Name Solution

Computer Architecture
EE 4720
Final Examination

5 May 1999, 7:30–9:30 CDT

Problem 1		(25 pts)
Problem 2		(25 pts)
Problem 3		(25 pts)
Problem 4		(25 pts)
Exam Total		(100 pts)

Alias

Good Luck!

Problem 1: With the following extension to the DLX ISA a program can get a count of how many times a particular address has been read or written by load and store instructions. The count is kept in a *load/store counter (LSC)* which is incremented whenever a load or store instruction uses a particular effective address. Two new instructions are added, `setlsc` (type I) and `getlsc` (type R). Instruction `setlsc r1,r2+#3` sets the effective address to `r1` and initializes the counter to `r2+#3`. Instruction `getlsc r3` copies the LSC to `r3`.

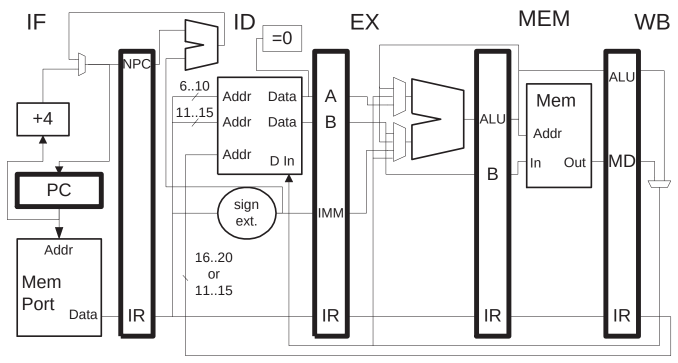
For example, consider the code below. The first instruction, `setlsc`, initializes the count to zero and sets the address to watch to the value of `r1`. When `lw` executes the count will be incremented because the addresses match. If `r1`=`r10-8` then the count will be incremented again when `sw` executes, otherwise the count will remain at one. The `getlsc` instruction will load `r2` with a two, if `r1`=`r10-8`, or a one, otherwise.

```
setlsc r1, r0+#0
lw r5, 0(r1)
sw 8(r10), r11
getlsc r2
add r3, r3, r2
add r4, r4, r2
```

(*a*) Show the changes necessary to add the two instructions to the pipeline below. (The illustration and program are repeated on the next page for convenience.)(20 pts)

- Include the hardware, including control, needed to set, increment, and read the LSC.
- The code above (and any other valid DLX program) must execute correctly and with as few stall cycles as possible.
- The solution can use basic gates, multiplexors, instruction recognizers ($\boxed{\texttt{=getlsc}}$, $\boxed{\texttt{=setlsc}}$, $\boxed{\text{=load/store}}$, etc.), equality testers ($\boxed{=}$), incrementers, and similar logic.
- Equality testers produce their output in $\frac{1}{2}$ cycle, instruction recognizers produce an output in much less than one cycle.

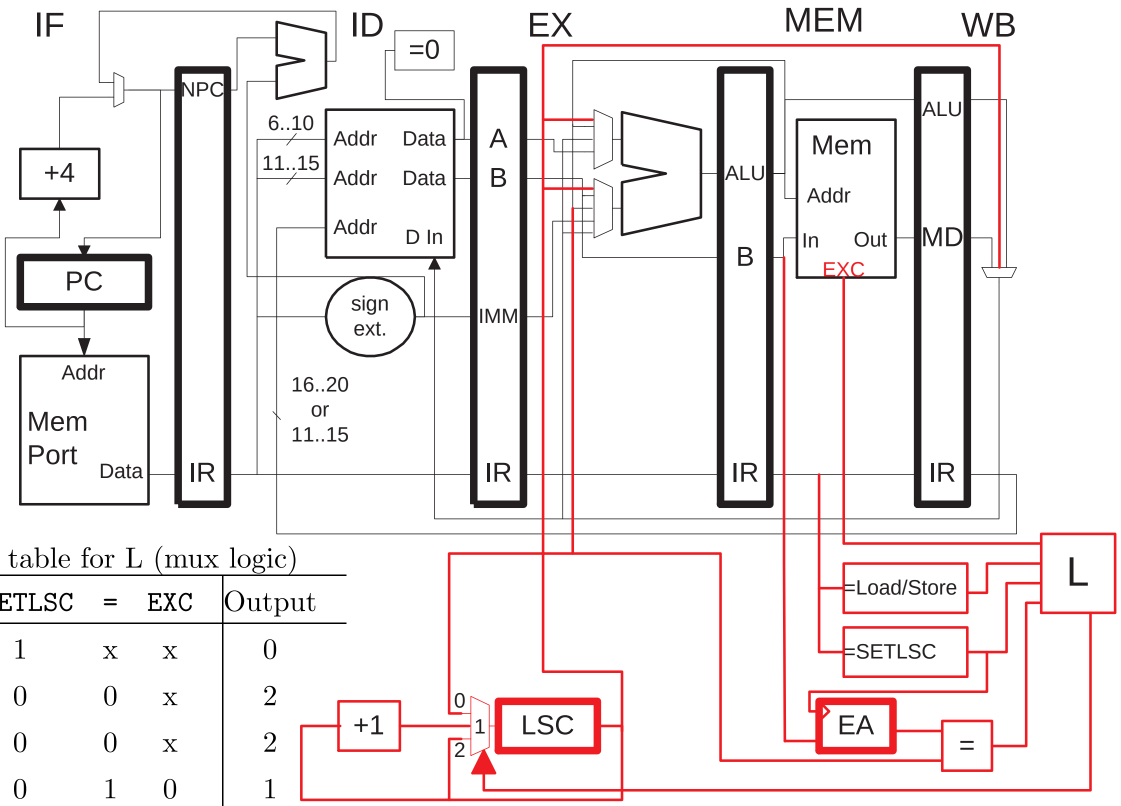
(*b*) If the solution above works correctly when instructions raise exceptions you've solved this part too. Otherwise, explain how an exception could result in incorrect execution and how it might be fixed. (Of course, the pipeline handled exceptions correctly before the changes for the first part.) (5 pts)



