Exercise 6.1. Register Update Unit

Figure **??** shows the Tomasulo processor structure, extended with the Re-Order Buffer, as presented in the lecture slides. Figure **??** shows a variation of this design, in which the Reservation Stations and the Re-Order Buffer are combined into a single structure, the Register Update Unit (RUU). This is the design simulated by SimpleScalar.

1. What are the pipeline stages for a register-to-register floating-point instruction?
2. Consider a design with two FP multipliers. When does the RUU design decide which multiplier to use for a particular instruction? How does this differ from the Tomasulo+ROB design?
3. In the Tomasulo+ROB design, the Issue stage is complex and may limit the design's cycle time, unless it is split over two cycles. How is the situation different in the RUU design?

Exercise 6.2. Software pipelining

In this question, assume a static-pipeline machine with the following characteristics:

- Integer arithmetic always takes just one clock cycle.
- Delayed conditional branch. No stalls are caused the branch.
- FP operations are identified at the ID stage and are dispatched to separate non-pipelined functional units. Results are routed direct to the WB stage where they are handled in issue order.
- There are two FP functional units, one for add/subtract, the other for multiply/divide.
- FP load takes 3 cycles, FP addition takes 4 cycles.
- There is full bypassing, i.e. wherever a hazard can be resolved using bypassing, the bypass circuitry is present.
- Since FP registers are separate from integer registers, updates (write-back) to FP registers can occur in the same cycle as updates to integer registers.

Consider the following loop:

```
LOOP: SD   0(R1),F4     ; stores into M[i]
      ADDD F4,F0,F2     ; adds to M[i+1]
      LD   F0,16(R1)    ; loads M[i+2]
      BLT  R1,R2,LOOP   ; delayed branch if R1 < R2
      ADD  R1,R1,#8     ; executed whether branch taken or not
```

Assume that R1 initially points to the start of an array `M` of doubles, and R2 points to the end.

1. Using your paper sideways, draw a diagram showing the timing for the execution of the loop. You will need to look at more than one iteration.
   - **Indicate clearly where forwarding occurs**
   - **No stalls should be necessary**
2. What is the average CPI when the loop is executed many times?
3. Explain what this loop does, and describe how it is likely to be used to perform a *straightforward* useful function. Show any initialisation needed. What is going on? How does the performance of this loop compare with a straightforward implementation?

