体系实习二实习报告

00748267 杨文新

一、 实习综述

1) 完成工作概况

本次实习主要完成了以下几点工作：

a. 设计了我自己的指令系统及其编码，并设计实现指令系统需要的数据通路，同时用设计的指令系统编写汇编实现 Bubble Sort 和 Matrix Multiplication 程序。

b. 针对该指令系统修改 simplescalar 模拟器的指令模板，主要是编写我的指令系统的指令定义，经过模拟器的验证，程序运行结果正确；

c. 分析了把我设计的指令系统转变为流水线技术所带来的冒险和消除这些冒险的方法。

d. 设计了一个翻译指令系统汇编代码为机器码的程序。

注：指令系统的设计文件见 related_source_code/instructions，而两个程序的汇编代码见 related_source_code/bubble_sort.s 和 related_source_code/matrix.s。模拟器的指令定义见 myisa/myisa.def 和 myisa/myisa.h。

2) 指令系统设计

本着按需设计的理念，首先分析了需要实现的应用如冒泡排序和矩阵乘法所必需的指令，然后再结合模拟器的执行过程设计出指令系统。其中部分指令与 MIPS 类似，不过指令的格式具体格式有较大不同（主要是寄存器的顺序）。比较指令引入 slet(Set on Less or Equal Than)，与分支指令 beqz(Branch on Equal to Zero)结合可以实现条件调转，跳转的地址范围可达 23 位。另外，考虑到矩阵乘法需要经常计算偏移量，因此引入了乘法指令 mult 和寄存器装载指令 lwr(Load Word according to Register)、寄存器存储指令 swr(Store Word according to Register)，可以简化读取过程。所有的指令均为 32 位。具体指令系统及编码说明如下：

1. add rs, rt, rd

31-26	25-21	20-16	15-11	10-0
op	rs	rt	rd	don't care
000000	*****	*****	*****	**********

加法指令，将 rs, rt 寄存器的值相加再存放入 rd 寄存器中。即 rs + rt → rd。

2. addi rs, rt, imm16

31-26	25-21	20-16	15-0
000001	*****	*****	****************

加立即数指令，将 rs 寄存器和 16 位立即数（为满足加负数需要将其做符号扩展）相加的结果存放入 rt 寄存器。即 rs + imm16 → rt。

3. sub rs, rt, rd

31-26	25-21	20-16	15-11	10-0
op	rs	rt	rd	don't care
000010	*****	*****	*****	**********

加法指令，将 rs 寄存器值减去 rt 寄存器值所得的结果存放入 rd 寄存器中。即 rs - rt → rd。

4. mult rs, rt, rd

31-26	25-21	20-16	15-11	10-0
op	rs	rt	rd	don't care
000011	*****	*****	*****	**********

乘法指令，将 rs 寄存器值和 rt 寄存器值相乘的结果存放入 rd 寄存器中。即 rs * rt → rd。

5. lw rs, rt, imm16

31-26	25-21	20-16	15-0
000100	*****	*****	**********

装载指令，rs 寄存器存放基址，16 位立即数存放偏移量，rt 存放结果。即 rs[imm16] → rt。

6. sw rs, rt, imm16

31-26	25-21	20-16	15-0
000101	*****	*****	**********

存储指令，rs 寄存器存放基址，16 位立即数存放偏移量，rt 存放需要写入内存的值。即 rt → rs[imm16]。

7. slet rs, rt, rd

31-26	25-21	20-16	15-11	10-0
op	rs	rt	rd	don't care
000110	*****	*****	*****	**********

比较指令，如果 rs 寄存器值小于或等于 rt 寄存器值就把 rd 寄存器置 1。硬件上的实现为 ALU 使用 sub 指令，得到的结果分两路比较：一是将得到的结果取最高位（最高位为 1 则表示负数）；二是将所得结果再跟 0 比较；最后把两路所得结果相或即可实现该指令。即

```
if(rs <= rt) rd = 1;
else rd = 0;
```

