6.823 Computer System Architecture
Module #1 Solutions

Last Updated: 2/20/2015

http:/csg.csail.mit.edu/6.823/

Problem M1.1: Self Modifying Code on the EDSACjr

Problem M1.1.A — Writing Macros For Indirection

One way to implement `ADDind n` is as follows:

```
.macro ADDind(n)
      STORE       orig_accum  ; Save original accum
      CLEAR                   ; accum <- 0
      ADD         n           ; accum <- M[n]
      ADD         _add_op     ; accum <- ADD M[n]
      STORE       _L1         ; M[_L1] <- ADD M[n]
      CLEAR                   ; accum <- 0
_L1:  CLEAR                   ; This will be replaced by
                              ; ADD M[n] and will have
                              ; the effect: accum <- M[M[n]]
      ADD         _orig_accum ; accum <- M[M[n]] + original accum
.end macro
```

The first thing we do is save the original accumulator value. This is necessary since the instructions we are going to use within the macro are going to destroy the value in the accumulator. Next, we load the contents of `M[n]` into the accumulator. We assume that M[n] is a legal address and fits in 11 bits.

After getting the value of `M[n]` into the accumulator, we add it to the `ADD` template at `_add_op`. Since the template has 0 for its operand, the resulting number will have the `ADD` opcode with the value of `M[n]` in the operand field, and thus will be equivalently an `ADD M[n]`. By storing the contents of the accumulator into the address `_L1`, we replace the `CLEAR` with what is equivalently an `ADD M[n]` instruction. Then we clear the accumulator so that when the instruction at `_L1` is executed, `accum` will get `M[M[n]]`. Finally, we add the original accumulator value to get the desired result, `M[M[n]]` plus the original content of the accumulator.

`STOREind n` can be implemented in a very similar manner.

```
.macro STOREind(n)
      STORE       _orig_accum ; Save original accum
      CLEAR                   ; accum <- 0
      ADD         n           ; accum <- M[n]
      ADD         _store_op   ; accum <- STORE M[n]
      STORE       _L1         ; M[_L1] <- STORE M[n]
      CLEAR                   ; accum <- 0
      ADD         _orig_accum ; accum <- original accum
_L1:  CLEAR                   ; This will be replaced by
                              ; STORE M[n], and will have the
                              ; effect: M[M[n]]<- orig. accum
.end macro
```

After getting the value of `M[n]` into the accumulator, we add it to the `STORE` template at `_store_op`. Since the template has 0 for its operand, the resulting number will have the `STORE` opcode with the value of `M[n]` in the operand field, and thus will be equivalently a `STORE M[n]` instruction. As before, we store this into `_L1` and then restore the accumulator value to its original value. When the PC reaches `_L1`, it then stores the original value of the accumulator into `M[M[n]]`.

`BGEind` and `BLTind` are very similar to `STOREind`. `BGEind` is shown below. `BLTind` is the same except that we use `_blt_op` instead of `_bge_op`.

```
.macro BGEind(n)
      STORE       _orig_accum ; Save original accum
      CLEAR                   ; accum <- 0
      ADD         n           ; accum <- M[n]
      ADD         _bge_op     ; acuum <- BGE M[n]
      STORE       _L1         ; M[_L1] <- BGE M[n]
      CLEAR                   ; accum <- 0
      ADD         _orig_accum ; accum <- original accum
_L1:  CLEAR                   ; This is replaced by BGE M[n]
.end macro
```

Problem M1.1.B Subroutine Calling Conventions

We implement the following contract between the caller and the callee:

1. The caller places the argument in the address slot between the function-calling jump instruction and the return address. Just before jumping to the subroutine, the caller loads the return address into the accumulator.
2. In the beginning of a subroutine, the callee receives the return address in the accumulator. The argument can be accessed by reading the memory location preceding the return address. The code below shows pass-by-value as we create a local copy of the argument. Since the subroutine receives the address of the argument, it's easy to eliminate the dereferencing and deal only with the address in a pass-by-reference manner.
3. When the computation is done, the callee puts the return value in the accumulator and then jumps to the return address.

