

Principles in Computer Architecture

CSE 240A Optional Homework

November 26, 2016

No part of this homework will be graded; we are providing these problems for you to practice for your final exam. No resemblance between these practice problems and final exam questions should be expected.

Problem Set

1 (Addressing Tradeoffs)

In this question we will be examining the tradeoffs associated with the set of addressing modes provided by ISAs. Below are 3 ISAs, each with a different set of addressing modes.

MIPS-like ISA: This is a RISC ISA that uses loads and stores to access memory. ALU operations only access registers and all memory accesses occur via load and store instructions, which utilize the displacement addressing mode. Both ALU instructions and LD can replace an operand by an immediate, by utilizing variant instruction opcodes. This ISA also provides PC-Relative branch instructions. It has 8 addressable registers.

2 Operand, 1 Memory Access ISA: This ISA has instructions with two operands, a source and a destination. As a result of only providing two operands, the instructions in this ISA will require fewer bits for specifying the operands and the additional bits can instead be used to provide more addressing modes, potentially allowing more complex operations to be performed in a single instruction. It provides 8 addressing modes, which correspond to all the ones listed in the table at the back of the exam. Like MIPS, this ISA uses PC-Relative branches. This ISA has 8 addressable registers.

3 Operand, 3 Memory Access ISA: This ISA uses extra bits to distinguish between a memory access and a register access for the operands in ALU operations. Due to the increased number of bits required, the number of addressing modes available for ALU operations is limited to register and pre-decrement; however, any of the operands, source or destination, could access memory and do it in the same instruction if desired as well. You may notice that there is no way of expressing an immediate in an ALU instruction due to the utilization of 3 operands. This ISA provides load and store instructions which can, like the two operand ISA of the previous paragraph, utilize the same 8 addressing modes. As with the other two ISAs, branches are PC-Relative. Due to the extra bit needed to store the addressing mode to use for each operand, there are only 4 addressable registers.

(Part A) For the following fragments of code, please compile and write the sequence of assembly instructions according to the specified ISAs. You may find the initial value of a variable f at the memory address $[f]$. For accessing array a , you can use the notation $[a]$ to indicate that you would like to use the base address. Keep in mind that accessing memory has a performance hit associated with it, so you should try to store values that will be getting reused frequently in registers, if possible. Please aim at making your code as space efficient as you can.

(a) Please compile and write this code in the *MIPS-like* and *3 Operand, 3 Memory Access ISA* ISAs.

```
for (i = 0; i < N; i++)
{
    x[i] = y[i] + z[i];
    if (i >= 3)
    {
        a[i] = a[i-1] + a[i-2] + a[i-3];
    }
    if (a[i] == 0)
    {
        a[i] = x[i];
    }
}
```

(b) Please compile and write this code in the *MIPS-like* and *2 Operand, 1 Memory Access ISA*s.

```
for (i = N - 1; i > 0; i++)
{
    x = a[i + 99];
    y = b[i];
    if (x > y)
    {
        c[i] = x - y;
    }
    else
    {
        c[i] = y - x;
    }
}
```

(Part B) The previous part of this question we hope gave you some practice with exploring tradeoffs in mapping code to different ISAs. Of course, you realize the previous part gave you some experience but its focal point was the length of the resulting code and perhaps a bit more evident characteristics such as number of memory accesses. While code size is important (particularly in the early history of computer science and continues to be important, particularly in embedded processors), there are other question to consider nowadays that mostly involve the pipelineability of the resulting code, with immediate effects on overall performance. In this part of the question, we want to prod you to both abstract a bit some of the software impacts of particular ISAs, but also get you to think about the hardware implications, pipelineability, and the consequent impacts on performance.

A number of attributes regarding pipelined hardware implementations' performance impact can be categorized as:

- (a) increase in forwarding paths
- (b) increased number of interlocks; this has an impact both on performance and interconnect and control complexity
- (c) increases in pipeline length variability
- (d) deeper (more stages) pipelines
- (e) increase in number of RAWs
- (f) incidence of WARs
- (g) incidence of WAWs
- (h) concurrent writes

In a similar vein, one can contemplate the software implications of particular ISAs. While you gleaned some ramifications of the software aspects by writing the codes in the previous part, somewhat higher level ramifications can be gleaned by looking at application domains such as:

- (a) numeric code [large number of array traversals]
- (b) codes with reuse windows [codes that bring in a particular array instance in one loop instantiation that can be reused multiple times in later instantiations of the same loop]
- (c) codes with many rapid reuses of variables

Given all these hardware and software attributes, could you provide a multidimensional assessment of the three ISAs listed? Please pay particular attention to characteristics such as memory accesses only through load/store, number of registers, ability to signal immediate values in an instruction, availability of addressing modes such as pre-decrement, and so on.

