**
0x40	

Rename Table (Latest)		
Name	**Before**	**After**
R1	P0	
R2	P5	
R3	P6	
R4	P7	

Physical Registers		
Name	**Value**	**Valid**
P0	45	1
P1	2	1
P2	-3	1
P3	100	1
P4	20	1
P5		
P6		
P7		
P8		
P9		
P10		

Free List
P8
P9
P10

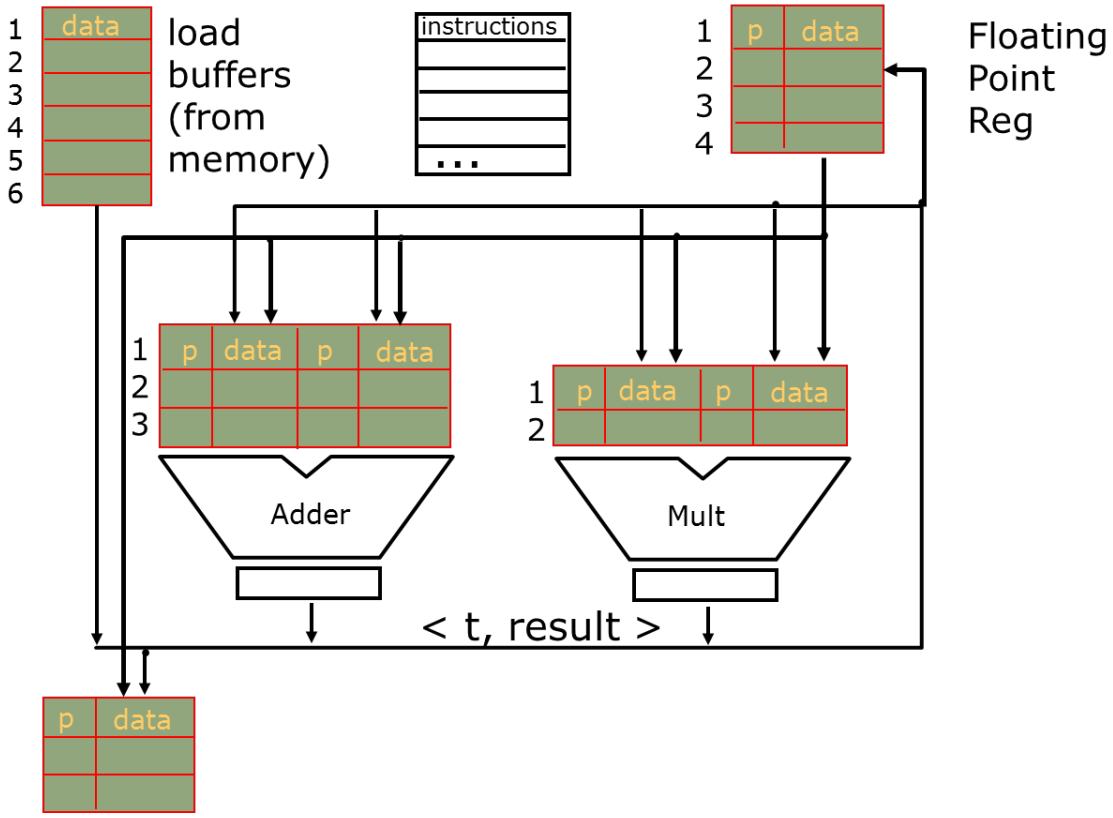
Rename Table (Snapshot 1)		Valid
		1
Name	**Before**	**After**
R1	P0	
R2	P5	
R3	P6	
R4	P3	

Reorder Buffer (ROB)									
use	**ex**	**op**	**p1**	**PR1**	**p2**	**PR2**	**Rd**	**LPRd**	**PRd**
1	1	lw	1	P3			r2	P1	P4
1	1	addi	1	P4			r2	P4	P5
1	1	lw	1	P3			r3	P2	P6
1		blez		P6					
1		addi		P5			r4	P3	P7

Next to commit

Next available

Problem M8.6: Out-of-order Processor Design (Spring 2014 Quiz 2, Part D)



You are designing an out-of-order processor similar to the IBM 360/91 Tomasulo design shown above. This design distributes the re-order buffer around the processor, placing entries near their associated functional units. In such a design, the distributed ROB entries are called "reservation stations". Entries are allocated when the instruction is decoded and freed when the instruction is dispatched to the functional unit.

Your design achieves an average throughput of 1.5 instructions per cycle. Two-thirds of instructions are adds, and one-third are multiplies. The latency of each instruction type *from allocation to completion* is 5 cycles for adds and 14 cycles for multiplies.

Type of operation	Fraction of instructions	Average latency
Add	2/3	5
Multiply	1/3	14

The adder and multiplier are each fully pipelined with full bypassing. *Once an instruction is dispatched to the FU*, the adder takes 2 cycles and the multiplier takes 5 cycles.

Throughput	Add latency	Multiply latency
1.5	2	5

Problem M8.6.A

How many entries are in use, on average, in the reservation station at each functional unit (adder, multiplier) in the steady state? Assume there are infinite entries available if needed. What is the average latency of an instruction in this machine? *For partial credit, feel free to give any formulae you believe may be important to answer this question.*

Problem M9.1: Memory Dependencies (Fall 2006)

Problem M9.1.A

For each of the following 3 instruction sequences, please give the condition for a memory dependency to exist or explain why there cannot be a dependency. Assume that there is no memory aliasing (i.e. all virtual memory pages are mapped to unique physical pages).

Instruction Pair	Condition under which Memory Dependency occurs
SW R2, 0(R3) LW R5, 0(R4)	
SW R2, 0(R3) LW R5, 4(R3)	
SW R2, 0(R3) LW R5, 4096(R3)	

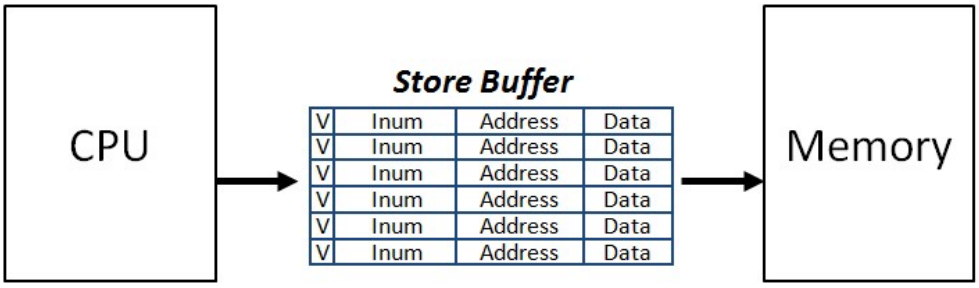
Problem M.9.1.B

If we allow memory aliasing to occur, how will it affect your answer in part A? Assume that the page size is 4KB and that the machine is byte addressed.

Instruction Pair	Condition under which Memory Dependency occurs
SW R2, 0(R3) LW R5, 0(R4)	
SW R2, 0(R3) LW R5, 4(R3)	
SW R2, 0(R3) LW R5, 4096(R3)	

Problem M9.2: Vector Store Buffers (Fall 2010)

Ben Bitdiddle designed an out-of-order vector machine with store buffers. This machine executes memory operations (both load and store) "in order", although other instructions can be executed out-of-order. (All the instructions are committed "in order".) There is a load/store issue queue to maintain the execution order of all the memory operations. Every load must check if the value it needs is in the store buffer, and to determine the proper value, every instruction is assigned a unique instruction number (Inum).



The machine has 4 vector registers, v1 through v4, and a special-purpose vector length register, vlr, which can be used like a general purpose register. Any MIPS integer operation (except jumps/branches) can be applied to the first vlr elements of one or more vector registers by prefixing a V to it; for example, if vlr is 16, the instruction VLW v1, 0 (r1) loads 16 consecutive words starting at the address in r1 into the first 16 locations in v1, and similarly the instruction VADD v3, v1, v2 adds each of the first 16 elements of v1 to the corresponding element of v2 and puts the result in the corresponding element of v3. Each vector register can hold at most 32 word values, and thus vlr can be at most 32. For the entire part, assume that **vlr has the value of 4**.

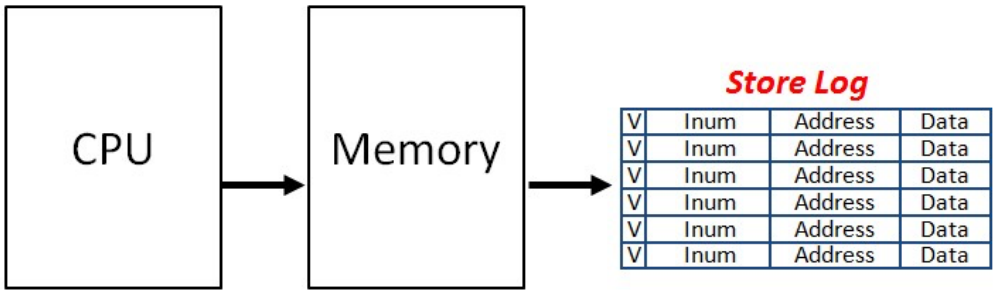
Problem M9.2.A

Suppose 10% of instructions are stores and the average lifetime of instructions in the store buffer is 100 cycles. Assuming the desired throughput of this machine to be 1 instruction per cycle, how many entries will the store buffer be holding at a given time on average?

Does this machine store into memory greedily or lazily?

Problem M9.2.B

Ben did not like the store buffers in the previous design, because the store buffers need to be looked up for every load instruction, and implementing a fast lookup to the buffers was too expensive. Thus, instead of using the store buffers, Ben decided to directly update the memory during execution (before the store instruction actually commits), and keep the old values in "**store logs**". **Each entry in the store logs consists of a valid bit, Inum, memory address and data value**. The data field holds the value that was in the memory before the store to the location writes to the memory.



Is this a greedy update or a lazy update?

