CS3410 Prelim 2 Review

Paul Upchurch

March 27, 2013

Memory hierarchy

Cache

Pipeline hazards

Assembly

Linkers and loaders

Calling conventions

Prelim 1 material (logic, FSM, memory, ALU, processor performance, MIPS, numbers)

a) [6 points] Calling Conventions: For each of the variables `a`, `b`, `c`, `d`, and `i`, state whether it is better to use a callee- or caller-save registers? Why?

```
int foo(int x) {
   int a = 0;
   int b = x;
   int c = b;

   for (int i = 0; i < 10; i++) {
      a = a + b;
      b = bar(a, i);
      c = b * i;

       int d = rab(a, b, c);
      a += d;
   }

   return c;
}
```

a) [6 points] Calling Conventions: For each of the variables `a`, `b`, `c`, `d`, and `i`, state whether it is better to use a callee- or caller-save registers? Why?

```
int foo(int x) {
   int a = 0;
   int b = x;
   int c = b;

   for (int i = 0; i < 10; i++) {
      a = a + b;
      b = bar(a, i);
      c = b * i;

       int d = rab(a, b, c);
      a += d;
   }

   return c;
}
```

d should be caller-save as its value is only used between functions. It is not a long-lived value, and thus it would be wasteful to spend saving and restoring its value in subroutines. In other words, it is okay if it gets clobbered during a function call. This variable is essentially temporary.

a, b, c, and i on the other hand should be callee-save as they are used both before and after function calls. Thus, we want to ensure that they are preserved across subroutines.

c) [3 points] Linkers: Bob links his Hello World program against 9001 static libraries. Amazingly, this works without any collisions. Why?

c) [3 points] Linkers: Bob links his Hello World program against 9001 static libraries. Amazingly, this works without any collisions. Why?

The linker chooses addresses for each library and fills in all the absolute addresses in each with the numbers that it chose.

d) [4 points] Arithmetic: Write a MIPS assembly subroutine that will multiply the value in register `$1` by `5` and store the value in `$2`. Do not write the whole routine, only the instructions that perform the multiply operation. NOTE: you **cannot** use the `MULT` or `MULTU` instruction.

d) [4 points] Arithmetic: Write a MIPS assembly subroutine that will multiply the value in register `$1` by `5` and store the value in `$2`. Do not write the whole routine, only the instructions that perform the multiply operation. NOTE: you **cannot** use the `MULT` or `MULTU` instruction.

```
SLL $2 $1 2
ADDU $2 $2 $1
```

e) [3 points] Numbers: Use the binary and decimal notation (below) to represent the following three amounts $\mathbf{2^{54}}$, $\mathbf{2^{15}}$, and $\mathbf{2^{34}}$?
Write your answer using ***both*** **binary** and **decimal** notation.
(***For example, for 2^{10} we would write 1 kB and 1 thousand-bytes***)

For *binary notation*, you may write the amount of bytes using k for kilo-, M for mega-, G for giga-, T for tera-, P for peta-, E for exa-, Z for zetta-, and Y for yotta-byte: kB, MB, GB, TB, PB, EB, ZB, YB, respectively.

For *decimal notation*, you may approximate and just write thousand-, million-, billion-, trillion-, quadrillion-, quintillion-, sextillion-, septillion-, octillion-**bytes** (…googol-bytes); or 10^3, 10^6, 10^9, 10^{12}, 10^{15}, 10^{18}, 10^{21}, 10^{24}, 10^{27} (…10^{100}), respectively.

e) [3 points] Numbers: Use the binary and decimal notation (below) to represent the following three amounts $\mathbf{2^{54}}$, $\mathbf{2^{15}}$, and $\mathbf{2^{34}}$?
Write your answer using ***both*** **binary** and **decimal** notation.
(***For example, for 2^{10} we would write 1 kB and 1 thousand-bytes***)

For *binary notation*, you may write the amount of bytes using k for kilo-, M for mega-, G for giga-, T for tera-, P for peta-, E for exa-, Z for zetta-, and Y for yotta-byte: kB, MB, GB, TB, PB, EB, ZB, YB, respectively.

