6.823 Computer System Architecture
Module #1 Problem Set #1 Solutions

Last Updated: 3/2/2017

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 M2.1: Cache Access-Time & Performance

Here is the completed Table M2.1-1 for M2.1.A and M2.1.B.

Component	Delay equation (ps)		DM (ps)	SA (ps)
Decoder	200×(# of index bits) + 1000	Tag	3400	3000
		Data	3400	3000
Memory array	$200 \times \log_2$ (# of rows) +	Tag	4217	4250
	$200 \times \log_2$ (# of bits in a row) + 1000	Data	5000	5000
Comparator	200×(# of tag bits) + 1000	▨	4000	4400
N-to-1 MUX	$500 \times \log_2 N + 1000$	▨	2500	2500
Buffer driver	2000	▨	▨	2000
Data output driver	500×(associativity) + 1000	▨	1500	3000
Valid output driver	1000	▨	1000	1000

Table M2.1-1: Delay of each Cache Component

Problem M2.1.A **Access time: DM**

To use the delay equations, we need to know how many bits are in the tag and how many are in the index. We are given that the cache is addressed by word, and that input addresses are 32-bit byte addresses; the two low bits of the address are not used.

Since there are 8 (2^3) words in the cache line, 3 bits are needed to select the correct word from the cache line.

In a 128 KB direct-mapped cache with 8 word (32 byte) cache lines, there are $4 \times 2^{10} = 2^{12}$ cache lines (128KB/32B). 12 bits are needed to address 2^{12} cache lines, so the number of index bits is 12. The remaining 15 bits (32 – 2 – 3 – 12) are the tag bits.

We also need the number of rows and the number of bits in a row in the tag and data memories. The number of rows is simply the number of cache lines (2^{12}), which is the same for both the tag and the data memory. The number of bits in a row for the tag memory is the sum of the number of tag bits (15) and the number of status bits (2), 17 bits total. The number of bits in a row for the data memory is the number of bits in a cache line, which is 256 (32 bytes × 8 bits/byte).

With 8 words in the cache line, we need an 8-to-1 MUX. Since there is only one data output driver, its associativity is 1.

Decoder (Tag) = 200 × (# of index bits) + 1000 $= 200 \times 12 + 1000$ = 3400 ps
Decoder (Data) = 200 × (# of index bits) + 1000 $= 200 \times 12 + 1000$ = 3400 ps

Memory array (Tag) $= 200 \times \log_2$(# of rows) + $200 \times \log_2$(# bits in a row) + 1000

= $200 \times \log_2(2^{12}) + 200 \times \log_2(17) + 1000$ ≈ 4217 ps

Memory array (Data) = $200 \times \log_2(\text{\# of rows}) + 200 \times \log_2(\text{\# bits in a row}) + 1000$
= $200 \times \log_2(2^{12}) + 200 \times \log_2(256) + 1000$ = 5000 ps

Comparator = $200 \times (\text{\# of tag bits}) + 1000$ = $200 \times 15 + 1000$= 4000 ps

N-to-1 MUX = $500 \times \log_2(N) + 1000$ = $500 \times \log_2(8) + 1000$ = 2500 ps

Data output driver = $500 \times (\text{associativity}) + 1000$ = $500 \times 1 + 1000$ = 1500 ps

To determine the critical path for a cache read, we need to compute the time it takes to go through each path in hardware, and find the maximum.

Time to tag output driver
= (tag decode time) + (tag memory access time) + (comparator time) + (AND gate time)
+ (valid output driver time)
≈ 3400 + 4217 + 4000 + 500 + 1000 = 13117 ps

Time to data output driver
= (data decode time) + (data memory access time) + (mux time) + (data output driver time)
= 3400 + 5000 + 2500 + 1500 = 12400 ps

The critical path is therefore the tag read going through the comparator. The access time is 13117 ps. At 150 MHz, it takes 0.013117×150, or 2 cycles, to do a cache access.

Problem M2.1.B Access time: SA

As in M2.1.A, the low two bits of the address are not used, and 3 bits are needed to select the appropriate word from a cache line. However, now we have a 128 KB 4-way set associative cache. Since each way is 32 KB and cache lines are 32 bytes, there are 2^{10} lines in a way (32KB/32B) that are addressed by 10 index bits. The number of tag bits is then (32 – 2 – 3 – 10), or 17.

The number of rows in the tag and data memory is 2^{10}, or the number of sets. The number of bits in a row for the tag memory is now quadruple the sum of the number of tag bits (17) and the number of status bits (2), 76 bits total. The number of bits in a row for the data memory is four times the number of bits in a cache line, which is 1024 (4×32 bytes $\times$ 8 bits/byte).

As in 1.A, we need an 8-to-1 MUX. However, since there are now four data output drivers, the associativity is 4.

