Name Solution

Computer Architecture
EE 4720
Midterm Examination

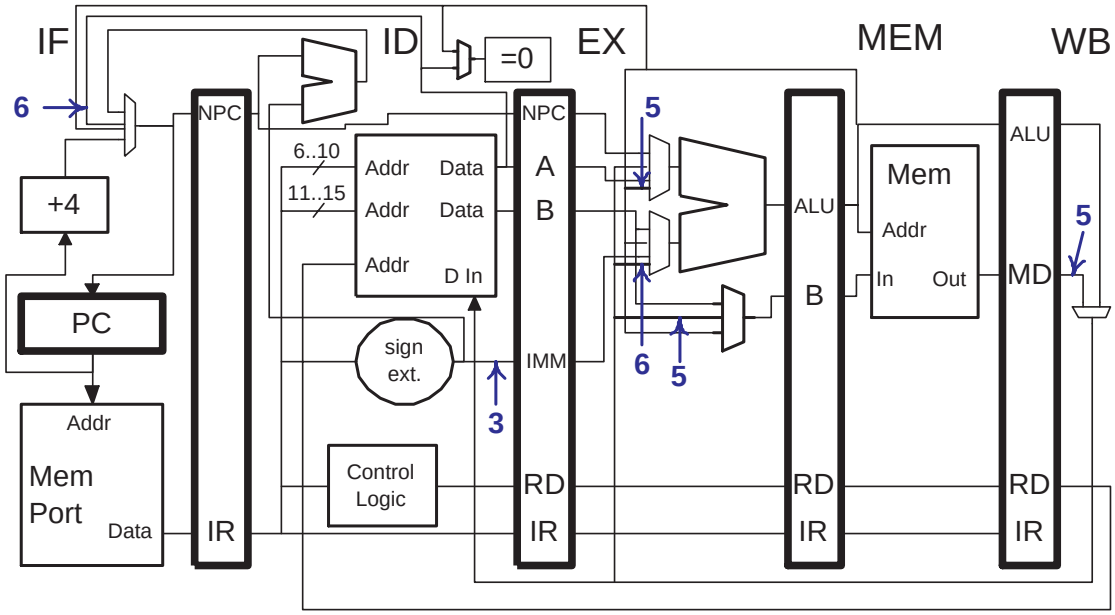
Wednesday, 21 March 2001, 13:40–14:30 CST

Problem 1		(35 pts)
Problem 2		(25 pts)
Problem 3		(40 pts)
Exam Total		(100 pts)

Alias Farewell, Mir!

Good Luck!

Problem 1: In the DLX implementation below six paths are marked with a number, a cycle in which the path will be used. Write a DLX program that uses those six paths at the indicated cycles. Some marked paths are *bypass* paths and some are not; the bypass paths can be used **only** at the cycles indicated. A pipeline execution diagram is provided for your convenience.



☑ [15 pts] Choose the register operands so the bypass paths are used **only** at the cycles indicated.

☑ [15 pts] Choose the instructions so that the marked paths will be used as indicated. There need be only one control transfer instruction.

☑ [5 pts] One instruction will be squashed. Show where by crossing out the segment labels (`ID`, etc.) in the diagram.

Solution:

```
    ! Cycle     0   1   2   3   4   5   6   7   8   9   10  11  12
nop             IF  ID  EX  MEM WB

lw   r1,0(r20)      IF  ID  EX  MEM WB

addi r2, r21, #1        IF  ID  EX  MEM WB

sw   0(r2), r1              IF  ID  EX  MEM WB

add  r22, r23, r2               IF  ID  EX  MEM WB

jr   r24                            IF  ID  EX  MEM WB

nop                                     IFx

nop                                             IF  ID  EX  MEM WB
    ! Cycle     0   1   2   3   4   5   6   7   8   9   10  11  12
```

Problem 2: The DLX code below runs on a dynamically scheduled system that uses reorder buffer entries to name destination registers (Method 1). The floating-point adder has **five stages**, `A1` through `A5`, and is fully pipelined. The common data bus (CDB) can handle an unlimited number of writebacks per cycle, but other parts of the implementation are ordinary. The cache is non-blocking. The `cvtftoi` instruction uses the FP adder; the `movfptoi` instruction uses the integer (`EX`) functional unit. The pipeline is fully bypassed.

