Homework Assignment 3

For chapter 3 ILP exploitation

College of Computer Science, Zhejiang University

时间：8-14h

充　也需要　，占分

1. You are tasked with designing a new processor microarchitecture and you are trying to determine how best to allocate your hardware resources. Which of the hardware and software techniques you learned in Chapter 3 should you apply? You have a list of latencies for the functional units and for memory, as well as some representative code. Your boss has been somewhat vague about the performance requirements of your new design, but you know from experience that, all else being equal, faster is usually better. Start with the basics. Figure below provides a sequence of instructions and list of latencies.

提示：假设取指、译码和发射共用同一个周期，WB 需要一个周期。store 和 branch 指令没有 WB 周期。需要注意，题目假设寄存器是上升沿写入。

| **Latencies beyond single cycle** | |
|---|---|
| Memory LD | +3 |
| Memory SD | +1 |
| Integer ADD, SUB | +0 |
| Branches | +1 |
| fadd.d | +2 |
| fmul.d | +4 |
| fdiv.d | +10 |

```
Loop:   fld     f2,0(Rx)
I0:     fmul.d  f2,f0,f2
I1:     fdiv.d  f8,f2,f0
I2:     fld     f4,0(Ry)
I3:     fadd.d  f4,f0,f4
I4:     fadd.d  f10,f8,f2
I5:     fsd     f4,0(Ry)
I6:     addi    Rx,Rx,8
I7:     addi    Ry,Ry,8
I8:     sub     x20,x4,Rx
I9:     bnz     x20,Loop
```

a. [10] What is the baseline performance (in cycles, per loop iteration) of the code sequence in the Figure if no new instruction’s execution could be initiated until the previous instruction’s execution had completed? Assume for now that execution does not stall for lack of the next instruction, but only one instruction/cycle can be issued. Assume the branch is taken, and that there is a one-cycle branch delay slot.

提示：本题没过程不给分。本题需要写出答案，并填写下面的表格。

| IF/IS | FU | WB |
|---|---|---|
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

```
Loop:   fld     f2,0(Rx)
I0:     fmul.d  f2,f0,f2
I1:     fdiv.d  f8,f2,f0
I2:     fld     f4,0(Ry)
I3:     fadd.d  f4,f0,f4
I4:     fadd.d  f10,f8,f2
I5:     fsd     f4,0(Ry)
I6:     addi    Rx,Rx,8
I7:     addi    Ry,Ry,8
I8:     sub     x20,x4,Rx
I9:     bnz     x20,Loop
```

b. [10] Think about what latency numbers really mean—they indicate the number of cycles a given function requires to produce its output. If the overall pipeline stalls for the latency cycles of each functional unit, then you are at least guaranteed that any pair of back-to-back instructions (a “producer” followed by a “consumer”) will execute correctly. But not all instruction pairs have a producer/consumer relationship. Sometimes two adjacent instructions have nothing to do with each other. How many cycles would the loop body in the code sequence in Figure require if the pipeline detected true data dependences and only stalled on those, rather than blindly stalling everything just because one functional unit is busy? (Hint: an instruction with latency +2 requires two <stall> cycles to be inserted into the code sequence.) Think of it this way: a one-cycle instruction has latency 1+0, meaning zero extra wait states. So, latency 1+1 implies one stall cycle; latency 1+N has N extra stall cycles.

提示：假设 core 是单发射的，每种 FU unit 有无限个。本题没过程不给分。

c. [15] Consider a multiple-issue design. Suppose you have two execution pipelines, each capable of beginning execution of one instruction per cycle, and enough fetch/decode bandwidth in the front end so that it will not stall your execution. Assume results can be immediately forwarded from one execution unit to another, or to itself. Further assume that the only reason an execution pipeline would stall is to observe a true data dependency. Now how many cycles does the loop require?