Decoder (Tag) = $200 \times (\text{\# of index bits}) + 1000$ = $200 \times 10 + 1000$ = 3000 ps
Decoder (Data) = $200 \times (\text{\# of index bits}) + 1000$ = $200 \times 10 + 1000$ = 3000 ps

Memory array (Tag) = $200 \times \log_2(\text{\# of rows}) + 200 \times \log_2(\text{\# bits in a row}) + 1000$

	$= 200 \times \log_2(2^{10}) + 200 \times \log_2(76) + 1000$	≈ 4250 ps
Memory array (Data)	$= 200 \times \log_2(\text{\# of rows}) + 200 \times \log_2(\text{\# bits in a row}) + 1000$	
	$= 200 \times \log_2(2^{10}) + 200 \times \log_2(1024) + 1000$	$= 5000$ ps
Comparator	$= 200 \times (\text{\# of tag bits}) + 1000$	$= 200 \times 17 + 1000 = 4400$ ps
N-to-1 MUX	$= 500 \times \log_2(N) + 1000$	$= 500 \times \log_2(8) + 1000 = 2500$ ps
Data output driver	$= 500 \times (\text{associativity}) + 1000$	$= 500 \times 4 + 1000 = 3000$ ps

Time to valid output driver
= (tag decode time) + (tag memory access time) + (comparator time) + (AND gate time)
+ (OR gate time) + (valid output driver time)
= 3000 + 4250 + 4400 + 500 + 1000 + 1000 = 14150 ps

There are two paths to the data output drivers, one from the tag side, and one from the data side. Either may determine the critical path to the data output drivers.

Time to get through data output driver through tag side
= (tag decode time) + (tag memory access time) + (comparator time) + (AND gate time)
+ (buffer driver time) + (data output driver)
= 3000 + 4250 + 4400 + 500 + 2000 + 3000 = 17150 ps

Time to get through data output driver through data side
= (data decode time) + (data memory access time) + (mux time) + (data output driver)
= 3000 + 5000 + 2500 + 3000 = 13500 ps

From the above calculations, it's clear that the critical path leading to the data output driver goes through the tag side.

The critical path for a read therefore goes through the tag side comparators, then through the buffer and data output drivers. The access time is 17150 ps. The main reason that the 4-way set associative cache is slower than the direct-mapped cache is that the data output drivers need the results of the tag comparison to determine which, if either, of the data output drivers should be putting a value on the bus. At 150 MHz, it takes 0.0175×150, or 3 cycles, to do a cache access.

It is important to note that the structure of cache we've presented here does not describe all the details necessary to operate the cache correctly. There are additional bits necessary in the cache which keeps track of the order in which lines in a set have been accessed. We've omitted this detail for sake of clarity.

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`			1A						no
`102`	10								no
`361`							36		no
`204`	20								no
`114`									yes
`1A4`									yes
`177`								17	no
`301`	30								no
`206`	20								no
`135`									yes

	D-map
Total Misses	10
Total Accesses	13

4-way									LRU
	line in cache								Hit?
Address	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`		1A							No
`102`			10						No
`361`				36					No
`204`									Yes
`114`									Yes
`1A4`									Yes
`177`							17		No
`301`			30						No
`206`									Yes
`135`									Yes

	4-way LRU
Total Misses	8
Total Accesses	13

4-way **Address**									FIFO
	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		1A							No
102			10						No
361				36					No
204									Yes
114									Yes
1A4									Yes
177							17		No
301	30								No
206		20							No
135									Yes

	4-way FIFO
Total Misses	9
Total Accesses	13

Problem M2.1.D — Average latency

The miss rate for the direct-mapped cache is 10/13. The miss rate for the 4-way LRU set associative cache is 8/13.

The average memory access latency is (hit time) + (miss rate) × (miss time).

For the direct-mapped cache, the average memory access latency would be (2 cycles) + (10/13) × (20 cycles) = 17.38 ≈ 18 cycles.

For the LRU set associative cache, the average memory access latency would be (3 cycles) + (8/13) × (20 cycles) = 15.31 ≈ 16 cycles.

The set associative cache is better in terms of average memory access latency.

For the above example, LRU has a slightly smaller miss rate than FIFO. This is because the FIFO policy replaced the {20} block instead of the {10} block during the 12th access, because the {20} block has been in the cache longer even though the {10} was least recently used, whereas the LRU policy took advantage of temporal/spatial locality.

LRU doesn't always have lower miss rate than FIFO. Consider the following counter example: A sequence accesses 3 separate memory locations A,B and C in the order of A, B, A, C, B, B, B, …. When this sequence is executed on a processor employing a fully-associative cache with 2 cache lines and LRU replacement policy, the execution ends up with 4 misses. On the other hand, the same sequence will only produces 3 misses if the cache uses FIFO replacement policy. (We assume the cache is empty at the beginning of the execution).