A call looks like

```
            ......                  ; preceding code sequence
            clear
            add         _THREE      ; accum <- 3
            bge         _here       ; skip over pointer
_hereptr    .fill       _here       ; hereptr = &here
_here       add         _hereptr    ; accum <- here+3 = return addr
            bge         _sub        ; jump to subroutine
                                    ; The following address location is
                                    ; reserved for argument passing and
                                    ; should never be executed as code:
_argument   .fill 6                 ; argument slot
            ......                  ; rest of program
```

(note that without an explicit program counter, a little work is required to establish the return address).

The subroutine begins:

```
_sub        store       _return     ; save the return address
            sub         _ONE        ; accum <- &argument = return address-1
            store       _arg        ; M[_arg] <- &argument = return address-1
```

```
            clear
            ADDind      _arg        ; accum <- *(&arg0)
            store       _arg        ; M[_arg] <- arg
```

And ends (with the return value in the accumulator):

```
            BGEind      _return
```

The subroutine uses some local storage:

```
_arg        clear                   ; local copy of argument
_return     clear                   ; reserved for return address
```

We need the following global constants:

```
_ONE        or          1           ; recall that OR's opcode is 00000
_THREE      or          3           ; so positive constants are easy to form
```

The following program uses this convention to compute fib(n) as specified in the problem set. It uses the indirection macros, templates, and storage from part M1.1.A.

```
;; The Caller Code Section
;;    ......                        ; preceding code sequence
_caller     clear
            add         _THREE      ; accum <- 3
            bge         _here
_hereptr    .fill       _here
_here       add         _hereptr    ; accum <- here+3 = return addr
            bge         _fib        ; jump to subroutine

;; The following address location is reserved for
;; argument passing and should never be executed as code
arg0        .fill       4           ; arg 0 slot.  N=4 in this example

_rtpnt      end

;; The fib Subroutine Code Section

; function call prelude
_fib        store       _return     ; save the return address
            sub         _ONE
            store       _n          ; M[_n] <- &arg0 = return address-1
            clear
            ADDind      _n          ; accum <- *(&arg0)
            store       _n          ; M[_n] <- arg0

; fib body
            clear
            store       _x          ; x=0
            add         _ONE
            store       _y          ; y=1

            clear                   ; if(n<2)
            add         _n
            sub         _TWO
            blt         _retn

            clear
            store       _i          ; for (i = 0;
```

```
_forloop    clear                   ; i < n-1;
            add         _n
            sub         _ONE
            sub         _i
            sub         _ONE
            blt         _done
_compute    clear
            add         _x
            add         _y
            store       _z          ; z = x+y
            clear
            add         _y
            store       _x          ; x = y
            clear
            add         _z
            store       _y          ; y = z

_next       clear                   ; i++)
            add         _i
            add         _ONE
            store       _i
            bge         _forloop

_retn       clear
            add   _n
            BGEind      _return     ; return n

_done       clear
            add         _z
            BGEind      _return     ; return z

;; Global constants (remember that OR's opcode is 00000)

_ONE        or 1
_TWO        or 2
_THREE      or 3
_FOUR       or 4

These memory locations are private to the subroutine

_return     clear       ; return address
_n          clear       ; n
_x          clear
_y          clear
_z          clear
_i          clear       ; index
_result     clear       ; fib
```

Now we can see how powerful this indirection addressing mode is! It makes programming much simpler.

The 1 argument-1 result convention could be extended to variable number of arguments and results by

1. Leaving as many argument slots in the caller code between the subroutine call instruction and the return address. This works as long as both the caller and callee agree on how many arguments are being passed.
2. Multiple results can be returned as a pointer to a vector (or a list) of the results. This implies an indirection, and so, yet another chance for self-modifying code.