Assume an arithmetic exception occurred, and thus, the processor state and memory need to be recovered appropriately. Which machine design (store buffer machine or store log machine) has the higher recovery cost, and why?

Problem M9.2.C

Now, we want to investigate how Ben's machine with the store logs changes the processor state. The diagram below shows the initial processor state. Assume that the Inum space is infinite.

[Initial State]

Vector Registers

V1	5,0,5,0,5,...
V2	1,1,1,1,1,...
V3	0,5,0,5,3,...
V4	0,1,2,3,4,...

Memory

0x1000	0,1,2,3,4,...
0x2000	5,5,0,0,5,...
0x3000	3,3,3,3,3,...

Store Logs

v	Inum	Address	Data
v	78	0x2000	1,1,1,1,1,...
v	83	0x3000	0,1,4,9,16,...
v	96	0x2000	2,2,2,2,2,...

In the following questions, **we always start from the same initial state**, and you have to update the diagram for each question **separately**, to reflect the processor state after each event has occurred. (The event specified in each question is the only event that takes place for that question.) There are extra blank boxes for vector registers and memory. Write down the new values in these boxes if the value changes due to the event specified in the question. You don't have to repeat the values if they do not change by the specified event. Also, cross out any entries in the store logs which are no longer valid.

Note that the vector length register (vlr) has the value of 4.

(i) The instruction **VADD V3, V1, V2** (Inum:99) is executed and committed.

Vector Registers

V1	5,0,5,0,5,...	
V2	1,1,1,1,1,...	
V3	0,5,0,5,3,...	
V4	0,1,2,3,4,...	

Memory

0x1000	0,1,2,3,4,...	
0x2000	5,5,0,0,5,...	
0x3000	3,3,3,3,3,...	

Store Logs

v	Inum	Address	Data
v	78	0x2000	1,1,1,1,1,...
v	83	0x3000	0,1,4,9,16,...
v	96	0x2000	2,2,2,2,2,...

(ii) The instruction **VST V4, 0x2000** (Inum:100) is executed.

Vector Registers

V1	5,0,5,0,5,...	
V2	1,1,1,1,1,...	
V3	0,5,0,5,3,...	
V4	0,1,2,3,4,...	

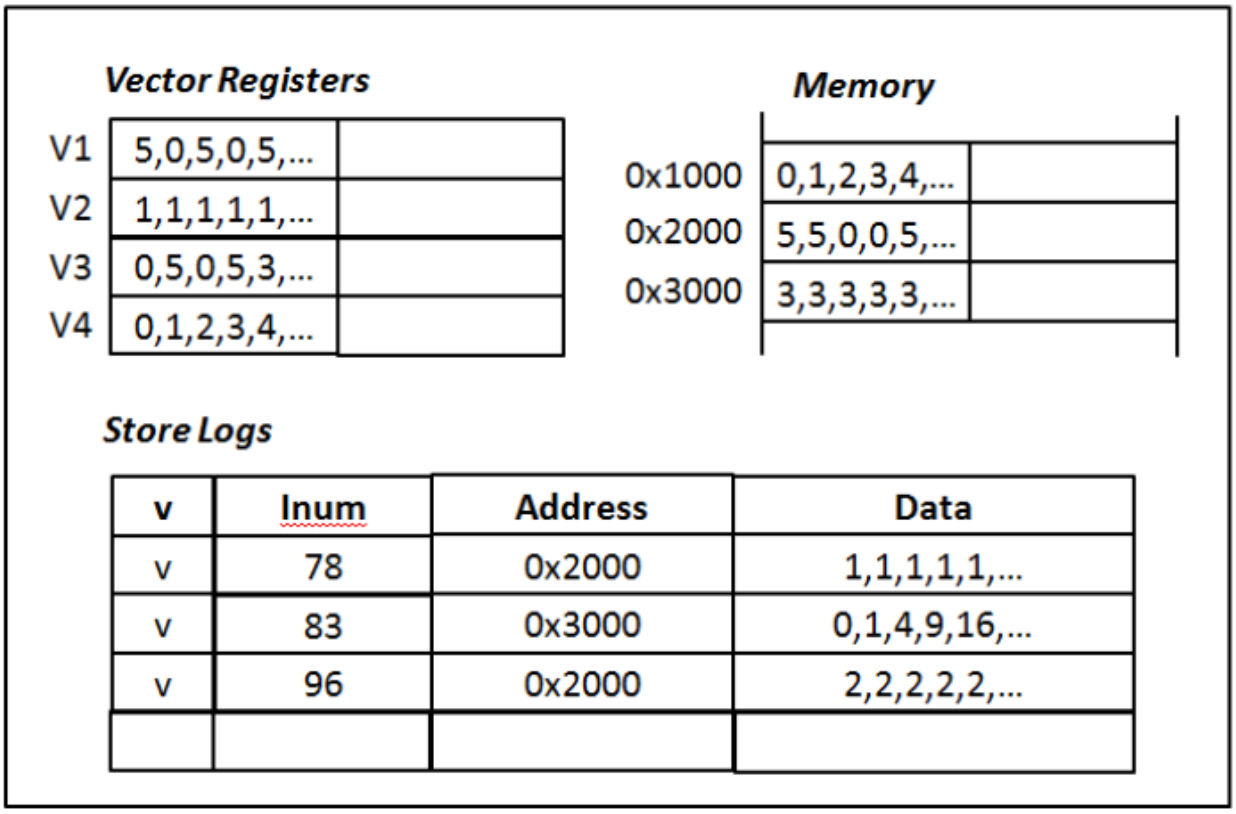
Memory

0x1000	0,1,2,3,4,...	
0x2000	5,5,0,0,5,...	
0x3000	3,3,3,3,3,...	

Store Logs

v	Inum	Address	Data
v	78	0x2000	1,1,1,1,1,...
v	83	0x3000	0,1,4,9,16,...
v	96	0x2000	2,2,2,2,2,...

(iii) An arithmetic exception occurs at the instruction ***with the Inum of 81***, and the recovery process takes place. (Do not worry about the vector register recovery.)



v	Inum	Address	Data
v	78	0x2000	1,1,1,1,1,...
v	83	0x3000	0,1,4,9,16,...
v	96	0x2000	2,2,2,2,2,...

Problem M9.2.D

For this machine with store logs, when will the processor clear the log entry, other than on exceptions?

Ben still did not like the fact that the store log requires so much extra storage. Thus, he decided to eliminate the store log and instead, use a direct-mapped writeback cache to hold the values. On a store operation that hits in the cache, the data is written only into the cache. On a cache miss, the data block is first fetched into the cache, and then the new value is written in the cache (write allocate policy). Memory is written only when the entry is evicted. Two new status bits are added in each cache line – "***dirty***" and "***committed***".

- A new bit "***dirty***" is 0 when the cache line has a clean copy (same with memory), and is set to 1 when the cache line is written so that it has a different copy than memory (i.e., memory has a stale copy).

- A new bit "***committed***" is 0 while the instruction that wrote the cache line has not been committed, and is set to 1 when the corresponding store instruction is committed.

As described in the beginning, this machine executes memory operations (both load and store) "in order", although other instructions can be executed out-of-order. (All the instructions are committed "in order".) There is a load/store issue queue to maintain the execution order of all the memory operations.

Now suppose the following signals are given.

Signal	Description
cache(address).tag	Returns the tag stored in the cache for the corresponding address
cache(address).dirty	0: the corresponding cache line has a clean copy (same with memory) / 1: the corresponding cache line has a dirty copy (i.e., memory has a stale copy)
cache(address).committed	0: the instruction that wrote the corresponding cache line is not committed / 1: the instruction that wrote the corresponding cache line is committed
QHead.OpCode	LD / ST (Opcode of the instruction at the head of the load/store issue queue)
QHead.memAddr	the effective memory address of the instruction at the head of the load/store issue queue
TAG(address)	A function that returns the tag of address

<u>For this part, assume the cache always hits.</u>

Problem M9.2.E

Under what state of the cache line can you writeback the line at the address A to memory?

Problem M9.2.F

How should the dirty bit and the committed bit be updated after writeback of the block at address A?

Problem M9.2.G

Alice P Hacker warns Ben that the machine may stall if it finds that it cannot use the cache block. Thus, in order to make progress, the machine may need to explicitly evict and writeback the corresponding cache line.

Write down the Boolean equation for the stall signal for the load/store issue queue.

Stall =

Problem M9.2.H

Write down the Boolean equation for the signal to force eviction (equivalently, writeback) of the cache line so that the machine gets unstalled.

Evict(writeback) =

Problem M11.1: Synchronization Primitives

One of the common instruction sequences used for synchronizing several processors are the LOAD RESERVE/STORE CONDITIONAL pair (from now on referred to as LdR/StC pair). The LdR instruction reads a value from the specified address and sets a local reservation for the address. The StC attempts to write to the specified address provided the local reservation for the address is still held. If the reservation has been cleared the StC fails and informs the CPU.

Problem M11.1.A

Describe under what events the local reservation for an address is cleared.

Problem M11.1.B

Is it possible to implement LdR/StC pair in such a way that the memory bus is not affected, i.e., unaware of the addition of these new instructions? Explain

Problem M11.1.C

Give two reasons why the LdR/StC pair of instructions is preferable over atomic read-test-modify instructions such as the TEST&SET instruction.

Problem M11.1.D

LdR/StC pair of instructions were conceived in the context of snoopy busses. Do these instructions make sense in our directory-based system in Handout #13? Do they still offer an advantage over atomic read-test-modify instructions in a directory-based system? Please explain.