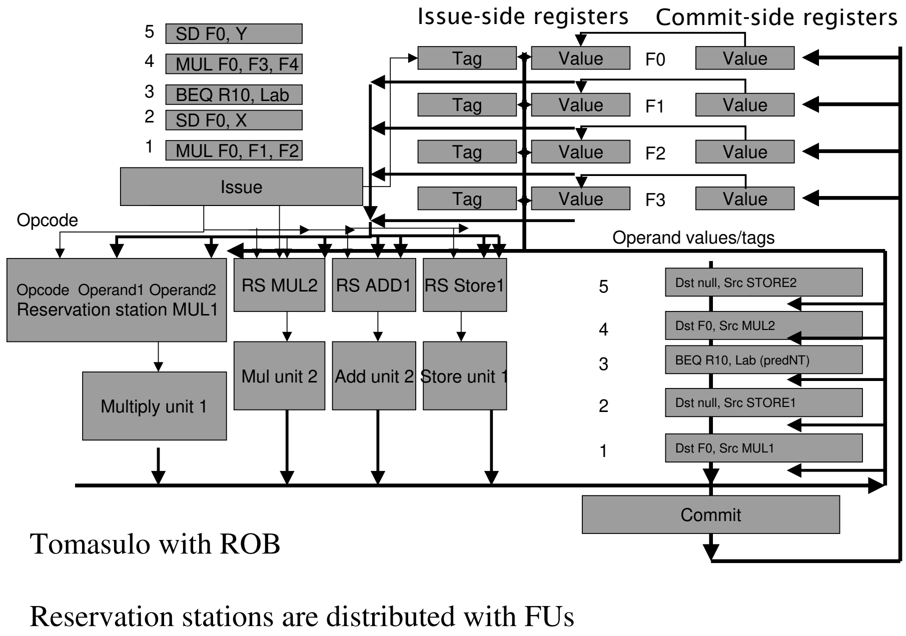

Figure 1: Tomasulo+ROB

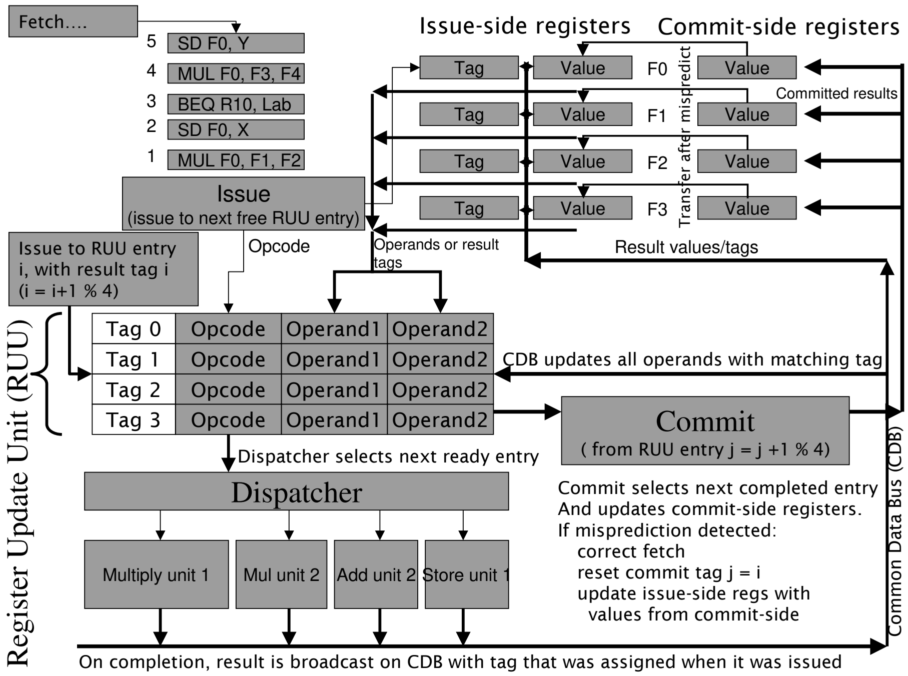

Figure 2: Register Update Unit design

Exercise 6.3

The loop in Exercise 6.2 is the core of a fast implementation of a simple operation. Write down the assembler code for the straightforward version of the loop. Consider how it would be executed on a dynamically-scheduled version of the machine using Tomasulo's algorithm with a re-order buffer, and compare its timing with the behaviour of the loop shown in Exercise 6.2. What do you conclude?

Paul Kelly, Imperial College, 2007

Exercise 6 — Suggested solutions

Exercise 6.1. Register Update Unit

1. What are the major pipeline stages for a register-to-register floating-point instruction?

   Fetch, Issue (which includes decode and register fetch), Read Operands (where the Reservation Station collects operands from completing instructions – possibly omitted if operands available at issue), Execute, Write Back (on the CDB), and Commit. Different sources give these stages confusingly different names. A stage may be split across two (or more) cycles for implementation reasons. Some designs introduce further stages to account for delays moving data around the chip. See, for example, the Pentium 4, which has two "Drive" stages in its pipeline[1].

2. Consider a design with two FP multipliers. When does the RUU design decide which multiplier to use for a particular instruction? How does this differ from the Tomasulo+ROB design?

   In the Tomasulo design, the RS's are co-located with the functional units. When an instruction is issued, the issue unit has to pick an RS fronting an FU appropriate for this instruction. In the RUU design, the RS's form a uniform pool - any onewould do. It is the Dispatch unit that assigns the ready instruction to an FU. This seems easier and more efficient as we know at dispatch time which FU is actually free (if we know service times in advance there might not be any actual performance advantage).

3. In the Tomasulo+ROB design, the Issue stage is complex and may limit the design's cycle time, unless it is split over two cycles. How is the situation different in the RUU design?

   The Tom+Rob Issue stage has to read the source registers, find a free RS, write the resulting tag to the destination register, and route the instruction to the RS and ROB.

