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

Spring 1999 John Kubiatowicz

Midterm II
Solutions
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]

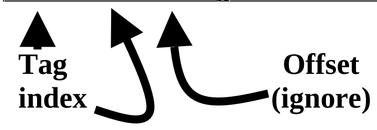
3.141592653589793238462643383279502884197169399375105820974944

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.

Bit Pattern	Address	Hit/Miss?	Miss Type?
00 **000** 111	0x7	Miss	Compulsory
01 **001** 101	0x4D	**Miss**	**Compulsory**
00 **101** 010	0x2A	**Miss**	**Compulsory**
01 **111** 001	0x79	**Miss**	**Compulsory**
10 **101** 011	0xAB	**Miss**	**Compulsory**
11 **001** 110	0xCE	**Miss**	**Compulsory**
00 **101** 110	0x2E	**Hit**	**N/A**
01 **001** 011	0x4B	**Hit**	**N/A**
01 **101** 101	0x6D	**Miss**	**Compulsory**
10 **001** 010	0x8A	**Miss**	**Compulsory**
10 **101** 111	0xAF	**Miss**	**Conflict**
00 **101** 001	0x29	**Miss**	**Conflict**
11 **001** 000	0xC8	**Miss**	**Conflict**
11 **001** 110	0xCE	**Hit**	**N/A**
01 **101** 010	0x6A	**Miss**	**Conflict**



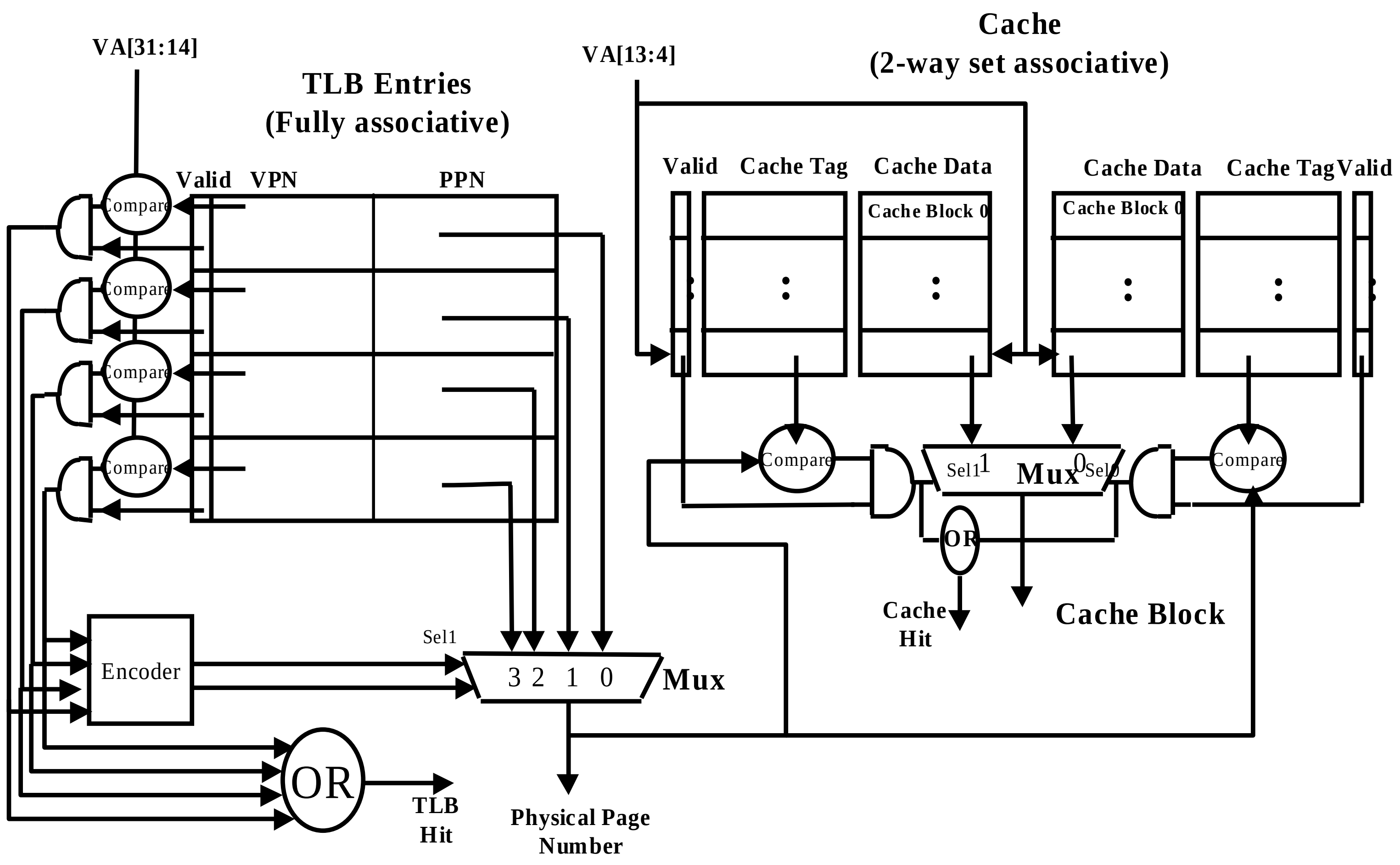
Ans: The trick with this type of cache simulation is the split the address into bit fields. Each cache block is 8 bytes⇒offset is 3 bits. This is the lowest 3 bits of the address, and should be completely ignored. The total blocks in the cache is 128 bytes/8 bytes=16 blocks. Since the cache is 2-way set associative, this means that the cache index selects among 16/2 = 8 blocks. So, index is 3 bits. Finally, the remaining two bits at the top are tag bits.