For *decimal notation*, you may approximate and just write thousand-, million-, billion-, trillion-, quadrillion-, quintillion-, sextillion-, septillion-, octillion-**bytes** (…googol-bytes); or 10^3, 10^6, 10^9, 10^{12}, 10^{15}, 10^{18}, 10^{21}, 10^{24}, 10^{27} (…10^{100}), respectively.

2^{54} = 16 PB = 16 quadrillion bytes (16 x 10^{15} bytes; e.g.16 to 32 racks worth of storage servers)
2^{15} = 32 kB = 32 thousand bytes (32 x 10^3 bytes; e.g. size of L1 cache)
2^{34} = 16 GB = 16 billion bytes (16 x 10^9 bytes; e.g. amount of storage in your phone)

Assume that we have a byte-addressed 32-bit processor with 32-bit words (i.e. a word is 4 bytes). Suppose further that we are using a direct mapped cache with 4096 128-**byte** cache lines.

b) [7 points] How many bits do we need for the cache tag? Index? And, offset?
Which bits are the cache tag? Which are the index? The offset of a word within the cache line? The offset of a byte within the word?

Assume that we have a byte-addressed 32-bit processor with 32-bit words (i.e. a word is 4 bytes). Suppose further that we are using a direct mapped cache with 4096 128-**byte** cache lines.

b) [7 points] How many bits do we need for the cache tag? Index? And, offset?
Which bits are the cache tag? Which are the index? The offset of a word within the cache line? The offset of a byte within the word?

Tag: 13 bits (tag is 32 – 12 – 7)
Index: 12 bits (4096 is 2^{12}; i.e. we need 12 bits to index into a cache with 4096 cache lines)
Offset: 7 bits (2 bits for offset within a word and 5 bits for offset of word within a cache line)

Tag: 19-31
Index: 7-18
Offset within a cache line: 2-6
Offset within a word: 0-1

d) [4 points] How large is the cache in bytes (data only)? How many words can it hold?

d) [4 points] How large is the cache in bytes (data only)? How many words can it hold?

The cache is 512 kB (128 bytes per cache line x 4096 cache lines = 2^7 x 2^{12} = 2^{19} = 512 kB) and can hold 2^{17} words (2^{19} bytes / 4 bytes per word = 2^{19} / 2^2 = 2^{17})

e) [3 points] How large does the cache need to be in bytes **including data and overhead**?

e) [3 points] How large does the cache need to be in bytes **including data and overhead**?

Data = 512 kB (4096 x sizeof(block) = 4096 x 128 bytes)
Overhead = 4096 x (sizeof(tag) + valid bit)
= 4096 x (13 + 1) = 4096 x 14 bits
= 57,344 bits
= 7,168 bytes (57,344 bits / 8 bits per byte)

Data + Overhead = 512 kB + 7,168 bytes
= 519 kB

f) [4 points] What is the number of cache hits and misses given the following memory references from homework4?

```
LW $1 ← M[221]
LW $2 ← M[2]
LW $3 ← M[40]
LW $1 ← M[68]
LW $2 ← M[80]
LW $3 ← M[0]
LW $1 ← M[44]
LW $2 ← M[72]
```

f) [4 points] What is the number of cache hits and misses given the following memory references from homework4?

```
LW $1 ← M[221]   M
LW $2 ← M[2]     M
LW $3 ← M[40]    H
LW $1 ← M[68]    H
LW $2 ← M[80]    H
LW $3 ← M[0]     H
LW $1 ← M[44]    H
LW $2 ← M[72]    H
```

Misses = 2
Hits = 6

g) [3 points] Is the performance good or bad (cache hits vs misses)? Why; in particular, name the property responsible for the performance?

g) [3 points] Is the performance good or bad (cache hits vs misses)? Why; in particular, name the property responsible for the performance?