8. beqz rs, target_address

31-26	25-21	20-0
000111	*****	**********

分支指令，如果 rs 寄存器值等于 0 则跳转到指定的地址。由于指令的地址 32 位的最后两位均为 0，为了使跳转地址范围更大所以将地址右移了两位，执行到该指令时现将地址再左移 2 位再设置 PC 值。可以跳转 2^{23} 位即 8M 的地址。硬件上的实现应当把 rs 寄存器值与$zero 运算，ALU 的控制信号设置为 or，如果得到的结果为 0 送给取指部件 1 信号(即 ALU 结果的 zero 信号)，如果得到的结果不为 0 则送给取指部件 0 信号。具体可见数据通路设计部分。即

```
if(rs == 0) PC = (target_address << 2);
else PC = PC + 4;
```

9. jump target_address

31-26	25-0
001000	**********

跳转指令，类似分支指令对地址进行了右移。可以跳转 2^{28} 即 256M 的地址。即

PC = (target_address << 2)

10. lwr rs, rt, rd

31-26	25-21	20-16	15-11	10-0
op	rs	rt	rd	don't care
001001	*****	*****	*****	**********

寄存器装载指令，rs 寄存器存放基址，rt 寄存器存放偏移量，将读入值放入 rd 中。在硬件上的实现与 lw 类似，不过 ALUsrc 控制信号应当控制选择 rt 寄存器作为偏移值，rd 作为目的寄存器。即

rs[rt] → rd

11. swr rs, rt, rd

31-26	25-21	20-16	15-11	10-0
op	rs	rt	rd	don't care
001010	*****	*****	*****	**********

寄存器存储指令，rs 寄存器存放基址，rt 寄存器存放偏移量，将 rd 寄存器值存入指定内存中。在硬件上的实现与 lw 类似，不过 ALUsrc 控制信号应当控制选择 rt 寄存器作为偏移值，rd 作为目的寄存器。即 rd → rs[rt]。