Problem M2.2: Victim Cache Evaluation

Problem M2.2.A — Baseline Cache Design

Component	Delay equation (ps)		FA (ps)
Comparator	200×(# of tag bits) + 1000		6800
N-to-1 MUX	$500 \times \log_2 N + 1000$		1500
Buffer driver	2000		2000
AND gate	1000		1000
OR gate	500		500
Data output driver	500×(associativity) + 1000		3000
Valid output driver	1000		1000

Table M2.2-1

The **Input Address** has 32 bits. The bottom two bits are discarded (cache is word-addressable) and bit 2 is used to select a word in the cache line. Thus the **Tag** has 29 bits. The **Tag+Status** line in the cache is 31 bits.

The **MUX**es are 2-to-1, thus N is 2. The associativity of the **Data Output Driver** is 4 – there are four drivers driving each line on the common **Data Bus**.

Delay to the **Valid Bit** is equal to the delay through the **Comparator**, **AND** gate, **OR** gate, and **Valid Output Driver**. Thus it is 6800 + 1000 + 500 + 1000 = 9300 ps.

Delay to the **Data Bus** is delay through MAX ((**Comparator**, **AND** gate, **Buffer Driver**), (**MUX**)), **Data Output Drivers**. Thus it is MAX (6800 + 1000 + 2000, 1500) + 3000 = MAX (9800, 1500) + 3000 = 9800 + 3000 = 12800 ps.

Critical Path Cache Delay: 12800 ps

Problem M2.2.B **Victim Cache Behavior**

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			A						N			N
10		1							N			N
C0					C				N			N
18									Y			N
20			2						N		A	N
8C	8								N	0		Y
28									Y			N
AC			A						N		2	Y
38				3					N			N
C4									Y			N
3C									Y			N
48					4				N	C		N
0C	0								N		8	N
24			2						N	A		N

Table M2.2-2

15% of accesses will take 50 cycles less to complete, so the average memory access improvement is 0.15 * 50 = 7.5 cycles.

Problem M2.3: Loop Ordering

Problem M2.3.A

Each element of the matrix can only be mapped to a particular cache location because the cache here is a Direct-mapped data cache. *Matrix A* has 64 columns and 128 rows. Since each row of matrix has 64 32-bit integers and each cache line can hold 8 words, each row of the matrix fits exactly into eight (64÷8) cache lines as the following:

0	A[0][0]	A[0][1]	A[0][2]	A[0][3]	A[0][4]	A[0][5]	A[0][6]	A[0][7]
1	A[0][8]	A[0][9]	A[0][10]	A[0][11]	A[0][12]	A[0][13]	A[0][14]	A[0][15]
2	A[0][16]	A[0][17]	A[0][18]	A[0][19]	A[0][20]	A[0][21]	A[0][22]	A[0][23]
3	A[0][24]	A[0][25]	A[0][26]	A[0][27]	A[0][28]	A[0][29]	A[0][30]	A[0][31]
4	A[0][32]	A[0][33]	A[0][34]	A[0][35]	A[0][36]	A[0][37]	A[0][38]	A[0][39]
5	A[0][40]	A[0][41]	A[0][42]	A[0][43]	A[0][44]	A[0][45]	A[0][46]	A[0][47]
6	A[0][48]	A[0][49]	A[0][50]	A[0][51]	A[0][52]	A[0][53]	A[0][54]	A[0][55]
7	A[0][56]	A[0][57]	A[0][58]	A[0][59]	A[0][60]	A[0][61]	A[0][62]	A[0][63]
8	A[1][0]	A[1][1]	A[1][2]	A[1][3]	A[1][4]	A[1][5]	A[1][6]	A[1][7]
• • •	• • •	• • •	• • •	• • •	• • •	• • •	• • •	• • •

Loop A accesses memory sequentially (each iteration of *Loop A* sums a row in *matrix* `A`), an access to a word that maps to the first word in a cache line will miss but the next seven accesses will hit. Therefore, *Loop A* will only have compulsory misses (128×64÷8 or 1024 misses).

The consecutive accesses in *Loop B* will use every eighth cache line (each iteration of *Loop B* sums a column in *matrix* `A`). Fitting one column of matrix A, we would need 128×8 or 1024 cache lines. However, our 4KB data cache with 32B cache line only has 128 cache lines. When Loop B accesses a column, all the data that the previous iteration might have brought in would have already been evicted. Thus, every access will cause a cache miss (64×128 or 8192 misses).

The number of cache misses for Loop A: 1024

The number of cache misses for Loop B: 8192

Problem M2.3.B

Since *Loop A* accesses memory sequentially, we can overwrite the cache lines that were previous brought in. *Loop A* will only require 1 cache line to run without any cache misses other than compulsory misses.