Problem M1.1.C — Subroutine Calling Other Subroutines

The subroutine calling convention implemented in Problem M1.1.B stores the return address in a fixed memory location `(_return)`. When `fib_recursive` is first called, the return address is stored there. However, this original return address will be overwritten when `fib_recursive` makes its first recursive call. Therefore, your program can never return to the original caller!

Problem M1.2: CISC, RISC, and Stack: Comparing ISAs

Problem M1.2.A CISC

How many bytes is the program? 19

How many bytes of instructions need to be fetched if b = 10?

(2+2) + 10*(13) + (6+2+2) = 144

Assuming 32-bit data values, how many bytes of data memory need to be fetched? Stored?

Fetched: the compare instruction accesses memory, and brings in a 4 byte word b+1 times: 4 * 11 = 44
Stored: 0

Problem M1.2.B RISC

Many translations will be appropriate, here's one. We ignore MIPS32's branch-delay slot in this solution since it hadn't been discussed in lecture. Remember that you need to construct a 32-bit address from 16-bit immediate values.

x86 instruction	label	MIPS32 instruction sequence
`xor %edx,%edx`		*xor r4, r4, r4*
`xor %ecx,%ecx`		*xor r3, r3, r3*
`cmp 0x8047580,%ecx`	*loop*	*lui r6, 0x0804* *lw r1, 0x7580 (r6)* *slt r5, r3, r1*
`jl L1`		*bnez r5, L1*
`jmp done`		*j done*
`add %eax,%edx`	*L1*	*add r4, r4, r2*
`inc %ecx`		*addi r3, r3, #1*
`jmp loop`		*j loop*
...	*done:*	...

How many bytes is the MIPS32 program using your direct translation?

10*4 = 40

How many bytes of MIPS32 instructions need to be fetched for b = 10 using your direct translation.

There are 2 instructions in the prelude and 7 that are part of the loop (we don't need to fetch the 'j done' until the 11th iteration). There are 5 instructions in the 11th iteration. All instructions are 4 bytes. 4(2+10*7+5) = 308.

Note: You can also place the label '`loop`' in two other locations assuming `r6` and `r1` hold the same values for the remaining of the program after being loaded. One location is in front of the `lw` instruction, and we reduce the number of fetched byte to 268. The other is in front of the `slt` instruction, and we further decrease the number of fetched bytes to 228.

How many bytes of data memory need to be fetched? Stored?

Fetched: 11 * 4 = 44 (or 4 if you place the label '`loop`' in front of the `slt` instruction)
Stored: 0

Problem M1.2.C Optimization

There are two ideas that we have for optimization.

1) We count down to zero instead of up for the number of iterations. By doing this, we can eliminate the slt instruction prior to the branch instruction.

2) Hold b value in a register if you haven't done it already.

```
            xor r4, r4, r4
            lui r6, 0x0804
            lw r1, 0x9580(r6)
            jmp dec
loop:       add r4, r4, r2
dec:        addiu r1, r1, #-1
            bgez r1, loop
done:
```

This modification brings the dynamic code size down to 144 bytes, the static code size down to 28 and memory traffic down to 4 bytes.

Problem M1.3: Addressing Modes on MIPS ISA

Problem M1.3.A — Displacement addressing mode

The answer is yes.

```
LW R1, 16(R2)      ➔      ADDI R3, R2, #16
                          LW R1, 0(R3)
```

(R3 is a temporary register.)

Problem M1.3.B — Register indirect addressing

The answer is yes once again.

```
LW R1, 16(R2)      ➔

lw_template:   LW   R1, 0       ; it is placed in data region
               ...
  LW_start:    LW   R3, lw_template
               ADDI R4, R2, #16
               ADD  R3, R3, R4  ; R3 <- "LW R1, addr"
               SW   R3, _L1     ; write the LW instruction
       _L1:    NOP              ; to be replaced by "LW .."
```

(R3 and R4 are temporary registers.)

