

# GEORGIA INSTITUTE OF TECHNOLOGY

College of Computing

## CS6290/CS4290 — High-Performance Computer Architecture

Fall 2000

---

CS6290/CS4290  
Homework 7

Issued: October 29, 2000  
Due: November 10, 2000

---

**Purpose:** This homework covers predication and introduces some issues in network performance.

**Reading:** H&P Sections 4.6, 7.1 and 7.2  
[Kubi93] paper

**Problems:**

1. Predication.
2. Effective Bandwidth.
3. Reading: [Kubi93].

**Collaboration:** (*As in the syllabus*) collaboration on projects and homework in **pairs** is encouraged. If you work in a pair, turn in one write-up with the names of both collaborators. You're welcome to discuss high-level concepts with other groups, but all homework solutions must be worked out and written up separately.

## Problem 1: Predication

This problem explores *predication* as an alternate to branches for encoding control flow.

Consider the following situation: here's a data structure for holding a value in multiple forms along with an accessor function for interpreting the data as a “double” regardless of its stored form.

```
struct blah
{
    int type;           /* +0:  INTEGER, FLOATING or COMPLEX */
    int ivalue;        /* +4:  value for INTEGER */
    double realvalue;  /* +8:  value for FLOATING or real part of COMPLEX */
    double imagvalue; /* +16: imaginary part for COMPLEX */
};

double magnitude(struct blah *ptr)
{
    double result;

    if (ptr->type == FLOATING)
        result = ptr->realvalue;
    else if (ptr->type == INTEGER)
        result = float(ptr->ivalue);
    else if (ptr->type == COMPLEX)
        result = sqrt(ptr->realvalue * ptr->realvalue
                      + ptr->imagvalue * ptr->imagvalue);
    else
        assert(0);

    return(result);
}
```

Here is an instruction sequence in DLX assembly for implementing the sequence. The target processor is a 2-way static superscalar – it can execute up to two instructions per cycle but cannot reorder them. I’ll show the whole scheduling process to hopefully make the reasoning plain.

Notes:

- The standard DLX instructions are summarized on p. 96 in the book.
- The (integer) argument is passed in R1.
- The (double floating) result is returned in F0.

- JR R31 implements `return()`.
- SQRD is what it sounds like.
- HACF, halt-and-catch-fire, will serve to implement `assert(0)`
- I'm assuming a one-cycle stall for loads and a (very modest) one-cycle stall FOP->FOP as well. I don't have to worry about the delay of SQRD in this code because it's not used in this code.

Step 1: here's a version of the code written sequentially. Convince yourself that this works.

```

magnitude:    LW      R2, [R1+0]
              SUBI    R3, INTEGER, R2
              BE      R3, isinteger

              SUBI    R3, FLOATING, R2
              BE      R3, isfloating

              SUBI    R3, COMPLEX, R2
              BE      R3, iscomplex

isbroken:     HACF

isinteger:    LF      F0, [R1+4]
              CVTI2D  F0, F0
              JR      R31

isfloating:   LD      F0, [R1+8]
              JR      R31

iscomplex:    LD      F0, [R1+8]
              LD      F2, [R1+16]
              MULD    F0, F0, F0
              MULD    F2, F2, F2
              ADDD    F0, F0, F2
              SQRD    F0, F0
              JR      R31

```

Step 2: here's the same code run on a two-way static superscalar with stalls indicated (“--”).

```

magnitude:    LW      R2, [R1+0]          --
              --                          --
              SUBI   R3, INTEGER, R2     --
              BE     R3, isinteger       SUBI   R3, FLOATING, R2
              BE     R3, isfloating      SUBI   R3, COMPLEX, R2
              BE     R3, iscomplex       --

isbroken:     HACF                          --

isinteger:    LF     F0, [R1+4]          --
              --                          --
              CVTI2D F0, F0              JR     R31

isfloating:   LD     F0, [R1+8]          JR     R31

iscomplex:    LD     F0, [R1+8]          LD     F2, [R1+16]
              --                          --
              MULD   F0, F0, F0          MULD   F2, F2, F2
              --                          --
              ADDD   F0, F0, F2          --
              --                          --
              SQRTD  F0, F0              JR     R31

```

Step 3: the stalls in the code above suggest opportunities to reschedule the code without introducing any additional delays. Here's a minor scheduling improvement: pull the structure loads up into the stall slots at the top of the schedule:

```

magnitude:    LW      R2, [R1+0]          LF      F0, [R1+4]
              LD      F2, [R1+8]          LD      F4, [R1+16]
              SUBI    R3, INTEGER, R2    --
              BE      R3, isinteger      SUBI    R3, FLOATING, R2
              BE      R3, isfloating     SUBI    R3, COMPLEX, R2
              BE      R3, iscomplex      --

isbroken:     HACF                      --

isinteger:    CVTI2D  F0, F0            JR      R31

isfloating:   MOVE    F0, F2            JR      R31

iscomplex:    MULD    F0, F2, F2        MULD    F2, F4, F4
              --
              ADDD    F0, F0, F2        --
              --
              SQRD    F0, F0            JR      R31

```

Use the last schedule above for the first three questions below (A, B and C).

