University of California, Berkeley
College of Engineering
Computer Science Division — EECS

Fall 1999 John Kubiatowicz

Midterm II
SOLUTIONS

November 17, 1999
CS152 Computer Architecture and Engineering

Your Name:	
SID Number:	
Discussion Section:	

Problem	**Possible**	**Score**
1	25	
2	25	
3	25	
4	25	
Total		

[This page left for π]

3.141592653589793238462643383279502884197169399375105820974944

Problem 1: Memory Hierarchy

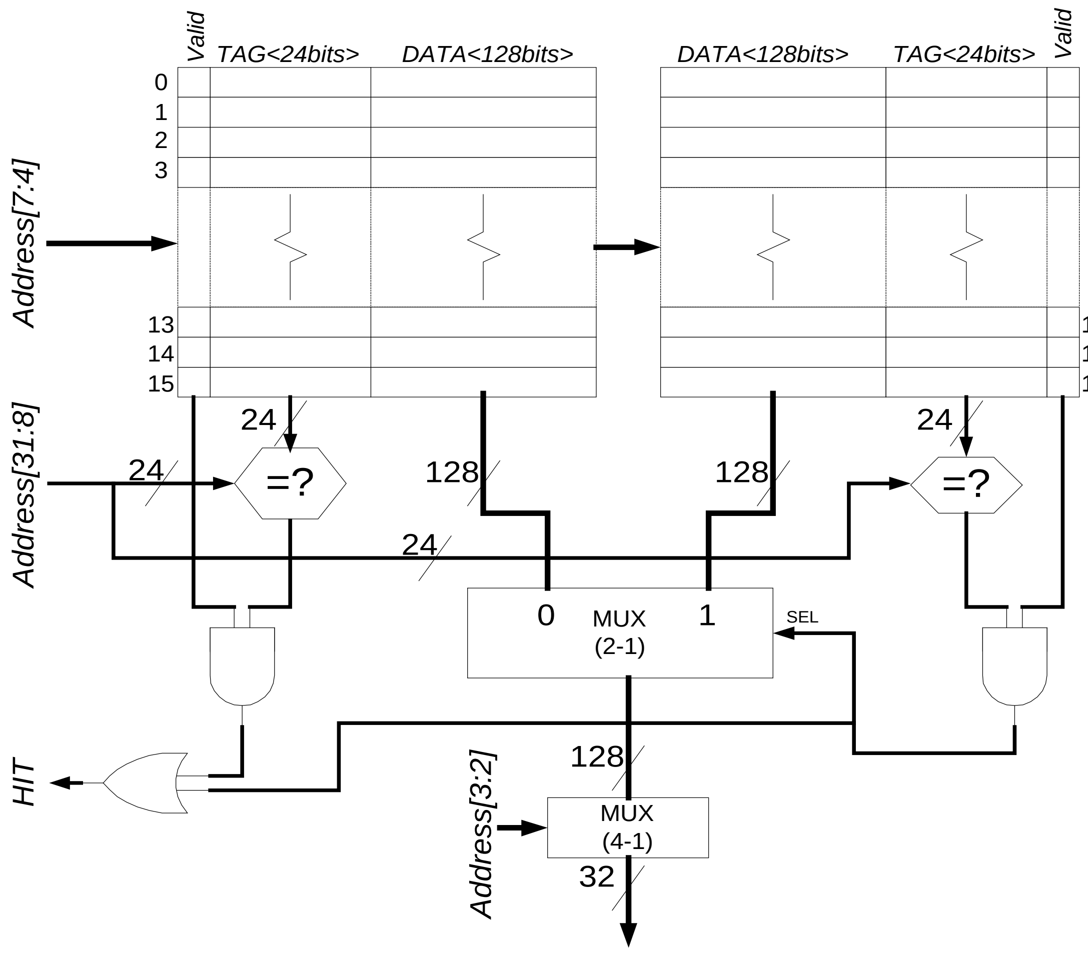
Problem 1a: Assume that we have a 32-bit processor (with 32-bit words) and that this processor is byte-addressed (i.e. addresses specify bytes). Suppose that it has a 512-byte cache that is two-way set-associative, has 4-word cache lines, and uses LRU replacement. Split the 32-bit address into “tag”, “index”, and “cache-line offset” pieces. Which address bits comprise each piece?

tag:	*bits 31-8*
index:	*bits 7-4*
cache-line offset:	*bits 3-0*

Problem 1b: How many sets does this cache have? Explain.