Problem 1b:
Calculate the miss rate and hit rate.
Miss Rate = 12/15 = 80%
Hit Rate = 3/15 = 20%

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.

Answer: The key observation is that, in order to do parallel access to TLB and cache data, you need to keep the cache index+offset ≤ page size (in bits). So, this means that we need 32K/16K sets (i.e. 2-way set associative). This is enough to draw our diagram:



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?

$AMAT = HT_{L1} + MR_{L1}*AMAT_{L2} + MR_{TLB}*MP_{TLB}$
$AMAT_{L2} = HT_{L2} + MR_{L2}*AMAT_{RAM}$
$AMAT_{RAM} = HT_{RAM} + MR_{RAM}*AMAT_{DISK}$

$HT_{L2} = 10\ cycles + 1\ cycle/64bits * 32 * 8\ bits = 14\ cycles$
$HT_{RAM} = 100ns + 25ns/8bytes * 128\ bytes = 500ns = 100\ cycles$
$HT_{DISK} = 50ms + 20ns/byte * 16Kbytes = 50.32768ms = 10.065536*10^6\ cycles$

$AMAT_{L2} = 3036.66\ cycles$
$AMAT_{inst} = 31.36\ cycles$
$AMAT_{data} = 152.88\ cycles$

Note: *HT = hit time*
MR = miss rate
MP = miss penalty
AMAT = Average Memory Access Time

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:

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

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:

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

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.`

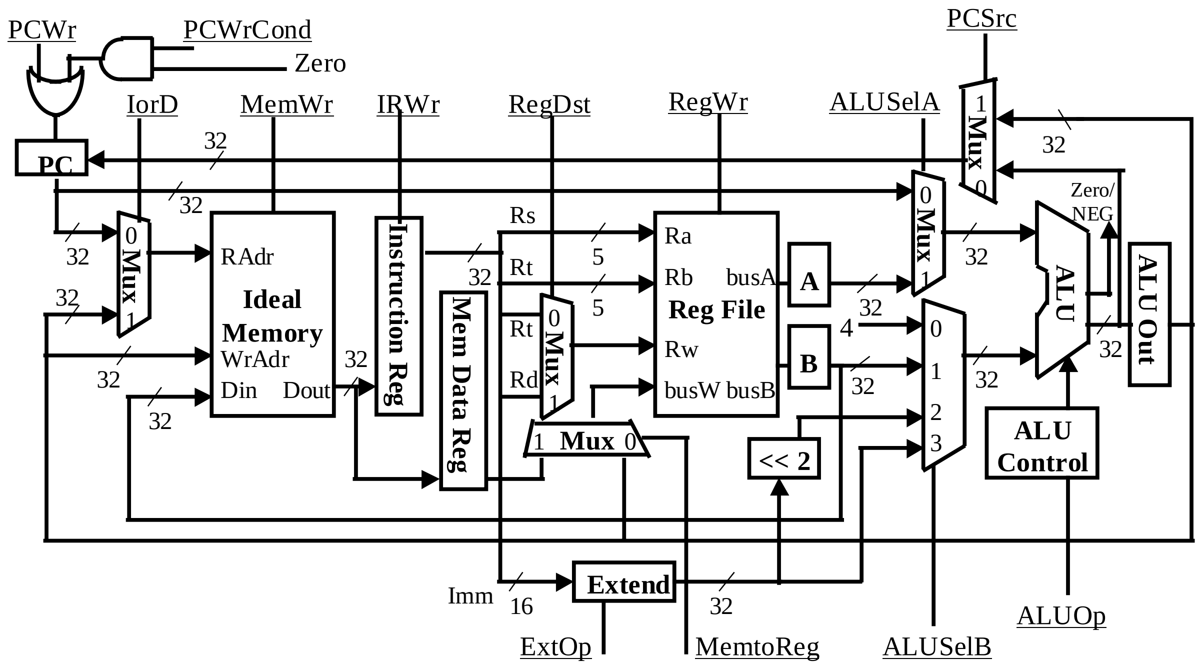


Figure 1: The Multicycle Data Path

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)

Problem 2 has many possible solutions. We will discuss some of them here. For 2b onwards, we will illustrate one possible solution.

Answer (#1): Without migrating computation out of the inner loop, there are 7 cycles of operations, since each address takes 2 cycles to compute (not forgetting to add the base, e.g. poly1!!!), and there is a multiply, add, and one decrement. Note that we moved the incrementing of indexdeg2 before the memory operations, since this takes care of the +1 part of the address computation. We must be careful to overlap memory operations, or we will have to take more cycles. Here is one iteration of the loop; the arrows show address computations:

Cycle 1	*Cycle 2*	*Cycle 3*	*Cycle 4*	*Cycle 5*	*Cycle 6*	*Cycle 7*
`indexdeg1 +1`	`poly1+ indexdeg1<<2` ➘	`indexdeg2 ++`	`poly2+ indexdeg2<<2` ➘	`indexdeg1 --`	`multiply`	`add`
		`poly1[]`		`poly2[]`		

Note that we must check the condition for the while in Cycle 1,2, or 3, since indexdeg2 has changed by cycle 3.

Answer(#2): Let’s move the “+1” functionality out of the inner loop. We can hope for a 6-cycle inner loop. Here is one “cycle” of the loop, assuming that we have incremented poly1 and poly2 by one element (i.e. by 4) outside the inner loop:

Cycle 1	*Cycle 2*	*Cycle 3*	*Cycle 4*	*Cycle 5*	*Cycle 6*
poly1+ indexdeg1<<2	*poly2+ indexdeg2<<2*	*indexdeg1--*	*indexdeg2++*	*multiply*	*add*
	poly1[]	*poly2[]*			

Answer(#3): Alternatively, we could increment indexdeg1 outside the loop and leave poly1 and poly2 alone. Then, by creative incrementing and decrementing,we can avoid the extra +1 computation. This means that we need to check for the boundary condition “indexdeg2 ≤ degree2” before cycle 3 (when indexdeg2 changes) and check for the boundary condition “indexdeg1 ≥ 1” before cycle 5 or check indexdeg1≥ 0 on cycle 5 or 6:

Cycle 1	*Cycle 2*	*Cycle 3*	*Cycle 4*	*Cycle 5*	*Cycle 6*
poly1+ indexdeg1<<2	*indexdeg2++*	*poly2+ indexdeg2<<2*	*indexdeg1--*	*multiply*	*add*
	poly1[]		*poly2[]*		

Answer(#4): Finally, if we are really clever, we could reduce this down to 4 cycles by using pointers that have the base values already added in. In the following, assume that: “point1 = poly1 + indexdeg1<<2” and that “point2=poly2+indexdeg2<<2”. Also assume that we indexdeg1 is ahead by 1 (i.e. point1 is ahead by 4) at the beginning of the loop. Then, in cycle 1 we check the boundary condition “point2≤poly2+degree2<<4” and in cycle 3 or 4 we check the boundary condition “point1≥poly1”:

Cycle 1	*Cycle 2*	*Cycle 3*	*Cycle 4*
point2+=4	*point1-=4*	*multiply*	*add*
point1[]	*point2[]*		
Cond2 check		*Cond1 check*	

Just to make clear what we have done, here would be modified code to reflect the last option:

```
point1 = poly1 + indexdeg1<<4 + 4;
point2 = poly2 + indexdeg2<<4;

/* (indexdeg1+indexdeg2)=resultdeg throughout loop */
accum = 0;
do {
   cond2 = (point2 == poly2);  /* Save this on cycle 1 */
   point2 = point2 + 4;
   /* THE “*” means to find value pointer*/
   accum = accum + (*point1)×(*point2);
   point1 = point1 – 4;
   cond1 = (point1 == poly1+degree2<<4)
} until (cond1 or cond2);
```

Registers needed for this code: point1, point2, poly2, (poly1+degree2<<4), accum, cond2; Note also that we will multiply everything by 4 to make this work as well.