提示：本小问使用类似 tomasulo 的算法，不需要拿到操作数就可以发射。

d. [20] Reorder the instructions to improve performance of the code in Figure. Assume the

two-pipe machine in Exercise c and that the out-of-order completion issues have been dealt with successfully. Just worry about observing true data dependences and functional unit latencies for now. How many cycles does your reordered code take?
提示：本题没过程不给分。
e. [10] In your reordered code from Exercise d, what fraction of all cycles, counting both pipes, were wasted (did not initiate a new op)?
f. [10] Loop unrolling is one standard compiler technique for finding more parallelism in code, in order to minimize the lost opportunities for performance. Hand-unroll two iterations of the loop in your reordered code from Exercise d.
提示：展开时，展开的那个循环中的指令请用不同颜色标出，新指令使用的寄存器 f0->f10, f1->f11...给出指令的时序和耗时。本题没过程不给分。
g. [10] What speedup did you obtain? (For this exercise, just color the N+1 iteration's instructions green to distinguish them from the Nth iteration's instructions; if you were actually unrolling the loop, you would have to reassign registers to prevent collisions between the iterations.)
h. [15] Computers spend most of their time in loops, so multiple loop iterations are great places to speculatively find more work to keep CPU resources busy. Nothing is ever easy, though; the compiler emitted only one copy of that loop's code, so even though multiple iterations are handling distinct data, they will appear to use the same registers. To keep multiple iterations' register usages from colliding, we rename their registers. Figure 3.48 shows example code that we would like our hardware to rename. A compiler could have simply unrolled the loop and used different registers to avoid conflicts, but if we expect our hardware to unroll the loop, it must also do the register renaming. How? Assume your hardware has a pool of temporary registers (call them T registers, and assume that there are 64 of them, T0 through T63) that it can substitute for those registers designated by the compiler. This rename hardware is indexed by the src (source) register designation, and the value in the table is the T register of the last destination that targeted that register. (Think of these table values as producers, and the src registers are the consumers; it doesn't much matter where the producer puts its result as long as its consumers can find it.) Consider the code sequence in Figure 3.48. Every time you see a destination register in the code, substitute the next available T, beginning with T9. Then update all the src registers accordingly, so that true data dependences are maintained. Show the resulting code. (Hint: see Figure 3.49.)

```
Loop:           fld             f2,0(Rx)
I0:             fmul.d          f5,f0,f2
I1:             fdiv.d          f8,f0,f2
I2:             fld             f4,0(Ry)
I3:             fadd.d          f6,f0,f4
I4:             fadd.d          f10,f8,f2
I5:             sd              f4,0(Ry)
```

**Figure 3.48** Sample code for register renaming practice.

```
I0:           fld               T9,0(Rx)
I1:           fmul.d            T10,F0,T9
...
```

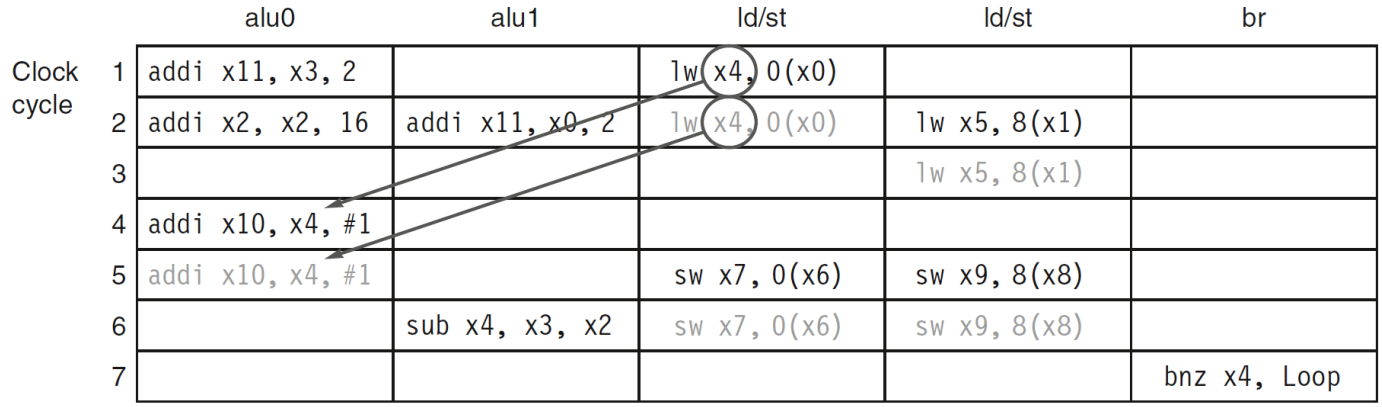
**Figure 3.49** Expected output of register renaming.

