6.823 Computer System Architecture
Datapath for DLX Spring 2002
Problem Set #2

Students are allowed to collaborate in groups of up to 3 people. A group hands in only one copy of the solution to a problem set. Homework assignments are due at the beginning of class on the due date. To facilitate grading, each problem must be stapled separately. Homework will not be accepted once solutions are handed out.

Problem 1: Microprogramming and Bus-Based Architectures

Problem 1.A

How many cycles does it take to execute the following instructions in the microcoded DLX machine? Use the states and control points from DLX-Controller-2 and assume Memory will not assert its busy signal.

Instruction	Cycles
`ADD  R3,R2,R1`	
`ADDI R2,R1,#4`	
`LW   R1,0(R2)`	
`BEQZ R1,label  # (R1 == 0)`	
`BEQZ R1,label  # (R1 != 0)`	
`J    label`	
`JR   R1`	
`JAL  label`	
`JALR R1`	

Which instruction takes the most cycles to execute? Which instruction takes the fewest cycles to execute?

Problem 1.B

Ben Bitdiddle needs to compute factorials for small numbers. Realizing there is no multiply instruction in the microcoded DLX machine, he uses the following code to calculate the factorial of an unsigned number n.

```
result = 1;

for (i = 0; i < n; i++) {
```

```
        temp = result;
        for (j = 0; j < i; j++) {
            result += temp;
        }
    }
```

The variables `i`, `j`, `n`, `temp`, and `result` are unsigned 32-bit values.

Write the DLX assembly that implements Ben's factorial code. Use only the DLX instructions that can be executed on the microcoded DLX machine (ALU, ALUi, LW, SW, J, JAL, JR, JALR, BEQZ, and BNEZ). The microcoded DLX machine does not have branch delay slots. Use `R1` for `n` and `R2` for `result`. At the end of your code, only `R2` must have the correct value. The values of all other registers do not have to be preserved.

How many DLX instructions are executed to calculate a factorial? How many cycles does it take to calculate a factorial? Again, use the states and control points from DLX-Controller-2 and assume Memory will not assert its busy signal.

Factorial	Instructions	Cycles
0		
1		
2		
3		
N		

Problem 1.C

Alyssa P. Hacker tells Ben that his factorial code will run much faster if he implements an unsigned multiply instruction in the microcoded DLX machine. Then Ben can replace the inner loop instructions with the new unsigned multiply instruction.

The details of Alyssa's new proposed unsigned multiply instruction are:

```
MULU Rd,Rs1,Rs2     # Rd <- Rs1 + ... + Rs1 (Rs2 times)
```

The value of `Rs1` is added `Rs2` times and the result stored into `Rd`. `Rs1` and `Rs2` are treated as unsigned 32-bit values. If `Rs2` or `Rs1` is 0, then the result of `Rd` will also be 0. The format of the MULU instruction is R-type.

In order to be able to write microcode for MULU, Alyssa adds an additional register, `T0` (33), to the register file. This register, like the PC register, is not visible to the programmer. She also adds 33 as an input to the register file multiplexer.

Using Worksheet 1, write microcode to implement Alyssa's new unsigned multiply instruction.

In Worksheet 1, the representation of the next state is different from what was presented in lecture. The last two columns specify the next state. The μBr (*micro*branch) column represents a 2-bit field with four possible values: N, J, Z, and D. If μBr is N (next), then the next state is simply (*current state* + 1). If it is J (jump), then the next state is *unconditionally* the state specified in the Next State column (i.e., it's an unconditional microbranch). If it is Z (branch-if-zero), then the next state depends on the value of the ALU's *zero* output signal (i.e., it's a conditional microbranch). If *zero* is asserted (== 1), then the next state is that specified in the Next State column, otherwise, it is (*current state* + 1). If μBr is D (dispatch), then the FSM looks at the opcode and function fields in the IR and goes into the corresponding state. For this problem set, we assume that the dispatch goes to the state labeled (DLX-instruction-name + "0"). For example, if the instruction in the IR is SW, then the dispatch will go to state SW0.