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!)*

Answer: Although the last option from the previous problem can be made to use the least additional hardware, it can be the hardest to understand. Instead, lets use the somewhat less optimal option #3 above. Further, lets take the outer "for" loop backwards, i.e. starting with resultdeg=degree3. This will eliminate the need for one more register, since we can decrement resultdeg and look for zero, rather than keeping degree3 around. So, new registers:

indexdeg1, indexdeg2, resultdeg, degree1, degree2, poly3, accum

All of the new registers must be muxed into the ALU as inputs. Further, they should be able to latch their values from output of the ALU. Also, the ALUOUT register should be fed back into the ALU as an input.

Note that poly1 and poly2 are always available in registers A and B, since they are continually fetched from the register file. So, since indexdeg1 must be added to poly1 (in reg A), indexdeg1 needs to go to the B input of the ALU. etc.

Other changes that are needed: we need to add a mux in front of the "RA" input of the register file so that we can read register RD once (to get poly3). Note that we assume that the "A" register of the ALU is always writing unless we assert "InitPoly3", in which case it holds its value.

Further, the value written to memory must be able to come from either the B register (for normal "store" instructions) or from the accum register (at the end of the inner loop). The address to memory should be able to come from either ALUOUT (as now) or from poly3 (when updating the polynomial). Finally, we need to add an extra register from data output and put enables on both registers; both of these registers (call them MemDataReg1 and MemDataReg2) need to be inputs to the ALU. To deal with the fact that we need to multiply indices by 4 on the way in, we will support x4 functionality on these registers as well (denoted by "<<2").

As far as conditionals are concerned, the "neg" output of the ALU will be tied into "cond3", the boundary condition (indexdeg1$\neq$4 and indexdeg2$\neq$degree2) will be tied into "cond1" and the condition (resultdeg $\neq 0$) will be tied into cond2. Both cond1 and cond2 are written in such a way as to indicate that the loop is not yet done (i..e we should continue with the loop).

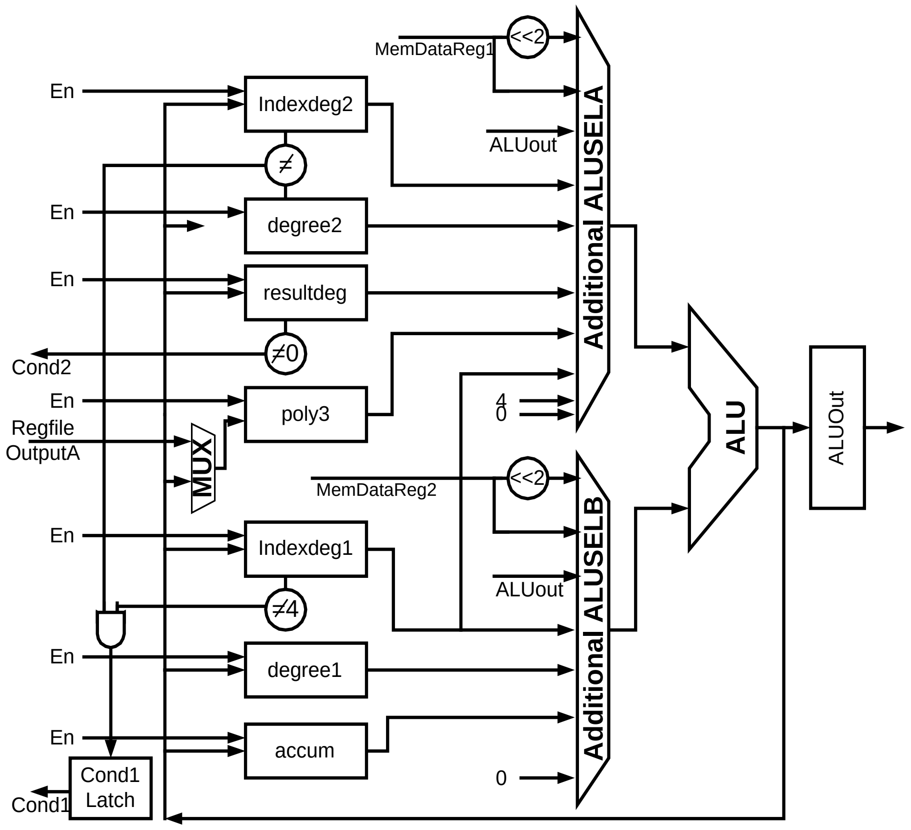
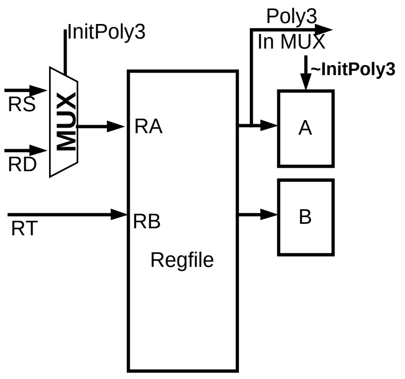


