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

Spring 1999 John Kubiatowicz

Midterm II

April 21, 1999
CS152 Computer Architecture and Engineering

Your Name:	Solution
SID Number:	
Discussion Section:	

Problem	**Possible**	**Score**
1	20	
2	30	
3	25	
4	25	
Total		

[This page left for π]

3.141592653589793238462643383279502884197169399375105820974944

Problem 1: Memory Hierarchy

Problem 1a:
Below is a series of memory read references set to a cache. The cache holds 128 bytes total. It has 2-word blocks (i.e. 64bits), is 2-way set associative, and uses a least-recently-used replacement policy. Assume that the cache is initially empty.

Classify each memory references as a hit or a miss. Identify each cache miss as either **compulsory, conflict, or capacity.** One example is shown below. Feel free to use space in the margin as scratch.

Address	**Hit/Miss?**	**Miss Type?**
0x7	Miss	Compulsory
0x4D		
0x2A		
0x79		
0xAB		
0xCE		
0x2E		
0x4B		
0x6D		
0x8A		
0xAF		
0x29		
0xC8		
0xCE		
0x6A		

Problem 1b:

Problem 1c:
Suppose you have a 32-bit processor, with a virtual-memory page-size of 16K. The data cache is 32K in size with 32-byte cache blocks. Finally, your TLB has 4 entries. Assume that you wish to do TLB lookups in parallel with cache lookups.

Draw a block diagram of the data cache and TLB organization, showing a virtual address as input and both a physical address and data as output. Include *cache hit* and *TLB hit* output signals. Include as much information about the internals of the TLB and cache organization as possible. Include, among other things, all of the comparators in the system and any muxes as well. You can indicate RAM as with a simple block, but make sure to label address widths and data widths. Make sure to use abstraction in your diagram so that we can understand it. Label the function of various blocks and the width of any buses.

Now, assume the following instruction mix:

Loads: 20%, Stores: 15%, Integer: 29%, Floating-Point: 16% Branches: 20%

Assume that you have a memory-hierarchy consisting of 2-levels of cache, 1 level of DRAM, and a DISK. The following parameters are appropriate. Assume a 200MHz processor:

Component	Hit Time	Miss Rate	Block Size
First-Level Cache	1 cycle	5% Data 1% Instructions	32 bytes
Second-Level Cache	10 cycles + 1 cycle/64bits	3%	128 bytes
DRAM	100ns+ 25ns/8 bytes	1%	16K bytes
DISK	50ms + 20ns/byte	0%	16K bytes

In addition, assume that there is a TLB which misses 0.1% of the time on data (doesn't miss on instructions) and which has a fill penalty of 50 cycles.

Problem 1d:
What is the average memory access time for Instructions? For Data?

Problem 2: Multicycle Polynomial Multiply

The VAX architecture from Digital Equipment Corporation was well known for its complex instruction set. One instruction that was often cited was the *polynomial multiply instruction.* This instruction took two polynomials and multiplied them together to get a third:

$$\left(X^2 + 3X + 2\right) \times \left(2X^5 + X^4 + 4\right) \Rightarrow \left(2X^7 + 7X^6 + 7X^5 + 2X^4 + 4X^2 + 12X + 8\right)$$

Let's represent polynomials as pointers to arrays of numbers in memory. The first number will be the "degree" of the polynomial (highest power of X). The following (degree+1) values will be the coefficients of the powers of X, starting with the lowest power. For example:

$$\left(2X^5 + X^4 + 4\right) = \left(4X^0 + 0X^1 + 0X^2 + 0X^3 + 1X^4 + 2X^5\right) \Rightarrow [5 \quad 4 \quad 0 \quad 0 \quad 0 \quad 1 \quad 2]$$

The first number (5) is the degree. The next 6 numbers are coefficients. Thus, a 5th degree polynomial is represented by 7 numbers in memory.