Problem M11.2: Implementing Directories

Ben Bitdiddle is implementing a directory-based cache coherence invalidate protocol for a 64-processor system. He first builds a smaller prototype with only 4 processors to test out the cache coherence protocol described in Handout #13. To implement the list of sharers, **S**, kept by **home**, he maintains a bit vector per cache block to keep track of all the sharers. The bit vector has one bit corresponding to each processor in the system. The bit is set to one if the processor is caching a shared copy of the block, and zero if the processor does not have a copy of the block. For example, if Processors 0 and 3 are caching a shared copy of some data, the corresponding bit vector would be `1001.`

Problem M11.2.A

The bit vector worked well for the 4-processor prototype, but when building the actual 64-processor system, Ben discovered that he did not have enough hardware resources. Assume each **cache block is 32 bytes**. What is the overhead of maintaining the sharing bit vector for a 4-processor system, as a **fraction of data storage bits**? What is the overhead for a 64-processor system, as a **fraction of data storage bits**?

Overhead for a 4-processor system: _____

Overhead for a 64-processor system: _____

Problem M11.2.B

Since Ben does not have the resources to keep track of all potential sharers in the 64-processor system, he decides to limit **S** to keep track of only 1 processor using its 6-bit ID as shown in Figure M11.2-A (**single-sharer scheme**). When there is a load [`C2P_Req(a) S`] request for a shared cache block, Ben invalidates the existing sharer to make room for the new sharer (home sends a invalidate request `[P2C_Req(a) I]` to the existing sharer, the existing sharer sends an invalidate response `[C2P_Rep(a) I]` to home, home replaces the exiting sharer's ID with the new sharer's ID and sends the load response `[P2C_Rep(a) I S]` to the new sharer).

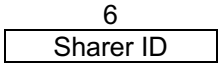


Figure M11.2-A

Consider a 64-processor system. To determine the efficiency of the bit-vector scheme and single-sharer scheme, **fill in the number of invalidate-requests** that are generated by the protocols for each step in the following two sequences of events. Assume cache block **B** is uncached initially for both sequences.

Sequence 1	**bit-vector scheme # of invalidate-requests**	**single-sharer scheme # of invalidate-requests**
Processor #0 reads **B**	0	0
Processor #1 reads **B**		
Processor #0 reads **B**		

Sequence 2	**bit-vector scheme # of invalidate-requests**	**single-sharer scheme # of invalidate-requests**
Processor #0 reads **B**	0	0
Processor #1 reads **B**		
Processor #2 writes **B**		

Problem M11.2.C

Ben thinks that he can improve his original scheme by adding an extra "**global bit**" to **S** as shown in Figure M11.2-B (**global-bit scheme**). The global bit is set when there is more than 1 processor sharing the data, and zero otherwise.

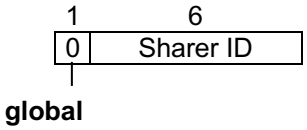


Figure M11.2-B

When the global bit is set, home stops keeping track of a specific sharer and assumes that all processors are potential sharers.

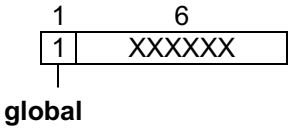


Figure M11.2-C

Consider a 64-processor system. To determine the efficiency of the global-bit scheme, **fill in the number of invalidate-requests** that are generated for each step in the following two sequences of events. Assume cache block **B** is uncached initially for both sequences.

Sequence 1	**global-bit scheme # of invalidate-requests**
Processor #0 reads **B**	0
Processor #1 reads **B**	
Processor #0 reads **B**	

Sequence 2	**global-bit scheme # of invalidate-requests**
Processor #0 reads **B**	0
Processor #1 reads **B**	
Processor #2 writes **B**	

Problem M11.3: Tracing the Directory-based Protocol

For the problem we will be using the following sequences of instructions. These are small programs, each executed on a different processor, each with its own cache and register set. In the following **R** is a register and **X** is a memory location. Each instruction has been named (e.g., B3) to make it easy to write answers.

Assume data in location X is initially 0.

Processor A	Processor B	Processor C
A1: ST X, 1	B1: R := LD X	C1: ST X, 6
A2: R := LD X	B2: R := ADD R, 1	C2: R := LD X
A3: R := ADD R, R	B3: ST X, R	C3: R := ADD R, R
A4: ST X, R	B4: R:= LD X	C4: ST X, R
	B5: R := ADD R, R	
	B6: ST X, R	

These questions relate to the directory-based protocol in Handout #13 (as well as Lecture 15). Unless specified otherwise, assume all caches are initially empty and *no voluntary responses are sent (i.e. responses are sent only on receiving a request).*

Problem M11.3.A

Suppose we execute Program A, followed by Program B, followed by Program C and all caches are initially empty. Write down the sequence of messages that will be generated. We have omitted ADD instructions because they cannot generate any messages. EO indicates the global execution order.

Processor A			Processor B			Processor C		
Ins	EO	Messages	Ins	EO	Messages	Ins	EO	Messages
A1	1	<M,A,Req,x,M> <A,M,Rep,x,I,M,0>	B1	4		C1	8	
A2	2		B3	5		C2	9	
A4	3		B4	6		C4	10	
			B6	7				

How many messages are generated? _____

Problem M11.3.B

Is there an execution sequence that will generate even fewer messages? Fill in the EO columns to indicate the global execution order. Also, fill in the messages.

Processor A			Processor B			Processor C		
Ins	EO	Messages	Ins	EO	Messages	Ins	EO	Messages
A1			B1			C1		
A2			B3			C2		
A4			B4			C4		
			B6					

How many messages are generated? _____

Problem M11.3.C

Can the number of messages in Problem M11.3.B be decreased *by using voluntary responses*? Explain.

Problem M11.3.D

What is the execution sequence that generates the most messages *without any voluntary responses*? Fill in the global execution order (EO) and the messages generated. Partial credit will be given for identifying a bad, but not necessarily the worst sequence.

Processor A			Processor B			Processor C		
Ins	EO	Messages	Ins	EO	Messages	Ins	EO	Messages
A1			B1			C1		
A2			B3			C2		
A4			B4			C4		
			B6					

How many messages are generated? _____

Problem M11.4: Snoopy Cache Coherent Shared Memory

In this problem, we investigate the operation of the snoopy cache coherence protocol in Handout #14.

The following questions are to help you check your understanding of the coherence protocol.

- Explain the differences between **CR, CI,** and **CRI** in terms of their purpose, usage, and the actions that must be taken by memory and by the different caches involved.
- Explain why **WR** is not snooped on the bus.
- Explain the I/O coherence problem that **CWI** helps avoid.

Problem M11.4.A Where in the Memory System is the Current Value

In Table M11.4-1, M11.4-2, and M11.4-3, column 1 indicates the initial state of a certain address X in a cache. Column 2 indicates whether address X is currently cached in any other cache. (The "cached" information is known to the cache controller only immediately following a bus transaction. Thus, the action taken by the cache controller must be independent of this signal, but state transition could depend on this knowledge.) Column 3 enumerates all the available operations on address X, either issued by the CPU (read, write), snooped on the bus (**CR, CRI, CI**. etc), or initiated by the cache itself (replacement). Some state-operation combinations are impossible; you should mark them as such. (See the first table for examples). In columns 6, 7, and 8 (corresponding to this cache, other caches and memory, respectively), **check all possible locations where up-to-date copies of this data block could exist after the operation in column 3 has taken place** and ignore column 4 and 5 for now. Table M11.4-1 has been completed for you. Make sure the answers in this table make sense to you.

Problem M11.4.B MBus Cache Block State Transition Table

In this problem, **we ask you to fill out the state transitions in Column 4 and 5.** In column 5, fill in the resulting state after the operation in column 3 has taken place. In column 4, list the necessary MBus transactions that are issued by the cache as part of the transition. Remember, the protocol should be optimized such that data is supplied using **CCI** *whenever possible*, and only the cache that ***owns*** a line should issue **CCI**.

Problem M11.4.C **Adding atomic memory operations to MBus**

We have discussed the importance of atomic memory operations for processor synchronization. In this problem you will be looking at adding support for an atomic fetch-and-increment to the MBus protocol.

Imagine a dual processor machine with CPUs A and B. Explain the difficulty of CPU A performing fetch-and-increment(x) when the most recent copy of x is cleanExclusive in CPU B's cache. You may wish to illustrate the problem with a short sequence of events at processor A and B.

Fill in the rest of the table below as before, indicating state, next state, where the block in question may reside, and the CPU A and MBus transactions that would need to occur atomically to implement a fetch-and-increment on processor A.

State	other cached	ops	actions by this cache	next state	this cache	other caches	mem
Invalid	yes	read					
		write					

initial state	other cached	ops	actions by this cache	final state	this cache	other caches	mem
Invalid	no	none	none	**I**			√
		CPU read	**CR**	**CE**	√		√
		CPU write	**CRI**	**OE**	√		
		replace	none	*Impossible*			
		CR	none	**I**		√	√
		CRI	none	**I**		√	
		CI	none	*Impossible*			
		WR	none	*Impossible*			
		CWI	none	**I**			√
Invalid	yes	none	same as above	**I**		√	√
		CPU read		**CS**	√	√	√
		CPU write		**OE**	√		
		replace		*Impossible*			
		CR		**I**		√	√
		CRI		**I**		√	
		CI		**I**		√	
		WR		**I**		√	√
		CWI		**I**			√

initial state	other cached	ops	actions by this cache	final state	this cache	other caches	mem
cleanExclusive	no	none	none	**CE**			
		CPU read					
		CPU write					
		replace					
		CR		**CS**			
		CRI					
		CI					
		WR					
		CWI					

Table M11.4-1

initial state	other cached	ops	actions by this cache	final state	this cache	other caches	mem
ownedExclusive	no	none	none	**OE**			
		CPU read					
		CPU write					
		replace					
		CR		**OS**			
		CRI					
		CI					
		WR					
		CWI					

initial state	other cached	ops	actions by this cache	final state	this cache	other caches	mem
cleanShared	no	none	none	**CS**			
		CPU read					
		CPU write					
		replace					
		CR					
		CRI					
		CI					
		WR					
		CWI					
cleanShared	yes	none	same as above				
		CPU read					
		CPU write					
		replace					
		CR					
		CRI					
		CI					
		WR					
		CWI					

Table M11.4-2

initial state	other cached	ops	actions by this cache	final state	this cache	other caches	mem
ownedShared	no	none	none	**OS**			
		CPU read					
		CPU write					
		replace					
		CR					
		CRI					
		CI					
		WR					
		CWI					
ownedShared	yes	none	same as above				
		CPU read					
		CPU write					
		replace					
		CR					
		CRI					
		CI					
		WR					
		CWI					

Table M11.4-3

Problem M11.5: Snoopy Cache Coherent Shared Memory

This problem improves the snoopy cache coherence protocol presented in Handout #14. As a **review** of that protocol:

> When multiple shared copies of a *modified* data block exist, one of the caches *owns* the current copy of the data block instead of the memory (the owner has the data block in the OS state). When another cache tries to retrieve the data block from memory, the owner uses *cache to cache intervention* (CCI) to supply the data block. CCI provides a faster response relative to memory and reduces the memory bandwidth demands. However, when multiple shared copies of a *clean* data block exist, there is no owner and CCI is *not* used when another cache tries to retrieve the data block from memory.