Figure 2a: Additions to the ALU path for polynomial multiply. Note that the ALU input muxes in this figure only include new inputs. Assume that the other inputs are still there.



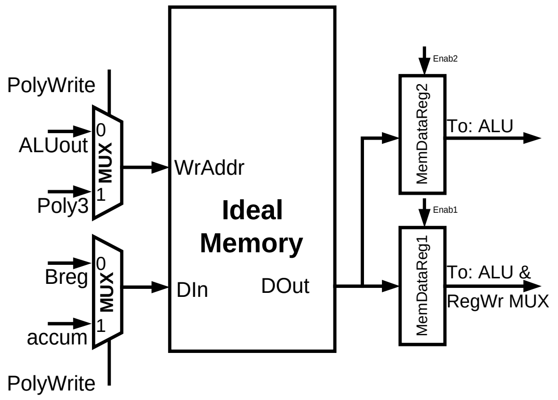


Figure 2b: Additions to the RegFile control to support polynomial multiply. The primary change is the addition of an extra address MUX to give us access to register RD. Also, added enable to A register.

Figure 2c: Additions to the Memory component to support polynomial multiply. Note that we have two new muxes to support writes of the accumulated value into Poly3. Also, we have added a temporary memory register.

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

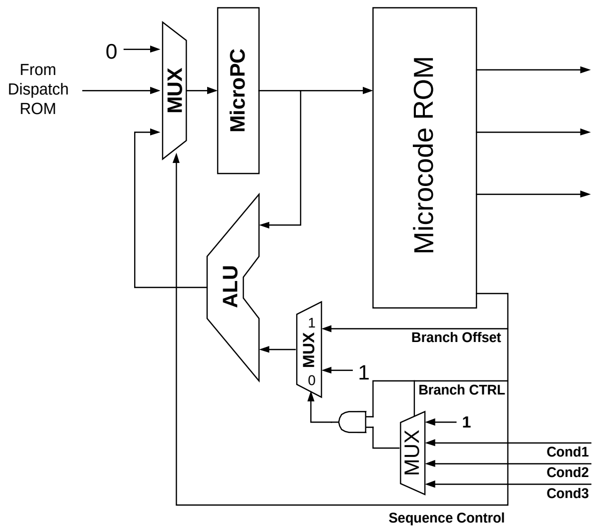
Label	*ALU*	*SRC1*	*SRC2*	*Dest.*	*Memory*	*Mem. Reg.*	*PC Write*	*Sequence*
Fetch:	Add	PC	4	—	Read PC	IR	ALU	SEQ
	Add	PC	Extshft	—	—	—	—	Dispatch
Rtype:	Func	rs	rt	—	—	—	—	Seq
	—	—	—	rd ALU	—	—	—	Fetch
Ori:	Or	rs	Extend0	—	—	—	—	Seq
	—	—	—	rt ALU	—	—	—	Fetch
Lw:	Add	rs	Extend	—	—	—	—	Seq
					Read ALU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

Answer: The sequencer needs to have the ability to branch. The version shown here includes a "branch always" option, to permit jumping. It also includes two conditions (see input mux in lower right corner of diagram).

Answers which permitted some form of branching (and which included an acknowledgment that there are two conditions of interest got most credit.



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.