2. [30] Very long instruction word (VLIW) designers have a few basic choices to make regarding architectural rules for register use. Suppose a VLIW is designed with self-draining execution pipelines: once an operation is initiated, its results will appear in the destination register at most L cycles later (where L is the latency of the operation). There are never enough registers, so there is a temptation to wring maximum use out of the registers that exist. Consider Figure 3.52. loads have a 1+2 cycle latency.

```
Loop:   lw      x1,0(x2);       lw      x3,8(x2)
        <stall>
        <stall>
        addi    x10,x1,1;       addi    x11,x3,1
        sw      x1,0(x2);       sw      x3,8(x2)
        addi    x2,x2,8
        sub     x4,x3,x2
        bnz     x4,Loop
```

**Figure 3.52 Sample VLIW code with two adds, two loads, and two stalls.**

a. [10] unroll this loop once. show the result.
b. [10] 观察下图。show how a VLIW capable of two loads and two adds per cycle can use the minimum number of registers, in the absence of any pipeline interruptions or stalls.
c. [10] Give an example of an event that, in the presence of self-draining pipelines, could disrupt this pipelining and yield wrong results.

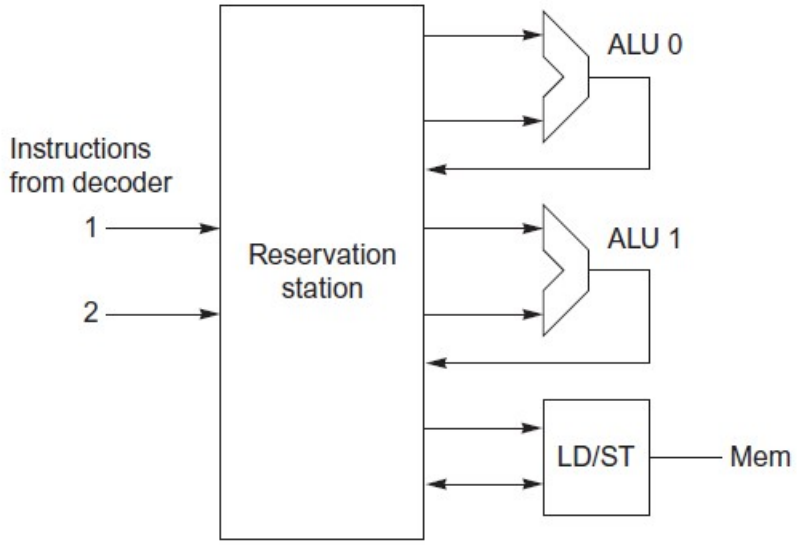



补充：[20] VLIW 是如何解决 RAW 的？VLIW 是如何解决 WAR 的？

3. [30] Assume a five-stage single-pipeline microarchitecture (fetch, decode, execute, memory, write-back) and the code. All ops are one cycle except LW and SW, which are 1+2 cycles, and branches, which are 1+1 cycles. There is no forwarding. 本题需要给出过程，否则不给分。

```
Loop:
lw x3,0(x0)
lw x1,0(x3)
addi x1,x1,1
sub x4,x3,x2
sw x1,0(x3)
bnz x4, Loop
```

a. [10] Show the phases of each instruction per clock cycle for one iteration of the loop. How many clock cycles per loop iteration are lost to branch overhead?
提示：画表，每行代表一条指令，每列代表一个 cycle，内容填写 FDEMW 五种阶段和 - (stall)。
b. [10] Assume a static branch predictor, capable of recognizing a backward branch in the Decode stage. Now how many clock cycles are wasted on branch overhead?
c. [10] Assume a dynamic branch predictor. How many cycles are lost on a correct prediction?
4. [85] Let’s consider what dynamic scheduling might achieve here. Assume a microarchitecture as shown in Figure. Assume that the arithmetic-logical units (ALUs) can do all arithmetic ops (fmul.d, fdiv.d, fadd.d, addi, sub) and branches, and that the Reservation Station (RS) can dispatch, at most, one operation to each functional unit per cycle (one op to each ALU plus one memory op to the fld/fsd).
提示：忽略取值、解码和写回，仅考虑 FU 阶段。