2 (WAR and RAW hazards in scoreboard)

The following piece of code is considered for **all the three parts** of this question. Assume for all referenced READ registers (that are not being written in the given code) that their values are ready at the initiation of the given code. Assume that both the ADDD and the SUBD instructions take 4 EX steps, while a MULTD takes 8 and a DIVD takes 20.

```
DIVD F4, F10, F2
MULTD F10, F2, F12
SUBD F6, F10, F8
ADDD F8, F2, F12
```

(Part A) In a scoreboard processor, once an instruction reaches the *Write Results* stage, it needs to check for possible WAR hazards before it can perform the actual *Write* operation. Yet differentiating whether this instruction is subject to a WAR hazard (and therefore needs to stall until the WAR hazard is cleared) or whether there exists a RAW hazard instead is not so simple, since in both cases the **Fi** field (the destination register) of that instruction matches the **Fj** or **Fk** field (the source registers) of the instructions in flight with which either hazard may be in effect. How does a scoreboard processor distinguish these two situations and figure out which one is a WAR hazard? Give a brief explanation. (*Hint: think about the MULTD and the ADDD instructions in the code above.*)

(Part B) In a scoreboard processor, one part of the scoreboard, which keeps track of the functional units status, has two fields **Qj** and **Qk**, which keep records as to which functional units will produce the source registers **Fj** and **Fk**. Meanwhile, another two fields, **Rj** and **Rk**, are used to indicate whether **Fj** and **Fk** are ready or not. Since the scoreboard does not rely on renaming and the operands are fetched from the register file directly, it would seem that only using the **Rj** and **Rk** fields would suffice. Therefore, one engineer in a design team has suggested that the **Qj** and **Qk** fields can be discarded to save hardware resources. The company's attempts to use this idea resulted eventually in this poor engineer getting fired. What is the reason? Why do we need **Qj** and **Qk** in scoreboard? (*Hint: think about the DIVD and the SUBD instructions in the code above.*)

(Part C) Now that we know **Qj** and **Qk** are needed to keep records of the producer of the source registers **Fj** and **Fk**, we are searching desperately for alternative ways to reduce the significant area overhead of the scoreboard. Engineers in the same company have now focused on the possible redundancy of the **Rj** and **Rk** fields, since they have noticed that **Qj** and **Qk** can also indicate whether **Fj** and **Fk** are ready or not. Now instead of discarding the two fields **Qj** and **Qk**, they are thinking of discarding the two bits **Rj** and **Rk**. Please evaluate this idea. Will the scoreboard work correctly or not with this modification?

3 (Renaming Registers)

Your company designs application specific systems, and a client comes to you with 3 software products, Sapphire, Ruby and Amethyst. Note that the instructions in all three require a special **imm[Rx*imm2]** addressing mode.

However, your compiler and hardware teams get into an argument over how to design the best system for each of the products. The four groups that emerge are:

1. The hardware uses an **IF ID EX* M WB** pipeline with forwarding, and has **only 4 FP registers**. There are different EX* stages the instructions can go through; EX_{addsub} takes **1 cycle**, EX_{addmul} (used for the strange addressing mode) takes **2 cycles**, and EX_{muldiv} takes **8 cycles**; these EX* stages are pipelined as well! The team wants to write a smart compiler that does register renaming, loop unrolling, and re-orders the instructions to achieve the best steady-state performance.
2. The hardware is the same as above, except for two differences; it has **12 FP registers** instead of 4, but the EX_{addmul} takes **8 cycles**. Again, this group wants to rely solely on the compiler to extract ILP.
3. The third and fourth groups want the compiler to do nothing fancy. The hardware uses the *original* Tomasulo's algorithm, **without ROB** (or any physical register renaming for that matter). The Functional Units are shown below. Note that the *Memory Unit* does the effective address computation and accesses memory for **both loads and stores**. The memory unit has **2 load buffers** and **2 store buffers**. Each FU has 2 reservation stations, and the machine has 4 FP registers. Account for a **1-cycle CDB delay**.

Functional Unit	Execution Cycles
Addsub	1
Memory Unit	8
Muldiv	5

4. This group agrees with the third group just discussed on most counts. Their only disagreement is that they are proposing a faster **Memory Unit that takes only 2 cycles**. To compensate for its extra cost, they are proposing to reduce the **number of reservation stations per FU down to 1**.