<u>Additions to SRC1:</u>
- MemData1
- MemData1<<2
- ALUout
- indexdeg1
- indexdeg2
- degree2
- resultdeg
- poly3
- 0
- 4
- poly1 (same as "rs")

<u>Additions to Dest:</u>
- InitPoly3
- LatchCond1

<u>Additions to SRC2:</u>
- MemData2
- MemData2<<2
- ALUout
- indexdeg1
- degree1
- accum
- 0
- poly2 (same as "rt")

<u>Additions to Sequence:</u>

jump	offset
bneg	offset
bfor	offset
bwhile	offset

<u>ALULatch:</u>
- indexdeg1
- indexdeg2
- resultdeg
- degree1
- degree2
- poly3
- accum

Additions to <u>Memory:</u>
- wr Poly3

Additions to <u>Memory Reg:</u>
- MemData1
- MemData2

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

Note that we ended up calling this part of the problem "extra credit" and gave a few extra points to people who had started down the path.

Label	ALU	SRC1	SRC2	ALULatch	Dest.	Memory	Mem. Reg.	PC Write	Seq
/***** Fetch address of destination polynomial and put in register poly3 */									
polymult:	—	—	—	—	*InitPoly3*	—	—	—	*Seq*
/***** Fetch poly1[0] and poly2[0] */									
	Add	*poly1*	*0*	—	—	—	—	—	*Seq*
	Add	*0*	*poly2*	—	—	*rd ALU*	*MemData1*	—	*Seq*
	—	—	—	—	—	*rd ALU*	*MemData2*	—	*Seq*
/***** Compute poly3[0] and initialize degree1 & degree2 with 4 * degree values */									
	Add	*MemData1*	*MemData2*	—	—	—	—	—	*Seq*
	Add	*MemData1<<2*	*0*	*degree1*	—	*wr Poly3*	—	—	*Seq*
	Add	*0*	*MemData2<<2*	*degree2*	—	—	—	—	*Seq*
/***** Start resultdeg at end (i.e. degree3). Of course, like degree1 and degree2, this is actually *4 */									
	Add	*degree2*	*degree1*	*resultdeg*	—	—	—	—	*Seq*
/***** Initial value of poly3 is at end of result polynomial (since we are going to work backwards) */									
	Add	*poly3*	*resultdeg*	*poly3*	—	—	—	—	*Seq*
	Add	*poly3*	*4*	*poly3*	—	—	—	—	*Seq*
*/***** Compute indexdeg1 and indexdeg2. Next 4 lines are combination of max function and following subtract */*									
forloop:	*Sub*	*resultdeg*	*degree1*	*indexdeg2*	—	—	—	—	*bneg forloop1*
	Add	*0*	*degree1*	*indexdeg1*	—	—	—	—	*jump forloop2*
forloop1:	*Add*	*resultdeg*	*0*	*indexdeg1*	—	—	—	—	*Seq*
	Add	*0*	*0*	*indexdeg2*	—	—	—	—	*Seq*
/***** Initialize accum variable for inner loop. */									
forloop2:	*Add*	*0*	*0*	*accum*	—	—	—	—	*Seq*
/***** Let indexdeg1 be ahead by one iteration (to avoid extra +1 in [] – see discussion in 2b) */									
	Add	*4*	*indexdeg1*	*indexdeg1*	—	—	—	—	*Seq*
/***** Next 6 microinstructions are the inner loop */									
whileloop:	*Add*	*poly1*	*indexdeg1*	—	*LatchCond1*	—	—	—	*Seq*
	Add	*indexdeg2*	*4*	*indexdeg2*	—	*rd ALU*	*MemData1*	—	*Seq*
	Add	*indexdeg2*	*poly2*	—	—	—	—	—	*Seq*
	Sub	*indexdeg1*	*4*	*indexdeg1*	—	*rd ALU*	*MemData2*	—	*Seq*
	Mul	*MemData1*	*MemData2*	—	—	—	—	—	*Seq*
	Add	*accum*	*ALUout*	*accum*	—	—	—	—	*bwhile whileloop*
/***** End of while loop. Write back result to poly3, update resultdeg and poly3 pointer */									
endfor:	*Sub*	*resultdeg*	*4*	*resultdeg*	—	*wr Poly3*	—	—	*Seq*
	Sub	*poly3*	*4*	*poly3*	—	—	—	—	*bfor forloop*
/***** Last dummy instruction is just for fetching */									
	—	—	—	—	—	—	—	—	*fetch*

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:

a. Floating-point multiply: **4 cycles**
b. Floating-point addition: **2 cycles**
c. 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?

Ans: 14 cycles: 9 instructions + 2 stall cycles before “muls” + 3 stall cycles before “adds”.

