# 16.482 / 16.561: Computer Architecture and Design

Spring 2015

Homework #6 Solution

*All problems deal with the following three threads. Note that you must determine the number of stall cycles between dependent instructions based on the instruction latencies given below:*

*Thread 1:*
```
L.D F0, 0(R1)
L.D F2, 8(R1)
ADD.D F4, F0, F2
SUB.D F6, F2, F0
S.D F4, 16(R1)
S.D F6, 24(R1)
DSUBUI R1, R1, #32
BNEZ R1, loop
```

*Thread 2:*
```
DADDUI R1, R1, #24
ADD.D F2, F0, F4
ADD.D F4, F6, F8
ADD.D F6, F0, F6
S.D F2, -24(R1)
S.D F4, -16(R1)
S.D F6, -8(R1)
BEQ R1, R7, end
```

*Thread 3:*
```
L.D F6, 0(R1)
ADD.D F8, F8, F6
S.D F8, 8(R1)
DADDUI R1, R1, #16
BNE R1, R2, loop
L.D F6, 0(R1)
ADD.D F8, F8, F6
S.D F8, 8(R1)
DADDUI R1, R1, #16
BNE R1, R2, loop
```

*Assume you are using a processor with the following characteristics:*

- *6 functional units: 3 ALUs, 2 memory ports (load/store), 1 branch*
- *The following instruction latencies:*
    - *L.D/S.D: 4 cycles (1 EX, 3 MEM)*
    - *ADD.D/SUB.D: 2 cycles*
    - *All other operations: 1 cycle*

**Solution:** When dealing with these threads, the first step is really to identify the dependences and the latency of the producing instructions. Doing so allows you to figure out both what instructions are independent, and how many cycles are required between dependent instructions.

Note that just writing the stalls as you normally would using in-order execution isn't sufficient. You may run into cases in which stalls that are hidden in a single issue processor show up in multithreading (or any multiple issue machine, for that matter) because you're executing multiple instructions in each cycle.

The breakdown starts on the next page. Note that each instruction has been numbered to make it easier to list the dependences. The number of cycles shown after each dependence is the number of cycles required between the producing and consuming instructions. If no other instructions are available during this time, the thread will stall.

Thread 1:

```
(1)      L.D F0, 0(R1)
(2)      L.D F2, 8(R1)
(3)      ADD.D F4, F0, F2
(4)      SUB.D F6, F2, F0
(5)      S.D F4, 16(R1)
(6)      S.D F6, 24(R1)
(7)      DSUBUI R1, R1, #32
(8)      BNEZ R1, loop
```

Dependences:

| | |
|---|---|
| (1) → (3) | 3 cycles |
| (2) → (3) | 3 cycles |
| (3) → (5) | 1 cycle |
| (4) → (6) | 1 cycle |
| (7) → (8) | 0 cycles |

Thread 2:

```
(1)      DADDUI R1, R1, #24
(2)      ADD.D F2, F0, F4
(3)      ADD.D F4, F6, F8
(4)      ADD.D F6, F0, F6
(5)      S.D F2, -24(R1)
(6)      S.D F4, -16(R1)
(7)      S.D F6, -8(R1)
(8)      BEQ R1, R7, end
```

Dependences:

| | |
|---|---|
| (1) → (5) | 0 cycles |
| (1) → (6) | 0 cycles |
| (1) → (7) | 0 cycles |
| (1) → (8) | 0 cycles |
| (2) → (5) | 1 cycle |
| (3) → (6) | 1 cycle |
| (4) → (7) | 1 cycle |

Thread 3:

```
(1)      L.D F6, 0(R1)
(2)      ADD.D F8, F8, F6
(3)      S.D F8, 8(R1)
(4)      DADDUI R1, R1, #16
(5)      BNE R1, R2, loop
(6)      L.D F6, 0(R1)
(7)      ADD.D F8, F8, F6
(8)      S.D F8, 8(R1)
(9)      DADDUI R1, R1, #16
(10)     BNE R1, R2, loop
```

Dependences:

| | |
|---|---|
| (1) → (2) | 3 cycles |
| (2) → (3) | 1 cycle |
| (4) → (5) | 0 cycles |
| (4) → (6) | 0 cycles |
| (6) → (7) | 3 cycles |
| (7) → (8) | 1 cycle |
| (9) → (10) | 0 cycles |