   In the RUU design the Issue stage's work is slightly simpler: it's easy to find a free RS - the RUU is managed as a circular buffer, with instructions being issued at one end and committed at the other (when an instruction is committed its RUU slot becomes free). Also, the instruction only has to be passed to the RUU - it doesn't have to be routed across the chip to the FU. Instead it is the Dispatch stage that does this.

   As we see in the Pentium 4, the optimum may be a compromise between these two extremes.

Exercise 6.2. Software pipelining

In this question, assume a static-pipeline machine with the following characteristics:

- Integer arithmetic always takes just one clock cycle.
- Delayed conditional branch. No stalls are caused the branch.
- FP operations are identified at the ID stage and are dispatched to separate non-pipelined functional units. Results are routed direct to te WB stage where they are handled in issue order.
- There are two FP functional units, one for add/subtract, the other for multiply/divide.
- FP load takes 3 cycles, FP addition takes 4 cycles
- There is full bypassing, i.e. wherever a hazard can be resolved using bypassing, the bypass circuitry is present.
- Since FP registers are separate from integer registers, updates (write-back) to FP registers can occur in the same cycle as updates to integer registers.

Consider the following loop:

[1] http://arstechnica.com/articles/paedia/cpu/p4andg4e.ars/6

```
LOOP: SD   0(R1),F4     ; stores into M[i]
      ADDD F4,F0,F2     ; adds to M[i+1]
      LD   F0,16(R1)    ; loads M[i+2]
      BLT  R1,R2,LOOP   ; delayed branch if R1 < R2
      ADD  R1,R1,#8     ; executed whether branch taken or not
```

Assume that R1 initially points to the start of an array M of doubles, and R2 points to the end.

1. Using your paper sideways, draw a diagram showing the timing for the execution of the loop. You will need to look at more than one iteration.

   Take care to identify and explain all possible hazards.

Suggested answer

```
SD   0(R1),F4  IF ID EX M
ADDD F4,F0,F2     IF ID EX EX EX EX WB
LD   F0,16(R1)      IF ID EX M  M  M  WB
BLT  R1,R2,LOOP        IF ID EX
ADD  R1,R1,#8              IF ID EX M  WB

SD   0(R1),F4                 IF ID EX M
ADDD F4,F0,F2                    IF ID EX EX EX EX WB
LD   F0,16(R1)                      IF ID EX M  M  M  WB
BLT  R1,R2,LOOP                        IF ID EX
ADD  R1,R1,#8                             IF ID EX M  WB
```

Basic idea: **(1 marks)**

No stalls occur. **(1 marks)**

Hazards:

(a) There are no data (RAW) hazards: each instruction uses a register generated by the next instruction in the loop, giving the maximum possible time (unless we unroll and use different registers) for the instruction to complete.

(b) There are no WAR hazards because, in a static pipeline like this one, instructions are always issued (and therefore pick up their operands) in order.

(c) There are no WAW hazards because, although we need concurrent WBs (which the hardware must accommodate), the registers in question are different.

(d) You were asked to assume no control hazards. This could be achieved by relying on branch prediction to assume the branch will be taken. In this machine, the branch will be resolved by the EX or perhaps ID stage, which is easily early enough to avoid the registers being updated with the results of wrongly-executed instructions.

Sensible comments about hazards: **(2 marks)**

---

2. What is the CPI when the loop is executed many times?

Suggested answer

Since there are no stalls, the CPI tends to 1 when the number of iterations is large. **(1 marks)**

---

3. Explain what this loop does, and describe how it is likely to be used to perform a useful function. Show any initialisation needed. What is going on? How does the performance of this loop compare with a straightforward implementation?

Suggested answer

The loop really does this:

```
for i = 0 to N do
  A[N] := A[N] + K
```

(2 marks)

The code is designed to allow as much time as possible for load and FP add operations to take place, without unrolling. To do this, instructions are migrated backwards to the earliest point they can be issued. An instruction can be migrated round the loop, so it initiates work belonging to a later iteration.