Problem M1.3.C Subroutine

Yes, you can rewrite the code as follows.

```
Subroutine: lw   R6, ret_inst ; r6 = "j 0"
            add  R6, R6, R31  ; R6 = "j return_addr"
            sw   R6, return   ; replacing nop with "j return_addr"

            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:     nop               ; will be replaced by "j return_addr"
L1:         add  R4, R4, R2   ; result += a
            addi R3, R3, #1   ; i++
            j    loop
ret_inst:   j    0            ; jump instruction template
```

Problem M1.4: Fully-Bypassed Simple 5-Stage Pipeline

Problem M1.4.A — Stall

We still need the logic for stalls, because we cannot prevent load-use hazard. If a load instruction is followed by an instruction which takes the loaded value as a source operand, we cannot avoid stalling for a cycle. The following instruction sequence illustrates this hazard.

```
LW  R1, 0(R2)     # R1 <- M[R2]
ADD R3, R5, R1    # R1 is a source operand of ADD (data dependency)
                  # The correct value of R1 is not available when
                  # ADD is in ID stage.  So it has to stall for a cycle.
```

Problem M1.4.B — Bypass Signal

Here are the bypass conditions.

Bypass $_{EX\text{->}ID}$ ASrc = $(rs_D=ws_E).we\text{-}bypass_E.re1_D$

Bypass $_{MEM\text{->}ID}$ = $(rs_D=ws_M).we_M.re1_D$

Bypass $_{WB\text{->}ID}$ = $(rs_D=ws_W).we_W.re1_D$

Priority: Bypass $_{EX\text{->}ID}$ > Bypass $_{MEM\text{->}ID}$ > Bypass $_{WB\text{->}ID}$
(In order to execute a given program correctly, the value from the latest producer must be taken if multiple bypass paths are active.)

Problem M1.4.C — Partial Bypassing

It is an open question and there is no single correct answer. Here are a couple of issues to consider as a guideline.

First, you may consider the penalty for not having all the bypass paths. If we don't have the bypass path EX→ID, we have to stall for three cycles for the hazard to be resolved. Likewise, not having MEM→ID results in a stall of two cycles, and not having WB→ID, in one. Therefore, you can conclude that the bypass path between EX→ID is the most beneficial.

Secondly, the best bypass path depends on the access patterns of data. The EX→ID bypass path is effective if a producer instruction is followed by a consumer, except load-use cases (See solution for M1.4.A). On the other hand, the MEM→ID bypass path works best if there are many load-use cases or many (producer, consumer) pairs have an independent instruction between them. Likewise, the WB→ID bypass path helps when many (producer, consumer) pairs are separated by exactly two independent instructions.

Problem M1.5: Basic Pipelining

Problem M1.5.A **Mux Control Signals (1)**

PCEn = (S==Execute)

IREn = (S==I-Fetch)

AddrSrc = Case S

I-Fetch => PC

Execute => ALU

Problem M1.5.B **Modified pipeline**

A stall can occur in 2 different cases.

1. A structural hazard in the shared memory.
   LD R1, 16(R2)
   Any instruction following this LD instruction should be stalled.

2. The other is caused by a control hazard, because we don't have a delay slot.
   J 200
   Any instruction following this J instruction should be flushed.

Problem M1.5.C **Mux Control Signals (2)**

PCEnable = not ((opcode == LW) or (opcode == SW))

AddrSrc = Case opcode

not (LW or SW) => PC

(LW or SW) => ALU

IRSrc = Case <u>opcode</u>

LW or SW or Jump or Br_{taken} => nop

Else => Mem

Problem M1.5.D

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	**I_3:108**	**I_2**	**0**	*****	**ALU**	**Nop**
t_3	**I_3:108**	**-**	**1**	**pc+4**	**PC**	**Mem**
t_4	**I_4:112**	**I_3**	**1**	**jabs**	**PC**	**Nop**
t_5	**I_7:312**	**-**	**1**	**pc+4**	**PC**	**Mem**
t_6	**I_8:316**	**I_7**	**1**	**pc+4**	**PC**	**Mem**

Problem M1.5.E Self-Modifying Code

The answer is no. The hazard is resolved by the datapath itself because (1) memory accesses are serialized by the stall logic at the shared memory and (2) memory write takes only one cycle.

