Lab Report - 01

PB20111686 Ruixuan Huang

Lab Goal

The task of the lab is to realize multiplication and write the machine code. The two operands are placed in R0 and R1 respectively, and the results should be stored in R7. We do not limit the state of other registers (i.e. unlimited end state).

Initial state: R0 and R1 store the number to be calculated, and all other registers are 0.

L-version Code

There's a obvious idea about multiplication. Here I give the pseudo code first.

```
// initial
initial R0, R1 as operand;
initial R2~R7 as zero;

// my algorithm
if (R1 > 0) {
    R7 += R0;
    R1 -= 1;
}
if (R1 < 0) {
    R7 -= R0;
    R1 += 1;
}
goto start;
```

According to the pseudo code above, I could give the machine code as follows.

```
0001 011 011 1 11111 ; x3000, R3<- -1
0001 010 001 1 00000 ; x3001, R2<-R1, set condition code

0000 110 000000011   ; x3002, if R1 <= 0, then go to x3006

0001 111 000 000 111 ; x3003, R7 <- R0 + R7
0001 001 011 000 001 ; x3004, R1 <- R3 + R1
0000 001 111111101   ; x3005, if R1 > 0, then goto x3003

0000 010 000000101   ; x3006, if R1 = 0, goto halt
1001 000 000 111111  ; x3007, R0 <- ~R0
0001 000 000 1 00001 ; x3008, R0 <- R0 + 1

0001 111 000 000 111 ; x3009, R7 <- R0 + R7
0001 001 001 1 00001 ; x3010, R1 <- R1 + 1
0000 100 111111101   ; x3011, if R1 < 0, then goto x3009
```

Optimization

Note that the contents of the two cyclic branch operations are similar, which inspires us to think about the practical principle of 2's complement multiplication, rather than just the value represented by 2's complement itself.

> ***Values interpreted by 2's complement and unsigned numbers***
>
> *We know that if an unsigned number is positive, its 2's complement representation is the same as itself. On the contrary, some changes should be made. Observe the following results (Since presenting a 16-bit code is rather complicated, here I use 4-bit as example).*

	1111	*1011*	*1000*
interpreted as 2's complement	*-1*	*-5*	*-8*
interpreted as unsigned	*15*	*11*	*8*

We can draw a conclusion that when a number is negative in 2's complement, its 2's complement representation and unsigned representation have the relation below.

$$[\text{unsigned}](code) = [\text{2's complement}](code) + 2^n$$

Here n is the length of the code of number.

What's the difference between unsigned and 2's complement multiplication

We know the principle of unsigned number mutiplication, the result of one unsigned number multiplies one another is regular (still an unsigned number, if not overflowing). Then we can transform the 2's complement multiplication to unsigned numbers multiplication, and since the value of LC-3 is stored in only 16 bits, the operations here are carried out in the case of module 65536. In this case, we can concluded below.

$$(m_1 + k_1 \cdot 2^n) \times (m_2 + k_2 \cdot 2^n) \equiv m_1 \times m_2 (\text{mod } 2^n)$$