The ALU performs operations specified by the ALUOp, which is determined by the ALU control logic block. Assume the ALU can perform the following operations:

ALUOp	ALU Result Output
COPY_A	A
COPY_B	B
INC_A_1	A+1
DEC_A_1	A-1
INC_A_4	A+4
DEC_A_4	A-4
ADD	A+B
SUB	A-B

How many cycles does it take for the MULU instruction for different values of `Rs2`?

`Rs2`	Cycles
0	
1	
2	
3	
N	

Problem 1.D

With Alyssa's new unsigned multiply instruction, Ben eliminates the inner loop of his original factorial code and simplifies it to the following.

```
result = 1;

for (i = 1; i <= n; i++) {
    result = result * i;
}
```

Help Ben write DLX assembly code to implement factorial using the new MULU instruction. Again, use `R1` for `n` and `R2` for `result`. At the end of your code, `R2` must contain the correct value. You do not have to preserve the values of any other registers.

How many DLX instructions are executed to calculate a factorial? How many cycles does it take to calculate a factorial? Again, assume Memory will not assert its busy signal.

Factorial	Instructions	Cycles
0		
1		
2		
3		
N		

Problem 1.E

Combining a microcontroller and the DLX bus-based datapath (L4-5) gives us a complete working computer that can run a subset of the DLX ISA.

Besides requiring much more memory, Alyssa tells Ben another reason why using the original DLX Microcontroller (L4-9) was a bad idea. A machine with the original controller would have a much longer cycle time than a machine using the second microcontroller (L4-15). Ben can't understand why this is true.

Below are the delays of the hardware parts used to implement the DLX bus-based datapath and the first version of the DLX Microcontroller (L4-9).

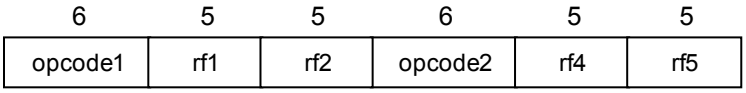
t_{setup} – setup time of registers IR, A, B, MA, μPC
$t_{c\text{-}q}$ – clock-to-q time of registers IR, A, B, MA, μPC
t_{ALU} – time ALU takes to generate result and zero
t_{extend} – delay of a sign extender
t_{mux} – delay of a multiplexer
t_{bus} – delay from when data is written to bus to when it becomes stable
$t_{tristate}$ – delay of a tri-state buffer
t_{RF} – delay of the register file
t_{Mem} – delay of memory
t_{ROM} – delay of the ROM
$t_{ALU\_control}$ – delay for ALU Control

Assume that t_{ALU}, $t_{ROM,}$ t_{RF}, t_{Mem} have comparable values and that these delays are bigger than the delays for other components.

Using the first version of the microcontroller, which microcode instruction invokes the critical path of the machine? Describe the critical path. What is the minimum clock period that the complete DLX bus-based machine can run at? Assume Memory will not assert its busy signal.

Problem 2: Pipeline Hacking

Inspired by his success with the `MACC` instruction in the last problem set, Ben Bitdiddle comes up with the following new instruction format called BIF (for Ben's Instruction Format) that he wants to add to the DLX ISA:



The semantics of the new instruction would be this:

```
rf5 <- (rf1 op1 rf2) op2 rf4
```

where `op1` and `op2` are ALU operations. For example, `ADD R1,R2: ADD R3,R4` would be computed as follows:

```
R4 <- (R1 + R2) + R3
```

To implement the new instruction format, Ben decides to add an ALU to the memory phase of the pipelined, fully-bypassed implementation of the DLX datapath discussed in lecture. Old-style DLX instructions would still use the ALU in the execute phase while BIF instructions would use both ALUs. The first ALU operation in a BIF instruction would execute on the ALU in the execute phase, while the second ALU operation would execute on the ALU in the memory phase. The new pipeline would look like this:

IF	ID	EX	MA	WB
Fetch phase	Decode and register fetch phase	Execute phase with original ALU	Memory phase with new ALU	Write-back phase

In addition, the register file in the old datapath is replaced with a register file with three read ports and one write port so that all three operands can be read at the same time.