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?

```
dotprod:  lw    $f5, 0($r1)      ; load element from first array
          lw    $f6, 0($r2)      ; load element from second array
          addi  $r1, $f1, 4      ; advance pointers
          addi  $f2, $r2, 4
          muls  $f7, $f5, $f6    ; multiply elements
          addi  $r3, $r3, -1     ; decrement element count
          bne   $r3, $zero, dotprod  ; loop
          adds  $f4, $f4, $f7    ; add elements to accumulator in f7
```

Now this gets 9 cycles/iteration: 8 instructions + one stall cycle before “adds”.

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?

```
dotprod:  lw    $f5, 0($r1)     ; load element from first array
          lw    $f6, 0($r2)     ; load element from second array
          lw    $f8, 4($r1)     ; load another element from first
          lw    $f9, 4($r2)     ; load another element from second
          muls  $f7, $f5, $f6
          addi  $r1, $r1, 8     ; advance pointers
          muls  $f10,$f8, $f9
          addi  $r2, $r2, 8
          adds  $f4, $f4, $f7   ; add elements to accumulator in f4
          addi  $r3, $r3, -2    ; decrement element count
          bne   $r3, $zero, dotprod  ; loop
          adds  $f4, $f4, $f10  ; add elements to accumulator in f7
```

Total cycles: 12 instructions, no stalls. So, 12cycles/2iterations = 6 cycles/iteration

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)*

Ans: 4.5 cycles/iteration. There are 4 substantive instructions/loop (2 loads, 1 muls, 1 adds). The remaining 4 instructions (3 addi, 1 bne) get distributed over all iterations. So: $4 \times 8 + 4 =$ 36 cycles/8 iterations.

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.*

```
dotprod:  adds  $f4, $f4, $f7   ; add elements to accumulator in f7
          muls  $f7, $f5, $f6   ; multiply elements
          lw    $f5, 0($r1)     ; load element from first array
          lw    $f6, 0($r2)     ; load element from second array
          addi  $r1, $r1, 4     ; advance pointers
          addi  $r3, $r3, -1    ; decrement element count
          bne   $r3, $zero, dotprod  ; loop
          addi  $r2, $r2, 4
```

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`? *7*
- the maximum execution latency for `adds`? *8*
- the maximum load-use latency (delay slots) for `lw`? *5*

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?

Less. Because things are more deeply pipelined, each stage is less complicated and could thus be run at lower voltage to keep at same clock rate.

Problem 4: Hazards and Advanced 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: *Read After Write - first instruction has not yet modified (written to) register file yet, but next instruction is trying to read that register.*

```
            add   $1, $2, $3
            addi  $4, $1, 50
```

Fix: Forward data from the pipeline as needed.

b) WAR: *Write After Read - first instruction is reading from a register that the next instruction has somehow already modified.*

```
            add   $1, $2, $3  # assume really long fetch or something
            add   $2, $5, $6  # assume writes really fast
```

Fix: Make each stage equal length in time and read the registers early and write late in the pipeline.

c) WAW: *Write After Write - later instruction writes to a register before the former instruction has modified it.*

```
            add   $1, $2, $3
            add   $1, $4, $5
```

Fix: Only modify the register file in the WB stage.

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

(1) *Waiting always fixes the problem, i.e. stall and bubble the pipeline.*
(2) *Branch Prediction is another more sophisticated solution.*
(3) *Another similar solution is to execute both branches.*
(4) *Changing the software model is also a valid solution, e.g. branch delay slot.*

Problem 4c:
Come up with two reasons why designers don't make 100-stage pipelines. Are there circumstances in which such a pipeline might make sense?

100 stage pipelines incur way too many data and control hazards, requiring way too much hardware overhead to fix these hazards. Having so many stages means having lots of registers which means higher area, bigger clock net, more power dissipation, etc.

Such a pipe may be feasible if there are very few dependencies and very little decision making such as in stream based processing like multimedia.

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

Precise exceptions occur when all instructions following the one that made the exception do not affect (modify) the state of the machine, and all instructions previous have finished completely (i.e. written back to register file etc.).

Precise interrupts are important because they make getting back from the exception easier to manage and easier to figure out which exception cause the problem and what the problem is.

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

A precise exception is achieved by keeping track of which instruction causes the exception and waiting till the write back stage to cause the exception. This ensures that all instructions following and including the instruction that caused the exception will not modify the state of the system; and all instructions before the exception-causing instruction are completed. To handle precise exception, extra hardware such as the Exception PC (EPC) and Cause register are needed.