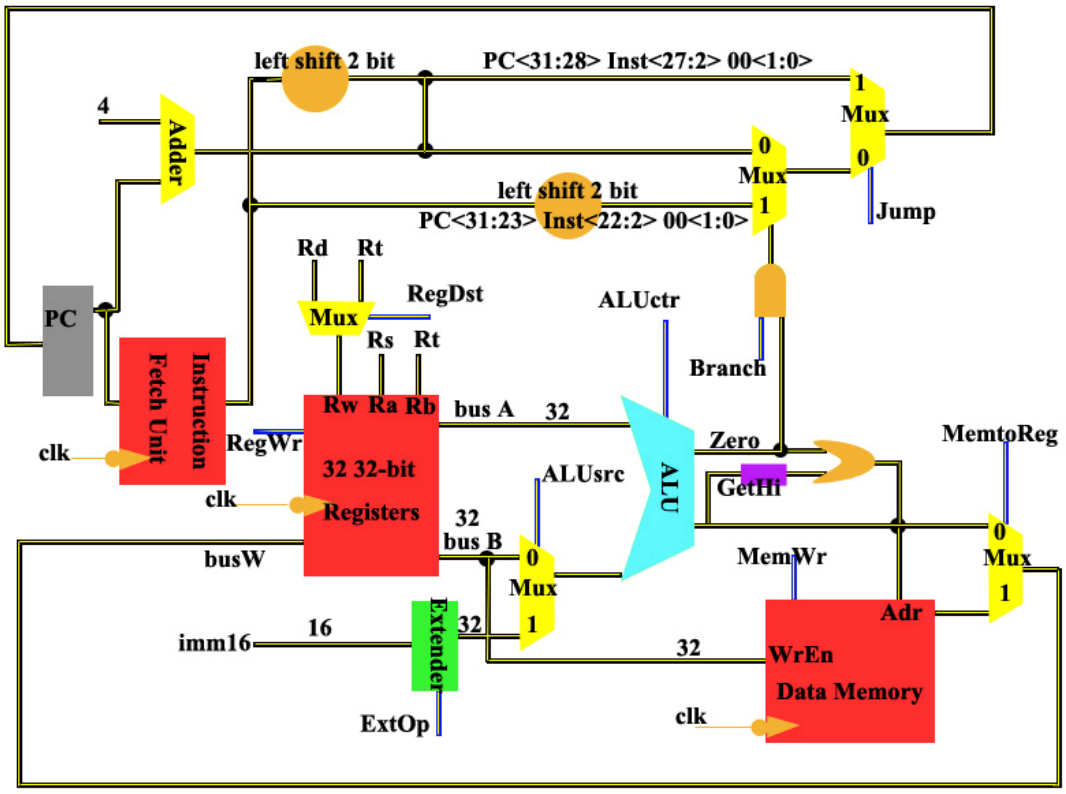
12. swi parameter

31- 26	25-0
111111	**************************

系统调用指令，与课程提供的一致，参数为 1 为系统停机，参数为 2 时为内存输出。

3) 数据通路设计

数据通路的设计我采用单周期数据通路，具体图如下：



由于文件大小问题，本处图可能比较模糊，原图请见包中的 alu.jpg（线路中出现中空状态主要是 photoshop 掌握不太好的原因……另外 ALU 画的比教材的少两条边，不过无伤大雅）。不过由于按照教材乘法指令是需要 32 周期的，扩展中的矩阵乘法用到了惩罚指令，本图中并没有给出，考虑的改进方案为把乘法指令删去，而用相应的循环加法来替代。

其中蓝色的线路表示控制信号，黄色梯形为 2:1 选择器，每个部件上都有相应的文字说明。其中 ALU 右侧的 GetHi 部件自定义的部件，用来实现 slet 指令，将 ALU 计算得到的结果取其最高位，硬件上的实现将最高位引出即可。实现原理见指令系统设计中的 slet。

对应各指令的控制信号值表如下：

op	**000000**	**000001**	**000010**	**000011**	**000100**	**000101**	**000110**	**000111**	**001000**	**001001**	**001010**
inst	add	addi	sub	mult	lw	sw	slet	beqz	jump	lwr	swr
RegDest	1	0	1	1	0	x	1	x	x	1	x
RegWr	1	1	1	1	1	0	1	0	0	1	1
ExtOp	x	Sign	x	x	Sign	Sign	x	x	x	x	x
ALUsrc	0	1	0	0	1	1	0	0	x	0	0
ALUctr	add	add	sub	mult	add	add	sub	or	x	add	add
MemWr	0	0	0	0	0	1	0	0	0	0	1
MemtoReg	0	0	0	0	1	x	0	x	x	1	x
Branch	0	0	0	0	0	0	0	1	0	0	0
Jump	0	0	0	0	0	0	0	0	1	0	0

4) 程序汇编代码

首先把程序的算法伪代码写出，然后根据伪代码可以方便快速的翻译为汇编代码。需要注意的是读取数据段的内容时不要忘记把得到的偏移量乘以 4 再加上基址。由于寄存器数量比较充足，将寄存器与变量固定关联也有助于编写时的条例清晰☺，如矩阵乘法中$7 用以存放变量 i, $8 用以存放变量 j 等等，翻译时可以很方便的找到。编写的汇编代码详见 related_source_code 文件夹中的.s 文件。

Bubble Sort 程序的伪码描述为：

```
BUBBLE-SORT(array A)
   for i <- 0 to length[A]-2
   {
      for j <- 1 to length[A]-i-1
      {
         if A[j] < A[j-1]
         {
            tmp <- A[j-1]
            A[j-1] <- A[j]
            A[j] <- tmp
         }
      }
   }
```

矩阵乘法程序的伪代码为：

```
MATRIX-MULTIPLICATION(matrix A, matrix B)
[C(m, n) <- A(m, l) * B(l, n)]
   for i <- 0 to m-1
   {
      for j <- 0 to n-1
      {
         c[i][j] <- 0
         for k <- 0 to l-1
            c[i][j] <- c[i][j]+a[i][k]*b[k][j]
      }
   }
```