4 bits in the index ⇒ 16 sets

Problem 1c: Draw a block diagram for this cache. Show a 32-bit address coming into the diagram and a 32-bit data result and “Hit” signal coming out. Include, all of the comparators in the system and any muxes as well. Include the data storage memories (indexed by the “Index”), the tag matching logic, and any muxes. You can indicate RAM with a simple block, but make sure to label address widths and data widths. Make sure to label the function of various blocks and the width of any buses.



[This page left for scratch]

Problem 1d: Below is a series of memory read references set to the cache from part (a). Assume that the cache is initially empty and classify each memory references as a hit or a miss. Identify each miss as either **compulsory, conflict, or capacity.** One example is shown. *Hint: start by splitting the address into components. Show your work.*

Address	Hit/Miss?	Miss Type?
`0x300`	Miss	Compulsory
`0x1BC`	*Miss*	*Compulsory*
`0x206`	*Miss*	*Compulsory*
`0x109`	*Miss*	*Compulsory*
`0x308`	*Miss*	*Conflict*
`0x1A1`	*Miss*	*Compulsory*
`0x1B1`	*Hit*	—
`0x2AE`	*Miss*	*Compulsory*
`0x3B2`	*Miss*	*Compulsory*
`0x10C`	*Hit*	—
`0x205`	*Miss*	*Conflict*
`0x301`	*Miss*	*Conflict*
`0x3AE`	*Miss*	*Compulsory*
`0x1A8`	*Miss*	*Conflict*
`0x3A1`	*Hit*	—
`0x1BA`	*Hit*	—

Problem 1e: Calculate the miss rate and hit rate.

$$\textit{Hit Rate} = \frac{4}{16} = 0.25$$

$$\textit{Miss Rate} = 1 - \textit{Hit Rate} = \frac{12}{16} = 0.75$$

[This page intentionally left blank]

Problem 1f: You have a 500 MHz processor with 2-levels of cache, 1 level of DRAM, and a DISK for virtual memory. Assume that it has a Harvard architecture (separate instruction and data cache at level 1). Assume that the memory system has the following parameters:

Component	Hit Time	Miss Rate	Block Size
First-Level Cache	1 cycle	4% Data 1% Instructions	64 bytes
Second-Level Cache	20 cycles + 1 cycle/64bits	2%	128 bytes
DRAM	100ns+ 25ns/8 bytes	1%	16K bytes
DISK	50ms + 20ns/byte	0%	16K bytes

Finally, assume that there is a TLB that misses 0.1% of the time on data (doesn’t miss on instructions) and which has a fill penalty of 40 cycles. What is the average memory access time (AMAT) for Instructions? For Data (assume all reads)?

$AMAT_{DISK} = (5\times10^{7} ns) + (16384\times20) = 50327680ns/2ns\text{-per-cycle} = 25163840$ *cycles*

$AMAT_{DRAM} = (100ns+25ns\times16)+ 0.01\times AMAT_{DISK} = (500ns+503276.8ns)/2ns= 251888.4$ *cycles*

$AMAT_{L2} = (20 + 8) + 0.02\times AMAT_{DRAM} = 5065.77$*cycles*

$AMAT_{INST} = (1+0.01\times AMAT_{L2})=51.66$ *cycles*

$AMAT_{DATA} = (1+0.04\times AMAT_{L2}+0.001\times40) = 203.67$ *cycles*

Why are these so high? Because our miss-rate for the disk (1%) is so high. This computer would technically be “thrashing”, i.e. spending all of its time moving pages to and from the disk.

