

CS 211 - HOMEWORK 3 Solutions

Ques.1: We use the notation $i \rightarrow j$ to mean statement j is control dependent on statement i .

(a) Control Dependencies: Do data references allow scheduling the dependent statement before the IF statement.

- $1 \rightarrow 2$ Yes.
- $1 \rightarrow 3$ No - could change the IF outcome
- $1 \rightarrow 4$ Yes
- $1 \rightarrow 5$ Yes
- $1 \rightarrow 6$ No - could change the IF outcome

(b) Data dependencies

- $1 \rightarrow 3$ WAR dependence on a
- $1 \rightarrow 6$ WAR dependence on c
- $2 \rightarrow 3$ RAW dependence on d
- $3 \rightarrow 7$ WAR dependence on b
- $3 \rightarrow 7$ RAW dependence on a
- $5 \rightarrow 6$ RAW dependence on f
- $5 \rightarrow 7$ RAW dependence on f

The group of statements that can be issued together are $\{1\}, \{2\}, \{3\}, \{4, 5\}, \{6\}, \{7\}$.

(c) Only b and c are live after the code segment, so values a,d,e and f are not needed subsequently. Values a and f are needed to compute b and a possible final value of c, so statements 1,2,3,5,6, and 7 are needed because they are part of the b and c chain of calculation. Statement 4 produces a value that does not contribute to the live values and is itself not live, thus this statement may be deleted.

(d) The opportunity to issue statements 4 and 5 together is lost when statement 4 is deleted. yet this was the only multiple issue opportunity. The available ILP is affected by the compiler technology. Compiler optimization may reduce the available ILP. the performance due to a hardware design option can be affected, both positively and negatively, by compiler actions.

Ques.2: There are really two parts to this question. First, examine the code as it is written and determine dependencies and whether it can be unrolled. Second, examine what the code is doing and determine if we can rewrite the code so it can be unrolled.

Examining the code as written, we have the following dependencies:

- WAR dependency between S1, S2 on B[i]. S1 has to read B[i] before S2 writes new value of B[i].

- Loop carried RAW dependency between S3 and S1 on A[i]. In iteration i , we need value of A[i] computed in previous iteration $i - 1$. Similarly, we need value of B[i] computed in previous iteration.
- Loop carried RAW dependency between S4 and S1 on A[i]. Similar to above, we need value of A[i] computed in previous iteration.

Since we have true loop carried dependencies, we cannot unroll the loop as written.

To determine if we can rewrite it to remove loop carried dependencies, let's examine the exact computations at iteration i , and $i - 1$.

```

.....
A[i-1] = A[i-1] + B[i-1];
B[i-1] = 2*B[i-1];
C[i-1] = A[i-2] + B[i-2];
D[i-1] = D[i-1] * A[i-2];
A[i] = A[i] + B[i];
B[i] = 2*B[i];
C[i] = A[i-1] + B[i-1];
D[i] = D[i] * A[i-1];
.....

```

This can be re-arranged as:

```

.....
C[i-1] = A[i-2] + B[i-2];
D[i-1] = D[i-1] * A[i-2];
// new loop body //
A[i-1] = A[i-1] + B[i-1];
B[i-1] = 2*B[i-1];
C[(i-1)+1] = A[i-1] + B[i-1];
D[(i-1)+1] = D[(i-1)+1] * A[i-1];
// end of new loop body //
A[i] = A[i] + B[i];
B[i] = 2*B[i];
.....

```

If we write the new loop body as shown above, and move the first computation of C[0] and D[0] outside the loop we get:

```

C[1] = A[0] + B[0];
D[1] = D[1] * A[0];
//note that neither A[0] nor B[0] are operated on inside the loop //
for (i=1; i <100; i++){// new loop body //
A[i] = A[i] + B[i];
B[i] = 2*B[i];
C[i+1] = A[i] + B[i];

```

```
D[i+1] = D[i+1] * A[i];
}
```

In the above code there is no loop carried RAW dependency and the loop can be unrolled. There is a big difference between the two pieces of code – in the first case (when we did not rewrite the code), the compiler only had to examine what was given to it. In the second case, the compiler had to figure out equivalent program code that could be unrolled – thus resulting in a more complex code optimization process.