Your profiling team identifies a loop in each that takes up most of the time for that product. They also guarantee that **exceptions won't occur** in these loops, so imprecise exceptions are okay.

You inspect the loops given, and have a hunch that no single method may turn out to be best for all three products. For each product, you need to now decide which of the four approaches works best. Explain why!

Note that you aren't expected to try all 4 methods for each loop. Inspect the loops before you start! Also, $R_1 = 1$ at the beginning of each loop.

(Part A) Ruby

```
for i = 1 to 15 {  
    A[i+1] *= A[i]  
}  
  
loop:  ld    F0,  0(R1*1)  
      ld    F1,  1(R1*1)  
      mul   F2,  F0, F1  
      sd    F2,  1(R1*1)  
      addui R1,  #1  
      bnz   R1,  #-16, loop // End loop when R1 == 16
```

(Part B) Sapphire

```
for i = 1 to 9 {  
    A[5i+1] += A[5i]  
}  
  
loop:  ld    F0,  0(R1*5)  
      ld    F1,  1(R1*5)  
      add   F2,  F0, F1  
      sd    F2,  1(R1*5)  
      addui R1,  #1  
      bnz   R1,  #-10, loop // End loop when R1 == 10
```

(Part C) Amethyst

```
for x = 1 to 3 {  
    A[5x+4] = A[5x+3]  
    A[5x+3] = A[5x+2]  
    A[5x+2] = A[5x+1]  
    A[5x+1] = A[5x]  
}  
  
loop:  ld    F0,  3(R1*5)  
       sd    F0,  4(R1*5)  
       ld    F0,  2(R1*5)  
       sd    F0,  3(R1*5)  
       ld    F0,  1(R1*5)  
       sd    F0,  2(R1*5)  
       ld    F0,  0(R1*5)  
       sd    F0,  1(R1*5)  
       addui R1,  #1  
       bnz   R1,  #-4, loop // End loop when R1 == 4
```

4 (The Last Drop of ILP)

The Scoreboard allows some degree of out-of-order execution, and it helps in the case of WAR dependencies. However, *just* because the scoreboard can do ILP doesn't mean the compiler cannot perform any optimizations to improve ILP!

(**Part A**) For the following blocks of code, perform explicit register renaming and re-order the instructions to **minimize execution time in a scoreboard**. There are only 6 registers, **F0, F2, F4, F6, F8, F10**, and **one** of each type of functional unit. Assume the following functional unit latencies:

Functional Unit	Execution Cycles
FP div	20
FP mul	8
FP addsub	3

Code 1.

```
F0 = F2 / F4
F4 = F0 * F6
F4 = F6 + F6
F8 = F4 / F2
...
```

No outstanding subsequent reads to F10 before write to F10

Code 2.

```
F0 = F2 / F4
F4 = F0 * F6
F4 = F6 + F6
F8 = F4 / F10
...
```

No outstanding subsequent reads to F4 before write to F4

Code 3.

```
F0 = F2 / F4
F2 = F4 * F4
F6 = F8 * F10
...
```

Outstanding subsequent reads to F0, F2, F4, F6, F8 and F10 before a write to the corresponding register.

(**Part B**) Of course, if there are compiler optimizations for the scoreboard, surely they can help with Tomasulo's algorithm! Perform explicit register renaming and re-order the instructions to **minimize execution time in Tomasulo's algorithm**. Again, there are only 6 registers, F0, F2, F4, F6, F8 and F10. There is a **one cycle CDB delay**. There is **only one of each type** of functional unit, and each has **2 reservation stations**. Assume the following functional unit latencies (*they are identical to Part A*):

Functional Unit	Execution Cycles
FP div	20
FP mul	8
FP addsub	3

Code 1.

```

F0 =  F2 / F4
F4 =  F0 * F6
F4 =  F6 + F6
F8 =  F4 / F10
...
```

No outstanding subsequent reads to F4 before write to F4

Code 2.

```

F0 =  F2 / F4
F2 =  F4 * F4
F6 =  F8 * F10
F4 =  F0 * F6
F6 =  F6 / F4
F10 = F2 + F10
...
```

Outstanding subsequent reads to F0, F2, F4, F6, F8 and F10 before a write to the corresponding register.