Problem 1g: Suppose that we measure the following instruction mix for benchmark “X”:

Loads: 20%, Stores: 15%, Integer: 30%, Floating-Point: 15% Branches: 20%

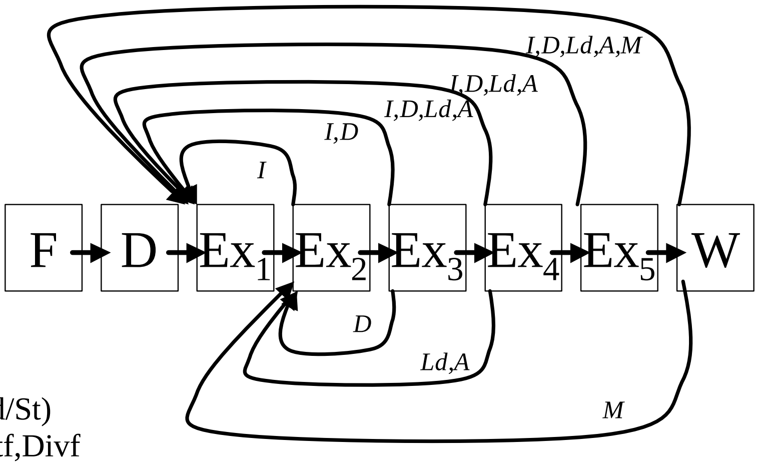
Assume that we have a single-issue processor with a minimum CPI of 1.0. Assume that we have a branch predictor that is correct 95% of the time, and that an incorrect prediction costs 3 cycles. Finally, assume that data hazards cause an average penalty of 0.7 cycles for floating point operations. Integer operations run at maximum throughput. What is the average CPI of Benchmark X, including memory misses (from part g)?

$$\begin{aligned} CPI &= CPI_{NORMAL}+CPI_{Compute\text{-}stalls}+CPI_{Memory\text{-}stalls} \\ &= 1 + (0.3\times0 + 0.15\times0.7 + 0.2\times0.05\times0.3) +(AMAT_{INST}+0.35\times AMAT_{DATA}) = \\ &= 1 + .135 + (51.66 + 0.35\times203.67) = 124.08 \text{ cycles/instruction} \end{aligned}$$

Problem #2: Superpipelining

Suppose that we have single-issue, *in-order* pipeline with one fetch stage, one decode stage, multiple execution stages (which include memory access) and a singe write-back stage. Assume that it has the following execution latencies (i.e. the number of stages that it takes to compute a value): **multf** (5 cycles), **addf** (3 cycles), **divf** (2 cycles), integer ops (1 cycle). Assume full bypassing and two cycles to perform memory accesses, i.e. loads and stores take a total of 3 cycles to execute (including address computation). Finally, branch conditions are computed by the first execution stage (integer execution unit).

Problem 2a:
Assume that this pipeline consists of a *single linear sequence* of stages in which later stages serve as no-ops for shorter operations. Draw each stage of the pipeline as a box (no internal details) and name each of the stages. Describe what is computed in each stage and show *all* of the bypass paths (as arrows between stages). *Your goal is to design a pipeline which never stalls unless a value is not ready.* Label each of these arrows with the types of instructions that will forward their results along these paths (i.e. use “M” for **multf,** “D” for **divf,** “A” for **addf,** “I” for integer operations). [*Hint:* be careful to optimize for information feeding into store instructions!]



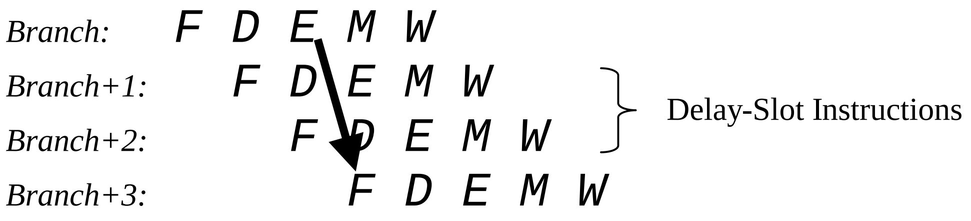
Stage:	
F	Fetch next instruction
D	Decode stage
EX_1	Integer Ops Address compute (for Ld/St) First stage of: Addf,Multf,Divf
EX_2	First stage of: Ld/St. Last Stage of: Divf, Second stage of: Addf, Mulff
EX_3	Last stage of: Ld/St, Addf Third stage of: Multf
EX_4	Fourth stage of: Multf
EX_5	Last stage of: Multf
W	Writeback stage

Problem 2b:
How many extra instructions are required between each of these instruction combinations to avoid stalls (i.e. assume that the second instruction uses a value from the first). Be careful!

Between a **divf** and an **store:**	*0*	Between a **multf** and an **addf:**	*4*
Between a **load** and a **multf:**	*2*	Between an **addf** and a **divf:**	*2*
Between two integer instructions:	*0*	Between an integer op and a **store:**	*0*

Problem 2c:
How many branch delay slots does this machine have? Explain.