To enable the use of CCI when multiple shared copies of a *clean* data block exist, we introduce a new cache data block state: *Clean owned shared* (COS). This state can only be entered from the clean exclusive (CE) state. The state transition from CE to COS is summarized as follows:

initial state	other cached	ops	actions by this cache	final state
cleanExclusive (CE)	no	**CR**	**CCI**	**COS**

There is no change in cache bus transactions but a slight modification of cache data block states. Here is a summary of the possible cache data block states (**differences from problem set highlighted in bold**):

- Invalid (**I**): Block is not present in the cache.
- Clean exclusive (**CE**): The cached data is consistent with memory, and no other cache has it. **This cache is responsible for supplying this data instead of memory when other caches request copies of this data.**
- Owned exclusive (**OE**): The cached data is different from memory, and no other cache has it. This cache is responsible for supplying this data instead of memory when other caches request copies of this data.
- Clean shared (**CS**): The data has not been modified by the corresponding CPU since cached. Multiple **CS** copies and at most one **OS** copy of the same data could exist.
- Owned shared (**OS**): The data is different from memory. Other **CS** copies of the same data could exist. This cache is responsible for supplying this data instead of memory when other caches request copies of this data. (Note, this state can only be entered from the **OE** state.)
- **Clean owned shared (COS): The cached data is consistent with memory. Other CS copies of the same data could exist. This cache is responsible for supplying this data instead of memory when other caches request copies of this data. (Note, this state can only be entered from the CE state.)**

Problem M11.5.A

Fill out the state transition table for the new COS state:

initial state	other cached	ops	actions by this cache	final state
COS	yes	none	none	**COS**
		CPU read		
		CPU write		
		replace		
		CR		
		CRI		
		CI		
		WR		
		CWI		

Problem M11.5.B

The COS protocol is not ideal. Complete the following table to show an example sequence of events in which multiple shared copies of a clean data block (*block B*) exist, but CCI is *not* used when another cache (*cache 4*) tries to retrieve the data block from memory.

cache transaction	source for data	state for data block B			
		cache 1	cache 2	cache 3	cache 4
0. *initial state*	**—**	**I**	**I**	**I**	**I**
1. cache 1 reads data block B	**memory**	**CE**	**I**	**I**	**I**
2. cache 2 reads data block B	**CCI**	**COS**	**CS**	**I**	**I**
3. cache 3 reads data block B	**CCI**	**COS**	**CS**	**CS**	**I**
4.					
5.					

Problem M11.5.C

As an alternative protocol, we could eliminate the CE state entirely, and transition directly from I to COS when the CPU does a read and the data block is not in any other cache. This modified protocol would provide the same CCI benefits as the original COS protocol, but its performance would be worse. **Explain the advantage of having the CE state.** You should not need more than one sentence.

Problem M11.6: Snoopy Caches

This part explores multi-level caches in the context of the bus-based snoopy protocol discussed in Lecture 14 (2017). Real systems usually have at least two levels of cache, smaller, faster L1 cache near the CPU, and the larger but slower L2. The two caches are usually inclusive, that is, any address in L1 is required to be present in L2. L2 is able to answer every snooper inquiry immediately but usually operates at 1/2 to 1/4th the speed of CPU-L1 interface. For performance reasons it is important that snooper steals as little bandwidth as possible from L1, and does not increase the latency of L2 responses.

Problem M11.6.A

Consider a situation when the L2 cache has a cache line marked Sh, and an ExReq comes on the bus for this cache line. The snooper asks both L1 and L2 caches to invalidate their copies but responds OK to the request, even before the invalidations are complete. Suppose the CPU ends up reading this value in L1 before it is truly discarded. What must the cache and snooper system do to ensure that sequential consistency is not violated here?

Hint: Consider how much processing can be performed safely on the following sequences after an invalidation request for x has been received

Ld x; Ld y; Ld x

Ld x; St y; Ld x

Problem M11.6.B

Consider a situation when L2 has a cache line marked Ex and a ShReq comes on the bus for this cache line. What should the snooper do in this case, and why?

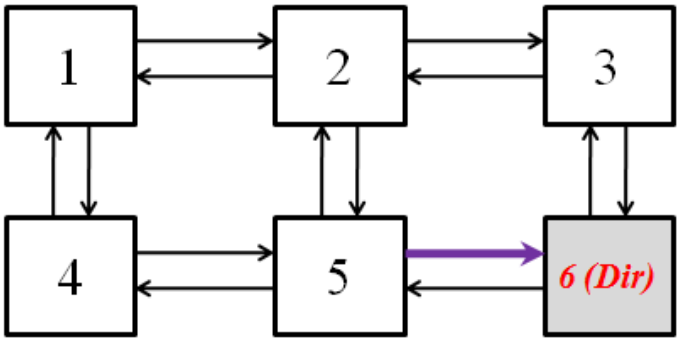
Problem M11.6.C

When an ExReq message is seen by the snooper and there is a Wb message in the C2M queue waiting to be sent, the snooper replies ***retry***. If the cache line is about to be modified by another processor, why is it important to first write back the already modified cache line? Does your answer change if cache lines are restricted to be one word? Explain.

Problem M11.7: Directory-based Protocol

Problem M11.7.A

The following questions deal with the directory-based protocol discussed in class. Assume XY routing, and message passing is FIFO. (**XY routing algorithm** first routes packets horizontally, towards their X coordinates, and then vertically towards their Y coordinates.) Protocol messages with the same source and destination sites are always received in the same order as that in which they were sent. **For this question, assume that the cache coherence protocol is free from deadlock, livelock and starvation**.



Assume the node 6 serves as the home directory, where the states for memory blocks are stored. Assume all caches are initially empty and no responses are sent voluntarily (i.e. every response is caused by a request)

Processor 1	Processor 4	Processor 5
I1.1: ST X, 10	I4.1: LD R1, X	I5.1: ST X, 20

Suppose the global execution order is as follows:

I4.1 => I5.1 => I1.1

Assume that the next instruction will start its execution only when the previous instruction has completed. For each instruction, list all protocol messages that are sent over the link 5 -> 6 (the purple link in the above figure).

I4.1:

I5.1:

I1.1:

Problem M11.7.B

For the directory protocol, we assume the message passing to be FIFO, meaning protocol messages with the same source and destination are always received in the same order as that in which they were sent. Now suppose messages can be delivered out-of-order for the same source and destination pairs. Describe one scenario that the cache coherence protocol will break due to this out-of-order delivery.

Problem M11.7.C

Under the 6823 directory-based protocol, a cache will receive a writeback request from the directory <M2C_Req, a, S> for address "a" when it is in state M and another cache wants a shared copy. Is it possible for a cache in the **S** state to receive <M2C_Req, a, S> ? Describe how this scenario can occur using the messages passed between the cache and the memory, and the state transitions.

Problem M11.8: Synchronicity (Spring 2014 Quiz 4, Part B)

You are writing a queue to be used in a multi-producer/single-consumer application. (Producer threads write messages that are read by one consumer.) We assume here a queue with infinite space. The basic code is shown below.

`TST rs, Imm(rt)` is the test-and-set instruction, which *atomically* loads the value at `Imm(rt)` into `rs`, and if the value is zero, updates the memory location at `Imm(rt)` to 1. This atomic instruction is useful for implementing locks: a value of 1 at the memory location indicates that someone holds the lock, and a value of 0 means the lock is free.

Producer pushes a message onto queue: (memory operations in bold)

```
void push(int** tail_ptr, int* tail_write_lock, int message) {
      while (lock_try(tail_write_lock) == false);
      **tail_ptr = message;
      *tail_ptr++;
      lock_release(tail_write_lock);
}

# R1 – contains address of data to enqueue
# R2 – contains the address of the tail pointer of queue
# R3 – address of tail pointer write lock
P1 SpinLock:TST R4, 0(R3)           # try to acquire tail write lock
P2          BNEZ R4, R4, SpinLock
P3          LD R4, 0(R2)            # get tail pointer
P4          ST R1, 0(R4)            # write message to tail
P5          ADD R4, R4, 4           # update tail pointer
P6          ST R4, 0(R2)
P7          ST R0, 0(R3)            # release lock
```

Consumer pops a message off queue: (memory operations in bold)

```
int pop(int** head_ptr, int** tail_ptr) {
      while (*head_ptr == *tail_ptr);
      int message = **head_ptr;
      *head_ptr++;
      return message;
}

# R1 – will receive address contained in message
# R2 – contains the address of the head pointer of queue
# R3 – contains the address of the tail pointer of the queue
C1 Retry:   LD R4, 0(R2)            # get head pointer
C2          LD R5, 0(R3)            # get tail pointer
C3          SUB R5, R4, R5          # is there a message?
C4          BNEZ R5, Pop
C5          JMP Retry
C6 Pop:     LD R1, 0(R4)            # read message from queue
C7          ADD R4, R4, 4           # update head pointer
C8          ST R4, 0(R2)
```

Problem M11.8.A

You are trying to port this code to an architecture that does not have the `TST` instruction (but, happily, the rest of the ISA is unchanged). Instead the new architecture has load-reserve/store-conditional instructions. Implement `TST rs, 0(rt)` using load-reserve/store-conditional:

```
LR rs, Imm(rt):
     rs ← Memory[(rt) + Imm]
     Track address (rt) + Imm

SC rs, Imm(rt):
     If (rt) + Imm modified:
          rs ← 0                          # Fail
     Else:
          Memory[(rt) + Imm] = (rs)       # Succeed
          rs ← 1
```

Problem M11.8.B

This new architecture is also *not* sequentially consistent. Give an example of memory orderings between the producer and consumer that would result in incorrect behavior. *Explain your answer fully or you will not receive credit.*

Your answer should look something like:
`P1, P3, P4, C1, C2, P6, P7, C1, C2, C6, C8`
(Except that this is a sequentially consistent ordering, so it is not a correct answer.)