a. [15] Suppose all of the instructions from the sequence in Figure (Problem 1) are present in the RS, with no renaming having been done. Highlight any instructions in the code where register renaming would improve performance. (Hint: look for write-after-read and write-after-write hazards. Assume the same functional unit latencies as in Figure (Problem 1).)
b. [20] Suppose the register-renamed version of the code from part (a) is resident in the RS in clock cycle N, with latencies as given in Figure (Problem 1). Show how the RS should dispatch these instructions out of order, clock by clock, to obtain optimal performance on this code. (Assume the same RS restrictions as in part (a). Also assume that results must be written into the RS before they’re available for use—no bypassing.) How many clock cycles does the code sequence take?
c. [20] Part (b) lets the RS try to optimally schedule these instructions. But in reality, the whole instruction sequence of interest is not usually present in the RS. Instead, various events clear the RS, and as a new code sequence streams in from the decoder, the RS must choose to dispatch what it has. Suppose that the RS is empty. In cycle 0, the first two register-renamed instructions of this sequence appear in the RS. Assume it takes one clock cycle to dispatch any op, and assume functional unit latencies are as they were for Exercise 3.2. Further assume that the front end (decoder/register-renamer) will continue to supply two new instructions per clock cycle. Show the cycle-by-cycle order of dispatch of the RS. How many clock cycles does this code sequence require now?
d. [10] If you wanted to improve the results of part (c), which would have helped most: (1) Another ALU? (2) Another LD/ST unit? (3) Full bypassing of ALU results to subsequent operations? or (4) Cutting the longest latency in half? What’s the speedup?
e. [20] Now let’s consider speculation, the act of fetching, decoding, and executing beyond

one or more conditional branches. Our motivation to do this is twofold: the dispatch schedule we came up with in part (c) had lots of nops, and we know computers spend most of their time executing loops (which implies the branch back to the top of the loop is pretty predictable). Loops tell us where to find more work to do; our sparse dispatch schedule suggests we have opportunities to do some of that work earlier than before. In part (d) you found the critical path through the loop. Imagine folding a second copy of that path onto the schedule you got in part (b). How many more clock cycles would be required to do two loops' worth of work (assuming all instructions are resident in the RS)? (Assume all functional units are fully pipelined.)

Appendix:

1. (50 points) Run the following code on Scoreboard with function units including two FP multipliers， one FP divider, one FP adder that can perform float add and sub operations, and two integer units that is responsible for memory accessing, integer ALU operations and branch).

(1) Fill the following tables with the scoreboard status at the end of **5th** clock cycle when the first L.D instruction just finish WB stage. The scoreboard status includes instruction status, function units status and FP register status.

```
L.D     F0,0(R1)     ; F0←MEM[R1+0]
L.D     F4,0(R2)     ; F4←MEM[R2+0]
MUL.D   F6,F0,F4     ; F6←F0 * F4
ADD.D   F8,F0,F2     ; F8←F0 + F2
S.D     F6, 0(R3)    ; MEM[R3 + 0] ← F6
S.D     F8, 0(R4)    ; MEM[R4 + 0] ← F8
```

| Instruction producing result | Instruction consuming result | Latency (in clock cycle) |
|---|---|---|
| FP operation | FP operation | 3 |
| FP operation | FP store（S.D） | 2 |
| FP load（L.D） | FP opereation | 1 |
| FP load（L.D） | FP store（S.D） | 0 |