Use the next page for the solution.

Problem 1, continued:

```
!Cycle             0   1   2   3   4   5   6   7   8   9
 setlsc r1, r0+#0  IF  ID  EX  MEM WB
 lw r5, 0(r1)          IF  ID  EX  MEM WB
 sw 8(r10), r11            IF  ID  EX  MEM WB
 getlsc r2                     IF  ID  EX  MEM WB
 add r3, r3, r2                    IF  ID  EX  MEM WB
 add r4, r4, r2                        IF  ID  EX  MEM WB
```



Truth table for L (mux logic)

=L/S	=SETLSC	=	EXC	Output
x	1	x	x	0
0	0	0	x	2
1	0	0	x	2
1	0	1	0	1
1	0	1	1	2

Added portions shown above in red. The truth table describes the control logic for the LSC multiplexor. Logic is not shown for the ALU-input and WB-stage multiplexors. The EXC output on the memory is 1 when the memory operation raises an exception.

Operation: the added hardware includes two registers, LSC and EA, both are in the MEM stage.

When a `setlsc` instruction is in the MEM stage the effective address will be in the EX/MEM.B latch and the count value will be in EX/MEM.ALU latch. These values will be clocked into the LSC and EA registers at the end of the cycle.

When a load or store instruction is in the MEM stage the effective address is in the EX/MEM.ALU latch; it is compared to the current value in EA, and based on the comparison LSC is either loaded with the same or an incremented value. (LSC could also have been implemented using a loadable counter.)

When a `getlsc` instruction is in the WB stage the current value of LSC will be selected from the WB-stage multiplexor.

LSC values can be bypassed back to the ALU using the existing WB-to-ALU path and a new LSC-to-ALU path. The new bypass path would be used by the first instruction after `getlsc` in the code above (while `getlsc` is in MEM and the add is in EX). The existing bypass path is used by the second add.

If a load or store instruction raises an exception the memory's EXC output is 1, the LSC multiplexor will select the unchanged value (input 2). If the incremented value were selected (that is, if the exception were ignored), then LSC would be incremented and incremented again when the instruction re-executes. The resulting value would be too high.

Correct operation can be verified by examining the pipeline execution diagram. For example, setlsc initializes LSC before it is incremented by the following lw and after it might have been incremented by any preceding instructions.

Problem 2: Provide pipeline execution diagrams for the code and machines indicated below. All machines have the following features in common:

- Dynamically scheduled processor using register renaming with a reorder buffer. Registers named using reorder buffer entries.
- Branch and branch target prediction (but no branch folding).
- Speculative execution past predicted branches with the reorder buffer used for misprediction recovery.
- Functional unit inputs connect to CDB (and reservation stations).
- Functional units and reservation stations are as listed below. The quantity of functional units is given for both an ordinary single-issue machine and a 4-way superscalar machine.

Quantity		Functional	Abbr.	Latency	Initiation	Reservation
Single	4-Way	Unit			Interval	Station Nums
1	1	Load/Store	L	1	1	0-1
1	4	Integer	EX	0	1	2-3,13-15
1	2	F.P. Add	A	1	1	4-6
1	1	F.P. Mul.	M	7	2	7-8
1	1	Branch	BR	0	1	9-10
1	1	F.P. Divide	DIV	22	23	11-12

(*a*) Show the execution of the code below on a single-issue (not superscalar) machine as described above.