5) 生成二进制可执行文件

起初设想是只把指令翻译成机器码，因为调转地址比较难处理，然后再把地址手工填入，不过实践中发现汇编代码 debug 时需要经常修改代码而重新生成二进制文件，每次都手工填入计算地址值太过于耗费时间。因此设计了一个可以基本完全翻译汇编代码为机器码的程序。源码可见 related_source_code/Generator.java，此处大概说明程序实现原理。

a. 首先读入每一条指令，然后分离出指令的操作码，根据操作码来分情况处理，得到各自对应的 rs, rt, rd, immediate, target_address 值。以 add 指令为例，其格式为：

   add rs, rt, rd

   先分离出 add，然后分别取出 rs, rt, rd，可以存放在 int 型整数中。而后先把指令存放变量 i 置 0，再分别与 add 的操作码、rs, rt, rd 相与可以得到指令的机器码：

   i |= 0x00000000;
   i |= (rs << 21);
   i |= (rt << 16);
   i |= (rd << 11);

此处所得 i 即为指令对应机器码。而后把每一条指令当作是一个整数，按顺序放入一个指令数组中。跳转指令则预留一个空位。

b. 对于数据段的处理：只需把每个数字存起来即可，同时记录偏移量，扫完数据段时便可填入文件的 32-63bit 的值，即 0x00000200 + (data_number - 2) * 4，此处 data_number 为最后一个数据段数字的偏移量。

c. 分支指令和跳转指令的处理：由于指令可能往回或者往前跳，因此等扫完一遍指令时再统一作处理（回填）。第一次遇到时只需要记录跳转的信息，包括类型(beqz 或者 jump)，跳转的目的标号，该指令所在的偏移量，如果是 beqz 则仍需要记录比较的寄存器。遇到标号时则记录该标号对应的偏移量（使用 HashMap 可以得到良好的查询效率）。

   当扫完一遍代码时，将记录的跳转指令一一处理即可。跳转指令获取与偏移量的关系为：

   addr = 0x00000200 + (offset - 1) * 4;
   addr >>= 2;
   addr &= 0x03FFFFFF; //此处为 jump，故取 26 位

d. 由于得到的指令数组为 int 型，需要以二进制形式写入文件。不过值得注意的是 Java 采用小尾存储，而模拟器需要的是将指令作为大尾存储的文件，因此我先将指令数组输出到一个文件中，再编写了另外一个 C++程序来完成二进制文件的转换。见 related_source_code/compiler.cpp。实现原理较为简单，把上一步所得的文件一次一个整数读入再以二进制写入目的文件。

6) simplescalar 模板实现

实习提供的 demo 有很大的帮助，给出了一些定义指令行为的方法，配合源程序中的注释可以快速的入门如何实现编码。分支和跳转指令我借鉴了原 arm.def 中的方法。需要注意的是原来的取立即数的定义并没有作符号扩展，因此并不能处理负数的情况，因此在 myisa.h 文件我修改了对其的定义：

```
#define IMME      (((signed long)((inst & 0xffff) << 16)) >> 16)
```

即先左移 16 位，再强转为符号数，最后再右移 16 位，完成符号扩展工作。不过后来发现头文件中有提供符号扩展的宏 SEXT(Num)。☹

另外为了实现自定义的 beqz 和 jump 指令，需要取得指令的后 21 和 26 位，同样的方法作符号扩展，不再赘述。各自对应的宏为 BEQZ_ADDR 和 JUMP_ADDR：

```
#define BEQZ_ADDR       (((signed long)((inst & 0x3fffff) << 12)) >> 12)
#define JUMP_ADDR       (((signed long)((inst & 0x3fffff) << 6)) >> 6)
```

指令模板源码具体可见 myisa/myisa.def 和 myisa.h。要使用两个文件只需把两个文件放入对应目录，再与 sim-safe 一起编译。经过模拟器验证冒泡排序跟矩阵乘法均执行正确。