(*a*) For this part no instructions raise exceptions.

☑ [7 pts] Show a pipeline execution diagram for the code. Reorder buffer and reservation station numbers **do not** have to be shown.

☑ [5 pts] Indicate where each instruction commits.

Check the code carefully for dependencies! Register `r1` *is part of a long chain of dependencies. Pay attention to the register equality and inequality comment.*

```
! Solution
! r4 = r2, r6 != r2, r6 != r1
!Cycle           0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
addf f0, f1, f2  IF ID A1 A2 A3 A4 A5 WC
cvtftoi f0, f0      IF ID RS          A1 A2 A3 A4 A5 WC
movdtoi r1, f0         IF ID RS                      EX WC
subd f0, f4, f6           IF ID A1 A2 A3 A4 A5 WB          C
sd 0(r1), f10                IF ID RS                   L1 L2 WC
sd 0(r2), f0                    IF ID L1       L2 WB             C
ld f12, 0(r4)                      IF ID L1       L2 WB             C
ld f14, 0(r6)                         IF ID L1                L2 WB   C
!Cycle           0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
```

Notes on the solution: Load and store instructions compute their effective addresses (use the `L1` stage) as soon as possible. The first `sd` does not move in to the `L1` stage until the rs1 operand, `r1`, is available in cycle 13. The second `sd` moves in to `L1` in cycle 7 but waits for the store value before moving to `L2`. The first load uses the `L2` stage in cycle 11 to bypass a result from the second `sd` (since their effective addresses are the same). The second `ld` must wait for the address of the first store to be computed. Without that address there is no way to tell whether the first store writes to the same location that the second load reads from.

(*b*) Suppose an arithmetic exception is discovered for the `subd` instruction when it is in the first FP adder stage, `A1`, in an execution of the program above.

☑ [6 pts] At what cycle is the reorder buffer flushed and the handler fetched?

When `subd` reaches the head of the reorder buffer, at cycle 14.

☑ [7 pts] Show the contents of the reorder buffer when the status of the `subd` instruction is set to exception. (For partial credit show the contents of the reorder buffer halfway through execution, be sure to indicate the cycle.)

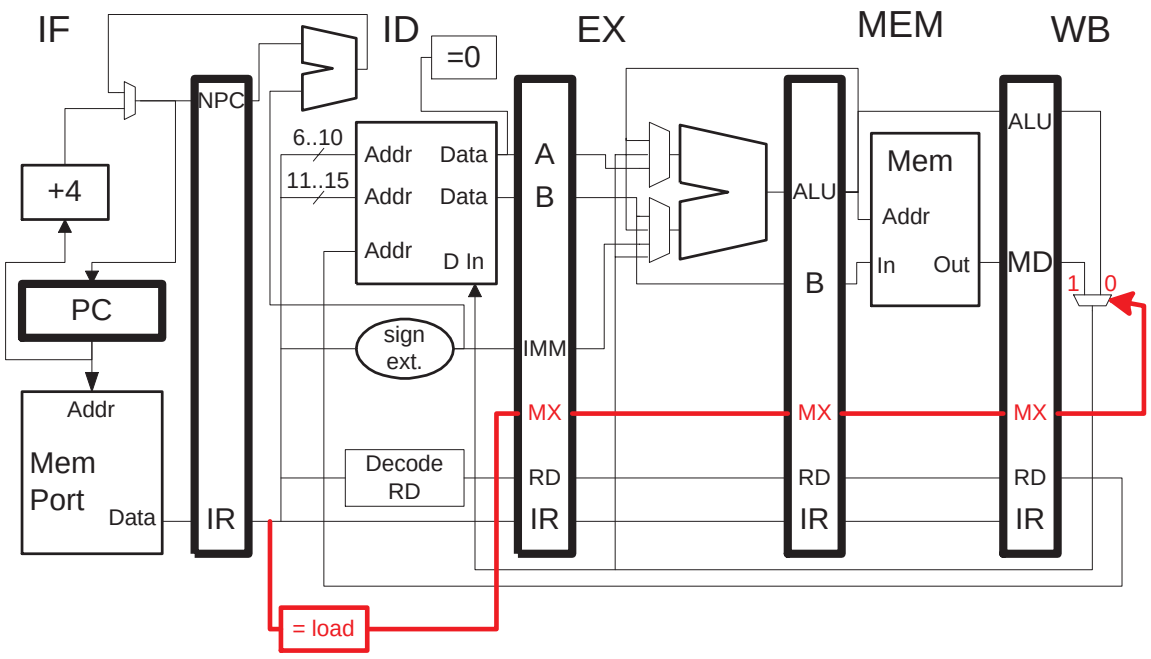
The status is set to exception when `subd` is in writeback, at cycle 10. The contents of the reorder buffer from head (next to commit) to tail (last to enter) is: `cvtftoi`, `movdtoi`, `subd`, `sd`, `sd`, `ld`, `ld`.