Problem M11.8.C

Show where memory fences should be added to the producer and consumer code to ensure correctness with a weak consistency model. Explain your answer fully.

```
P1 SpinLock:TST R4, 0(R3)          # try to acquire tail write lock

P2          BNEZ R4, R4, SpinLock

P3          LD R4, 0(R2)           # get tail pointer

P4          ST R1, 0(R4)           # write message to tail

P5          ADD R4, R4, 4          # update tail pointer

P6          ST R4, 0(R2)

P7          ST R0, 0(R3)           # release lock



C1 Retry:   LD R4, 0(R2)           # get head pointer

C2          LD R5, 0(R3)           # get tail pointer

C3          SUB R5, R4, R5         # is there a message?

C4          BNEZ R5, Pop

C5          JMP Retry

C6 Pop:     LD R1, 0(R4)           # read message from queue

C7          ADD R4, R4, 4          # update head pointer

C8          ST R4, 0(R2)
```

Problem M11.8.D

Let's next consider performance with a single producer thread and consumer thread. The following happens repeatedly:

1. The producer executes all instructions to push a message on the queue.
2. The consumer executes all instructions to pop a message off the queue.

Assume data, head, and tail pointers all lie in different, non-conflicting cache blocks.

First, after a few messages have been sent through the queue, will the consumer ever miss reading the head pointer? Will the producer ever miss reading the tail write lock, or fail to acquire the tail write lock? Explain in one or two sentences.

Problem M11.8.E

We'll now focus on the tail pointer only. Assuming a MSI invalidate coherence protocol, show the state of the tail pointer in the producer and consumer cache after each operation in the sequence below. Show any data or permissions transfers, e.g. "Memory→C" or "C invalidates P".

Operation	Producer tail pointer state	Consumer tail pointer state	Transfers
	I	I	
P1 TST try lock			
P3 LD tail_ptr			
P4 ST message			
P6 ST new_tail			
P7 ST release lock			
C1 LD head_ptr			
C2 LD tail_ptr			
C6 LD message			
C7 ST new_head			
P1 TST try lock			
P3 LD tail_ptr			
P4 ST message			
P6 ST new_tail			
P7 ST release lock			
C1 LD head_ptr			
C2 LD tail_ptr			
C6 LD message			
C7 ST new_head			

How many state transitions occur per message in the steady state?

Problem M11.8.F

Stay focused on the tail pointer only. Assume an update coherence protocol where the state of each line is either valid (V) or invalid (I). Show the state of the tail pointer in the producer and consumer cache after each operation in the sequence below in the steady state. Show any data or permissions transfers, e.g. "Memory→C" or "C invalidates P".

Operation	Producer tail pointer state	Consumer tail pointer state	Transfers
	I	I	
P1 TST try lock			
P3 LD tail_ptr			
P4 ST message			
P6 ST new_tail			
P7 ST release lock			
C1 LD head_ptr			
C2 LD tail_ptr			
C6 LD message			
C7 ST new_head			
P1 TST try lock			
P3 LD tail_ptr			
P4 ST message			
P6 ST new_tail			
P7 ST release lock			
C1 LD head_ptr			
C2 LD tail_ptr			
C6 LD message			
C7 ST new_head			

How many state transitions occur per message in the steady state?

Problem M11.8.G

Your new architecture supports "remote access" for cached lines. This lets you assign a "home cache" for lines so that all memory operations will be sent *over the network* to operate remotely on the line *without allocating it in the requesting cache*.

For example, if line 0x100 is homed to processor A, and processor B writes 0x100, then *processor A's cache will be updated* and processor B's will be unchanged.

Assume the tail pointer is mapped to the producer's cache, and the cache uses an MSI invalidate protocol (similar to Question 5). Once again, show the state of the tail pointer for the sequence of operations in the steady state and data/permission transfers:

Operation	Producer tail pointer state	Consumer tail pointer state	Transfers
	I	I	
P1 TST try lock			
P3 LD tail_ptr			
P4 ST message			
P6 ST new_tail			
P7 ST release lock			
C1 LD head_ptr			
C2 LD tail_ptr			
C6 LD message			
C7 ST new_head			
P1 TST try lock			
P3 LD tail_ptr			
P4 ST message			
P6 ST new_tail			
P7 ST release lock			
C1 LD head_ptr			
C2 LD tail_ptr			
C6 LD message			
C7 ST new_head			

How many state transitions occur per message in the steady state?

Problem M11.9: Cache Coherence (Spring 2015 Quiz 3, Part B)

Ben Bitdiddle is designing a snoopy-based, write-invalidate MSI protocol for write-back caches. Under the standard MSI protocol, when a cache observes a Bus Read Exclusive message (BusRdX), it has to invalidate its own copy of the cache block. Ben instead proposes an optimization, called delayed invalidation, to potentially reduce the number of read misses. The optimization works as follows:

Delayed invalidation: When a cache observes a Bus Read Exclusive message (BusRdX) and it has a copy of the block in the Shared (S) state, the cache delays the invalidation of the block until before a cache miss happens. In other words, the cache will treat any subsequent requests from its own processor as if the BusRdX had not happened, until one of those requests causes a miss. At that point, all pending invalidations are performed before processing the miss.

Problem M11.9.A

Suppose processors P1 and P2 are have private, snoopy caches. Both caches are initially empty. Consider the following sequence of accesses:

```
I0   P2: read   A
I1   P1: write  A
I2   P2: read   A
I3   P1: write  A
I4   P2: read   A
I5   P2: read   B
I6   P2: read   A
```

Assume blocks A and B do not conflict in the cache. Compare Ben's delayed invalidation optimization with the standard MSI protocol by filling the states (on the next page) for each cache block after each operation is done and calculate the number of misses in both cases.

Assume we use the standard MSI protocol. Fill in the following table.

Standard MSI Protocol				
	Processor P1's Cache		Processor P2's Cache	
Initial State	A: I	B: I	A: I	B: I
After P2 reads A	A: I	B: I	A: S	B: I
After P1 writes A	A:	B:	A:	B:
After P2 reads A	A:	B:	A:	B:
After P1 writes A	A:	B:	A:	B:
After P2 reads A	A:	B:	A:	B:
After P2 reads B	A:	B:	A:	B:
After P2 reads A	A:	B:	A:	B:

How many misses occur in the two caches?

Assume we adopt Ben's delayed invalidation optimization. Fill in the following table. If there is a delayed invalidation, write it in the invalidation queue (the "Inv Queue" column). For example, "Inv L" means there is a delayed invalidation on block L.

MSI Protocol with Delayed Invalidation						
	Processor P1's Cache			Processor P2's Cache		
	MSI state		Inv Queue	MSI state		Inv Queue
Initial State	A: I	B: I		A: I	B: I	
After P2 reads A	A: I	B: I		A: S	B: I	
After P1 writes A	A:	B:		A:	B:	
After P2 reads A	A:	B:		A:	B:	
After P1 writes A	A:	B:		A:	B:	
After P2 reads A	A:	B:		A:	B:	
After P2 reads B	A:	B:		A:	B:	
After P2 reads A	A:	B:		A:	B:	

How many misses occur in the two caches?

Problem M11.9.B

Does Ben's delayed invalidation optimization violate cache coherence rules? Please explain your answer in one or two sentences.

Problem M11.9.C

Suppose the original system guarantees sequential consistency. Does adding the delayed invalidation optimization break sequential consistency? Please explain your answer in one or two sentences. If your answer is yes, please provide a sequence of load/store operations that violates sequential consistency.

Problem M11.9.D

Ben only applies delayed invalidation on cache blocks that are in the S state. When a cache observes a Bus Read Exclusive message (BusRdX) and the associated cache block is in the Modified (M) state, it sends out the data in response to a BusRdX message and changes the cache state to Invalid (I).

Is it possible to delay invalidation when the cache block is in the Modified (M) state? If it is not, please explain why. If it is possible, please describe how to make delayed invalidations work when the block is in the M state. In other words, please describe the actions the cache needs to take when the cache observes a BusRdX message, how to handle subsequent read and write accesses if the invalidation is delayed, and when the invalidation needs to be processed.

Problem M11.10: Cache Coherence (Spring 2015 Quiz 3, Part C)

Please use Handout #15 to answer the questions in this part.

Problem M11.10.A

Ben designs an architecture that does not have the atomic compare-and-swap (`CAS`) instruction but has load-reserve (`LR`) and store-conditional (`SC`) instructions.

Help Ben implement a Boolean compare-and-swap instruction `BCAS old, new, Imm(base)` using load-reserve and store-conditional instructions:

```
LR rs, Imm(rt):
     <flag, addr> ← <1, rt + Imm>
     rs ← Memory[rt + Imm]

SC rs, Imm(rt):
     If <flag, addr> == <1, rt + Imm>:
          Memory[rt + Imm] ← rs
          rs ← 1                  # Succeed
     Else:
          rs ← 0                  # Fail
```

BCAS is a simplified CAS instruction that only deals with values 0 and 1. You can use temporary registers (`tmp1, tmp2, tmp3…`) and any algorithmic, logical, memory, and branch instructions in the MIPS instruction set.