With that representation, a polynomial multiplication can be described with the following straightforward pseudo-code, where poly1 – poly3 are pointers to 32-bit words in memory:

```
polynomial_mult(poly1,poly2) ⇒ poly3
{
   degree1 = poly1[0];          /* Degree of poly1 */
   degree2 = poly2[0];          /* Degree of poly2 */
   degree3 = degree1 + degree2; /* Compute degree of poly3 */
   poly3[0]=degree3;            /* Save into result */

   for (resultdeg = 0; resultdeg ≤ degree3; resultdeg++) {
      indexdeg1 = MIN(resultdeg,degree1);
      indexdeg2 = resultdeg – indexdeg1;

      /* (indexdeg1+indexdeg2)=resultdeg throughout loop */
      accum = 0;
      while ((indexdeg1 ≥ 0) and (indexdeg2 ≤ degree2)) {
         accum = accum + poly1[indexdeg1+1] × poly2[indexdeg2+1];
         indexdeg1--;   /* Decrement */
         indexdeg2++;   /* Increment */
      }
      poly3[resultdeg+1] = accum;/* Place final coeff into poly3 */
   }
}
```

Note: In reading this code, assume that the polynomial coefficients are 32-bit values. This means that you must scale indexes by 4 before using them, i.e. poly1[6] is at address poly+6x4!

The way that this algorithm works is that the "for" loop generates each coefficient of the result, starting with the lowest. The inner "while" loop adds up all terms in the product that are of the same degree. To see this, consider the $7X^5$ term of the result in the example above:

$$\left(1X^2 \times 0X^3\right) + \left(3X \times X^4\right) + \left(2X^0 \times 2X^5\right) = 7X^5 \quad \textit{or} \quad (1\times 0) + (3\times 1) + (2\times 2) = 7$$

When computing this term, **`resultdeg`**=5**.** Before the "while" loop, **`indexdeg1`** starts at MIN(5,2)=2 and **`indexdeg2`** starts at 5-2=3. Throughout the "while" loop, the sum (**`indexdeg1+indexdeg2) =`** `5.`