Problem M1.5.F

Due to this rerouting we will now have to stall even if it is an ALU instruction.

Problem M1.5.G Architecture Comparison

The Princeton architecture is cheaper than the Harvard architecture, but the Harvard architecture is faster than the Princeton architecture.

Problem M1.6: A 5-Stage Pipeline with an Additional Adder

Problem M1.6.A — Elimination of a hazard

The new datapath is trying to eliminate the hazard that occurs when a load instruction is immediately followed by an ALU instruction that requires the value that was loaded. In the original datapath, a pipeline interlock (stall) is needed for this type of an instruction sequence, an example of which is shown below. In Ben's datapath, this load-use interlock is not required because the data from the load instruction can be immediately forwarded to the ALU.

```
LW R1, 0(R3)
ADDI R1, R1, #5
```

Problem M1.6.B — New Hazard

The new hazard occurs when the result of an ALU operation is needed to calculate the address of a load or store instruction.

```
ADDI R1, R1, #5
LW R3, 3(R1)
```

Problem M1.6.C — Comparison

Now an address-generation interlock is needed for the LW instruction in the sequence in M1.6.B. Note that this new hazard affects both load and store instructions, while the original hazard only affected load instructions. This is a disadvantage of the modified pipeline. Also, the new datapath requires more hardware (another adder) than the original datapath. However, the load-use hazard illustrated in Problem M1.6.A has been eliminated. If we examine the behavior of typical programs, we will see that the percentage of load instructions resulting in the load-use interlock from Problem M1.6.A is higher than the percentage of all loads and stores resulting in the address-generation interlock from Problem M1.6.B. This is because many address calculations are based on values that change infrequently (e.g. the stack pointer does not change while a procedure is being executed). If a base address register has not been recently changed, then there will be no address-generation interlock. By contrast, when a load is issued, the load value is usually required within a few cycles, so a load-use interlock is much more likely. Whether performance is better on the original pipeline or on the modified pipeline will depend on the specific program.

Problem M1.6.D — Stall Logic

The stall equation for only the new hazard is given below. The **op** signal is used to determine the instruction opcode.