Problem 2.A

Give a code example that shows how you can get better performance using BIF instructions. Provide both the original old-style DLX code and the code that uses the BIF instructions. The original code should contain at least six instructions.

What is the maximum possible improvement in performance using BIF instructions?

Problem 2.B

Show all data hazards that can cause stalls and provide a code example for each case. You should consider both old-style DLX instructions and instructions using Ben's new format. You may assume that the datapath is fully-bypassed. Do not consider jumps or branches for now.

Still ignoring jumps and branches, how would performance change if non-BIF instructions used the new ALU in the MA phase instead of the original ALU in the execute phase?

Problem 2.C

Write the equations for ws, we, re1, re2 for the new datapath with non-BIF instructions using the original ALU in the execute phase. Write the stall signal using ws, we, re1, and re2. You may need other signals. The signals for the original datapath are provided here for your convenience. Again, do not consider jumps or branches for now.

c_{dest}

ws = *case opcode*

ALU	⇒ rf3
ALUi, LW	⇒ rf2
JAL, JALR	⇒ R31

we = *case opcode*

ALU, ALUi, LW	⇒ (ws ≠ R0)
JAL, JALR	⇒ on
…	⇒ off

c_{re}

re1 = *case opcode*

ALU, ALUi, LW, SW, BEQZ, JR, JALR	⇒ on
J, JAL	⇒ off

re2 = *case opcode*

ALU, SW	⇒ on
(	(off

cstall

stall = (rf1D = wsE)((opcodeE = LWE)((wsE (R0)(re1D +
(rf2D = wsE)((opcodeE = LWE)((wsE (R0)(re2D

Problem 2.D

Now consider jumps and branches. What additional hazards can occur? Give an example for each case.

With the new instruction format, Ben thinks that we can speed up conditional branches if we allow an instruction that combines the compare with the branch. For example,

```
SLT R1,R2: BEQZ label
```

would mean:

```
if (R1 < R2) then branch to label
```

The first instruction (in this example, the SLT instruction) would be performed on the ALU in the EX phase. The result (a 0 or 1), instead of being written to the register file, would be passed to the MA phase where the zero test would be performed on the new ALU. How many delay slots will this type of instruction require to avoid any stalls?

How could you reduce the number of delay slots that are needed, without introducing any new stall conditions or killing instructions in the pipeline? For each case that you consider, argue what effect it will have on the clock period. Each case should correspond to an implementation with a different number of delay slots.

Given the options you investigated above, argue in a few sentences which of these options is the best. Consider delay slots, stalls, circuit size, and clock period.

Problem 2.E

Ben finds that the additional read port on the register file is increasing the length of the critical path on the processor, and that they cannot clock the new datapath at as high a speed as the original. To try and solve this problem, he is going to try and use the original register file.

Since the original register file only has two read ports, only two of the operands can be read in the ID phase. The third operand is going to be read in the EX phase, in parallel with the first ALU operation.

What other changes are needed to make this scheme work? How do these changes affect the performance of the processor?

Alyssa thinks that Ben can solve his problem by adding a second register file, identical to the first. How would this scheme work? How does the performance of Alyssa's scheme compare to the two that Ben tried?

Problem 2.F

With the new BIF instructions, how will code size change? Will this have an affect on performance?