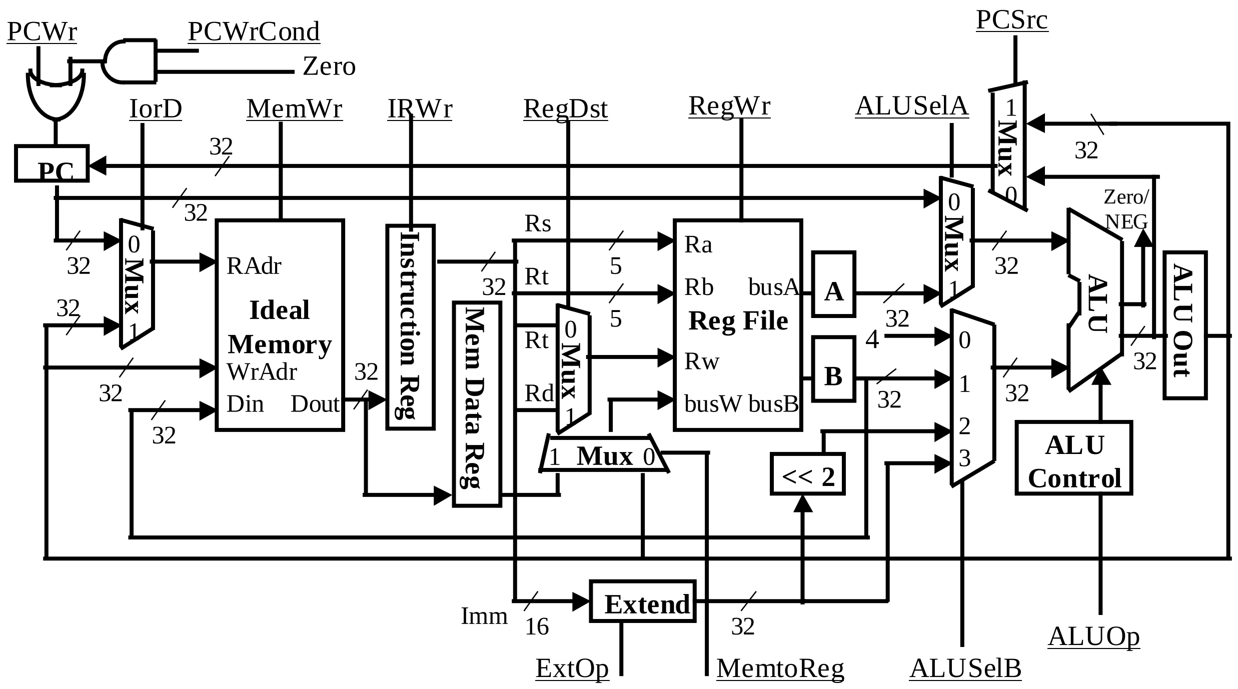

Figure 1: The Multicycle Data Path

Problem 2a:
Let the ALU support multiplication. You cannot change or duplicate the memory component, or change or duplicate the ALU component, but *are* allowed to add muxes, registers, equality comparitors, and random logic. Estimate the minimum number of cycles (on average) that you can hope to achieve in the inner "while" loop. Justify your answer by discussing the operations that must be performed on each iteration and showing a timing diagram for three iterations of the inner loop. Don't try to change the datapath yet. You will do that in (2b)

(Hints: You can recognize the last loop of the "while" condition by checking for "=0" and "=degree2", since this loop will always be executed at least once. Also, make sure you understand where each of the computations come from – there may be ways of moving them out of the inner loop!)

Problem 2b:
Assume that our new instruction is specified as follows:

polymult $r3, $r1, $r2

Where this is an R-TYPE instruction. Here, registers r1 and r2 hold pointers to the source polynomials, and r3 holds a pointer to memory for the destination polynomial. Let’s assume that there is enough memory at the location specified by r3 to hold any result. Assume also that the registers should not be changed during execution.

Change the data path to support polynomial multiply with the same rate in the inner loop as specified in (2a)? As before, you cannot change or duplicate the memory component, or change or duplicate the ALU component, but *are* allowed to add muxes, registers, equality comparitors, and random logic. Be explicit and try to be minimize the hardware/minimize the total number of cycles for the complete operation as much as possible. Show all new control points. (*Note: the computation of initial values of indexdeg1 and indexdeg2 can be done with one ALU operation and some muxes!)*

Table 1: Symbolic Definitions for Microcode

Field Name	*Values for Field*	*Function of Field with Specific Value*
ALU	**Add**	**ALU adds**
	Subt.	**ALU subtracts**
	Func code	**ALU does function code**
	Or	**ALU does logical OR**
SRC1	**PC**	**1st ALU input = PC**
	rs	**1st ALU input = Reg[rs]**
SRC2	**4**	**2nd ALU input = 4**
	Extend	**2nd ALU input = sign ext. IR[15-0]**
	Extend0	**2nd ALU input = zero ext. IR[15-0]**
	Extshft	**2nd ALU input = sign ex., sl IR[15-0]**
	rt	**2nd ALU input = Reg[rt]**
destination	**rd ALU**	**Reg[rd] = ALUout**
	rt ALU	**Reg[rt] = ALUout**
	rt Mem	**Reg[rt] = Mem**
Memory	**Read PC**	**Read memory using PC**
	Read ALU	**Read memory using ALU output**
	Write ALU	**Write memory using ALU output**
Memory register	**IR**	**IR = Mem**
PC write	**ALU**	**PC = ALU**
	ALUoutCond	**IF ALU Zero then PC = ALUout**
Sequencing	**Seq**	**Go to sequential µinstruction**
	Fetch	**Go to the first microinstruction**
	Dispatch	**Dispatch using ROM.**

Table 2: Microcode for Simple Instructions

Label	*ALU*	*SRC1*	*SRC2*	*Dest.*	*Memory*	*Mem. Reg.*	*PC Write*	*Sequence*
Fetch:	**Add**	**PC**	**4**	—	**Read PC**	**IR**	**ALU**	**SEQ**
	Add	**PC**	**Extshft**	—	—	—	—	**Dispatch**
Rtype:	**Func**	**rs**	**rt**	—	—	—	—	**Seq**
	—	—	—	**rd ALU**	—	—	—	**Fetch**
Ori:	**Or**	**rs**	**Extend0**	—	—	—	—	**Seq**
	—	—	—	**rt ALU**	—	—	—	**Fetch**
Lw:	**Add**	**rs**	**Extend**	—	—	—	—	**Seq**
					Read ALU			**Seq**
	—	—	—	**rt MEM**	—	—	—	**Fetch**
Sw:	**Add**	**rs**	**Extend**	—	—	—	—	**Seq**
	—	—	—	—	**WriteALU**	—	—	**Fetch**
Beq:	**Subt**	**rs**	**rt**	—	—	—	—	**Fetch**

Assume that we are going to microcode this instruction. For your reference, Tables 1 and 2 list the symbolic names that we have given to fields of the microinstructions, as well as the microcoded versions of some of the simple instructions.