2 delay slots. By the time the branch decision is made in the execute stage, there are already two instructions in the pipeline. Or, you could say that the two delay slots are necessary to avoid a “backward-in-time” arrow:



Probem 2d:
Could branch prediction increase the performance of this pipeline? Why or why not?

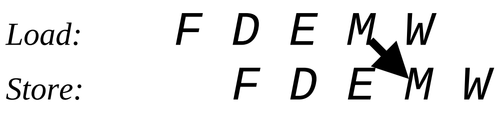
Yes. If we expose two delay slots to the programmer or compiler, they have to be filled either with useful instructions of NOPs. Unfortunately, it is hard enough to find one useful instruction to put in a delay slot, much less two. With branch prediction, we change the specification of the instruction set so that there are no branch delay slots from the standpoint of the programmer/compiler. Then, we let the prediction hardware figure out how to make use of those instruction slots. If branch prediction is good, then we can do better than the compiler at “filling” the hardware slots.

Problem 2e:
In the 5-stage pipeline that we discussed in class, a load into a register followed by an immediate store of that register to memory would not require any stalls, i.e. the following sequence could run *without* stalls:

```
lw   r4, 0(r2)
sw   r4, 0(r3)
```

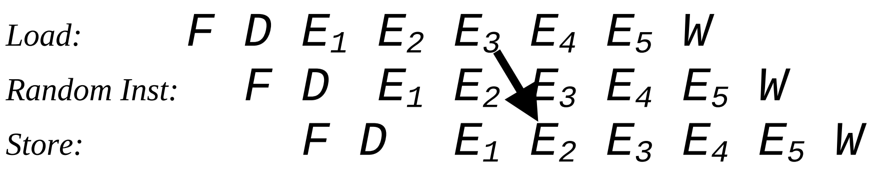
Explain why this was true for the 5-stage pipeline.

Both loads and stores happen in the memory stage, so the forwarding of information can be forward in time:



Problem 2f:
Is this still true for the superpipelined processor? Explain.

No. Because there are now two memory stages, hence we need to insert one extra instruction so that we can forward from the end of the load (in stage E_3) to the data portion of the store (E_2):


Load: F D E1 E2 E3 E4 E5 W
Random Inst: F D E1 E2 E3 E4 E5 W
Store: F D E1 E2 E3 E4 E5 W

Problem #3: Fixing the loops

For this problem, assume that we have a superpipelined architecture like that in problem (2) with the following use latencies (these are not the right answers for problem #2b!):

Between a **multf** and an **addf:**	**3 insts**	Between a **load** and a **multf/divf:**	**2 insts**
Between an **addf** and a **divf:**	**1 insts**	Between a **divf** and a **store:**	**7 insts**
Between an int op and a **store:**	**0 insts**	Number of branch delay slots:	**1 insts**

Consider the following loop which performs a restricted rotation and projection operation. The array based at register **r10** contains pairs of double-precision (64-bit) values which represent x,z coordinates. The array based at register **r20** receives a projected coordinate along the observer's horizontal direction:

```
loop:   ldf     $F20, 0($r10)
        multf   $F6,  $F20, $F1
        addf    $F12, $F6, $F2
        ldf     $F10, 8($r10)
        divf    $F13, $F12, $F10
        stf     0($r20), $F13
        addi    $r10, $r10,#16
        addi    $r20, $r20, #8
        subi    $r1, $r1, #1
        bne     $r1, $zero, loop
          nop
```

Problem 3a: How many cycles does this loop take per iteration? Indicate stalls in the above code by labeling each of them with a number of cycles of stall:

11 instructions + 14 stalls = 25 cycles/iteration

Problem 3b: Reschedule this code to run with as few cycles per iteration as possible. Do not unroll it or software pipeline it. How many cycles do you get per iteration of the loop now?

```
loop:  ldf   $F20, 0($r10)
       ldf   $F10, 8($r10)
       multf $F6, $F20, $F1
       addf  $F12, $F6, $F2
       addi  $r10, $r10, #16
       divf  $F13, $F12, $F10
       addi  $r20, $r20, #8
       subi  $r1, $r1, #1
       bne   $r1, $zero, loop
         stf -8($r20), $F13
```

There are many ways to rearrange/pipeline this code; one is shown above. Note that all of the optimal ones will result in 8 stalls. So:

10 instructions + 8 stalls = 18 cycles/iteration.