7) 流水线技术分析

假如使用教材的五级流水线技术来实现我的指令系统，经分析，汇编代码实际上存在大量的数据冒险和控制冒险。例如 Bubble Sort 的代码：

CODE SEG:

```
1   addi $0, $3, 512
2   lw $4, 0($3)
3   addi $4, $10, -2
4   addi $3, $1, 4
5   add $0, $4, $2
6   swi 2     #above display original array
7   add $0, $0, $5      #outer loop
8   outer:
9   slet $5, $10, $9
10  beqz $9, exit
11  addi $0, $6, 1      #inner loop
12  add $0, $3, $12
13  addi $12, $12, 4
14  sub $4, $5, $11
15  addi $11, $11, -1   #set $11 = length - i - 1
16  inner:
17  slet $6, $11, $9
18  beqz $9, next_i
19  lw $8, 0($12)
```

```
20  lw $7, 4($12)
21  slet $7, $8, $9
22  beqz $9, next_j
23  add $0, $8, $9
24  add $0, $7, $8
25  add $0, $9, $7
26  sw $8, 0($12)
27  sw $7, 4($12)
28  next_j:
29  addi $6, $6, 1
30  addi $12, $12, 4
31  jump inner
32  next_i:
33  addi $5, $5, 1
34  jump outer
35  exit:
36  addi $3, $1, 4
37  add $0, $4, $2
38  swi 2     #above display sorted array
39  swi 1
CODE END
```

其中 1, 2, 3 行红色标记处存在两处数据冒险，因为 2 需要等待 1 计算出$3 的值，不过采用数据旁路的方法即可消除所有的冒险；而 2, 3 和 19, 20, 21 行为加载-使用型数据冒险，对流水线的影响更大；黄色标记处存在控制冒险。

对于红色标记处，可以考虑重新排列代码的方法，将不会造成流水线阻塞的代码插入其中；而对于黄色标记处可以采用预测分析的方法消除冒险，也可以采用延迟处理的技术消除冒险。则采用代码重排的方法，上面的代码第一处红色部分可以优化为如下：

```
1   addi $0, $3, 512
2   lw $4, 0($3)
3   addi $3, $1, 4
4   addi $0, $0, $5
5   addi $4, $10, -2
6   add $0, $4, $2
7   swi 2
```

而第二部分的红色可以通过把后面的代码往前提的方法（其实是无论跳转还是不跳转都必须执行的代码），完全消除了冒险，优化为：

```
21  lw $7, 4($12)
22  lw $8, 0($12)
23  addi $6, $6, 1
24  addi $12, $12, 4
25  slet $7, $8, $9
26  beqz $9, next_j
```

优化过的适合于流水线的 Bubble Sort 程序请见 related_source_code 文件夹下的 optimized_bubble_sort.s 文件，此处不再一一列举分析所有的冒险。

总的来说，经过优化消除了三处加载-使用数据冒险，其中有两处是在循环内部的，假设数组的长度为 n，那么冒泡排序的平均比较次数为：(n - 1) + (n - 2) +…+1 = n(n-1)/2。加上刚程序开始部分消除的一个冒险，因此总的提前了 1 + n(n-1)/2 个周期完成程序。

二、 解决的困难和解决方法

1) 指令系统设计：本来刚开始想设计仿 VAX 的指令系统，不过后来考虑到变长指令的流水线性能比较差而且变长指令设计的难度比较大（如果能给一个变长指令系统的 demo 估计难度会小很多☺），因此后来改设计仿 MIPS 指令系统☹；
2) 二进制机器码生成：这个真是体力活……起初采用手动翻译分支和跳转指令的办法，不过到后来出 bug 调试时才发现每一次都要手动填一次太过麻烦，仔细分析后采用拉链回填的技术成功解决了这个问题；
3) 指令模板编写：主要从实习给的 demo 获得入门，而且 simplescalar 源码中的注释非常详细，多花点时间阅读会大有裨益，可谓磨刀不误砍柴工。
4) 汇编代码调试：个人感觉这个比较恶心，sim-safe 模拟器说是提供了反汇编功能对于自己设计的指令系统并不能工作，因此得一步一步的跟踪程序的运行和寄存器的状态。主要使用 step + regs + dump 等指令相结合来跟踪调试程序，不过使用的习惯了也是蛮好用的☺。

三、 实习体会

本次实习让我切实体会了当指令系统设计师的感觉，从指令系统设计到指令实现到数据通路的构建等方方面面。虽然只是从模拟器实现了两个简单的程序而并没有在硬件上具体构建，不过从全局的视觉来看本次实习让我受益匪浅☺。