Practice Problem 2:(not graded) Assumption: When the while statement is implemented, we assume that branch taken means the code inside the while loop is executed.

The first loop branch condition is evaluated 101 times – for $i = 0, i = 1, \dots, i = 99, i = 100$. The branch b1 is taken the first 100 times and not taken the last time (with $i = 100$).

The second loop branch condition b2 is evaluated 50 times – for $i = 100, i = 101, \dots, i = 149, i = 150$. It is taken 49 times and not taken the last time.

Let us consider behaviour using a 1-bit predictor. The predictor always predicts branch Taken. Therefore, for branch 1 the first 100 iterations are predicted correctly but the last is incorrect. The second branch is correctly predicted 49 times and incorrect the last time. Therefore total accuracy is 149/151.

Using the 2-bit predictor, with history bits set to NT, the first prediction is Not Taken which is a wrong prediction. The actual outcome is T which takes it to state TT which predicts taken. The last iteration, the prediction is Taken (since state is TT) but the actual outcome is Not Taken. Therefore for branch b1, the accuracy is 99/101. Similarly, you can compute the accuracy and predictions for the other branch.

Ques.Practice Problem 2:

The given code is:

```
i1: load R0, (R4)
i2: add R1, R0, #10
i3: load R2, (R5)
i4: mul R0, R2, #10
i5: mul R4, R2, R1
i6: store R0, (R5)
```

Assumptions:

- We assume 1 cycle for issue, and then the latency of the operation. Therefore, a load that issues in cycle 1, completes end of cycle 3, releases unit and instruction needing this result can start in cycle 4. (You could also assume that the latencies given include issue latency since the question did not specify whether issue time is included in latencies given.)
- We assume two reservation stations per unit. We assume the following tags for the reservation stations: 1: load/store; 2: load/store, 3: add, 4: add; 5: mul; 6: mul.
- We assume that the execution units are NOT pipelined. Therefore if a Load is sent for execution to the Load functional unit then no other Load can be sent to it for 2 more cycles. Note that if we assume that the units are pipelined then we can send one each cycle to the execution unit and you will get a different schedule.

The following is a typical Tomasulo schedule, assuming two instructions issued per cycle (if you assume only one issue per cycle then you will get a different schedule):

- Cycle 1: Issue instructions i1, i2. Inst. i1, Load R0,(R4), is issued to reservation station 1 (with tag 1). The data in R4 is ready. Instruction i2 is issued to res. station with tag 3 (Add), with value 10 in one operand field and a tag of 1 in the other operand field for R0 (since result comes from the Load operation).
- Cycle 2: Issue instructions i3,i4. Inst i1 is executing. i3 is sent to Load station with tag 2. i4 is sent to Mul station with tag 5. The operand for the mul is tagged with 2 (result comes from Load unit 2), and the other operand is the value 10.
- Cycle 3: i3 cannot start executing since the load execution unit is not free. i1 completes execution at the end of cycle 3. Issue instruction i5 to station with tag 6 (mul). The operands are both tags, one operand comes from tag 2 (output of inst i3) and the other operand has tag 3 (the add). We cannot issue i6 since it needs a Load/Store unit and both are busy (with instruction i1,i3).
- Cycle 4: The result from i1 is available. Therefore i2 starts executing in the adder (since it needed data from tag 2). Register R0 is written from output of station 1. Issue instruction i6 with tag 1, but its data comes from tag 5 (mul station). Since the Load/Store functional unit is free, we start execution of inst i3. Since execution latency of inst i2 is 1 cycle, it completes at end of this cycle.
- you can complete the remaining cycles..