Problem M11.10.B

Suppose the hardware where the shared-memory queue from Handout #15 is executed has a weak consistency model that relaxes all the orderings of reads and writes. Give an example of memory orderings between the producer and consumer that would result in incorrect behavior. *Please fully explain your answer to get full credit.*

Your memory ordering example should look something like:
`P1, C2, P2, C4, P4, C5, C7, C9, C10`

Problem M11.10.C

Please add the minimum number of memory fences ($\texttt{FENCE}_{WR}$, $\texttt{FENCE}_{RW}$, $\texttt{FENCE}_{WW}$, or $\texttt{FENCE}_{RR}$) to the producer and consumer codes to ensure correctness with a weak consistency model. Please explain your answer fully.

Code for producer to enqueue a message:

```
P1:  LD  R3, 0(R2)  # get tail pointer

P2:  ST  R1, 0(R3)  # write message to tail

P3:  ADD R3, R3, 4  # update tail pointer

P4:  ST  R3, 0(R2)
```

Code for consumer to dequeue a message:

```
C1: SpinLock: MOV  R6, R0         # set R6 to 0

C2:           CAS  R6, R5, 0(R4) # try to acquire lock

C3:           BNEZ R6, SpinLock

C4:           LD   R7, 0(R2)     # get head pointer

C5: Retry:    LD   R8, 0(R3)     # get tail pointer

C6:           BEQ  R7, R8, Retry # is there a message?

C7:           LD   R1, 0(R7)     # read message from queue

C8:           ADD  R7, R7, 4     # update head pointer

C9:           ST   R7, 0(R2)

C10:          ST   R0, 0(R4)     # release lock
```

Problem M12.1: Networks-on-Chip

Problem M12.1.A

Consider a flow control method similar to circuit switching but where the request message 'reserves' each channel for a fixed period of time in the future (for example, for 10 cycles since a reservation is made). At each router along the path, a reservation is made if a request from a neighbor can be accommodated. If the request cannot be accommodated a NACK is sent that cancels all previous recommendations for the connection, and the request is retired. If a request reaches the destination, an acknowledgement is sent back to the source, confirming all reservations.

Draw a time-space diagram of a situation that demonstrates the advantage of reservation circuit switching over conventional circuit switching.

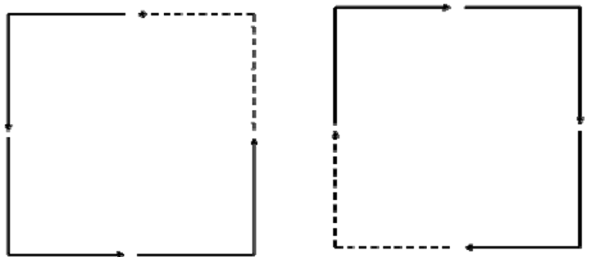
Problem M12.1.B

Determine whether the following oblivious routing algorithms are deadlock-free for the 2-D mesh. There is only one virtual channel per link and no 180-degree turns are allowed for (c).

(a) Randomized dimension-order: All packets are routed minimally. Half of the packets are routed completely in the X dimension before the Y dimension and the other packets are routed Y before X.

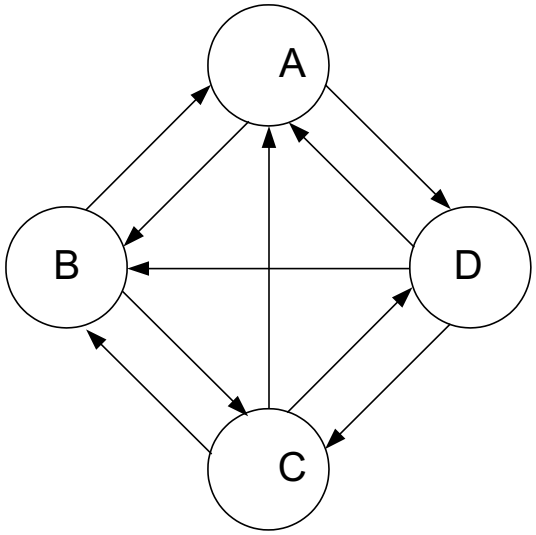
(b) Less randomized dimension-order: All packets are routed minimally. Packets whose minimal direction is increasing in both X and Y, always route X before Y. Packets whose minimal direction is decreasing in both X and Y, always route Y before X. All other packets randomly choose between X before Y and vice versa.

(c) All packets are prohibited to take the two turns in dash:



Problem M12.2: Non-mesh Networks

We have the following network topology with 4 network nodes and 10 links.



Note that each link is unidirectional, and only one link exists between A and C (only a link from C to A (not from A to C), and only from D to B between B and D. Each link can transfer 1 flit per cycle and there is only one virtual channel per link. For all parts, 180-degree turns are not allowed.

Problem M12.2.A

Fill in the following table of the properties of this network.

Diameter	
Average Distance	
Bisection Bandwidth	

Problem M12.2.B

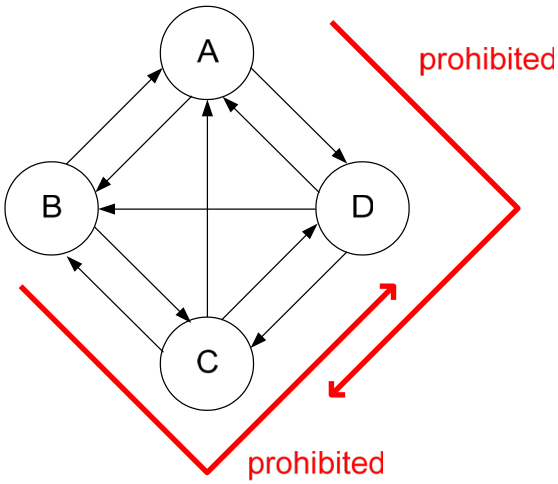
Draw the channel dependency graph of this network.

Problem M12.2.C

Is a **minimal** routing on this network deadlock-free? Show your reasoning and give a deadlock scenario if it is not deadlock -free.

Problem M12.2.D

Now, we use a possibly **non-minimal** routing on this network. Plus, we prohibited the following two movements on the non-minimal routing: 1) A to D then D to C and 2) B to C then C to D.



Is this routing deadlock-free? Show your reasoning and give a deadlock scenario if it is not deadlock -free.

Problem M12.2.E

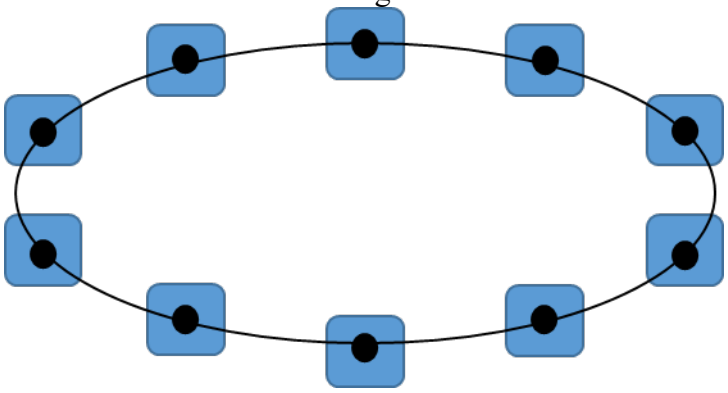
Still having the two movements in M12.2.D prohibited, we added another restriction in routing: the link from C to A can be used only by packets generated at C, before the packets are transferred to any other nodes (it should be the first link those packets ever take). Also, the link from D to B can be used only by packets generated at D with the same condition (however, routes may be non-minimal).

Is this routing deadlock-free? Show your reasoning and give a deadlock scenario if it is not deadlock -free.

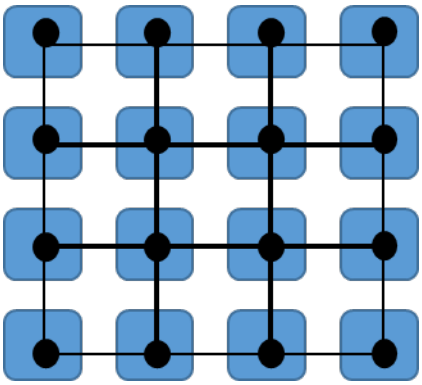
Problem M12.3: Network Effects (Spring 2014 Quiz 4, Part A)

You are choosing between several network topologies for your on-chip network, shown below.

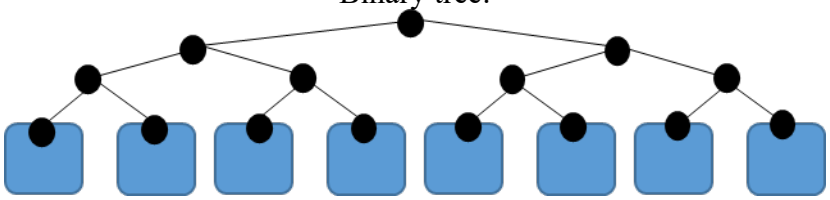
Ring:



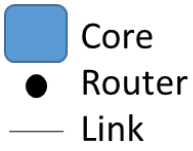
Mesh:



Binary tree:



Legend:

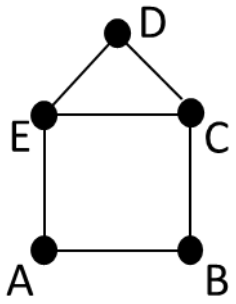


Problem M12.3.A

Your first task is to evaluate these topologies along several important dimensions. Fill in the table below as a function of the number of nodes in the network, *N*. You can safely assume *N* is an even power of 2, giving a complete mesh and binary tree. *For partial credit, give the asymptotic growth instead.*

	Ring	Mesh	Tree
Number of links			
Diameter			
Average distance			
Bisection bandwidth			

In a sudden flash of inspiration, you decide to use the following topology:



Having decided upon a topology, you now want to make sure your system works properly. All links are bidirectional.