The branch is predicted taken and its target is correctly predicted. However, the branch is in fact not taken. Show execution until the last instruction completes. Show when each instruction commits; indicate canceled instructions with an X. (Note: `ltf f0,f3` sets the floating-point condition register to `f0<f3`; `bfpt SKIP` branches to `SKIP` if the floating-point condition is true. The `ltf` instruction uses the FP add unit. (9 pts)

```
!Cycle             0    1    2    3    4    5    6    7   8   9   10   11  12  13  14  15
 addf  f0, f1, f2  IF   ID   A1   A2   WC
 ltf   f0, f3           IF   ID   RS   A1   A2   WC
 bfpt  SKIP                  IF   ID   RS   RS   BR   WC
 lf    f0, 0(r1)                                          IF  ID  L1   L2  WC
SKIP:
 addf  f4, f0, f5                 IF   ID   A1   A2   WX      IF  ID   RS  A1  A2  WC
 addi  r1, r1, #4                      IF   ID   EX   X           IF   ID  EX  WB      C
```

(*b*) Show the execution of the code below on a single-issue (not superscalar) machine as described above. The branch is correctly predicted not taken. Show execution until the last instruction completes. Show when each instruction commits. Show the state of the reorder buffer when `addf` reaches writeback. (8 pts)

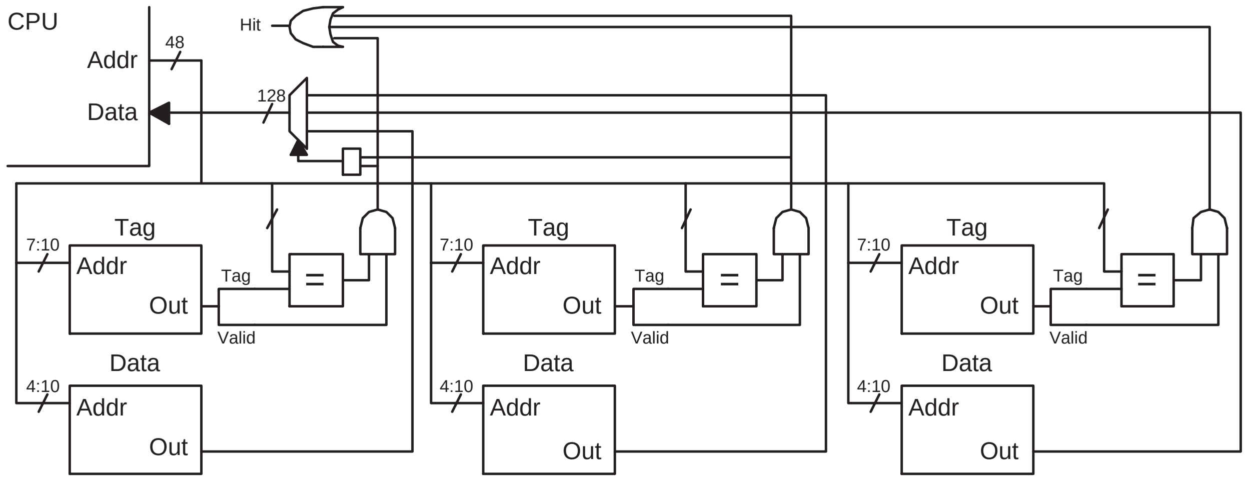
```
!Cycle             0  1   2    3    4    5    6    7    8    9    10   11  12  13 14 15 16
 multf f0, f1, f2  IF ID  M1   M1   M2   M2   M3   M3   M4   M4   WC
 ltf   f0, f3         IF  ID 4:RS 4:RS 4:RS 4:RS 4:RS 4:RS 4:RS   A1   A2  WC
 bfpt  SKIP               IF   ID 9:RS 9:RS 9:RS 9:RS 9:RS 9:RS 9:RS 9:RS  BR  WC
 lf    f0, 0(r1)               IF   ID   L1   L2   WB                             C
SKIP:
 addf  f4, f0, f5                   IF   ID 4:RS   A1   A2   WB                      C
 addi  r1, r1, #4                        IF   ID   EX   WB                              C
```

