6.823 Computer System Architecture
Module #1 Solutions

Last Updated: 2/24/2016

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.)

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.

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

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

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

Priority: $\text{Bypass}_{\text{EX->ID}} > \text{Bypass}_{\text{MEM->ID}} > \text{Bypass}_{\text{WB->ID}}$
(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 opcode

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: Processor Design (Short Yes/No Questions)

Problem M1.6.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 M1.6.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 M1.6.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.