Name Solution

Computer Architecture
EE 4720
Final Examination

11 December 2001, 7:30–9:30 CST

Problem 1		(20 pts)
Problem 2		(20 pts)
Problem 3		(20 pts)
Problem 4		(40 pts)
Exam Total		(100 pts)

Alias Solution!!!

Good Luck!

Problem 1: The code fragment below executes as shown on a single-issue dynamically scheduled system using a physical register file (Method 3). Initially register `f0` contains a 0, `f2` contains a 0.2, and `f4` contains a 0.4. The value computed by each instruction is shown near the right margin. (20 pts)

☑ Show where each instruction commits.

☑ Complete the ID- and commit-stage register map tables.

☑ Complete the physical register file table.

☑ Be sure to show the initial values for `f0`, `f2`, and `f4` in the register maps and the register file.

☑ In the physical register file use a [to indicate when a register is removed from the free list and use a] to indicate when it's put back.

```
!Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22
addd f0, f0, f2    IF ID Q                          A1 A2 WC                         (.2)
addd f4, f2, f4       IF ID Q  A1 A2 WB                      C                       (.6)
addd f2, f0, f8          IF ID Q                          A1 A2 WC                   (1.)
addd f0, f4, f8             IF ID Q  A1 A2 WB                      C                 (1.4)
addd f4, f2, f8                IF ID Q                          A1 A2 WC             (1.8)
!Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22
```

```
 !Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22
 ID:
 f0   0                3        6
 f2   1                      5
 f4   2                   4        7
 !Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22
 Commit:
 f0   0                                                    3        6
 f2   1                                                          5
 f4   2                                                       4        7

 !Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22
 PRF:
 0    0                                                    ]
 1    .2                                                         ]
 2    .4                                                      ]
 3                     [                                   .2       ]
 4                        [           .6                               ]
 5                           [                                   1.
 6                              [           1.4
 7                                 [                                   1.8
 8
 9
 !Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22
```

Problem 2: The code fragment below executes on three different systems, I, L, and G, each using a different branch predictor:

I: One-level predictor, 2^{16}-entry BHT. L: Sixteen-bit (two-level) local history. G: Sixteen-bit (two-level) gshare.

In the spaces provided below show the accuracy of, and the number of entries used by, the indicated branch on the indicated system. The accuracy and number of entries should be after warmup. The number of entries used for the gshare scheme can be approximate. If there is more than one table, show the number of entries in each table separately.(20 pts)

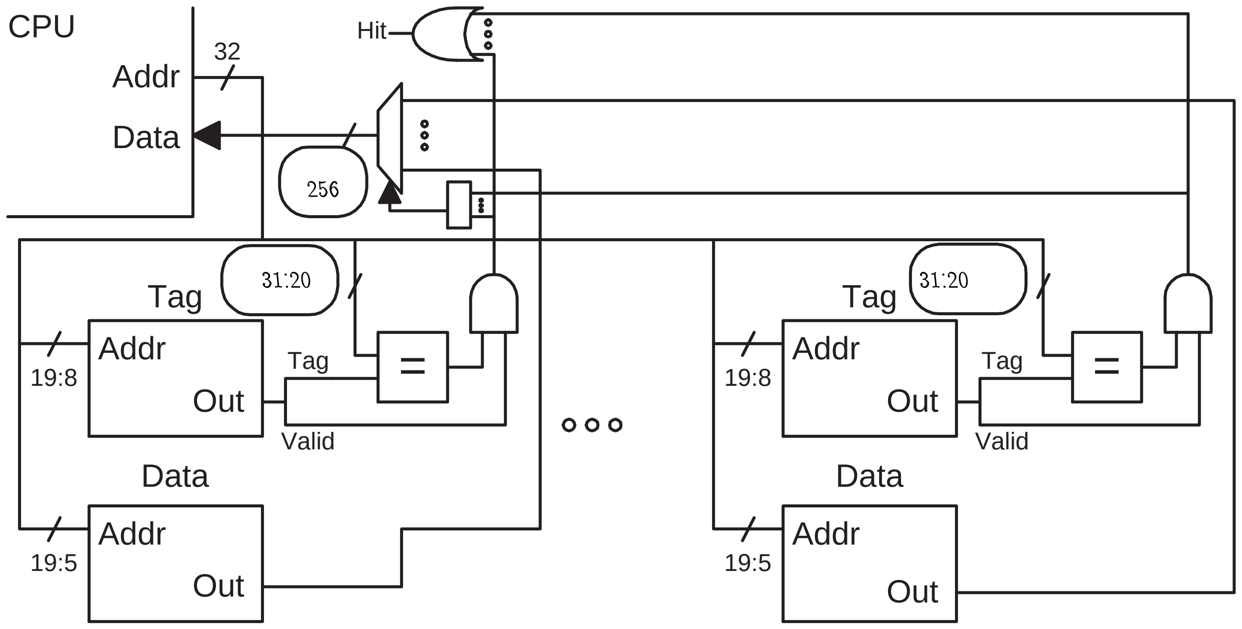
✓ B1 on I: Accuracy: 50%	Entries: One entry.
✓ B2 on I: Accuracy: 99%	Entries: One entry.
✓ B3 on I: Accuracy: $66\frac{2}{3}$%	Entries: One entry.
✓ B1 on L: Accuracy: 50%	Entries: $1 + 2^{16}$
✓ B2 on L: Accuracy: 99%	Entries: $1 + 17$
✓ B3 on L: Accuracy: 100%	Entries: $1 + 3$
✓ B1 on G: Accuracy: 50%	Entries: $\approx 2^8$
✓ B2 on G: Accuracy: 99%	Entries: $\approx 2^8$
✓ B3 on G: Accuracy: 100%	Entries: $\approx 1 + 2^9$

```
BIGLOOP:

LOOP2: ! Iterates 100 times
 ! Random, 0 or 1
    lw r1, 0(r2)
    addi r2, r2, #4
B1: bnez r1, SKIP1
    add r10, r10, r11
SKIP1:
    sub r3, r2, r12
B2: bnez r3, LOOP2

    addi r1, r0, #3
LOOP3:
    subi r1, r1, #1
B3: bnez r1, LOOP3

j BIGLOOP
```