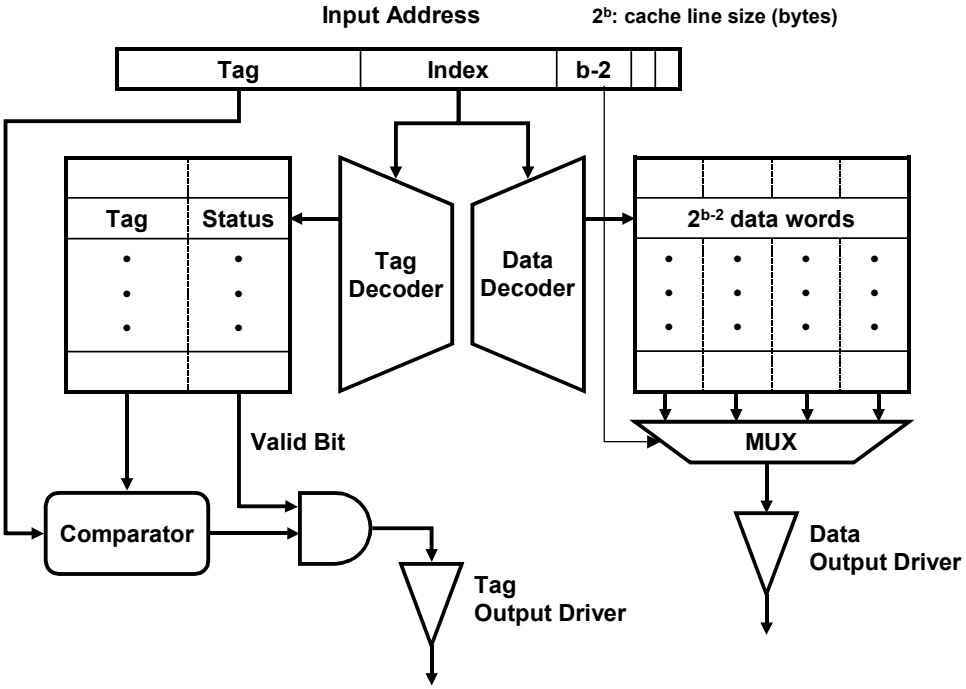
Problem 3: Cache Access-Time & Performance

Ben is trying to determine the best cache configuration for a new processor. He knows how to build three kinds of caches: direct-mapped caches, 2-way set-associative caches, and small fully associative caches. The goal is to find the best cache configuration with the given building blocks.

Since he only knows how to build very small fully associative caches, Ben decided to use either direct-mapped or 2-way set-associative as the basic cache configuration. He wants to know how these two different configurations affect the clock speed and the cache miss-rate, and choose the one that provides better performance in terms of average latency for a load.

Problem 3.A — Access time: DM

The following diagram shows how a direct-mapped cache is organized. To read a word from the cache, the input address is set by the processor. Then the index portion of the address is decoded to access the proper row in the tag memory array and in the data memory array. The selected tag is compared to the tag portion of the input address to determine if the access is a hit or not. At the same time, the corresponding cache block is read and the proper line is selected through a MUX.



In the memory array, each row corresponds to a row in the cache. For example, a row in the tag memory array contains one tag and two status bits (valid and dirty) for the cache line. For direct-mapped caches, a row in the data array holds one cache line.

Now we want to compute the access time of the cache. Assume a 32-KB cache with 8-word (32-byte) cache lines. The address is 32 bits, and two LSB of the address is ignored since a cache access is word-aligned. The data output is also 32 bits, and the MUX selects one word out of the eight words in a cache line. Using the delay equations given in Table 3-1, fill in the column for the direct-mapped (DM) cache in Table 3-1. *In the equation for the data output driver, 'associativity' refers to the associativity of the cache (1 or direct-mapped caches, A for A-way set-associative caches).*

Component	Delay equation (ps)		DM (ps)	SA (ps)
Decoder	200×(# of index bits) + 1000	Tag		
		Data		
Memory array	200×$\log_2$ (# of rows) +	Tag		
	200×$\log_2$ (# of bits in a row) + 1000	Data		
Comparator	200×(# of tag bits) + 1000			
N-to-1 MUX	500×$\log_2$ N + 1000			
Buffer driver	2000			
Data output driver	500×(associativity) + 1000			
Valid output driver	1000			