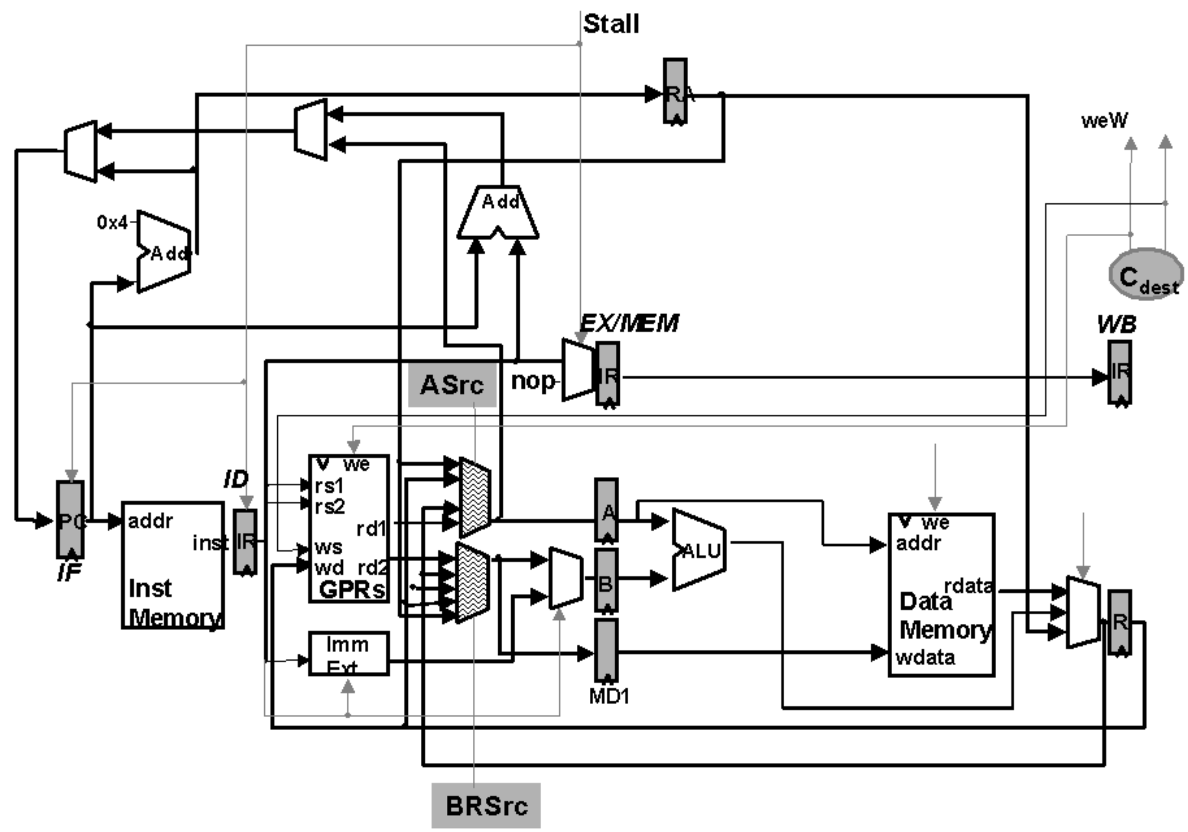
Stall $= ((op_{ID} = LW) + (op_{ID} = SW)).(rs_{ID} = ws_{AC}).((op_{AC} = ALU) + (op_{AC} = ALUi)).(ws_{AC} \neq 0)$

Problem M1.6.E — Datapath Improvement

If we eliminated the displacement addressing mode from the MIPS ISA and only supported register indirect addressing, then we would no longer need to compute an effective address for loads and stores. We could improve the datapath by eliminating the AC (effective address calculation) stage from Ben's modified pipeline, resulting in the following stages

IF	ID	EX/MEM	WB
Instruction fetch	Instruction decode and register fetch	Execution of ALU operations or memory access	Write-back to register file

A diagram showing the new pipeline is given below.



This new datapath does not have either of the hazards from Ben's original or modified pipelines. Thus, bubbles would not need to be inserted into the pipeline regardless of the instruction sequence, improving instruction throughput. As a side note, the latency of a single instruction has also been reduced since there are now only 4 stages instead of 5. Although this does not improve performance in the steady state, a fewer number of stages does help because fewer pipeline registers and bypass paths are required. However, this instruction set is limited in that it only supports register indirect addressing. This means that displacement addressing would have to be synthesized from simpler instructions (see Problem M1.6.F).

Problem M1.6.F Displacement Addressing Synthesizing

Programmers could synthesize a displacement load/store instruction using the ADDi instruction, a scratch register, and the register indirect load/store instruction. For example, to synthesize the following instruction with displacement addressing

```
LW R1, 4(R2)
```

we could use the following equivalent instruction sequence, where R3 is a temporary register

```
ADDI R3, R2, #4
LW R1, (R3)
```

The same programs could be written as before using this technique. However, using this limited ISA may increase the number of instructions in the program as compared to the original ISA.

Problem M1.6.G Jumps and Branches

If Ben uses the ALU to resolve conditional branches in both his original pipeline and his modified pipeline shown in Problem M1.6.A, then there will be an additional cycle of branch delay in the new datapath because the ALU is now one stage later in the pipeline. If we don't worry about duplicating logic, then we can put a comparator in any stage of the pipeline (except Instruction Fetch, as the register file has not yet been read in this stage) in order to resolve conditional branches. The table shown below compares each possible placement of the comparator.