5 (Multiple Issue)

You are tasked with designing a new processor microarchitecture, and you are trying to figure out how best to allocate your hardware resources. Which of the hardware and software techniques you learned in Chapter 3 should you apply? You have a list of latencies for the functional units and for memory, as well as some representative code. Your boss has been somewhat vague about the performance requirements of your new design, but you know from experience that, all else being equal, faster is usually better. Start with the basics. In the table below you will find a list of non-unit latencies followed up by a sequence of instructions on which you will be working.

Memory LD	+4
Memory SD	+1
Integer ADD/SUB	+0
Branches	+1
ADDD	+1
MULTD	+5
DIVD	+12

Table 1: Latencies beyond single cycle

Loop:	LD	F2, 0(Rx)
I0:	DIVD	F8, F2, F0
I1:	MULTD	F2, F6, F2
I2:	LD	F4, 0(Ry)
I3:	ADDD	F4, F0, F4
I4:	ADDD	F10, F8, F2
I5:	ADDI	Rx, Rx, #8
I6:	ADDI	Ry, Ry, #8
I7:	SD	F4, 0(Ry)
I8:	SUB	R20, R4, Rx
I9:	BNZ	R20, Loop

Table 2: Code

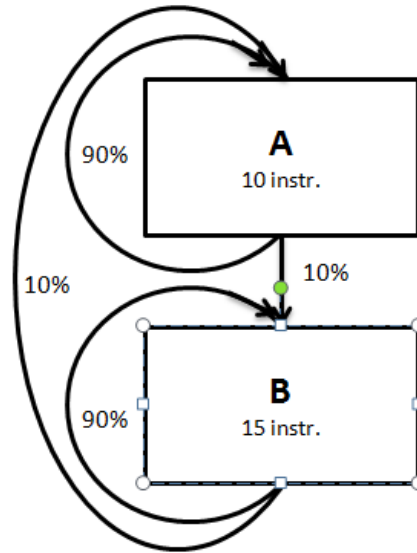
(Part A) What would be the baseline performance (in cycles, per loop iteration) of the code sequence if no new instruction's execution could be initiated until the previous instruction's execution had completed? Ignore front-end fetch and decode. Assume for now that execution does not stall for lack of the next instruction, but only one instruction/cycle can be issued. Assume the branch is taken, and that there is a one cycle branch delay slot. Please show the state of the pipeline as the instruction(s) are executed.

(Part B) Consider a multiple-issue design. Suppose you have two execution pipelines, each capable of beginning execution of one instruction per cycle, and enough fetch/decode bandwidth in the front end so that it will not stall your execution. Assume results can be immediately forwarded from one execution unit to another, or to itself. Further assume that the only reason an execution pipeline would stall is to observe a true data dependency. Now how many cycles does the loop require? Please show the state of the pipeline as the instruction(s) are executed.

(Part C) Reorder the instructions to improve performance of the code. Assume the two-pipe machine in (Part B) and that any issues with out-of-order completion have been dealt with successfully. Just worry about observing true data dependences and functional unit latencies for now. How many cycles does your reordered code take? Please show the state of the pipeline as the instruction(s) are executed.

6 (Branches and Caches)

You have the following loop constantly running on a processor:



Note that this processor is specially designed to handle unorthodox cache sizes, so the hardware to compute the index modulo any value exists.

(Part A) Assume the instruction cache is a direct-mapped cache, where each block can hold 1 instruction. For each size, how often is there a miss in the instruction cache in steady-state? Explain your reasoning.

1. Cache holds 5 instructions.
2. Cache holds 10 instructions.
3. Cache holds 15 instructions.
4. Cache holds 20 instructions.

(Part B) If a block instead holds 2 instructions, how often is there a miss if:

1. The cache has 5 blocks.
2. The cache has 10 blocks.

(**Part C**) In an attempt to improve performance, you introduce a **local predictor**. The local predictor has the following characteristics:

- The branches at the end of A and B are **not** aliased; they map to different predictors.
- Until the prediction is made, no instructions are fetched.
- The **two subsequent instructions** fetched after the prediction is made are **speculatively issued**. The prediction is verified later (*so the third instruction to issue after a prediction will be from the correct path*).
- The predictor has 100% accuracy when A repeats, 50% accuracy when A continues to B, and 50% accuracy both ways for the branch at the end of B.

Given these characteristics, how often is there a miss in the instruction cache in steady-state **if the cache holds 12 instructions** and each block holds 1 instruction? Show the breakdown of how you arrived at that number.

If you could change the block size of the cache but keep the cache size constant at 12 instructions, how many instructions/block would be ideal? Explain your reasoning.