For *Loop B* to run without any cache misses other than compulsory misses, the data cache needs to have the capacity to hold one column of matrix A. Since the consecutive accesses in *Loop B* will use every eighth cache line and we have 128 elements in a *matrix A* column, *Loop B* requires 128×8 or 1024 cache lines.

Data-cache size required for Loop A: ______1______ cache line(s)

Data-cache size required for Loop B: ______1024______ cache line(s)

Problem M2.3.C

Loop A still only has compulsory misses (128×64÷8 or 1024 misses).

Because of the fully-associative data cache, *Loop B* now can fully utilize the cache and the consecutive accesses in *Loop B* will no longer use every eighth cache line. Fitting one column of *matrix A*, we now would only need 128 cache lines. Since 4KB data cache with 8-word cache lines has 128 cache lines, *Loop B* only has compulsory misses (128×(64÷8) or 1024 misses).

The number of cache misses for Loop A:______1024______

The number of cache misses for Loop B:______1024______

Problem M2.4: Cache Parameters

Problem M2.4.A

TRUE. Since cache size is unchanged, the line size doubles, the number of tag entries is halved.

Problem M2.4.B

FALSE. The total number of lines across all sets is still the same, therefore the number of tags in the cache remain the same.

Problem M2.4.C

TRUE. Doubling the capacity increases the number of lines from N to 2N. Address i and address i+N now map to different entries in the cache and hence, conflicts are reduced.

Problem M2.4.D

FALSE. The number of lines doubles but the line size remains the same. So the compulsory "cold-start" misses stays the same.

Problem M2.4.E

TRUE. Doubling the line size causes more data to be pulled into the cache on a miss. This exploits spatial locality as subsequent loads to different words in the same cache line will hit in the cache reducing compulsory misses.

Problem M2.5: Microtags

Problem M2.5.A

A direct-mapped cache can forward data to the CPU before checking the tags for a hit or a miss. A set-associative cache has to first compare cache tags to select the correct way from which to forward data to the CPU.

Problem M2.5.B

tag	Index	offset

# of bits in the tag: ____21________

# of bits in the index: _____6________

# of bits in the offset: _____5________

32-byte line requires 5 bits to select the correct byte.
An 8KB, 4-way cache has 2KB in each way, and each way holds 2KB/32B=64 lines, so we need 6 index bits.
The remaining 32-6-5=21 bits are the tag.

Problem M2.5.C

If the loTags are not unique, then multiple ways can attempt to drive data on the tristate bus out to the CPU causing bus contention.

(It is possible to have a scheme that speculatively picks one of the ways when there is as match in loTags, but this would require additional cross-way logic that would slow the design down, and would also incur extra misses when the speculation was wrong.)

Problem M2.5.D

The loTag has to be unique across ways, and so in a 4-way cache with 2-bit tags the tags would never be able to hold addresses that were different from a direct-mapped cache of the same capacity. The conflict misses would therefore be identical.

Problem M2.5.E

When a new line is brought into the cache, any existing line in the set with the same loTag must be chosen as the victim. If there is no line with the same loTag, any conventional replacement policy can be used.

Problem M2.5.F

No. The full tag check is required to determine whether the write is a hit to the cached line.

Problem M2.5.G

A 16KB page implies 14 untranslated address bits. An 8KB, 4-way cache requires 11 index+offset bits, leaving 3 untranslated bits for loTag.

Problem M2.5.H

If the loTags include translated virtual address bits, then each cache line must store the physical page number (PPN) as the hiTag. An access will hit if loTag matches, and the PPN in hiTag matches. The replacement policy has to maintain two invariants: 1) no two lines in a set have the same loTag bits and 2) no two lines have the same PPN. If two lines had the same PPN, there might be a virtual address alias. Because a new line might have the same loTag as an existing line, and also the same PPN as a different line, two lines might have to be evicted to bring in one new line.
A slight improvement is to only evict a line with the same PPN if the untranslated part of loTag is identical. If the untranslated bits are different, the two lines cannot be aliases.

Problem M3.1: Virtual Memory Bits

Problem M3.1.A

The answer depends on certain assumptions in the OS. Here we assume that the OS does everything that is reasonable to keep the TLB and page table coherent. Thus, any change that OS software makes is made to both the TLB and the page table.

However, the hardware can change the U bit (whenever a hit occurs this bit will be set) and the M bit (whenever a page is modified this bit will be set). Thus, these are the only bits that need to be written back. Note that the system will function correctly even if the U bit is not written back. In this case the performance would just decrease.

It is also important to note, that if the entry is laid out properly in memory, all the hardware-modified bits in the TLB can be written back to memory with a single memory write instruction. Thus it makes no difference whether one or two bits have been modified in the TLB, because writing back one bit or two bits still requires writing back a whole word.

Problem M3.1.B

An advantage of this scheme is that we do not need the TLB Entry Valid bit in the TLB anymore. One bit savings is not very much.