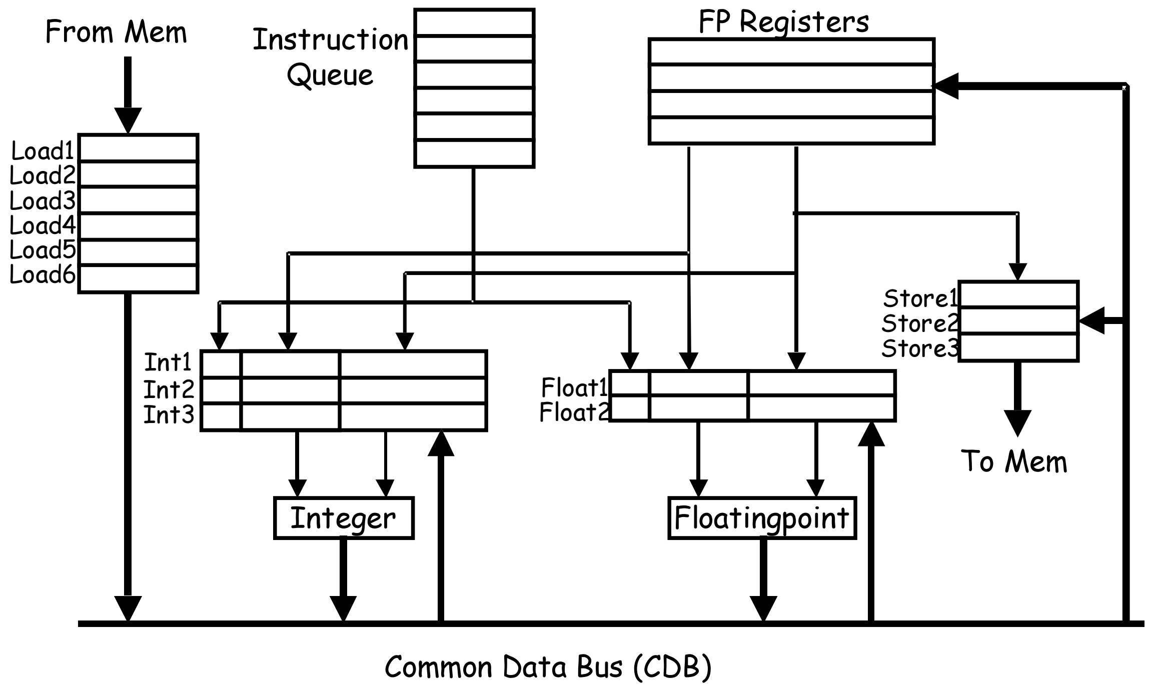


Figure 2: A basic Tomasulo architecture

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:
Fetch instructions from memory in program order and place into Instruction Queue.

b) Issue:
Get next instruction from Instruction Queue and send to appropriate reservation station, replacing registers with values or tags (if the value is the pending result of an instruction in some reservation station).

c) Execute:
Dispatch instructions queued in reservation units to execution units if when register (tag) values are available. Mark the reservation stations as available.

d) Writeback:
Broadcasts result on the CDB (Common Data Bus). Any instructions waiting for the result will grab the value. This will also update the values in the register file.

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

RAW – *wait for values to be broadcast on CDB before dispatching dependent instructions from the reservation stations to execution units.*

WAR – *Because of register renaming, there is no WAR hazards.*

WAW – *Because of register renaming, there is no WAW 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?

Answer: The CPI we are looking for is 2. Consider the following timing diagram:

```
EXE | WB
        | EXE | WB
                   | EXE | WB
```

You can only dispatch a instruction waiting in the reservation station once every two cycles because the WB takes a complete cycle to update the values of the tag and also decide which instruction in the reservation station to dispatch to the Integer unit. Thus, one integer instruction is retired every two cycles.

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.

Answer: To have the throughput of 1 cycle per instruction, we need to overlap the WB and EXE stages, as shown in the following diagram:

```
EXE | WB
    | EXE | WB
          | EXE | WB
```

In this scheme, during the last stage of EXE stage, the decision about which instruction to dispatch in the reservation station is also computed. As soon as the data become avaiable, the instruction can be dispatched to the execution unit without additional delay.

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?
Problem: As the second instruction is being issued, with tags from the register file, the first instruction is finishing up and will remove tag (because the $r1 is no longer busy). Thus, the second instruction will be waiting for a tag that does not exist at the end of the cycle and deadlocks.

Solutions:
AS in MIPS register file, do write in the first half of the cycle and read in the second half of the cycle. This way, the second instruction will just get the actual value of $r1, instead of the a tag that points to $r1.