Comparator In Stage	Number of Branch Delay Cycles	Additional Stall Condition	Change in Clock Period
WB	4	None	Will remain unchanged since comparator is simpler than ALU operation so it cannot be the critical path.
EX/MEM	3	None	Will remain unchanged since comparator is simpler than ALU operation so it cannot be the critical path.
AC	2	1 cycle stall when the ALU output or result of a load is used for the branch	Will remain unchanged since comparator is simpler than ALU operation so it cannot be the critical path.
ID	1	2 cycle stall when the ALU output or result of a load is used for the branch	Will likely **increase** the clock period since it now could be on the critical path (fetch register value + compare)

Obviously placing the comparator in the Write-Back stage makes no sense since this doesn't provide an advantage over placing the comparator in the Execute/Memory stage, and in fact, it increases the number of branch delay cycles by 1. Placing the comparator in the Address Calculation stage instead of the Execute/Memory stage reduces the number of branch delay cycles by 1, but introduces a potential stall condition. Since the branch delay affects all branches, while the stall condition would only affect some of the branches, placing the comparator in the Address Calculation stage is to be preferred over the Execute/Memory stage. Finally, the comparator could be placed in the Instruction Decode stage. If this doesn't lengthen the critical path, then this would be the best placement, as the number of branch delay cycles is reduced to 1. However, if it does lengthen the critical path—and it likely will—then the increased cycle time would probably not be worth the reduction in the branch delay, as now *all* instructions will run more slowly.

Problem M1.7: Dual ALU Pipeline

Problem M1.7.A — ALU Usage

Instruction	ALU1 or ALU2?
`add r1, r2, r3`	**ALU1**
`lw r4, 0(r1)`	
`add r5, r4, r6`	**ALU2**
`add r7, r5, r8`	**ALU2**
`add r1, r2, r3`	**ALU1**
`lw r4, 0(r1)`	
`add r5, r1, r6`	**ALU1**

The following timeline shows the execution of the instructions, with the stage where each instruction produces its result highlighted in bold, and the bypassing between instructions shown by arrows.

add_1	IF	ID	**EX1**	EX2	WB						
lw_1		IF	ID	EX1	**MEM**	WB					
add_2			IF	ID	EX1	**EX2**	WB				
add_3				IF	ID	EX1	**EX2**	WB			
add_4					IF	ID	**EX1**	EX2	WB		
lw_2						IF	ID	EX1	**MEM**	WB	
add_5							IF	ID	**EX1**	EX2	WB

The pipeline is initially idle, so the first add reads its operands from the register file in ID and uses ALU1. The second add uses the result of the lw which is not available by the end of ID; therefore the add uses ALU2, and the load data is bypassed to it at the end of EX1. The third add uses the result of the second, so its data is not available by the end of ID; it also uses ALU2, allowing the data to be bypassed to it at the end of EX1. The fourth add has no dependencies on the previous instructions; it reads its operands from the register file in ID and uses ALU1. The fifth add uses the result of the fourth add. This value is bypassed to it at the end of ID from EX2/MEM, and it uses ALU1.

Problem M1.7.B **Control Signal**

$$\begin{aligned}\mathbf{alu2_{ID}} = (& ((\mathbf{OP_{ID}} = \mathbf{ALU}) + (\mathbf{OP_{ID}} = \mathbf{ALUi})) \\ & \cdot((\mathbf{rs_{ID}} = \mathbf{ws_{EX1}}) + (\mathbf{rt_{ID}} = \mathbf{ws_{EX1}})\cdot \mathbf{re2_{ID}}) \\ & \cdot(\mathbf{ws_{EX1}} \neq 0) \\ & \cdot(\ \underline{(\mathbf{OP_{EX1}} = \mathbf{LW}) + \mathbf{alu2_{EX1}}}\ ) \\ )\end{aligned}$$

An ALU instruction uses ALU2 if its operands are not available by the end of ID. This occurs if the ALU instruction (in ID) uses the result of its immediately preceding instruction (in EX1) as a source, but the instruction will not produce its result until EX2/MEM. The two classes of instructions which do not produce a result until EX2/MEM are LW instructions and ALU instructions which use ALU2.