1. *(25 points) Determine how long the code will take using fine-grained multithreading. Assume the processor uses in-order scheduling.*

In fine-grained multithreading, we alternate threads every cycle. This technique takes 18 cycles to execute all three threads.

| Cycle | ALU1 | ALU2 | ALU3 | Mem1 | Mem2 | Branch | |
|---|---|---|---|---|---|---|---|
| 1 | | | | T1: L.D | T1: L.D | | |
| 2 | T2: DADDUI | T2: ADD.D | T2: ADD.D | | | | |
| 3 | | | | T3: L.D | | | |
| 4 | T2: ADD.D | | | T2: S.D | T2: S.D | | |
| 5 | T1: ADD.D | T1: SUB.D | | | | | |
| 6 | | | | T2: S.D | | T2: BEQ | |
| 7 | T3: ADD.D | | | | | | |
| 8 | T1: DSUBUI | | | T1: S.D | T1:S.D | | |
| 9 | T3: DADDUI | | | T3: S.D | | | |
| 10 | | | | | | T1: BNEZ | |
| 11 | | | | T3: L.D | | T3: BNE | |
| 12 | | | | | | | T3 stall |
| 13 | | | | | | | T3 stall |
| 14 | | | | | | | T3 stall |
| 15 | T3: ADD.D | | | | | | |
| 16 | | | | | | | T3 stall |
| 17 | T3: DADDUI | | | T3: S.D | | | |
| 18 | | | | | | T3: BNE | |

*2. (25 points) Determine how long the code will take using coarse-grained multithreading. Assume the processor uses in-order scheduling, and switch threads on any stall longer than 1 cycle (stalls of 2 or more cycles).*

In coarse-grained multithreading, we switch threads on any stall requiring more than 1 cycle—namely, the three-cycle stalls in both Thread 1 and Thread 3. Note that, with this technique, Thread 2 will run to completion without being switched out. However, we do have to be careful when scheduling that thread, as there are dependences between the ADD.D and S.D instructions that must be satisfied. Overall, coarse-grained and fine-grained multithreading perform similarly, as the threads take a total of 21 cycles to complete.

| Cycle | ALU1 | ALU2 | ALU3 | Mem1 | Mem2 | Branch | |
|---|---|---|---|---|---|---|---|
| 1 | | | | T1: L.D | T1: L.D | | |
| 2 | T2: DADDUI | T2: ADD.D | T2: ADD.D | | | | |
| 3 | T2: ADD.D | | | | | | |
| 4 | | | | T2: S.D | T2: S.D | | |
| 5 | | | | T2: S.D | | T2: BEQ | |
| 6 | | | | T3: L.D | | | |
| 7 | T1: ADD.D | T1: SUB.D | | | | | |
| 8 | | | | | | | T1 stall |
| 9 | T1: DSUBUI | | | T1: S.D | T1:S.D | | |
| 10 | | | | | | T1: BNEZ | |
| 11 | T3: ADD.D | | | | | | |
| 12 | | | | | | | T3 stall |
| 13 | T3: DADDUI | | | T3: S.D | | | |
| 14 | | | | T3: L.D | | T3: BNE | |
| 15 | | | | | | | T3 stall |
| 16 | | | | | | | T3 stall |
| 17 | | | | | | | T3 stall |
| 18 | T3: ADD.D | | | | | | |
| 19 | | | | | | | T3 stall |
| 20 | T3: DADDUI | | | T3: S.D | | | |
| 21 | | | | | | T3: BNE | |

*3. (25 points) Determine how long the code will take using simultaneous multithreading. Assume the processor uses in-order scheduling, and that thread 1 is the preferred thread, followed by threads 2 and 3.*

Thread 1 is the preferred thread, followed by Threads 2 and 3. Simultaneous multithreading allows for the best overall usage of functional units and takes only 16 cycles for all three threads to complete.