Problem 3c: Unroll the loop once and schedule it to run with as few cycles as possible per iteration of the original loop. How many cycles do you get per iteration now?

Total = (16+4)/2 =10 cycles/iter

```
loop:  ldf   $F20, 0($r10)
       ldf   $F10, 8($r10)
       ldf   $F22, 16($r10)
       multf $F6, $F20, $F1
       ldf   $F12, 24($r10)
       multf $F7, $F22, $F1
       addf  $F12, $F6, $F2
       divf  $F13, $F12, $F10
       addf  $F16, $F7, $F2
       divf  $F14, $F16, $F12
       addi  $r10, $r10, #32
       addi  $r20, $r20, #16
       subi  $r1, $r1, #2
       stf   -16($r20), $F13
       bne   $r1, $zero, loop
         stf -8($r20), $F14
```

Problem 3d: Your loop in (3c) will not run without stalls. Without going to the trouble to unroll further, what is the minimum number of times that you would have to unroll this loop to avoid stalls? Explain. How many cycles would you get per iteration then?

If we have ≥ 4 iterations, we can groups all the loads together, followed by the multiplies, adds, divides, integers, and stores. There will be no stalls except for the stores. Unfortunately, 4 iterations is not quite enough to avoid all stalls, since the first store will stall for one cycle: `..DDDDIIISSSBS` *has 6 instructions between loads and stores. Thus, we need 5 iterations.* ***Cycles: (6×5+4)/5=6.8cycles/iteration.***

Problem 3e: Software pipeline the original loop to avoid stalls. Overlap 5 different iterations. What is the average number of cycles per iteration? Your code should have no more than one copy of the original instructions. Ignore startup and exit code.

```
loop:  stf     0($r20), $F13      } Phase – 5 [1 instruction]
       divf    $F13, $F12, $F10   } Phase – 4 [1 instruction]
       ldf     $F10, 40($r10)     } Phase – 3 [2 instructions]
       addf    $F12, $F6, $F2     }
       multf   $F6, $F20, $F1     } Phase – 2 [1 instruction]
       ldf     $F20, 64($r10)     }
       addi    $r10, $r10, #16    }
       subi    $r1, $r1, #1       } Phase – 1 [5 instructions]
       bne     $r1, $zero, loop   }
         addi  $r20, $r20, #8     }
```

This software pipelining problem had some subtleties with respect to the load placements that some of you got, but which we didn’t enforce. In particular, the division into phases as shown above. This division is enforced by data flow: phase-5 uses information generated in phase-4, phase-4 from phase-3, etc. Note the careful generation of offsets as well. There are 2 iterations between phase-1 and phase-3. In those two iterations, $r10 will have gained 32. Thus, to correct for this, 64($r10) in phase-1 becomes 32($r10) in phase 3. Since the phase-3 load is 8 bytes further than that, the phase-3 offset is (32+8)($r10) = 40($10).

This runs without stalls. So, the ***cycles/iteration = 10****.*

Extra Credit (Problem 3X):
Assume that you have a Tomasulo architecture with functional units of the same execution latency (number of cycles) as our deeply pipelined processor (*be careful to adjust use latencies to get number of execution cycles!).* Assume that it issues one instruction per cycle and has an unpipelined divider with a small number of reservation stations. Suppose the other functional units are duplicated with many reservation stations and that there are many CDBs. . What is the minimum number of divide reservation stations to achieve one instruction per cycle with the optimized code of (3b)? Show your work. *[hint: assume that the maximum issue rate is sustained and look at the scheduling of a single iteration]*

Answer: 2. The best way to understand this is to actually look at the timing of issue slots. First, we take the use latencies from the beginning of this problem to extract the execution latencies (number of execution stages) for the different operations:

Load: 3 cycles, Add: 2 cycles, Multiply: 4 cycles, Divide: 9 cycles (careful here!)

