

Principles in Computer Architecture

CSE 240A Homework One

January 15, 2015

*Only Problem Set Two will be graded. **Turn in only Problem Set Two** on January 28, 2015 (**Wednesday**) by **1:00 pm**.*

*Submit homework via csemoodle <csemoodle.ucsd.edu> as a **PDF**. Other formats may or may not be viewed and evaluated accurately.*

Problem Set One

1. Hennessy & Patterson (5th Ed.) 1.1 (*Chip Manufacturing*)
2. Hennessy & Patterson (5th Ed.) 1.8 (*Moore's Law*)
3. Hennessy & Patterson (5th Ed.) 1.14 (*Speedups*)
4. Hennessy & Patterson (5th Ed.) A.8 (*Instruction Encoding*)
5. Hennessy & Patterson (5th Ed.) A.18 (*ISA Styles*)
6. Hennessy & Patterson (5th Ed.) C.4 (*Hazards of Pipelining*)
7. Hennessy & Patterson (5th Ed.) C.5 (*Addressing Modes*)
8. Hennessy & Patterson (5th Ed.) C.6 (*Register-memory Pipelining*)
9. Hennessy & Patterson (5th Ed.) C.13 (*Scoreboards*)

Problem Set Two

1. The Funny Thing About ISAs...

You are given 3 ISAs.

Stack ISA: This is a typical stack ISA with **add**, **mult**, **sub**, **push X**, **pop Y**, **bz Z**. **X** may be an immediate value or a [memory address], so *Push X* can therefore put an immediate value or a variable from memory on the top of the stack. **Y** may only be a [memory address], so *Pop Y* can save the top-of-stack to a location in memory.

BZ Z takes **Z** as a **PC-relative** address (so +1 = next instruction, -4 = 4 instructions back). The branch is *taken iff top-of-stack is 0*. **The branch instruction also pops the top of stack.**

Accumulator ISA: This is an accumulator ISA with **add X**, **mult X**, **sub X**, **ld X**, **st Y**, **bz Z**. **X** may be an immediate value or a [memory address]. The instructions use the accumulator as the other operand and the destination. **Y** may only be a [memory address].

The branch instruction semantics are similar to the same instruction in *Stack ISA*. **Z** is a **PC-relative** target address, and the branch is taken iff the Accumulator == 0.

RISC ISA: The standard MIPS ISA.

For all three ISAs, assume the ISA is implemented in a non-pipelined manner; you do not need to worry about NOOP insertion or hazards introduced by the implementation. In addition, in order to simplify things, the initial value of variable *f* can be found at memory location [*f*].

(Part A) Write code in all three ISAs for the given block of code. Also list the number of memory references each incurs.

When writing code, number instructions in increasing numerical order and use the difference for PC-relative addressing. For *Stack ISA*, show the initial state of the stack. For *RISC ISA*, list which registers you assume to contain the values of which variables (e.g. $R_1 \leftarrow a$).

```
a = b + c;
d = 8 * a;
e = d * 5;
f = a - 7;
g = e - f;
```

(Part B) Do the same for this second block of code without any behavioral optimizations/alterations. (Write code in all three ISAs, list memory references, show initial stack state for *Stack ISA*, and list initial register-variable mapping for *RISC ISA*.)

```
n = p + q;
x = f - 5;
if x:
    m = w - n;
else:
    m = n - w;
endif
```

2. -MIPSs

We are augmenting the standard MIPS ISA and its 5-stage pipeline architecture with an additional addressing mode: *auto-decrement*. This enhancement allows typical array-traversal operations to be faster, since the LD or ST instructions can decrement the memory address in the same instruction that they access memory. You decide to make it a **pre-decrement**, meaning the value is decreased by 1 *before* using it as an address. You are also only allowing this addressing mode in LD and ST instructions. This addressing mode however raises a few interesting questions regarding the implementation of auto-decrement. Mainly, there are two decisions to be made. The first is whether to write the decrement to the register file in the **ID** stage or pass the value down the pipeline and only write in **WB**. The other is how much a second write port on the register file would help. One thing you remember to keep in mind when making both decisions is that, in the standard pipeline, the ‘concurrent’ decode and writeback in the register file are resolved by doing the writeback in the first half of the cycle and the read in the second.