Problem M12.3.B

Show how deadlock could arise in the network by drawing an example on the graph above. Explain your answer in one or two sentences.

Problem M12.3.C

Draw the channel dependency graph (CDG) for your topology.

Show an example of how to eliminate routes to prevent deadlock on the CDG.

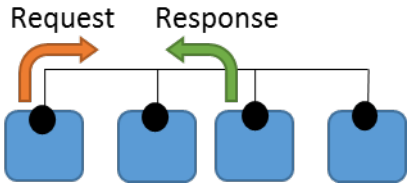
Problem M12.4: The Truth Will Set You Free (Spring 2014 Quiz 4, Part III)

Problem M12.4.A — Peanuts

Snoopy coherence protocols rely on broadcast communication to detect sharing and updates. These are conventionally implemented using bus networks that allow for one message to be sent at a time to all nodes on the network.

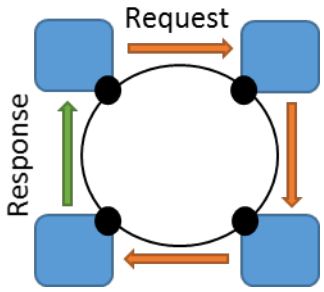
Ben Bitdiddle is implementing a bus-based snoopy coherence protocol. One fifth of instructions access memory, and one quarter of these miss in the core's local cache (either because the line is invalid or doesn't have necessary permissions). Assuming each memory operation consists of a request and acknowledgement, the network traffic per core is therefore: $\frac{1}{5} \times \frac{1}{4} \times 2 = \frac{1}{10} \frac{\text{messages}}{\text{instruction}}$. Assume all messages fit within a single network flit.

Assuming a fixed IPC of 1, perfect bus arbitration, and infinite buffers, how many cores can the bus support?



Problem M12.4.B **... To rule them all**

Ben needs to build a larger system than the bus network will allow, so he changes the system to use a unidirectional ring network. In this design, the core issuing the memory operation sends the request around the ring, and each node along the way either forwards the request or replaces it with its response. Assuming fixed IPC of 1 and a single-cycle per hop in the network, at how many cores will this design saturate?



Problem M12.4.C **Matryoshka**

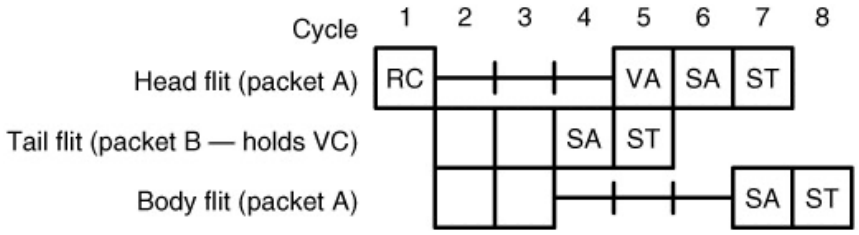
Ben next explores the tradeoffs in cache design between an inclusive cache, where the parent always has a copy of every line in the child's cache, and non-inclusive caches, where this isn't guaranteed.

Give one advantage and one disadvantage of a non-inclusive cache design.

Problem M12.5: Network-on-chip (Spring 2015 Quiz 3, Part A)

Problem M12.5.A

Consider the router in Handout 16. Assume this router **has one virtual channel per physical link**. Suppose two packets, A and B, are traversing the router. Both are routed to output unit 2, as shown in the following waterfall diagram.



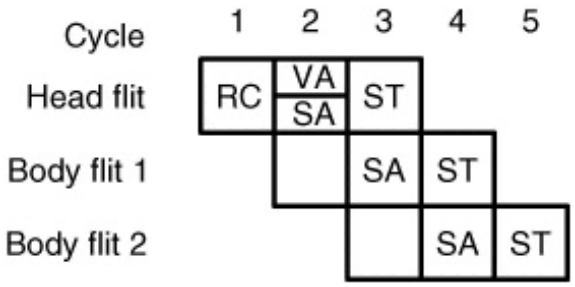
Before cycle 1, packet B's head flit has finished RC and VA. In the following cycles, packet B's four flits traverse SA and ST without stalls. Packet A's head flit completes routing computation at cycle 1 and tries to allocate an output virtual channel starting at cycle 2. Unfortunately, the only output virtual channel compatible with its route is occupied by packet B, so packet A's head flit fails to allocate a VC and is unable to make progress until packet B's tail flit releases the VC.

Fill in the following table showing the state of packet A's input virtual channel.

Cycle	**G**	**R**	**O**
1	R	-	-
2			
3			
4			
5			
6			
7			
8			

Problem M12.5.B

Suppose the router in Handout 16 is improved with **speculative switch allocation.** Head flits attempt VC and switch allocation in the same cycle. If both succeed, the head flit traverses the switch on the next cycle, as shown in the waterfall diagram below.



Consider the same scenario as in question 1, with packets A and B going to the same output unit. Assume that **non-speculative switch allocation requests are always prioritized over speculative ones** (i.e., those from flits without a VC). Fill in the following waterfall diagram to show how packet A is routed.

Cycle	1	2	3	4	5	6	7	8
A: Head Flit	RC							
A: Body Flit 1								
A: Body Flit 2								
B: Body Flit 1	SA	ST						
B: Body Flit 2	-	SA	ST					
B: Body Flit 3	-	-	SA	ST				
B: Tail Flit	-	-	-	SA	ST			

Problem M12.5.C

Consider the same speculative switch allocation optimization as in question 2. Unfortunately, always prioritizing non-speculative switch allocation requests over speculative ones increases the critical path too much, so we opt for a **simpler switch allocator that is oblivious to whether requests are speculative**.

We want to analyze the performance of this simpler design under the following scenario:

- All packets in the router are single-flit packets.
- The probability that a packet successfully obtains a VC on its first try is 75%.
- The probability that a flit successfully allocates the switch on its first try is 80%.
- If a packet fails either virtual channel or switch allocation on its first try, it always succeeds on its second try.

1) What percentage of allocated timeslots on the switch goes unused?

2) What is the average latency to go through this speculative router?

3) Briefly explain the effect of this optimization on network performance at both very low loads and very high loads (near saturation).

Problem M12.5.D

Ben Bitdiddle wants to implement the Valiant routing algorithm, which routes each packet through a randomly-chosen intermediate node. He uses routers with two virtual channels per physical link. He decides to use X-Y routing between the source node and intermediate node, and Y-X routing between the intermediate node and the destination node. However, Alyssa points out this routing algorithm will not work without further modification. Explain why this is the case and provide a solution for Ben.

Problem M12.6: Network-on-chip (Spring 2016 Quiz 3, Part D)

Problem M12.6.A

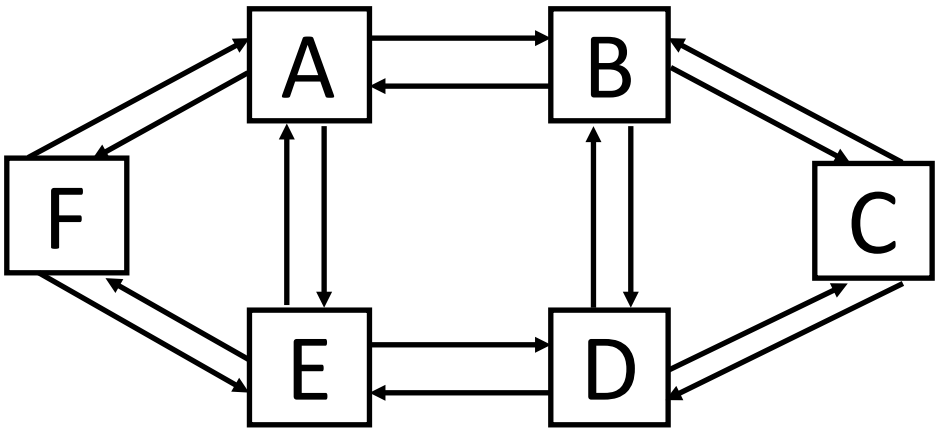
Determine whether the following routing algorithms are deadlock-free for a 2D-mesh. State your reasoning.

a) (3 points) Randomized dimension-order: All packets are routed minimally. Half of the packets are routed completely in the X dimension before the Y dimension, and the other packets are routed in the Y dimension before the X dimension.

b) (3 points) Less randomized dimension-order: All packets are routed minimally. Packets whose minimal direction is increasing in both X and Y always route **X before Y**. Packets whose minimal direction is decreasing in both X and Y always route **Y before X**. All other packets choose randomly between X before Y and vice-versa.

Problem M12.6.B

Consider the following topology:



(a) (2 points) What is the diameter of this topology?

(b) (2 points) What is the bisection bandwidth (in flits/cycle) of this topology?

(c) (5 points) Assume that 180-degree turns are prohibited. No other turns are prohibited. Show how deadlock could arise in the given topology.

(d) (10 points) We now restrict all routes to be minimal and disallow the following turns on the mesh (among the nodes A, B, E, D): north-to-east, north-to-west, south-to-east, south-to-west. Is the routing strategy deadlock-free? Draw the CDG to justify your answer.

Problem M13.1: Sequential Consistency

For this problem we will be using the following sequences of instructions. These are small programs, each executed on a different processor, each with its own cache and register set. In the following **R** is a register and **X** is a memory location. Each instruction has been named (e.g., B3) to make it easy to write answers.

Assume data in location X is initially 0.

Processor A	Processor B	Processor C
A1: ST X, 1	B1: R := LD X	C1: ST X, 6
A2: R := LD X	B2: R := ADD R, 1	C2: R := LD X
A3: R := ADD R, R	B3: ST X, R	C3: R := ADD R, R
A4: ST X, R	B4: R:= LD X	C4: ST X, R
	B5: R := ADD R, R	
	B6: ST X, R	

For each of the questions below, please circle the answer and provide a short explanation assuming the program is executing under the SC model. **No points will be given for just circling an answer!**

Problem M13.1.A

Can X hold value of 4 after all three threads have completed? Please explain briefly.