Next, we show the timing of two+ iterations. We assume that the write back of one instruction and the scheduling of a dependent instruction can occur in the same cycle (e.g. first ldf write back in cycle 5 means that dependent multf can start executing in cycle 6):

Name	Issue	Start Execution	End Execution	Write Back
ldf	1	2	4	5
ldf	2	3	5	6
multf	3	6	9	10
addf	4	11	12	13
addi	5	6	6	7
addi	6	7	7	8
subi	7	8	8	9
divf	8	14	22	23
bne	9	10	10	11
stf	10	24	26	27
ldf	11	12	14	15
ldf	12	13	15	16
multf	13	16	19	20
addf	14	21	23	24
addi	15	16	16	17
addi	16	17	17	18
subi	17	18	18	19
divf	18	25	33	34
bne	19	20	20	21
stf	20	35	37	38
ldf	21	22	23	24
ldf	22	23	24	25
multf	23	25	28	29
addf	24	30	32	33
addi	25	26	26	27
addi	26	27	27	28

Looking at this table, we see that we only need 2 reservation stations: one that is running, and one waiting (the first divide is done in cycle 23, second divide overlaps, but third is issued until cycle 28).

Problem #4: Short Answers

Problem 4a: Give a simple definition of precise interrupts/exceptions:

A precise interrupt/exception is one which generates a single instruction in the instruction stream for which all preceding instructions have completed and committed their results and for which the designated instruction and all following instructions have not committed any results (i.e. have not modified machine state).

Problem 4b: Explain how the presence of delayed branches complicates the description of a precise exception point (*Hint: what if there is a divide instruction in a delay slot that gets a divide by zero exception)?*

To describe the precise exception point, one needs more than one PC. For instance, if there is a single delay slot, you need two PCs to describe the precise exception point (these are often called the PC and nPC – for next PC). For a machine with **n** *delay slots, you need* **n+1** *PCs.*

The reason that you need multiple instructions is to properly restart the pipeline. Consider the case in which the precise exception point is at the delay slot instruction. In that case, the next instruction to execute on return is clearly the delay slot instruction. However, the following instruction might either be PC+4 or the target of the branch. Hence the need for the nPC.

Problem 4c: Explain the relationship between support for precise exceptions and support for branch prediction. What hardware structure supports both of these mechanisms in a modern out-of-order pipeline?

With out-of-order execution, precise exceptions require rolling back operations that have already occurred after the exception point. Branch prediction requires the same support (to rollback to the branch). The simplest structure to support rollback is the reorder buffer.

Problem 4d: Explain how pipelining can save power (and energy) for multimedia (streaming) applications:

Multimedia applications consists of large numbers of independent operations. Hence, if one can successfully pipeline the execution units, one can keep the overall clock rate constant and get the "same" throughput (i.e. there are no stalls due to data hazards). After pipelining, there is less logic between registers; as a result, we can lower the voltage without lowering the clock rate. The net result is a power savings without a reduction in throughput.

Problem 4e: A PalmPilot is a portable computing device that holds calendars and addresses. It has a micro-power mode that stops the clock and shuts down power to the processor when it is idle. Suppose that it also recognized when the battery was getting low and ran the clock at lower than normal speed during busy periods. Would this extend battery life? Why or why not?

No. It would not extend battery life. The reason is that slowing the clock down increases the amount of time that the PalmPilot must be "on" in order to complete a given task. The way to look at this is that the total number of transitions for a given operation (say looking up a name in the address book) is constant. Total consumed energy is dependent on number of transitions and voltage, neither of which have changed. If we changed the voltage as well, the answer would be different (but we didn't specify this – in fact the PalmPilot versions 1-5 don't allow voltage variation). Note that laptops change their clock when idle only because the software which runs on them is not particularly intelligent (i.e. it consumes energy during idle loops!).