Problem 3: The diagram below is for an 8 MiB (2^{23} byte) set-associative cache for a system with 8-bit (how ordinary) characters.

(*a*) Answer the following, formulæ are fine as long as they consist of literals and grade-time constants. (10 pts)

☑ Fill in the blanks in the diagram.



☑ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

	Tag	Index	Offset
Address:	31 … 20	19 … 8	7 … 5 | 4 … 0

☑ Associativity:

Eight-way.

☑ Line Size (Indicate Unit!):

It's 2^8 characters.

☑ Memory Needed to Implement (Indicate Unit!):

It's 2^{23} characters (data) plus $8 \times 2^{12}(12 + 1)$ bits for the tag store.

☑ Show the bit categorization for a direct mapped cache with the same capacity and line size.

	Tag	Index	Offset
Address:	31 … 23	22 … 8	7 … 5 | 4 … 0

Problem 3, continued: Continue to use the set-associative cache from the previous part.

When the code below starts running the cache is empty. Consider only accesses to the array.

```
double *d = 0x100000;  // sizeof(double) = 8 characters
double sum;
int i;
int ISTRIDE = 1;
int ILIMIT = 64;

for(i=0; i<ILIMIT; i++)
  sum += d[ ISTRIDE * i ];
```

(*b*) Answer the following. (10 pts)

✓ What is the hit ratio for the program above?

It's $\frac{31}{32}$. (Note that the size of a double is 8 characters.)

✓ Modify `ISTRIDE` and `ILIMIT` so that the cache, which remember is set-associative, is completely filled *with the minimum number of accesses.*

Set `ISTRIDE` so that successive values of `i` access successive lines. That would be the line size divided by the array element size, $\texttt{ISTRIDE} = \frac{2^8}{2^3} = 2^5$. Set ILIMIT to the number of lines, which is the associativity times the number of sets (two to the power of the number of index bits): $\texttt{ILIMIT} = 8 \times 2^{20-8} = 2^{3+20-8} = 2^{15}$.

Problem 4: Answer each question below.

(*a*) The contents of the load/store queue in a dynamically scheduled system is shown below for $t = 4720$. At this particular time the cache is empty, but with a load/store queue full of instructions it won't be empty for long. The instruction in entry `L0` is the oldest.

The EA field indicates the effective address, a `#1` in this field indicates that the effective address cannot be computed until the instruction in ROB entry #1 (not shown) completes. It completes at $t = 7700$.

The Data field indicates the data that will be written by the store instruction in the entry.

The cache is nonblocking and the load/store queue can process an unlimited number of instructions per cycle.

☑ (5 pts) For each load instruction at $t = 4721$ if it's complete show what data it has loaded otherwise indicate the first thing it's waiting for.

Solution under the "Data loaded or reason for waiting" column.

```
    Type  EA      Data   Data loaded or reason for waiting.
L8: load  #1             Need value from instruction in ROB entry #1 for EA
L7: load  0x1000         Loaded 55
L6: store 0x2000  66
L5: store 0x1000  55
L4: load  0x1000         Waiting for L3 to compute its address (could be 0x1000)
L3: store #1      33
L2: store 0x1000  22
L1: load  0x2000         Waiting for the cache.
L0: load  0x1000         Waiting for the cache.
```

(*b*) The exception recovery mechanism in the R10000 does not need a commit map, the one in Method 3 does.

☑ (5 pts) But the R10000 takes longer to recover. Why does it take longer and how much longer does it take?

It takes longer because rather than replacing the ID map with a commit map it re-creates the older ID map by reversing the changes made by the soon-to-be-squashed instructions four at a time. If there are sixteen instructions in the ROB that would take 4 steps.

(*c*) In some schemes for recovering from branch mispredictions on dynamically scheduled systems recovery can start at completion rather than waiting for commitment.