Problem 2c:
First, how does the sequencer box have to change in order to support this instruction? Draw a block diagram showing the MicroPC, the logic around it, and the ROM.

Problem 2d:
Next, make changes to Table 1 to reflect your new hardware. Make sure that you are clear about what you are adding/changing.

Problem 2e:
Finally, write microcode for the polynomial multiply instruction. (You are now an official CISC system designer!).

Problem 3: Speeding up the Loops

For the following problem, assume an in-order, MIPS-style pipelined architecture with up to 4 cycles in the EX stage, but *full forwarding for operations that take less than 4 cycles*. Assume the following number of execution cycles are required:

1. Floating-point multiply: **4 cycles**
2. Floating-point addition: **2 cycles**
3. Integer operations: **1 cycle**

Assume as well that there is **one branch delay slot,** that there is no delay between integer operations and dependent branch instructions, and that the load-use latency (or number of load delay slots) is **2 cycles.**

One possible pipeline that might behave this way could appear as follows:

IF	ID | BR	EX$_1$	EX$_2$ | MEM$_1$	EX$_3$ | MEM$_2$	EX$_4$	WR

Now, given this pipeline, the following code computes a dot-product. Assume tha `r1` and `r2` contain addresses of arrays of floating-point numbers, and that `r3` contains the length of the arrays (in elements). Assume that `r4` is initialized to zero. Then, the dot product can be computed as follows:

```
dotprod: lw   $f5, 0($r1)    ; load element from first array
         lw   $f6, 0($r2)    ; load element from second array
         muls $f7, $f5, $f6  ; multiply elements
         adds $f4, $f4, $f7  ; add elements to accumulator in f7
         addi $r1, $r1, 4    ; advance pointers
         addi $r2, $r2, 4
         addi $r3, $r3, -1   ; decrement element count
         bne  $r3, $zero, dotprod  ; loop
         nop                 ; Do nothing (branch delay slot)
```

Problem 3a:
How many cycles on average does each iteration take, without rearranging the code?

Problem 3b:
Rearrange the code so that it gets as few cycles per iteration as possible (don’t unroll the loop). Show the scheduled code. How many cycles per iteration does it get now?

Problem 3c:
Unroll the given loop once, and schedule it to completely avoid stalls. Show your code. How many cycles per iteration does it get now?

Problem 3d:
If you were to unroll the loop 8 times, how many cycles per iteration would this achieve? (*hint: you do not need to actually perform the unrolling, but justify your answer)*

Problem 3e:
Now, assume that you want to design a new processor that is more deeply pipelined, i.e. which has larger latencies for all of the operations. Maximize the latencies of instructions that the loop can tolerate by rewriting the loop with *software pipelining.* Do not unroll the loop (i.e. there will be only 8 instructions). Only show code for the loop; you can ignore any startup or cleanup instructions outside the loop. *Hint: this code will overlap 3 different iterations of the loop.*

Problem 3f:
For the software-pipelined version of the loop, assuming that the loop runs without stalls, what is

- the maximum execution latency for `muls`?
- the maximum execution latency for `adds`?
- the maximum load-use latency (delay slots) for `lw`?
-

Problem 3g:
Assuming that most of the power in your original processor was consumed in the execute stages, is the new processor likely to consume more, the same, or less power than the original? Why?

Problem 4: Hazards and Advanced Pipelining

This problem brings together a number of different elements of pipelining.

Problem 4a:
There are three different types of data hazards, RAW, WAR, and WAW. Define them, giving a short code sequence to illustrate each, and describe how a 5-stage pipeline removes them:

a) RAW:

b) WAR:

c) WAW:

Problem 4b:
What are control hazards? Name and explain two different techniques for getting rid of them.

Problem 4c:
What are precise exceptions and why are they important?