A disadvantage of this scheme is that the kernel needs to ensure that all TLB entries always are valid. During a context switch, all TLB entries would need to be restored (this is time-consuming). And, in general, whenever a TLB entry is invalidated, it will have to be replaced with another entry.

Problem M3.1.C

Changes to exceptions: “Page Table Entry Invalid” and “TLB Miss” exceptions are replaced with exceptions “TLB Entry Invalid” and “TLB No Match”

The TLB Entry Invalid exception will be raised if the VPN matches the TLB tag but the (combined) valid bit is false. When this exception is raised the kernel will need to consult the page table entry to see if this is a TLB miss (valid bit in page table entry is true), or an access of an invalid page table entry (valid bit in page table entry is false). Depending on what the cause of the exception was, it will then have to perform the necessary operations to recover.

The TLB No Match exception will be raised if the VPN does not match any of the TLB tags. If this exception is raised the kernel will do the same thing it did when a TLB Miss occurred in the previous design.

Problem M3.1.D

When loading a page table entry into the TLB, the kernel will first check to see if the page table entry is valid or not. If it is valid, then the entry can safely be loaded into the TLB. If the page table entry is not valid, then the Page Table Entry Invalid exception handler needs to be called to create a valid entry before loading it into the TLB. Thus we only keep valid page table entries in the TLB. If a page table entry is to be invalidated, the TLB entry needs to be invalidated.

Changes to exceptions: Page Table Entry Invalid exception is not raised by the TLB anymore.

Problem M3.1.E

The solution for Problem M3.1.C ends up taking two exceptions, if the PTE has the combined valid bit set to invalid. The first exception will be the TLB No Match exception, which will call a handler. The handler will load the corresponding PTE into the TLB and restart the instruction. The instruction will cause **another** exception right away, because the valid bit will be set to invalid. The exception will be the TLB Entry Invalid exception.

The solution for Problem M3.1.D will only take one exception, because the handler for Page Table Entry Invalid exception will get called by the TLB Miss handler. When the instruction that caused the exception is restarted, it will execute correctly, because the handler will have created a valid PTE and put it in the TLB.

Thus Bud Jet's solution in M3.1.D will be faster.

Problem M3.1.F

Yes, the R bit can be removed in the same way we removed the V bit in 5.1.D. When loading a page table entry into the TLB we check if the data page is resident or not. If it is resident, we can write the entry into the TLB. If it is not resident, we go to the nonresident page handler, loading the page into memory before loading the entry into the TLB. Thus, we only keep page table entries of resident pages in the TLB. In order to preserve this invariant, the kernel will have to invalidate the TLB entry corresponding to any page that gets swapped out. There's no performance penalty since the page was going to be loaded in from disk anyway to service the access that triggered the fault.

Problem M3.1.G

The OS needs to check the permissions before loading the entry into the TLB. If permissions were violated, then the Protection Fault handler is called. Thus, we only keep page table entries of pages that the process has permissions to access.

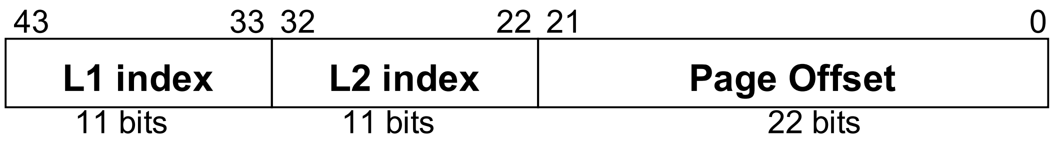
Problem M3.1.H

Whenever a page table entry is loaded into the TLB the U bit in the page table PTE can be set. Thus, we do not need the U bit in the TLB entry anymore.

Whenever a Write Fault happens (store and W bit is 0) the kernel will check the page table PTE to see if the W bit is set there. If it is not set the old Write Fault handler will be called. If the W bit is set, then the kernel will set the M bit in the PTE, set the W bit in the TLB entry to 1, and restart the store instruction. Thus, the M bit is not needed in the TLB either, and hence, TLB entries do not need to be written back to the page table anymore.

Problem M3.2: Page Size and TLBs

Problem M3.2.A



The L1 index and L2 index fields are the same, but the Page Offset field subsumes the L3 index and increases to 22 bits.

Problem M3.2.B **Page Table Overhead**

$$\mathbf{PTO_{4KB}} = \frac{16\text{ KB} + 16\text{ KB} + 8\text{ KB}}{3\text{ MB}} = \frac{40\text{ KB}}{3\text{ MB}} = \mathbf{1.3\%}$$

$$\mathbf{PTO_{4MB}} = \frac{16\text{ KB} + 16\text{ KB}}{4\text{ MB}} = \frac{32\text{ KB}}{4\text{ MB}} = \mathbf{0.8\%}$$

For the 4KB page mapping, one L3 table is sufficient to map the 768 pages since each contains 1024 PTEs. Thus, the page table consists of one L1 table (16KB), one L2 table (16KB), and one L3 table (8KB), for a total of 40 KB. The 768 4KB data pages consume exactly 3MB. The total overhead is 1.3%.