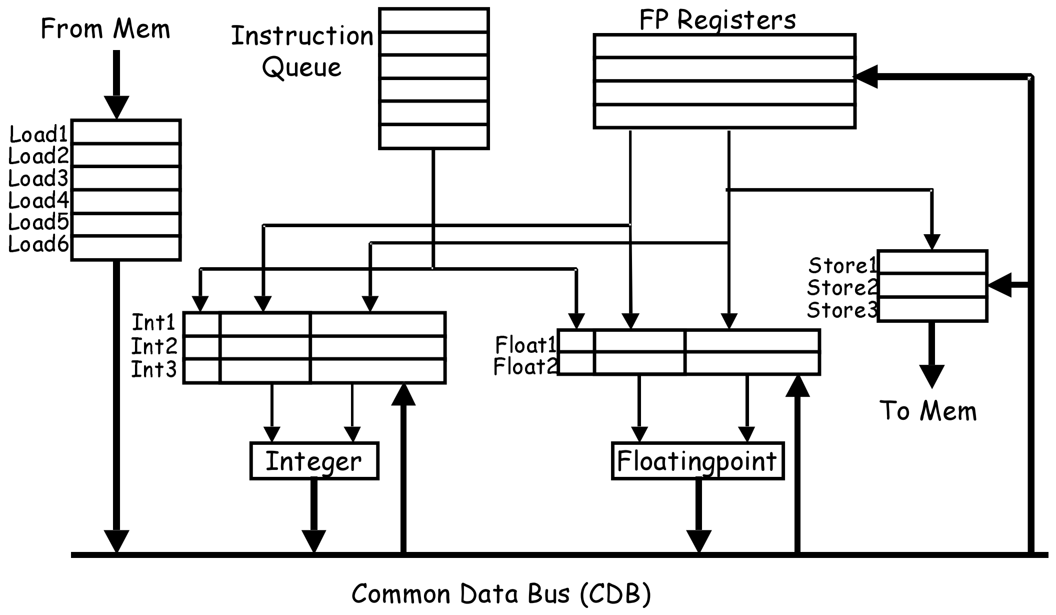


Figure 1: A basic Tomasulo architecture

Problem 4f: The Tomasulo architecture (shown above) replaces a normal 5-stage pipeline with 4 stages: *Fetch, Issue, Execute, and Writeback.* One of its strengths is that it is able to *Execute* instructions in a different order than the programmer originally specified. The simplest version of this architecture also performs Writeback out-of-order as well. However, the Fetch and Issue stages of the Tomasulo architecture are always handled in program order. Why?

We have to Fetch and Issue in order so that we can analyze the dataflow between instructions and maintain the semantics of the program. Alternatively, you could think of the fact that register renaming requires handling each instruction in order so that data will flow properly between instructions which produce values and instructions that use these values.

Problem 4g: Pipelined architectures have three different types of data hazards with respect to registers. Name and define them. For *each* type, give a short code sequence that illustrates the hazard and describe how a Tomasulo architecture removes this hazard.

RAW: *A "Read After Write" hazard occurs which an instruction produces a value which is used by a later instruction. Tomasulo uses the CDB to forward data to correct this type of hazard. Example:*

```
add   $r1, $r2, $r3
sub   $r5, $r1, $r6
```

WAR: *A "Write After Read" hazard occurs when a later instruction writes a register that needs to be read by an earlier instruction. If data flows from the later to earlier instruction, we will get incorrect operation. Tomasulo prevents this with register renaming. Example:*

```
sub   $r5, $r1, $r6
add   $r1, $r2, $r3
```

WAW: *A "Write After Write" hazard occurs when two instructions write a register. If these instructions write out of order, then the register will have an incorrect value in it. Tomasulo prevents this with register renaming. Example:*

```
add   $r1, $r2, $r3
sub   $r1, $r4, $r5
```

Problem 4h: What is register renaming, why is it desirable, and how is it accomplished in the Tomasulo architecture?

During register renaming, the register names in instructions are changed (“renamed”) to either a value or a pointer to a physical register slot in the machine. This is desirable because it eliminates all WAR and WAW hazards.

It is accomplished in the Tomasulo architecture by maintaining an indirection table for every register. When issuing an instruction, we:

1. *Look up each operand in this table to get either an actual value for the register (if ready), or a pointer to the reservation slot which will produce the value some time in the future.*
2. *Update the register indirection slot for the destination register of the new instruction to point at the reservation station where we are placing the new instruction.*

Problem 4i: Why does a Tomasulo architecture need branch prediction?

Because the Tomasulo architecture gets its performance by executing instructions out of order. Out-of-order execution depends on **issuing** *enough instructions that we can find ones that are not blocked awaiting results. However, in order to issue enough instructions, we need to scan forward in the instruction stream. Since there is a branch every 5 or 6 instructions (typically), we need to predict the direct of branches so that we can issue past them even before we know their outcomes.*

As the simplest example, consider the fact that, if we want to overlap multiple iterations of a loop, we need to issue bast several instances of the loop branch. If that branch instruction depends on the results of the loop (not uncommon), then we will not be able to issue past the branch unless we make a guess of the direction that the branch is going to take.