Problem 3: Answer each question below.

(*a*)

☑ [8 pts] Design the control logic for the WB-stage multiplexor. The control logic itself must be in the ID stage. Show the connection to the mux and number the mux inputs. *Hint: This is an easy problem.*

Changes shown in red.



(*b*) In the pipeline execution diagram below `multd` is delayed because of a true dependency with the `addd` instruction.

☑ [8 pts] Why can't any realistic implementation do things this way?

```
addd  f0, f2, f4   IF ID A1 A2 A3 A4 WB
multd f6, f0, f8               IF ID M1 M2 M3 M4 M5 M6 WB
subd  f10, f12, f14               IF ID A1 A2 A3 A4 WB
```

Because the true dependency could not be detected before the instruction is fetched and decoded.

(*c*) Unlike DLX, many ISAs, such as SPARC, use a condition code register for integer branch conditions.

☑ [4 pts] In what way is DLX's method more flexible?

Since any register can be used for a condition the code can test one condition, another condition, and then the first one without recomputing it.

☑ [4 pts] In what way is a condition code register more flexible?

Condition code registers typically have flags indicating that a value is negative, zero, overflowed, etc, and these can be set by an instruction doing normal computation. A branch instruction can test various combinations of these conditions. After executing a single set condition-code instruction, several different tests can be made on the conditions, such as ≥ 0, overflow, etc.

(*d*) An ISA may have variable-width instructions, fixed-width instructions, and bundled instructions.

☑ [4 pts] How do the different alternatives affect displacement branches?

If the instructions are variable width then the displacement in displacement branches must be the number of characters away the branch target is. If the instructions are fixed width the displacement is the number of instructions to skip, which spans a larger area of memory (assuming instructions are aligned). With bundled instructions the displacement is the number of bundles to skip, and so an even larger area of memory is covered.

[4 pts] Name an ISA category (type) that uses each instruction format:

☑ Variable-Width Instructions:

CISC

☑ Fixed-Width Instructions:

RISC

☑ Bundled Instructions:

VLIW

(*e*) Packed-operand data types and instructions are *de rigueur* for any *au courant fin-de-siècle* ISA. (Are a must-have for any up-to-date end-of-the-century ISA.)

☑ [8 pts] What are packed-operand data types and instructions? Show a short program that would benefit from these. The program can be in a high-level language or even pseudo code. Do not show the packed-operand instructions, just explain what they would do.

A data type in which several values are held in a single register, for example four eight-bit values in a 32-bit register. A packed-operand instruction operates on all of the values in parallel. For example, a single `add` would, using the last example, sum four pairs of eight-bit numbers.

```
extern unsigned char *a, *b, *c;
for(i=0; i<1024; i++) a[i] = b[i] + c[i];
```