☑ (5 pts) What additional hardware is needed for this? How is it used?

Storage for backup copies of the ID map. When a branch is predicted a backup copy of the ID map is stored. In the writeback stage of a mispredicted branch the ID map is replaced by the backup copy corresponding to the mispredicted branch.

(*d*)

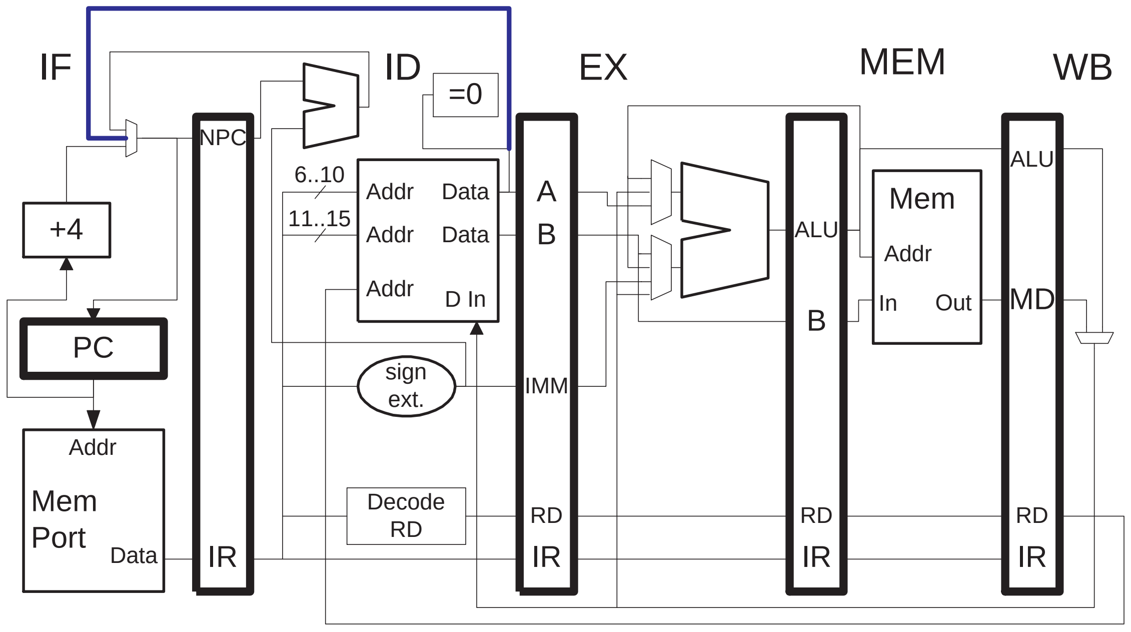
☑ (5 pts) Show how bits are categorized in a real address and a physical address in a system with a 64-bit virtual address space, a 40-bit real address space and 2^{14}-character pages.

Bits 0 to 13 in real and virtual addresses are the offset. In a virtual address bits 14 to 63 are the virtual page number and in a real address bits 14 to 39 are the real page number.

(*e*) Connections for the `jr` instruction are omitted from the pipeline below.

☑ (5 pts) Add them.

Change shown in blue below.



☑ (5 pts) Show a pipeline execution diagram for the code below consistent with the modified hardware above.

```
 slli r10, r1, 2    IF ID EX ME WB
 addi r10, r10, r2     IF ID EX ME WB
 lw   r10, 0(r10)         IF ID EX ME WB
 jr   r10                    IF ID ----> EX ME WB
 ...                            IF ---->x

TARG: !Target of jr.
 add r20, r21, r22                       IF ID EX ...
```

(*f*) Here are the IA-64 instructions used in the solution to Homework 3: `ld1` (load byte), `cmp.eq` (compare equal), `cmp.le` (compare less than or equal), `cmp.gt` (compare greater than), `add` (add of course), `st1` (store byte), and `br` (branch).

☑ (5 pts) Convert the DLX code below to IA-64, be sure to use predicated instructions.

☑ Show the stops.

```
 sle r1, r2, r3
 beqz r1, SKIP
 add  r10, r10, r11
 j CONT
SKIP:
 sub r12, r12, r13
CONT:
 add r14, r12, r10
```

```
      cmp.le  p1,p2 = r2, r3;;
 (p1) add r10 = r10, r11
 (p2) sub r12 = r13, r13;;
      add r14 = r12, r10
```

☑ (5 pts) Besides needing fewer instructions, how does predication speed execution? (Not necessarily in IA-64 implementations.) Modify the DLX program (without changing the branch and jump) so that predication would be less useful.

With predication branches can be eliminated, along with their inefficiencies. The inefficiencies include the time needed to fetch the target and the un-executed instructions in the fetch group following a taken branch.

```
 sle r1, r2, r3
 beqz r1, SKIP
 add  r10, r10, r11
 j CONT
SKIP:
 sub r12, r12, r13
CONT:
 add r14, r12, r10
```