For the first 4 parts of this question, assume you choose to go with only 1 write port on the register file and do the auto-decrement in the **ID** stage (since you already have an adder in the ID stage to calculate the target address of a branch, you think you can re-use it!).

(Part A) What forwarding paths, if any, do you need to add? (Please specify the source pipeline register (i.e. ID/M or M/EX) and destination units (i.e. ALU, Register File or Mem) when you define a path).

(Part B) Are there any interlocks you cannot resolve? If so, what are they?

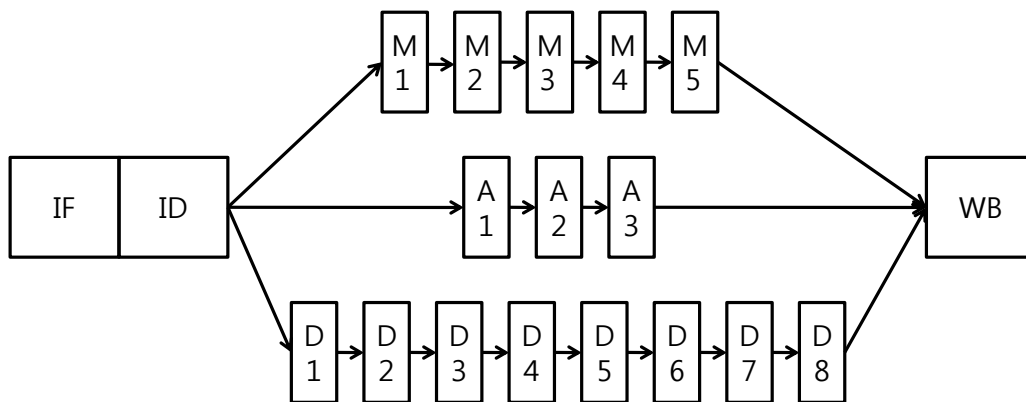
(Part C) Are there any register Write-After-Write hazards that you cannot resolve through forwarding which the new addressing mode introduces? If no, why? If yes, what sequence of instructions might trigger it?

(Part D) Is there the possibility of a concurrent-write stall? If no, why? If yes, what sequence of instructions might trigger it?

(Part E) Confident that you have considered all possible ramifications of implementing the pre-decrement in the **ID** stage with 1 write port, what would change regarding forwarding, interlocks, WAW hazards and concurrent-write stalls if you chose to write the decrement to the register file in the **WB** stage and consequently use 2 write ports?

3. Exceptional Pipelining

The fundamental rule of exception handling is that the faulting instruction should not be allowed to change the processor into an **unrecoverable state** when an exception occurs. One simple way to ensure this result is as we discussed in the class through *precise exceptions* and *in-order exception* taking. Obviously, once we get past the confines of the simple 5-stage classical pipeline we discussed early in the quarter, in-order exception taking becomes problematic to implement, since instructions can finish at widely divergent times and implementation of strictly precise exceptions, wherein you can simply restart the pipeline from the exception taking instruction onwards (once the exception has cleared of course), becomes costly. Nonetheless, there are ways as we discussed to some extent in class to simulate the process of precise exceptions and preserve semantic correctness, efficiency and performance. This question invites you to explore these ideas within the confines of a variable-stage pipelined schematic shown below.



For the variable stage pipeline architecture shown above, out-of-order commit is allowed, and registers can be updated before the preceding instructions are committed. Recall that each floating point instruction undergoes the IF stage, ID stage, the variable length EX stages, and the WB stage (we have merged the MEM and WB stages in this example in order to simplify things). In this process, two types of hazards, *RAW* and *WAW* hazards, are checked at the ID stage. As you remember from the discussion in class *WAW* hazards are not allowed to proceed, stalling the pipeline at that point. Since all reads happen in the ID stage and all writes in the WB stage, no *WAR* hazards are possible in the standard execution stream (unless they are introduced by exceptions!).