| Instructions | 发射（Issue） clock | 作数(RO) clock | 执行（EXE） clock | 结 (WB) clock |
|---|---|---|---|---|
| L.D F0, 0(R1) | 1 | 2 | 3-4 | 5 |
| L.D F4, 0(R2) | | | | |
| MUL.D F6,F0,F4 | | | | |
| ADD.D F8,F0,F2 | | | | |
| S.D F6, 0(R3) | | | | |
| S.D F8, 0(R4) | | | | |

| Function units | Function Unit Status | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Busy | Op | Fi | Fj | Fk | Qj | Qk | Rj | Rk | A |
| Integer1 | | | | | | | | | | |
| Integer2 | | | | | | | | | | |
| FPMult1 | | | | | | | | | | |
| FPMult2 | | | | | | | | | | |
| FPDivider | | | | | | | | | | |
| FPAdder | | | | | | | | | | |

| | Register Status | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | F0 | F2 | F4 | F6 | F8 | F10 | F12 | …… | F30 |
| FU | | | | | | | | | |

(2) Fill the following table with instruction status at the end of **6th** Clock Cycle as the first line in the table.

| | 发射（Issue） | 作数(RO) | 执行(EXE) | 结　(WB) |
|---|---|---|---|---|
| L.D F0, 0(R1) | 1 | 2 | 3-4 | 5 |
| L.D F4, 0(R2) | | | | |
| MUL.D F6,F0,F4 | | | | |
| ADD.D F8,F0,F2 | | | | |
| S.D F6, 0(R3) | | | | |
| S.D F8, 0(R4) | | | | |

| Function units | Function Unit Status | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Busy | Op | Fi | Fj | Fk | Qj | Qk | Rj | Rk | A |
| Integer1 | | | | | | | | | | |
| Integer2 | | | | | | | | | | |
| FPMult1 | | | | | | | | | | |
| FPMult2 | | | | | | | | | | |
| FPDivider | | | | | | | | | | |
| FPAdder | | | | | | | | | | |

| | Register Status | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | F0 | F2 | F4 | F6 | F8 | F10 | F12 | …… | F30 |
| FU | | | | | | | | | |

2.（50 points）For the following instruction sequence:

L.D F6, 34(R2)
L.D F2, 45(R3)
MUL.D F0, F2, F4
SUB.D F8, F2, F6
DIV.D F10, F0, F6

ADD.D F6, F8, F2

(1) Fill in the tables that Tomasulo algorithm used when the first instruction completes and finishes writing result. Assume the memory access unit needs two clock cycles to do execution: one for address calculation and one for memory access.

| Instruction status | | | |
|---|---|---|---|
| | ISSUE | EXECUTE | WRITE RESULT |
| 1 | | | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |

| Reservation stations | | | | | | | |
|---|---|---|---|---|---|---|---|
| NAME | BUSY | Op | Vj | Vk | Qj | Qk | A |
| Load1 | | | | | | | |
| Load2 | | | | | | | |
| Add1 | | | | | | | |
| Add2 | | | | | | | |
| Add3 | | | | | | | |
| Mult1 | | | | | | | |
| Mult2 | | | | | | | |

| Register status | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Field | F0 | F2 | F4 | F6 | F8 | F10 | F12 | … | F30 |
| Qi | | | | | | | | | |

(2) Assume the following latencies: load is 1 clock cycle; add is 2 clock cycles; multiply is 10 clock cycles; divide is 40 clock cycles. Fill in the tables when the instruction MUL.D is about to write result ( write result in next cycle).

| Instruction status | | | |
|---|---|---|---|
| | ISSUE | EXECUTE finish | WRITE RESULT |

| | Clock | Clock | Clock |
|---|---|---|---|
| 1 | | | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |

| Reservation stations | | | | | | | |
|---|---|---|---|---|---|---|---|
| NAME | BUSY | Op | Vj | Vk | Qj | Qk | A |
| Load1 | | | | | | | |
| Load2 | | | | | | | |
| Add1 | | | | | | | |
| Add2 | | | | | | | |
| Add3 | | | | | | | |
| Mult1 | | | | | | | |
| Mult2 | | | | | | | |

| Register status | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Field | F0 | F2 | F4 | F6 | F8 | F10 | F12 | … | F30 |
| Qi | | | | | | | | | |