The page table for the 4MB page mapping, requires only one L1 table (16KB) and one L2 table (16KB), for a total of 32 KB. A single 4MB data pages is used, and the total overhead is 0.8%.

Problem M3.2.C **Page Fragmentation Overhead**

$$\mathbf{PFO_{4KB}} = \frac{0}{3\text{ MB}} = \mathbf{0\%}$$

$$\mathbf{PFO_{4MB}} = \frac{1\text{ MB}}{3\text{ MB}} = \mathbf{33\%}$$

With the 4KB page mapping, all 3MB of the allocated data is accessed. With the 4MB page mapping, only 3MB is accessed and 1MB is unused. The overhead is 33%.

Problem M3.2.D

	Data TLB misses	Page table memory references (per miss)
4KB:	**768**	**3**
4MB:	**1**	**2**

The program sequentially accesses all the bytes in each page. With the 4KB page mapping, a TLB miss occurs each time a new page of the input or output data is accessed for the first time. Since the TLB has more than 3 entries (it has 64), there are no misses during the subsequent accesses within each page. The total number of misses is 768. With the 4MB page mapping, all of the input and output data is mapped using a single page, so only one TLB miss occurs.

For either page size, a TLB miss requires loading an L1 page table entry and then loading an L2 page table entry. The 4KB page mapping additionally requires loading an L3 page table entry.

Problem M3.2.E

[✓ **1.01×**] 10× 1,000× 1,000,000×

Although the 4KB page mapping incurs many more TLB misses, with either mapping the program executes 2M loads, 1M adds, and 1M stores (where M = 2^{20}). With the 4MB mapping, the single TLB miss is essentially zero overhead. With the 4KB mapping, there is one TLB miss for every 4K loads or stores. Each TLB miss requires 3 page table memory references, so the overhead is less than 1 page table memory reference for every 1000 data memory references. Since the TLB misses likely cause additional overhead by disrupting the processor pipeline, a 1% slowdown is a reasonable but probably conservative estimate.

Problem M3.3: Page Size and TLBs

Problem M3.3.A

If all data pages are 4KB

Address translation cycles = 100 + 100 +100 (for L1, L2 and L3 PTE)

*Data access cycles = 4K * 100*
(there is no cache, this assumes that memory access is byte-wise)

If all data pages are 1MB

Address translation cycles = 100 + 100 (for L1, L2 PTE)

*Data access cycles = 1M * 100*
(there is no cache, this assumes that memory access is byte-wise)

Problem M3.3.B

*Address translation cycles = (256*3 + 3 + 1) * 100*
*(Note that the arrays are contiguous and share some PTE entries. 256 L3 PTEs per array * 3 arrays, 1 L2 PTE per array * 3 arrays, 1 L1 PTE)*

*Data access cycles = 3M*100*

Problem M3.3.C

No. For the sample program given, all L3 PTEs are used only once.

Problem M3.3.D

4. (1 for L1 and 3 for L2)

Problem M3.4: 64-bit Virtual Memory

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

Problem M3.4.A — Single level page tables

12 bits are needed to represent the 4KB page. There are 64-12=52 bits in a VPN. Thus, there are 2^{52} PTEs. Each is 8 bytes. $2^{52} * 2^3 = 2^{55}$, or 32 petabytes!

Problem M3.4.B — Let's be practical

2^2 segments * $2^{(44-12)}$ virtual pages = 2^{34} PTEs. 2^3 (bytes/PTE) * 2^{34} PTEs = $\mathbf{2^{37}}$ bytes.

It is possible to interpret the question as there being 3 segments of 2^{44} bytes. Thus we'd need:

3 segments * $2^{(44-12)}$ virtual pages = $2^{33} + 2^{32}$ PTEs. $2^3*(2^{33}+ 2^{32}) = \mathbf{2^{36} + 2^{35}}$ bytes.

Problem M3.4.C — Page table overhead

The smallest possible page table overhead occurs when all pages are resident in memory. In this case, the overhead is

$8(2^{11} + 2^{11}*2^{11} + 2^{11}*2^{11}*2^{10}) / 2^{44} \approx 2^{35} / 2^{44} \approx 1 / 2^9$

The largest possible page table overhead occurs when only one data page is resident in memory. In this case, we need 1 L0 page table, 1 L1 page table, 1 L2 page table in order to get data page. Thus the overhead is:

$8(2^{11} + 2^{11} + 2^{10}) / 2^{12} = 10$

Problem M3.4.D — PTE Overhead

PPN is 40-12=28 bits. 28+1+1+3=33 bits.