Performance is great due to spatial locality and large block sizes.

a) [10 points] Assuming the same 5-stage pipeline with branch/jump done in the decode stage, identify all of the data and control hazards in this code.

```
memcpy:
      # $a0 contains the starting destination address
      # $a1 contains the starting source address
      # $a2 contains the count of bytes to copy
   1  ADD   $t0, $a0, $a2   # calculate ending dest address
   2  BEQZ  $a2, END # skip everything if count is zero
TOP 3  LB    $t1, 0($a1)
   4  SB    $t1, 0($a0)
   5  ADDIU $a0, $a0, 1
   6  ADDIU $a1, $a1, 1
   7  BNE   $a0, $t0, TOP
END 8  JR    $ra
```

a) [10 points] Assuming the same 5-stage pipeline with branch/jump done in the decode stage, identify all of the data and control hazards in this code.

```
memcpy:
      # $a0 contains the starting destination address
      # $a1 contains the starting source address
      # $a2 contains the count of bytes to copy
 1    ADD   $t0, $a0, $a2   # calculate ending dest address
 2    BEQZ  $a2, END # skip everything if count is zero
TOP 3 LB    $t1, 0($a1)
 4    SB    $t1, 0($a0)
 5    ADDIU $a0, $a0, 1
 6    ADDIU $a1, $a1, 1
 7    BNE   $a0, $t0, TOP
END 8 JR    $ra
```

line 1, 2: no hazards
line 3: first loop has control hazard from line 2
line 3: other loops have data hazard from line 6
line 4: data hazard from line 3
line 5, 6: no hazards
line 7: data hazard from 5
line 8: control hazard from 7

+2 points each hazard:

b) [4 points] If all hazards are resolved ***entirely by stalling*** (no forwarding and ***cannot*** write to and read from the register file during the same cycle), how many cycles are needed for each *loop* iteration? (For partial credit, show your work by indicating where and how many stalls happen in the code. Also, you can use the multi-cycle graph if you want, but do not have to use it.)

b) [4 points] If all hazards are resolved ***entirely by stalling*** (no forwarding and ***cannot*** write to and read from the register file during the same cycle), how many cycles are needed for each *loop* iteration? (For partial credit, show your work by indicating where and how many stalls happen in the code. Also, you can use the multi-cycle graph if you want, but do not have to use it.)

before line 3: 1 stall
before line 4: 3 stalls
before line 7: 2 stalls
before line 8: 1 stall

loop iteration takes 11 cycles (5 instructions + 6 stalls)

c) [5 points] Assume the pipeline now has a *hazard detection unit* and can automatically insert NOPs (stalls) and flush instructions required for correct execution.

Fill in the multi-clock cycle graph with the required stalls for the original instructions to execute correctly. What is the **minimal** number of cycles it will take for the code to completely clear the MIPS pipeline if the processor automatically inserts the required NOPs and none of the branches are taken?

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
1	ADD	IF	ID	Ex	M	W																	
2	***BEQZ***																						
3	***LB***																						
4	***SB***																						
5	***ADDIU***																						
6	***ADDIU***																						
7	***BNE***																						
8	***JR***																						

c) [5 points] Assume the pipeline now has a *hazard detection unit* and can automatically insert NOPs (stalls) and flush instructions required for correct execution.

Fill in the multi-clock cycle graph with the required stalls for the original instructions to execute correctly. What is the **minimal** number of cycles it will take for the code to completely clear the MIPS pipeline if the processor automatically inserts the required NOPs and none of the branches are taken?

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
1	ADD	IF	ID	Ex	M	W																	
2	*BEQZ*		*IF*	*ID*	*Ex*	*M*	*W*																
3	*LB*			*IF*	*IF*	*ID*	*Ex*	*M*	*W*														
4	*SB*					*IF*	*ID*	*ID*	*ID*	*ID*	*Ex*	*M*	*W*										
5	*ADDIU*						*IF*	*IF*	*IF*	*IF*	*ID*	*Ex*	*M*	*W*									
6	*ADDIU*										*IF*	*ID*	*Ex*	*M*	*W*								
7	*BNE*											*IF*	*ID*	*ID*	*ID*	*Ex*	*M*	*W*					
8	*JR*												*IF*	*IF*	*IF*	*IF*	*ID*	*Ex*	*M*	*W*			