Note that the feedback dependence of $\mathtt{alu2_{ID}}$ on $\mathtt{alu2_{EX1}}$ means that a sequence of ALU instructions following a LW will continue to use ALU2 as long as each instruction uses the result of its predecessor.

Problem M1.7.C **Instruction Sequences Causing Stalls**

	Stall?	Explanation
`add` **`r1`**`, r2, r3` `lw  r4, 0(`**`r1`**`)`	**No**	**The add (in EX1) uses ALU1 and bypasses its result to the LW (in ID).**

`lw  `**`r1`**`, 0(r2)`
`add r3, `**`r1`**`, r4`
`lw  r5, 0(`**`r1`**`)`
	No	**The first LW (in EX2/MEM) bypasses its result to the add (in EX1) which will use ALU2, and also to the second LW (in ID).**

`lw  `**`r1`**`, 0(r2)` `lw  r3, 0(`**`r1`**`)`	**Yes**	**The result of the first LW (in EX1) is not available in time for the second LW (in ID), so the second LW must stall.**
`lw  `**`r1`**`, 0(r2)` `sw  `**`r1`**`, 0(r3)`	**No**	**The LW (in EX2/MEM) bypasses its result to the SW (in EX1) in time for it to store the data in EX2/MEM.**

`lw  `**`r1`**`, 0(r2)`
`add `**`r3`**`, `**`r1`**`, r4`
`sw  r5, 0(`**`r3`**`)`
	Yes	**The LW (in EX2/MEM) bypasses its result to the add (in EX1) which will use ALU2. But, the result of the add (in EX1) is not available in time for the SW (in ID), so the SW must stall.**

`lw  `**`r1`**`, 0(r2)` `add r3, `**`r1`**`, r4`	**No**	**The LW (in EX2/MEM) bypasses its result to the add (in EX1) which will use ALU2.**

Note that the base address operand for both LW and SW must be available by the end of ID, but the data operand for SW must only be available by the end of EX1.

Problem M1.7.D **Stall Equation**

$$
\begin{aligned}
stall_{ID} = (\ & ((OP_{ID} = LW) + (OP_{ID} = SW)) \\
& \cdot (rs_{ID} = ws_{EX1}) \\
& \cdot (ws_{EX1} \neq 0) \\
& \cdot ((OP_{EX1} = LW) + alu2_{EX1}) \\
&)
\end{aligned}
$$

Since all instruction results are produced by the end of EX2/MEM, the operands for an instruction are always available by the end of EX1 even if it uses the result of its immediately preceding instruction as a source.

The only stall condition is when the base address operand for a memory instruction is not available by the end of ID. This occurs if the memory instruction (in ID) uses the result of its immediately preceding instruction (in EX1) as its base address, but the instruction will not produce its result until EX2/MEM. The two classes of instructions which do not produce a result until EX2/MEM are LW instructions and ALU instructions which use ALU2.

Note that ALU instructions never need to stall the pipeline. They either use ALU1 if their operands will be available by the end of ID, or ALU2 if their operands will be available by the end of EX1.

Problem M1.8: Processor Design (Short Yes/No Questions)

Problem M1.8.A — Interlock vs. Bypassing

No. Data dependencies are preserved with either interlocks or bypassing, so the processors always generate the same results. Bypassing improves performance by eliminating stalls.

Problem M1.8.B — Delay Slot

Yes. The instruction following a taken branch is executed on processor A, but killed on processor B so the processors can generate different results.

Problem M1.8.C — Structural Hazard

No. Both processors retrieve the same data values. There is only a performance difference because processor A must stall an instruction fetch to allow a load instruction to access memory.