Problem 4d:
Explain how to achieve precise exceptions in a standard 5-stage pipeline. Be explicit.

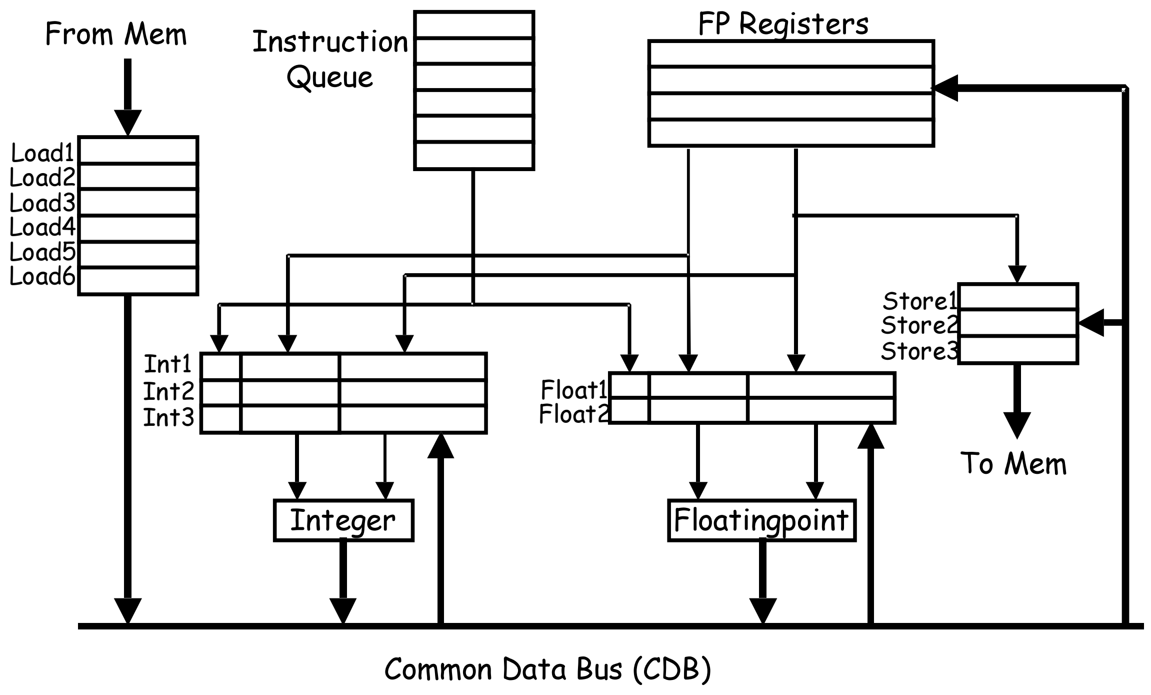

Figure 2: A basic Tomasulo architecture

Problem 4e:
Figure 2 shows the basic components of a Tomasulo architecture. This architecture replaces the normal 5-stages of execution with 4 stages: Fetch, Issue, Execute, and Writeback. Explain what happens to an instruction in each of them (be as complete as you can):

a) Fetch:

b) Issue:

c) Execute:

d) Writeback:

Problem 4f:
Explain how the Tomasulo architecture handles the three different types of data hazards:

Problem 4g:
Assume that you have a long chain of dependent instructions, such as the following:

```
add $r1, $r2, $r3
add $r3, $r1, $r4
add $r7, $r3, $r5
      →
```

Also assume that the integer execution unit takes one cycle for adds. What CPI would you achieve for this sequence with the basic Tomasulo architecture, assuming that each of the stages from (4f) are non-overlapped and take a complete cycle?

Problem 4h:
Assume that associative matching on the CDB is a slow enough operation that it takes much of a cycle. How can you still get a throughput of one instruction per cycle for long dependent chains of operations such as given in (4g)? Only well-thought-out answers will get credit.

Problem 4i:
Finally, the Tomasulo algorithm has one interesting “bug” in it. Consider the situation where one instruction uses a value from another one. Suppose the first instruction is *issued* on the same cycle as the one that it depends on is in *writeback.*

```
add $r1, $r2, $r3     ← The result is broadcast
      ...
add $r4, $r1, $r1     ← This one is being issued
```

What is the problem? Can you fix it easily?