**A:** The length of this instruction sequence in cycles depends on the mix of structure types with which it is called. How many cycles are required if the structures are all the same type? Assume that since the incoming types are all the same, the branches can be predicted perfectly:

- i. Best-case cycles for INTEGER structure
- ii. Best-case cycles for FLOATING structure
- iii. Best-case cycles for COMPLEX structure

**B:** The cycle counts computed in the previous question were only best cases. If the branch mispredict penalty is 5 cycles, what are the *worst case* times to compute a result for each of the structure types?

- i. Worst-case cycles for INTEGER structure
- ii. Worst-case cycles for FLOATING structure
- iii. Worst-case cycles for COMPLEX structure

Now, reconsider the instruction sequence when the type mix is exactly even: each valid type arrives with a probability of 1/3. There are no invalid types. To make things as simple as possible, assume the branch predictors are one-bit: they predict whichever way the branch went last time. In that scenario, the branches predictors will predict taken with the following probabilities:

- The first branch is predicted taken with probability 1/3.
- The second branch is predicted taken with probability 1/2. Note that `INTEGER` types never reach this branch, so half the branches that do are of type `FLOATING`.
- The third branch is predicted taken with probability 1.

**C:** If the branch mispredict penalty is 5 cycles, compute the average execution time for the block when each valid type arrives with probability 1/3. How much of this time is due to misprediction penalties?

**D:** Reschedule the code for best performance in the scenario in which each valid type arrives with probability 1/3.

Finally, consider a DLX architecture extended with support for *predication*. With predication, every instruction is made conditional: the instruction always goes through the execute stage of the processor but we make a decision at the last stage whether to commit the results of the instruction or not.

For our predicated DLX (DLX-P), we'll assume every instruction is extended with an additional register number and a `Z/  $\overline{NZ}$`  (zero/not zero) bit.<sup>1</sup> The instruction is executed if the indicated register matches the `Z/  $\overline{NZ}$`  bit.

; Examples:

```

ADD R1, R2, R3    (R4:NZ)    ; commits only if R4 != 0
BR loop          (R4:Z)      ; replaces 'BNE R4, loop'
ADD R1, R2, R3    (R0:Z)      ; always executed (R0 is zero)
ADD R1, R2, R3    ; usual way to write the above
ADD R1, R2, R3    (R0:NZ)    ; never executed
NOP              ; usual way to write the above

```

The standard way to apply predication in a case like this is to remove *all* branches and instead use predication to choose the result. The main advantage of this approach is that no prediction is required so that misprediction penalties are removed.

<sup>1</sup> Never mind that this change adds six bits to our 32-bit instructions.

**E:** Write a predicated version of the above code. How many cycles are required when for each structure type? How many cycles are required on average when each structure type arrives randomly with probability  $1/3$ ?

**F:** Finally, a compiler must choose whether to emit code with branches or code using predication. What general rules should the compiler use to make this decision?

## Problem 2: Effective Bandwidth

**A:** Problem 7.1 in the book. Also, what message size is required to achieve half of the boilerplate<sup>1</sup> bandwidth of the network (half of 9Mbits/sec, according to the problem)?

**B:** What is the bandwidth of FedEx when used to deliver a box containing 100GB (an even  $10^{11}$  bytes) on tape sent from San Jose to Atlanta assuming 6pm dropoff and 10am delivery, local time? Note that San Jose's time zone is three hours later. What is the bandwidth the other direction?

## Problem 3: Reading

Read the [Kubi93] paper on the Alewife message passing mechanism.

**A:** Figure 8 shows the instruction sequence for composing and launching a message with a one-word payload of data where the data is written directly out of a register. Show the sequence for sending a message with an additional payload of 256 bytes from location `foo` in memory.

**B:** Figure 8 also shows the detailed pipeline timing for launching a message: three cycles per `stio` plus one cycle for the `ipilaunch` plus two cycles of overhead. The processor runs at 33MHz; the network is byte-wide and runs at 66MHz (528Mbit/S). The instruction sequence shown launches one 32-bit word of payload in nine cycles. What bandwidth (in Mbits/S) is achieved if we repeat this sequence continuously? What is the maximum bandwidth if we send longer messages via the same mechanism (more `stios` from registers)?<sup>2</sup> What length of message is required to achieve half the boilerplate bandwidth of the network (half of 528Mbit/S)?

---

<sup>1</sup> That's "boilerplate" as in "maximum"; from the specification plate on a steam engine.

<sup>2</sup> The network is sufficiently fast that the processor is always the bottleneck.