| **Cycle** | **ALU1** | **ALU2** | **ALU3** | **Mem1** | **Mem2** | **Branch** | |
|---|---|---|---|---|---|---|---|
| **1** | T2: DADDUI | T2: ADD.D | T2: ADD.D | T1: L.D | T1: L.D | | |
| **2** | T2: ADD.D | | | T3: L.D | | | |
| **3** | | | | T2: S.D | T2: S.D | | |
| **4** | | | | T2: S.D | | T2: BEQ | |
| **5** | T1: ADD.D | T1: SUB.D | | | | | |
| **6** | T3: ADD.D | | | | | | |
| **7** | T1: DSUBUI | | | T1: S.D | T1:S.D | | |
| **8** | T3: DADDUI | | | T3: S.D | | T1: BNEZ | |
| **9** | | | | T3: L.D | | T3: BNE | |
| **10** | | | | | | | T3 stall |
| **11** | | | | | | | T3 stall |
| **12** | | | | | | | T3 stall |
| **13** | T3: ADD.D | | | | | | |
| **14** | | | | | | | T3 stall |
| **15** | T3: DADDUI | | | T3: S.D | | | |
| **16** | | | | | | T3: BNE | |

*4. (25 points) Determine how long the code will take using simultaneous threading if the processor uses dynamic (out-of-order) scheduling. Assume that the processor can issue four instructions per thread in each cycle, and that the order of preferred threads remains the same.*

Let's re-examine the threads and look at when each instruction is issued, and therefore how early each one can execute. Keep in mind that the idea of out-of-order scheduling is to identify instructions that can be issued without stalling behind instructions on which they do not depend:

Thread 1:

```
    (1)  L.D F0, 0(R1)
    (2)  L.D F2, 8(R1)
    (3)  ADD.D F4, F0, F2
    (4)  SUB.D F6, F2, F0
    (5)  S.D F4, 16(R1)
    (6)  S.D F6, 24(R1)
    (7)  DSUBUI R1, R1, #32
    (8)  BNEZ R1, loop
```

First cycle: Issue (1)-(4)
  Same dependences as original
Second cycle: Issue (5)-(8)
  (7) independent of (1)-(6); can start as early as cycle 2
  (8) can start 1 cycle after (7)

Thread 2:

```
    (1)  DADDUI R1, R1, #24
    (2)  ADD.D F2, F0, F4
    (3)  ADD.D F4, F6, F8
    (4)  ADD.D F6, F0, F6
    (5)  S.D F2, -24(R1)
    (6)  S.D F4, -16(R1)
    (7)  S.D F6, -8(R1)
    (8)  BEQ R1, R7, end
```

First cycle: Issue (1)-(4)
  All independent
Second cycle: Issue (5)-(8)
  (5)-(8) have same dependences as original
  (8) depends on (1) but not (2)-(7); can start one cycle after (1)

Thread 3:

```
    (1)  L.D F6, 0(R1)
    (2)  ADD.D F8, F8, F6
    (3)  S.D F8, 8(R1)
    (4)  DADDUI R1, R1, #16
    (5)  BNE R1, R2, loop
    (6)  L.D F6, 0(R1)
    (7)  ADD.D F8, F8, F6
    (8)  S.D F8, 8(R1)
    (9)  DADDUI R1, R1, #16
    (10) BNE R1, R2, loop
```

First cycle: Issue (1)-(4)
  (1)-(3) have same dependences as original
  (4) independent of (1)-(3)
Second cycle: Issue (5)-(8)
  Same dependences as original
Third cycle: Issue (9)-(10)
  (9) depends on (4), can start one cycle later

The list on the previous page shows that there aren't many instructions that can be moved, but they do have a significant impact on the overall performance, cutting the total execution time down to 10 cycles.

The biggest impact is seen in Thread 3, where moving the first DADDUI instruction earlier allows the second half of that thread to execute much sooner than it would otherwise. Note that this thread essentially represents two iterations of a loop, so the example shows how multiple issue can dramatically increase the performance benefit of dynamic scheduling with speculation.

| Cycle | ALU1 | ALU2 | ALU3 | Mem1 | Mem2 | Branch |
|---|---|---|---|---|---|---|
| 1 | T2: DADDUI | T2: ADD.D | T2: ADD.D | T1: L.D | T1: L.D | |
| 2 | T2: ADD.D | T1: DSUBUI | T3: DADDUI | T3: L.D | | T2: BEQ |
| 3 | T3: DADDUI | | | T2: S.D | T2: S.D | T1: BNEZ |
| 4 | | | | T2: S.D | T3: L.D | T3: BNE |
| 5 | T1: ADD.D | T1: SUB.D | | | | T3: BNE |
| 6 | T3: ADD.D | | | | | |
| 7 | | | | T1: S.D | T1:S.D | |
| 8 | T3: ADD.D | | | T3: S.D | | |
| 9 | | | | | | |
| 10 | | | | T3: S.D | | |

T3 stall