(*c*) Show the execution of the code below on a 4-way superscalar processor as described above. Instructions are fetched in aligned blocks of four. The branch is correctly predicted taken. All targets are correctly predicted. Show execution until the last instruction completes. Show when each instruction commits. (8 pts)

```
LINE0: ! LINE0 = 0x100c
 beqz r1, LINE1   IF  ID  BR  WC
 addf f0, f0, f1      IF  x
 j LINE2              IF  x
LINE1
 addf f0, f0, f2      IF  ID  A1   A2   WC
 add  r2, r2, r3      IF  ID  EX   WB   C
 and  r2, r2, r4          IF  ID   EX   WC
LINE2
 addf  f0, f0, f3         IF  ID 4:RS   A1    A2   WC
 addf  f5, f5, f0         IF  ID 5:RS 5:RS  5:RS   A1   A2   WC
 addf  f6, f6, f0         IF  ID 6:RS 6:RS  6:RS   A1   A2   WC
 addf  f7, f7, f0             IF   ID 4:RS  4:RS 4:RS   A1   A2  WC
 addf  f8, f8, f0             IF   ID ---------> 5:RS   A1   A2  WC
 and   r5, r6, r7             IF   ID   EX    WB                C
 or    r8, r9, r10            IF   ID   EX    WB                C
 xor   r11, r12, r13               IF --------->   ID   EX    WB      C
 sub   r14, r15, r16               IF --------->   ID   EX    WB      C
```

Problem 3:

(*a*) A system is equipped with the cache shown below.



Determine a value for each of the following. **Be sure to specify units (bits, bytes, etc.)** (10 pts)

Character Size (Size of item at a single address.):8 bits

Associativity:3

Block Size:128 bytes or 1024 bits

Number of Sets: 16

Cache Capacity (Amount of data that can be stored.):$3\times = 6144$ bytes

Memory Needed to Implement Cache: $3 \times 16 \times (128 \times 8 + (37 + 1))$ = 50976 bits = 6372 bytes

The problems on this page are for the cache described below, **not** *the cache from the previous page.*

Consider a system with a 64-bit address space ($a = 64$) which addresses the usual 8-bit (1-byte) characters ($c = 8$) and is equipped with a 2-way set-associative cache with 64-byte lines and 256 sets. The cache uses LRU replacement.

(b) Initially the cache is empty. What is the hit ratio for accesses to the array in the code below. Ignore all other memory accesses. (7 pts)

```
extern char *the_array;
// Note: sizeof(char) = 1 byte.
// &the_array[0] = 0x10000
for(i=0; i<16; i++) a += the_array[ i * 8 ];
```

Hit ratio will be $\frac{7}{8} = 0.875$.

(c) Choose values for `i_limit` and `stride` so that the program below completely fills the cache with the minimum number of accesses. Assume the cache is initially empty and only include accesses to the array. (8 pts)

```
extern char *the_array;
// Note: sizeof(char) = 1 character. (What else?)
// &the_array[0] = 0x10000

int i_limit =

int stride  =

for(i=0; i<i_limit; i++) a += the_array[ i * stride ];
```

Solution: `i_limit` = 512 and `stride` = 64.

Problem 4: Answer each question below.

(*a*) Show how the DLX instructions below are encoded. That is, write the instructions as numbers. Make up numbers for opcode and func fields, but use actual values for other fields. *Hint: If you can't remember field sizes look at the illustration for problem 1.* (5 pts)

```
 beqz r7, SKIP
 add r1, r2, r3
 xori r4, r5, #6
SKIP:
```

opcode	rs1	rd	immediate
beqz	7	?	8
0 5	6 10	11 15	16 31

opcode	rs1	rs2	rd	func
Type R	2	3	1	add
0 5	6 10	11 15	16 20	21 31

opcode	rs1	rd	immediate
xori	5	4	6
0 5	6 10	11 15	16 31

(*b*) Describe an advantage of VLIW over superscalar processors. Describe a disadvantage of VLIW over superscalar processors. (Use the VLIW ISA described in class.) (5 pts)

Advantage of VLIW: dependencies specified, superscalar machine devotes alot of silicon to finding dependencies. Advantage of superscalar: compatible with existing code.

(*c*) What is a translation lookaside buffer (TLB) and what does it do? (5 pts)

It is a cache indexed by virtual page number that stores real page numbers.

(*d*) Show how an address can be constructed for the branch history table in an (m, n) two-level correlating branch predictor. (Two ways were presented in class, show either one.) (5 pts)

Maintain the outcome of the last m branches in a shift register. Exclusive or the contents of the shift register with the low-order m bits past the alignment bits of the address of the branch instruction being predicted. Voilá.

(*e*) Re-write the program below using DLX instructions. Additional registers may be used. (5 pts)

```
lw  r1,@(r2)         ! Memory indirect addressing.
add r3, r4, (r5)     ! Register indirect addressing.
ld  f0, (0xfedcba01) ! Direct addressing.
```

Solution:

```
lw  r21, 0(r2)
lw  r1, 0(r21)
lw  r25, 0(r5)
add r3, r4, r25
lh  r22, 0xfedc
ld  f0, 0xba01(r22)
```