Table 3-1: Delay of each Cache Component

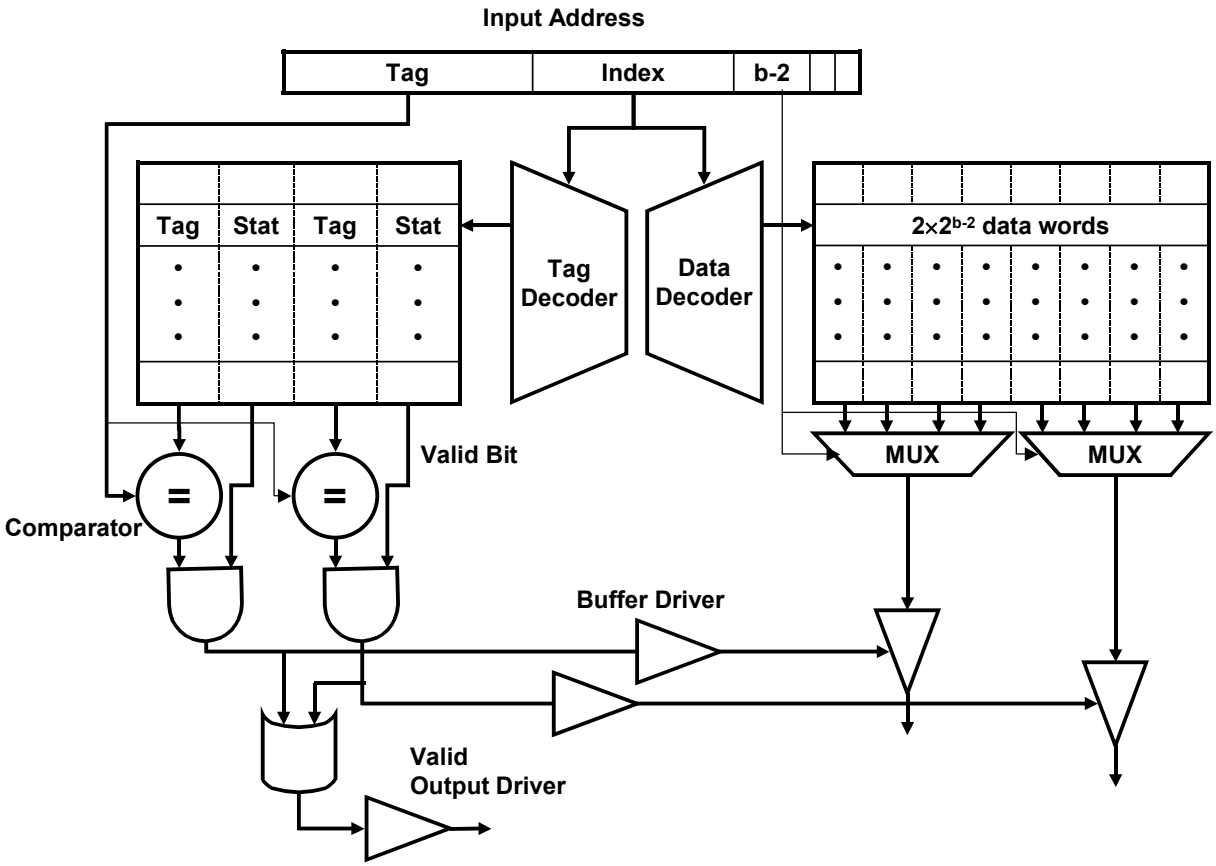
What is the critical path of this direct-mapped cache for a cache read? What is the access time of the cache (the delay of the critical path)? To compute the access time, assume that a gate (AND, OR) delay is 500 (ps). If the CPU clock is 150 MHz, how many CPU cycles does a cache access take?