It will take 19 cycles for the code to completely clear the MIPS pipeline

d) [6 points] Suppose instead that **branches**, **jumps**, and **memory loads** have one ***delay slot***, and other data hazards are resolved by forwarding. Re-order the instructions to achieve the highest performance possible. You can insert NOPs if needed, and change the instructions in minor ways (e.g. changing constants, changing offsets, using different registers, …) but don't change them to different operations or add any new instructions.

```
memcpy:
     # $a0 contains the starting destination address
     # $a1 contains the starting source address
     # $a2 contains the count of bytes to copy
     ADD   $t0, $a0, $a2   # calculate ending dest address
     BEQZ  $a2, END # skip everything if count is zero
TOP: LB    $t1, 0($a1)
     SB    $t1, 0($a0)
     ADDIU $a0, $a0, 1
     ADDIU $a1, $a1, 1
     BNE   $a0, $t0, TOP
END: JR    $ra
```

d) [6 points] Suppose instead that **branches**, **jumps**, and **memory loads** have one ***delay slot***, and other data hazards are resolved by forwarding. Re-order the instructions to achieve the highest performance possible. You can insert NOPs if needed, and change the instructions in minor ways (e.g. changing constants, changing offsets, using different registers, …) but don't change them to different operations or add any new instructions.

```
memcpy:
      # $a0 contains the starting destination address
      # $a1 contains the starting source address
      # $a2 contains the count of bytes to copy
      ADD   $t0, $a0, $a2   # calculate ending dest address
      BEQZ  $a2, END # skip everything if count is zero
TOP:  LB    $t1, 0($a1)
      SB    $t1, 0($a0)
      ADDIU $a0, $a0, 1
      ADDIU $a1, $a1, 1
      BNE   $a0, $t0, TOP
END:  JR    $ra
```

```
memcpy:
      BEQZ  $a2, END
      ADD   $t0, $a0, $a2 # in BEQ delay slot
TOP:  LB    $t1, 0($a1)
      ADDIU $a0, $a0, 1   # in LB delay slot
      SB    $t1, -1($a0)  # ADDIU moved earlier, so use -1 offset
      BNE   $a0, $t0, TOP
      ADDIU $a1, $a1, 1   # in BNE delay slot
END:  JR    $ra
      NOP
```

e) [3 points] TRUE/FALSE? A fully associative cache with LRU replacement always has a lower (better) miss rate than a direct-mapped cache.

Provide a convincing argument if true, or a counterexample if false.

e) [3 points] TRUE/FALSE? A fully associative cache with LRU replacement always has a lower (better) miss rate than a direct-mapped cache.

Provide a convincing argument if true, or a counterexample if false.

FALSE. For example: with a 4-line cache and 256-byte block size, access the following addresses.

0x1000, 0x1100, 0x1200, 0x1300, 0x1400, 0x1000, 0x1100, 0x1200, 0x1300, 0x1400

The direct-mapped cache has 3 hits out of 10, but the fully-associative cache has none!

a) [5 points] Assume that we have a byte-addressed 64-bit processor with **64-bit words**. Suppose that this processor has a 48-word, three-way, set-associative cache (LRU replacement) with 2-word cache lines. Split the 64-bit address into "tag", "index", and "cache-line offset" pieces. Which address bits comprise each piece (one is given)?

tag:
index:
cache-line offset: **bits 3 – 0 (Given)**

a) [5 points] Assume that we have a byte-addressed 64-bit processor with **64-bit words**. Suppose that this processor has a 48-word, three-way, set-associative cache (LRU replacement) with 2-word cache lines. Split the 64-bit address into "tag", "index", and "cache-line offset" pieces. Which address bits comprise each piece (one is given)?

tag:	*bits 63 – 7*
index:	*bits 6 – 4*
cache-line offset:	**bits 3 – 0 (Given)**

b) [4 points] How many sets does this cache have? Explain.

b) [4 points] How many sets does this cache have? Explain.