There are 31 wasted bits in a 64 bit page table entry. It turns out that some of the "wasted" space is recovered by the OS to do bookkeeping, but not much.

Problem M3.4.E Page table implementation

The top level has 1024 = 2^{10} entries. Next level also has 1024 = 2^{10} entries. The 3rd level has 512 = 2^9 entries. So the table is as follows:

Index	Length (bits)
Top-level ("page directory")	*10*
2nd-level	*10*
3rd-level	*9*

Problem M3.4.F Variable Page Sizes

Minimum = 4KB * 64 = 256KB
Maximum = 16MB * 64 = 1GB

Problem M3.4.G Virtual Memory and Caches

Alyssa's suggestion solves the homonym problem. If we add a PID as a part of the cache tag, we can ensure that two same virtual addresses from different processes can be distinghuished in the cache, because their PIDs will be different.

Putting a PID in the tag of a cache does not solve the synonym problem. This is because the synonym problem already deals with different virtual addresses, which presumably would have different tags in the cache. In fact, those two virtual addresses would usually belong to different processes, which would have different PIDs.

Ben is wrong in thinking that changing the cache to be direct mapped helps in any way. The homonym problem still happens, because same virtual addresses still receive the same tags. The synonym problem still happens because two different virtual addresses still receive different tags.

One way to solve both these problems is to make the cache physically tagged, as described in Lecture 5.

Problem M3.5: Cache Basics

Problem M3.5.A

Index	V	Tags (way0)	V	Tags (way1)
0	**1**	0x45	**0**	
1	**1**	0x3D	**0**	
2	**1**	0x2D	**1**	0x25
3	**1**	0x1D	**0**	

Problem M3.5.B

0x34 (hit: index 2)
-> 0x38 (miss: index 3)
-> 0x50 (miss: index 2)
-> 0x54 (hit: index 2)
-> 0x208 (hit: index 1)
-> 0x20C (hit: index 1)
-> 0x74 (miss: index 2)
-> 0x54 (hit: index 2)

Because there are 5 hits and 3 misses,
Average memory access time = 1 + 3 / 8 * 16 = 7 cycles

Problem M3.6: Handling TLB Misses

Problem M3.6.A

Virtual address 0x00030 -> Physical address (0x00D40)

VPN	PPN
0x0100	0x0F01
0x0003	0x00D4

TLB states

Problem M3.6.B

Virtual address 0x00050 -> Physical address (0x00E20)