When this happens, it is also necessary to migrate the instruction into the start-up sequence, so that the first two iterations are primed appropriately:

```
      LD   F0,(R1)      ; loads M[N]
      ADDD F4,F0,F2     ; compute new M[N] ready for SD
      LD   F0,8(R1)    ; loads M[N+1] ready for ADDD

Loop: SD   0(R1),F4     ; stores into M[i]
      ADDD F4,F0,F2     ; adds to M[i+1]
      LD   F0,16(R1)   ; loads M[i+2]
      BLT  R1,R2,LOOP
      ADD  R1,R1,#8
```

(2 marks)

You might also want to handle the last two iterations separately to avoid loading elements beyond the end of M).

A straightforward implementation of the loop above would look like:

```
Loop: LD   F0,0(R1)
      ADDD F4,F0,F2     ; adds to M[i]
      SD   0(R1),F4     ; stores into M[i]
      BLT  R1,R2,LOOP
      ADD  R1,R1,#8
```

This implementation suffers stalls in a static pipeline:

```
cycle            1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
LD   F0,0(R1)    IF ID EX M  M  M  WB
ADDD F4,F0,F2       IF ID          EX EX EX EX WB (stall due to F0)
SD   0(R1),F4          IF                ID EX M  WB (forwarding of F4 to M stage)
BLT  R1,R2,LOOP                          IF ID EX M  WB
ADD  R1,R1,#8                               IF ID EX M  WB


LD   F0,0(R1)                                  IF ID EX M  M  M  WB
...
```

Exactly how many stalls depends on the design of the floating point units. In a static-schedule design the constraint that ID stages access registers in-order leads to very poor performance.

The simple loop gets a CPI of 10/5=2.0. With a 10GHz clock, the clever loop would take 100μs to do 1M iterations, the simple one 200μs. The simple one is 100% slower, the clever one takes 100/200=50% as long, so it's 50% faster.

Total for 6.2: **(8 marks)**

(This exercise is based on an example in Hennessy and Patterson (first ed), page 325).

Exercise 6.3

The loop in Exercise 6.2 is the core of a fast implementation of a simple operation. Write down the assembler code for the straightforward version of the loop. Consider how it would be executed on a dynamically-scheduled version of the machine using Tomasulo's algorithm, and compare its timing with the behaviour of the loop shown in Exercise 6.2. What do you conclude?

Suggested answer

```
Loop: LD   F0,0(R1)
      ADDD F4,F0,F2     ; adds to M[i]
      SD   0(R1),F4     ; stores into M[i]
      ADD  R1,R1,#8
      BLT  R1,R2,LOOP
```

(1 marks)

```
               1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
LD   F0,0(R1)  IF IS RO M  M  M  WB
ADDD F4,F0,F2     IF IS          RO EX EX EX EX WB   (F0 fwded from LD via CDB)
SD   0(R1),F4        IF IS                     RO M (F4 fwded from ADDD via CDB)
ADD  R1,R1,#8           IF IS RO EX WB
BLT  R1,R2,LOOP            IF IS RO EX WB

LD   F0,0(R1)                 IF IS RO M  M  M     WB (loop-carried structural hazard due to CDB)
ADDD F4,F0,F2                    IF IS             RO EX EX EX EX WB
SD   0(R1),F4                       IF IS                          RO M
ADD  R1,R1,#8                          IF IS RO EX    WB
BLT  R1,R2,LOOP                           IF IS RO EX    WB

LD   F0,0(R1)                                IF IS RO M  M  M  WB
ADDD F4,F0,F2                                   IF IS          RO EX EX EX EX WB
SD   0(R1),F4                                      IF IS                     RO M
ADD  R1,R1,#8                                         IF IS RO EX    WB
BLT  R1,R2,LOOP                                          IF IS RO EX    WB
```

(3 marks) (As in the previous question, various variations on this answer are acceptable).

Conclude: you can sometimes, maybe often, get the performance advantage of dynamic scheduling via compile-time instruction scheduling. **(2 marks)**

Note the need for RO to initiate two operations in one cycle.

There is contention for the CDB, which delays forwarding the some iterations (eg iteration 2, cycles 12-13). However, delay does not accumulate since at most one instruction is issued per cycle, and the CDB handles one result per cycle.

Total for 5.3: **(6 marks)**

Total: **20 marks**

Paul Kelly, Imperial College, 2007