There are 8 sets in this cache (size of 2^{index}).

c) [5 points] Assume that the processor makes the following byte accesses. Label each reference address as a Hit (H) or a Miss (M). Also, identify each cache miss as a **compulsory** (i.e. cold), **conflict**, or **capacity** miss.

Byte Address	Hit/Miss?	Miss Type
38 (0x026)	Miss	Compulsory
172 (0x0AC)		
144 (0x090)		
85 (0x055)		
424 (0x1A8)		
111 (0x06F)		
174 (0x0AE)		
551 (0x227)		
90 (0x05A)		
32 (0x020)		
428 (0x1AC)		
544 (0x220)		
96 (0x060)		
422 (0x1A6)		
170 (0x0AA)		

c) [5 points] Assume that the processor makes the following byte accesses. Label each reference address as a Hit (H) or a Miss (M). Also, identify each cache miss as a **compulsory** (i.e. cold), **conflict**, or **capacity** miss.

Byte Address	Hit/Miss?	Miss Type
38 (0x026)	Miss	Compulsory
172 (0x0AC)	*Miss*	*Compulsory*
144 (0x090)	*Miss*	*Compulsory*
85 (0x055)	*Miss*	*Compulsory*
424 (0x1A8)	*Miss*	*Compulsory*
111 (0x06F)	*Miss*	*Compulsory*
174 (0x0AE)	*Hit*	–
551 (0x227)	*Miss*	*Compulsory*
90 (0x05A)	*Hit*	–
32 (0x020)	*Miss*	*Conflict*
428 (0x1AC)	*Miss*	*Conflict*
544 (0x220)	*Hit*	–
96 (0x060)	*Hit*	–
422 (0x1A6)	*Hit*	–
170 (0x0AA)	*Miss*	*Conflict*

e) [5 points] You have a 1 GHz processor with 2 levels of cache, 1 level of DRAM, and a DISK for virtual memory. Assume that it has a unified second-level cache. Assume that the access time to read a page from disk is 1,000,000 cycles. Assume that the memory system has the following parameters:

Component	Hit Time	Miss Rate	Block Size
L1 Data Cache	1 cycle	5% data	64 bytes
L2 Unified Cache	20 cycles + 1 cycle/64 bits	2%	128 bytes
DRAM	100 cycles + 25 cycles/8 bytes	0.1% (page faults)	16K bytes (page size)

What is the average memory access time (AMAT) for the L1 Data Cache?
[1M = 1K × 1K = 1024×1024]
[hint: write the general formula for AMAT]

e) [5 points] You have a 1 GHz processor with 2 levels of cache, 1 level of DRAM, and a DISK for virtual memory. Assume that it has a unified second-level cache. Assume that the access time to read a page from disk is 1,000,000 cycles. Assume that the memory system has the following parameters:

Component	Hit Time	Miss Rate	Block Size
L1 Data Cache	1 cycle	5% data	64 bytes
L2 Unified Cache	20 cycles + 1 cycle/64 bits	2%	128 bytes
DRAM	100 cycles + 25 cycles/8 bytes	0.1% (page faults)	16K bytes (page size)

What is the average memory access time (AMAT) for the L1 Data Cache?
[1M = 1K × 1K = 1024×1024]
[hint: write the general formula for AMAT]

$AMAT_{dram} = (100+25ns\times16)cycles + 0.001\times AMAT_{disk} = 400+1000cycles = 1500\ cycles$
$AMAT_{2lev} = (20+8)cycles + 0.02\times AMAT_{dram} = 58\ cycles$
$AMAT_{1lev\ Data} = 1 + 0.05\times AMAT_{2lev} = 3.9\ cycles$

b) [3 points] What is the problem or flaw with sending virtual memory addresses directly to cache without using a TLB to convert the virtual address to a physical address first?

b) [3 points] What is the problem or flaw with sending virtual memory addresses directly to cache without using a TLB to convert the virtual address to a physical address first?

The synonym problem. One physical address may map to 2 virtual addresses and be present in the cache twice.