VPN	PPN
0x0100	0x0F01
0x0101	0x0F02
0x0005	0x00E2

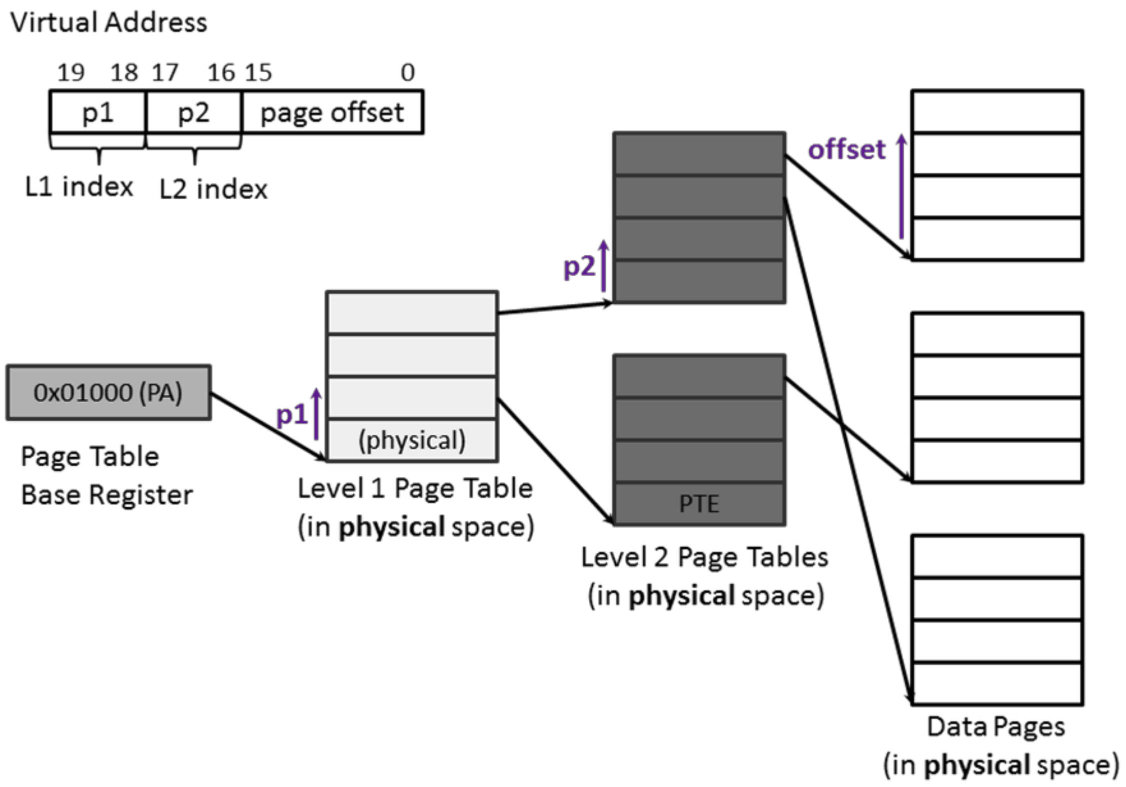
TLB states

Problem M3.6.C

New CPI = 2 + (0.01+0.02)*20 = 2.6

Problem M3.7: 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).

31 20	19 16	15 0
0	Physical Page Number (PPN)	Status Bits

Problem M3.7.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)

Virtual Address:

0xE:17B0 (1110:0001011110110000)

VPN	PPN
0x8	0x3
0xE	0x6

Final TLB states

VA 0xE17B0 => PA __________0x617B0___________

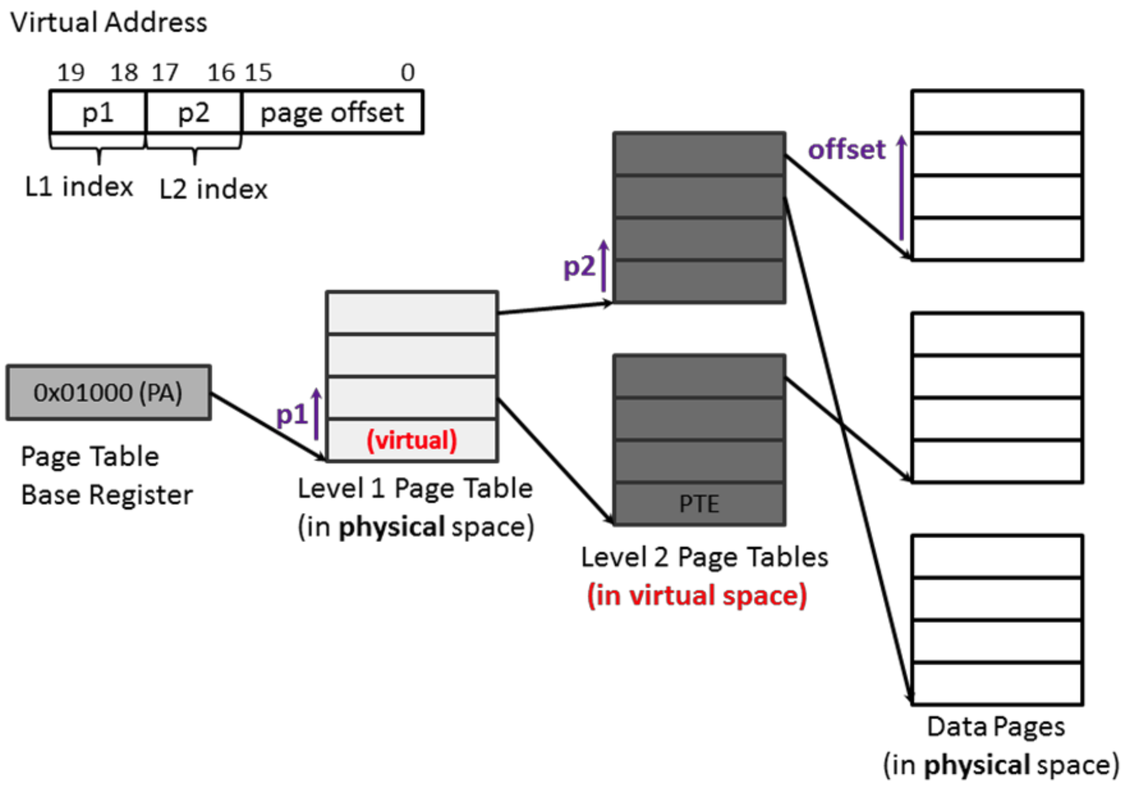
Problem M3.7.B

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

4 * 4 (level 1) + 4 * 4* 4 (level 2) = 80 bytes

Problem M3.7.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 M5.8.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 ":".***

.

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)

VPN	PPN
0x8	0x1

Initial TLB states

<u>**Virtual Address**</u>:

0xA:0708 (1010:0000011100001000)

VPN	PPN
0x8	0x1
0x2	0x1
0xA	0xF

Final TLB states

VA 0xA0708 => PA ____________0xF0708____________

Problem M3.7.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.

1. When the TLB is empty
2. When the VPN of the virtual address and the VPN of the level 1 page table entry are the same