Problem 3.B **Access time: SA**

The implementation of a 2-way set-associative cache is shown in the following diagram. The index part of the input address is again used to find the proper row in the data memory array and the tag memory array. In this case, however, each row corresponds to two cache lines (one cache set). A row in the data memory holds two cache lines (for 32-bytes cache lines, 64 bytes), and a row in the tag memory array contains two tags and status bits for those tags (2 bits per cache line). The tag memory and the data memory are accessed in parallel, but the output data driver is enabled only if there is a cache hit.

Assume the total cache size is 32-KB (each way is 16-KB) and all other parameters (such as the input address, cache line, etc.) are the same as part 3.A. Compute the delay of each component, and fill in the column for a 2-way set-associative cache in Table 3-1.

What is the critical path of the 2-way set-associative cache? What is the access time of the cache (the delay of the critical path)? What is the main reason that the 2-way set-associative cache is slower than the direct-mapped cache? If the CPU clock is 150 MHz, how many CPU cycles does a cache access take?



Problem 3.C **Miss-rate analysis**

Now Ben is studying the effect of set-associativity on the cache performance. Since he now knows the access time of each configuration, he wants to know the miss-rate of each one. For the miss-rate analysis, Ben is considering two small caches: a direct-mapped cache with 4 lines with 16 bytes/line, and a 2-way set-associative cache, using a least recently used replacement policy, with 4 lines with 16 bytes/line.

Ben tests the cache by accessing the following sequence of hexadecimal byte addresses, starting with empty caches. Complete the following tables for both the direct-mapped cache and the 2-way set-associative cache showing the progression of cache contents as accesses occur (in the tables, 'inv' = invalid, and the column of a particular cache line contains the {tag,index} contents of that line; e.g. for the address '110,' L1 gets the value '11'). *You only need to fill in elements in the table when a value changes.*

D-map	Problem 3.C				
	line in cache				hit?
Address	L0	L1	L2	L3	
110	inv	11	inv	inv	no
101	10				
123			12		
201	20				
15C					
102					
136					
202					
137					
15D					
103					
114					
203					

Problem 3.E	
line	VC hit?
VC	
inv	No
10	

	D-map
Total Misses	
Total Accesses	

2-way	Problem 3.C				
	line in cache				hit?
Address	Set 0		Set 1		
	MRU	**LRU**	**MRU**	**LRU**	
110	inv	inv	11	inv	no
101	10				
123	12	10			
201					
15C					
102					
136					
202					
137					
15D					
103					
114					
203					

Problem 3.E	
line	VC hit?
VC	
inv	no

	2-way
Total Misses	
Total Accesses	

Problem 3.D **Average latency**

Assume that the results of the above analysis can represent the average miss-rates of the direct-mapped and the 2-way 32-KB caches studied in 3.A and 3.B. What would be the average memory access latency in CPU cycles for each cache (assume that a cache miss takes 20 cycles)? Which one is better?

Problem 3.E **Victim caches**

In order to improve performance, Ben has decided to add a victim cache. This will be one line of 16 bytes that always holds the most recently evicted line from the main cache. When an item is found in the victim cache, it takes twice as long as the main cache access in order to bring it back into the main cache and return it. Fill out the remaining tables above. If we assume that the result of this analysis represents the average miss-rate for each case, what is the average memory access latency for each configuration with a victim cache? Does victim cache help? Overall, which cache configuration is the best?

State	PseudoCode	ld IR	Reg Sel	Reg W	en Reg	ld A	ld B	ALUOp	en ALU	ld MA	Mem W	en Mem	Ex Sel	en Imm	μB r	Next State
FETCH0	MA ← PC	0	PC	0	1	0	0	*	0	1	*	0	*	0	N	*
	IR ← Mem	1	*	*	0	0	0	*	0	0	0	1	*	0	N	*
	A ← PC	0	PC	0	1	1	0	*	0	0	*	0	*	0	N	*
	PC ← A+4	0	PC	1	1	0	0	INC_A_4	1	0	*	0	*	0	D	*
. . .																
NOP0	microbranch back to FETCH0	0	*	*	0	0	0	*	0	0	*	0	*	0	J	FETCH0
MULU0																

Worksheet 1