Yes / No

Problem M13.1.B

Can X hold value of 5 after all three threads have completed?

Yes / No

Problem M13.1.C

Can X hold value of 6 after all three threads have completed?

Yes / No

Problem M13.1.D

For this particular program, can a processor that reorders instructions but follows local dependencies produce an answer that cannot be produced under the SC model?

Yes / No

Problem M13.2: Relaxed Memory Models

Consider a system which uses Weak Ordering, meaning that a read or a write may complete before a read or a write that is earlier in program order if they are to different addresses and there are no data dependencies.

Our processor has four fine-grained memory barrier instructions:

- **MEMBAR$_{RR}$** guarantees that all read operations initiated before the MEMBAR$_{RR}$ will be seen before any read operation initiated after it.
- **MEMBAR$_{RW}$** guarantees that all read operations initiated before the MEMBAR$_{RW}$ will be seen before any write operation initiated after it.
- **MEMBAR$_{WR}$** guarantees that all write operations initiated before the MEMBAR$_{WR}$ will be seen before any read operation initiated after it.
- **MEMBAR$_{WW}$** guarantees that all write operations initiated before the MEMBAR$_{WW}$ will be seen before any write operation initiated after it.

We will study the interaction between two processes on different processors on such a system:

P1	P2
`P1.1: LW R2, 0(R8)`	`P2.1: LW R4, 0(R9)`
`P1.2: SW R2, 0(R9)`	`P2.2: SW R5, 0(R8)`
`P1.3: LW R3, 0(R8)`	`P2.3: SW R4, 0(R8)`

We begin with following values in registers and memory (same for both processes):

register/memory	Contents
R2	0
R3	0
R4	0
R5	8
R8	0x01234567
R9	0x89abcdef
M[R8]	6
M[R9]	7

After both processes have executed, is it possible to have the following machine state? Please circle the correct answer. If you circle **Yes**, please provide sequence of instructions that lead to the desired result (one sequence is sufficient if several exist). If you circle **No**, please explain which ordering constraint prevents the result.

Problem M13.2.A

Memory	contents
M[R8]	7
M[R9]	6

Yes **No**

Problem M13.2.B

memory	Contents
M[R8]	6
M[R9]	7

Yes **No**

Problem M13.2.C

Is it possible for M[R8] to hold 0?

Yes **No**

Now consider the same program, but with two **MEMBAR** instructions.

P1	P2
P1.1: LW R2, 0(R8)	P2.1: LW R4, 0(R9)
P1.2: SW R2, 0(R9)	$MEMBAR_{RW}$
$MEMBAR_{WR}$	P2.2: SW R5, 0(R8)
P1.3: LW R3, 0(R8)	P2.3: SW R4, 0(R8)

We want to compare execution of the two programs on our system.

Problem M13.2.D

If both M[R8] and M[R9] contain 6, is it possible for R3 to hold 8?

Without **MEMBAR** instructions? **Yes** **No**

With **MEMBAR** instructions? **Yes** **No**

Problem M13.2.E

If both M[R8] and M[R9] contain 7, is it possible for R3 to hold 6?

Without **MEMBAR** instructions? **Yes** **No**

With **MEMBAR** instructions? **Yes** **No**

Problem M13.2.F

Is it possible for both M[R8] and M[R9] to hold 8?

Without **`MEMBAR`** instructions? **Yes** **No**

With **`MEMBAR`** instructions? **Yes** **No**

Problem M13.3: Memory Models

Consider a system which uses **Sequential Consistency (SC)**. There are three processes, **P1**, **P2** and **P3**, on different processors on such a system (the values of R_A, R_B, R_C were all zeros before the execution):

P1	P2	P3
P1.1: ST (A), 1 P1.2: LD R_C, (C)	P2.1: ST (B), 1 P2.2: LD R_A, (A)	P3.1: ST (C), 1 P3.2: LD R_B, (B)

Problem M13.3.A

After all processes have executed, it is possible for the system to have multiple machine states. For example, $\{R_A, R_B, R_C\} = \{1,1,1\}$ is possible if the execution sequence of instructions is P1.1→P2.1→P3.1→P1.2→P2.2→P3.2. Also, $\{R_A, R_B, R_C\} = \{1,1,0\}$ is possible if the sequence is P1.1 → P1.2 → P2.1 → P3.1 → P2.2 → P3.2.

For each state of $\{R_A, R_B, R_C\}$ below, specify the execution sequence of instructions that results in the corresponding state. If the state is **NOT** possible with SC, just put X.

{0,0,0} :

{0,1,0} :

{1,0,0} :

{0,0,1} :

Problem M13.3.B

Now consider a system which uses **Weak Ordering(WO)**, meaning that a read or a write may complete before a read or a write that is earlier in program order if they are to different addresses and there are no data dependencies.

Does WO allow the machine state(s) that is not possible with SC? If yes, provide an execution sequence that will generate the machine states(s).

Problem M13.3.C

The WO system in Problem M13.3.B provides four fine-grained memory barrier instructions. Below is the description of these instructions.

- **$\text{MEMBAR}_{\text{RR}}$** guarantees that all read operations initiated before the $\texttt{MEMBAR}_{\texttt{RR}}$ will be seen before any read operation initiated after it.
- **$\text{MEMBAR}_{\text{RW}}$** guarantees that all read operations initiated before the $\texttt{MEMBAR}_{\texttt{RW}}$ will be seen before any write operation initiated after it.
- **$\text{MEMBAR}_{\text{WR}}$** guarantees that all write operations initiated before the $\texttt{MEMBAR}_{\texttt{WR}}$ will be seen before any read operation initiated after it.
- **$\text{MEMBAR}_{\text{WW}}$** guarantees that all write operations initiated before the $\texttt{MEMBAR}_{\texttt{WW}}$ will be seen before any write operation initiated after it.

Using the minimum number of memory barrier instructions, rewrite **P1**, **P2** and **P3** so the machine state(s) that is not possible with SC by the original programs is also not possible with WO by your programs.

P1	**P2**	**P3**
P1.1: ST (A), 1	P2.1: ST (B), 1	P3.1: ST (C), 1
P1.2: LD R_C, (C)	P2.2: LD R_A, (A)	P3.2: LD R_B, (B)

Problem M13.4: Memory consistency models (Spring 2016 Quiz 3, Part B)

Consider two processes P1 and P2 running on two different processors.
Assume that memory locations X and Y contain initial value 0.

P1	P2
P1.1: LD R1 ← (Y) P1.2: LD R2 ← (X)	P2.1: ST (X) ← 1 P2.2: ST (Y) ← 1

Problem M13.4.A

Out of the following possible final values of (X, Y, R1, R2), circle the ones that could occur if the system is **Sequentially Consistent (SC)**.

(0,0,0,0) **(0,0,0,1)** **(0,0,1,0)** **(0,0,1,1)**

(1,1,0,0) **(1,1,0,1)** **(1,1,1,0)** **(1,1,1,1)**

Problem M13.4.B

Out of the following possible final values of (X, Y, R1, R2), circle the ones that could occur if the system enforces **RMO**, a weak memory model where **loads and stores can be reordered after prior loads or stores.**

(0,0,0,0) **(0,0,0,1)** **(0,0,1,0)** **(0,0,1,1)**

(1,1,0,0) **(1,1,0,1)** **(1,1,1,0)** **(1,1,1,1)**

Problem M13.4.C

The RMO machine has the following fine-grained barrier instructions:

- **MEMBAR$_{RR}$** guarantees that all reads initiated before MEMBAR$_{RR}$ will be performed before any read initiated after it.
- **MEMBAR$_{RW}$** guarantees that all reads initiated before MEMBAR$_{RW}$ will be performed before any write initiated after it.
- **MEMBAR$_{WR}$** guarantees that all writes initiated before MEMBAR$_{WR}$ will be performed before any read initiated after it.
- **MEMBAR$_{WW}$** guarantees that all writes initiated before MEMBAR$_{WW}$ will be performed before any write initiated after it.

Use the **minimum number of memory barrier instructions**, rewrite **P1** and **P2** such that the **RMO machine produces the same outputs as the SC machine for the given code.**

P1	**P2**
P1.1: LD R1 ← (Y)	P2.1: ST (X) ← 1
P1.2: LD R2 ← (X)	P2.2: ST (Y) ← 1

Again, consider two processes P1 and P2 running the code below on two different processors. **Assume that memory locations X, Y, and Z contain initial value 0.**

P1	P2
P1.1: LD R1 ← (Z) P1.2: ST (Y) ← 1 P1.3: LD R2 ← (X)	P2.1: ST (X) ← 1 P2.2: LD R3 ← (Y) P2.3: ST (Z) ← 1

Problem M13.4.D

Out of the following possible final values of (R1, R2, R3), circle the ones that could occur if the system is **Sequentially Consistent (SC)**.

(0,0,0) **(0,1,0)** **(1,0,0)** **(1,1,0)**

(0,0,1) **(0,1,1)** **(1,0,1)** **(1,1,1)**

Problem M13.4.E

Out of the following possible final values of (R1, R2, R3), circle the ones that could occur if the system enforces **RMO** (**loads and stores can be reordered after prior loads or stores**).

(0,0,0) **(0,1,0)** **(1,0,0)** **(1,1,0)**

(0,0,1) **(0,1,1)** **(1,0,1)** **(1,1,1)**

Problem M13.4.F

Using the **minimum number of memory barrier instructions** (given in Question 3), rewrite **P1** and **P2** such that the **RMO machine produces the same outputs as the SC machine for the given code.**

P1	**P2**
P1.1: LD R1 ← (Z)	P2.1: ST (X) ← 1
P1.2: ST (Y) ← 1	P2.2: LD R3 ← (Y)
P1.3: LD R2 ← (X)	P2.3: ST (Z) ← 1