(Part A) To ensure precise exceptions, the easiest way would be to stall the issuing instruction until all the instructions in the pipeline finish (or at least get past their last exception taking point). This scheme would create a huge number of stalls in the pipeline (and essentially turn it into a non-pipelined execution model). A new idea of a relaxed issue is being proposed, which allows an instruction to be issued *only if the WB stage of the instruction is guaranteed to appear later than any other active instructions' WB*. This guarantees that the issuing instruction cannot update a register for any exception raised by preceding instructions. Exceptions are also taken in-order, matching the *precise* exception semantics.

(a) Please suggest a piece of hardware for keeping track of the necessary information for doing this checking. Your answer should try to minimize hardware while additionally minimizing checking time to ascertain the possible validity of the condition outlined above. You probably should try to identify and describe a strategy for what is being requested first and then follow this up by mapping it to frugal hardware that can be easily checked.

(b) For the following four instruction sequences, give brief explanations as to at which cycles the third instruction of each sequence can be issued (refer to the schematic given in the previous picture for the number of cycles per operations). Which sequence(s) benefit by the proposed scheme? (*Watch out for data dependencies and the stalls they may introduce, particularly with this variable execution-stage pipeline.*)

1.	ADDD	F0, F2, F4
	MULD	F2, F0, F6
	ADDD	F8, F4, F8

2.	ADDD	F0, F0, F4
	ADDD	F2, F0, F4
	DIVD	F6, F0, F8

3.	ADDD	F0, F2, F2
	DIVD	F6, F8, F2
	MULD	F4, F8, F2

4.	DIVD	F0, F2, F2
	ADDD	F6, F8, F2
	MULD	F8, F4, F2

(Part B) You remain unsatisfied with the speedup afforded to you by being able to issue instructions that do their write back later. You want to stall less and do more instructions. Since exceptions occur very rarely, you decide that it is okay if you are allowed to issue more instructions, and if an exception is raised by a previous instruction, the instructions that were issued after the exception-taking instruction is issued (and may have committed before the exception is taken) are re-executed after the exception is handled. (*You note that, since this is for a production system, you aren't concerned with whether the register values when an exception is handled are the same values they would be if instructions were finished in-order.*)

However, you realize that being able to re-execute these 'early issue' instructions means that you cannot simply allow any instruction to be issued early; there are certain constraints.

(a) If you have an instruction that may still generate an exception, when can you issue subsequent instructions? What are the constraints on which instructions you can choose to issue?

(b) For the following four instruction sequences, give brief explanations as to at which cycles the third instruction of each sequence can be issued (refer to the schematic given in the previous picture for the number of cycles per operations). Which sequence(s) benefit from the new scheme?

1.

ADDD	F0, F2, F4
MULD	F2, F0, F6
ADDD	F8, F4, F8

2.

ADDD	F0, F0, F4
ADDD	F2, F0, F4
DIVD	F6, F0, F8

3.

ADDD	F0, F2, F2
DIVD	F6, F8, F2
MULD	F4, F8, F2

4.

DIVD	F0, F2, F2
ADDD	F6, F8, F2
MULD	F8, F4, F2

(Part C) The longest duration along the standard execution path is 11 cycles, so an architect proposes adding many more than 11 *hidden registers*. In a non-exceptional case, instructions that need R1 first check if the value of R1 is available in a hidden register, and use that if available; if not, they use the value in R1 in the register file. When instructions write back, if all instructions before them have finished writing back, they write to the register file. Otherwise, they write to a new hidden register, and label it R1; this value remains in the hidden register (and instructions that need R1 in the meantime read from it) until all previous instructions have written back, at which point the labeled registers in the hidden register file are restored.

(a) If an instruction raises an exception, there are 3 conditions to consider.

- instructions before it may still be executing
- some instructions in the future may have already finished writeback
- some instructions after the exceptional instruction may not have completed yet, but are currently executing.

When the exception occurs, with this new *hidden register* scheme, what is the optimal thing to do with instructions in all 3 scenarios? When you take the exception, what should you do with the *hidden register* file? What is the procedure for when you return from exception handling?

(b) You notice that with just the specification above, there is no guarantee that a long instruction that raises an exception will have its exception raised before a short instruction that comes after it. This means that the precise exception semantics are not met yet because exceptions are not necessarily taken in-order. However, the architect who proposed the hidden register method says that in-order exceptions are also possible to implement in this scheme. How would this work?