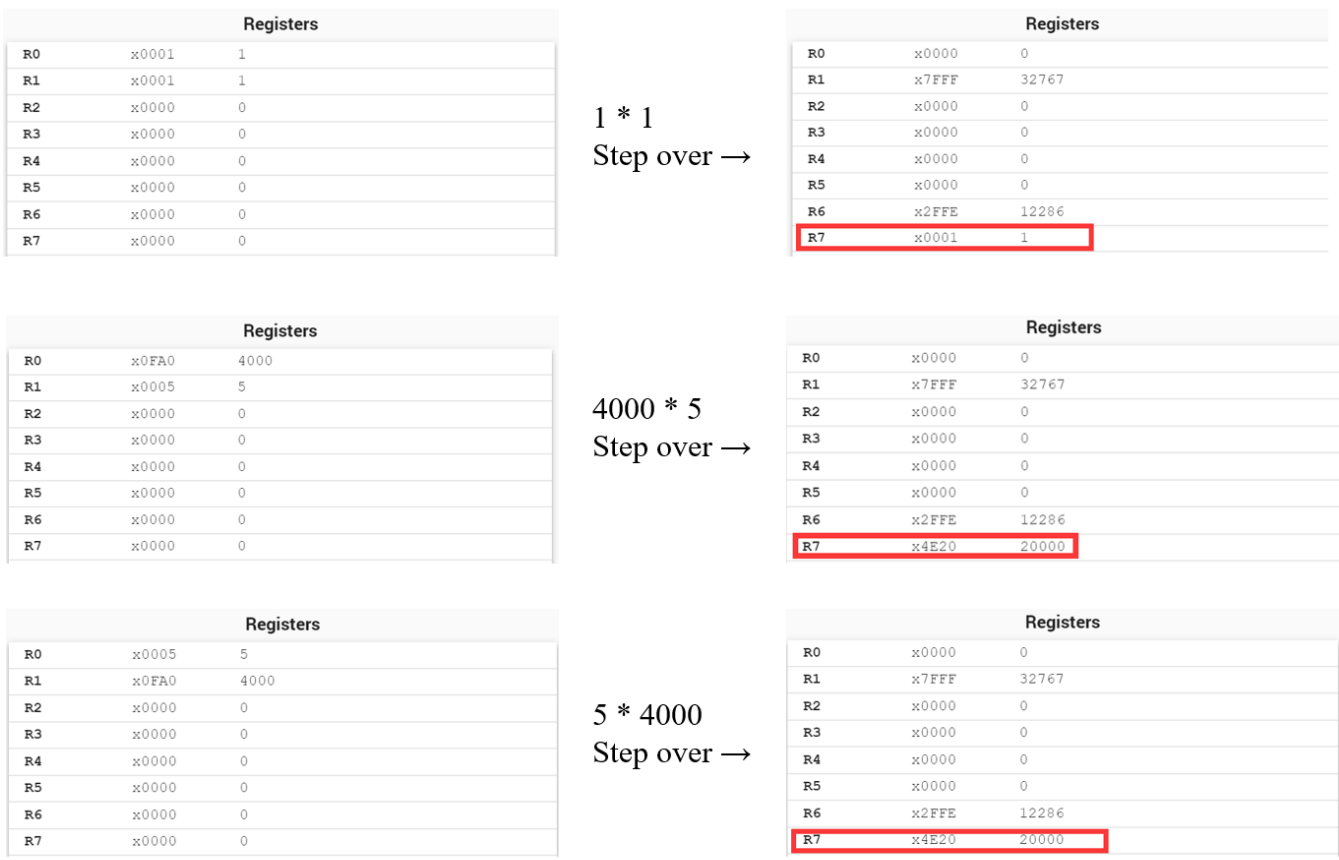
Note that whether a or $a + 2^n$ is a representative element of $\{a + k \cdot 2^n\}$, so unsigned mutiplication is equaled with the 2's complement multiplication. We can just write loop as if R0 and R1 are unsigned number!

According to the principle in the quotation, I could give the machine code as follows.

```
0001 111 000 000 111 ; x3000, R7 = R0 + R7
0001 001 001 1 11111 ; x3001, R0 = R0 - 1
0000 101 111111101   ; x3002, if R0 != 0, go to x3000
```

The original L-version program used 12 lines of instructions. After optimization, the final L-version program used 3 lines of instructions.

Verify the samples in LC-3tools



Registers		
R0	xFE0C	65036
R1	x01B1	433
R2	x0000	0
R3	x0000	0
R4	x0000	0
R5	x0000	0
R6	x0000	0
R7	x0000	0

-500 * 433
Step over →

Registers		
R0	x0000	0
R1	x7FFF	32767
R2	x0000	0
R3	x0000	0
R4	x0000	0
R5	x0000	0
R6	x2FFE	12286
R7	xB24C	45644

Using a program in C + + to verify the correctness of this test.

```
#include <iostream>

int main() {
    int tmp = (-500 * 433) % 65536;
    std::cout << tmp + 65536;
}
```

```
Microsoft Visual Studio 调试控制台
45644
C:\Users\Sprout\source\repos\大一
要在调试停止时自动关闭控制台，请
按任意键关闭此窗口. . .
```

Registers		
R0	xFF8E	65422
R1	xFF17	65303
R2	x0000	0
R3	x0000	0
R4	x0000	0
R5	x0000	0
R6	x0000	0
R7	x0000	0

-114 * -233
Step over →

Registers		
R0	x0000	0
R1	x7FFF	32767
R2	x0000	0
R3	x0000	0
R4	x0000	0
R5	x0000	0
R6	x2FFE	12286
R7	x67C2	26562

P-version Code

The average number of instructions of my L-version program is directly proportional to the size of R0/R1. That's to say if R0 is really big, the average number of instructions of my program will be unimaginable. But looking back on our own multiplication process, we adopt the column vertical method (as the picture below), and the number of loops should not exceed the width of the instruction. The P-version code will be designed by the idea above. More specifically, based on the fact that we can adopt the column vertical method to multiply two unsigned numbers and unsigned mutiplication is equaled with the 2's complement multiplication, we can adopt the column vertical method to multiply two 2's complement numbers.

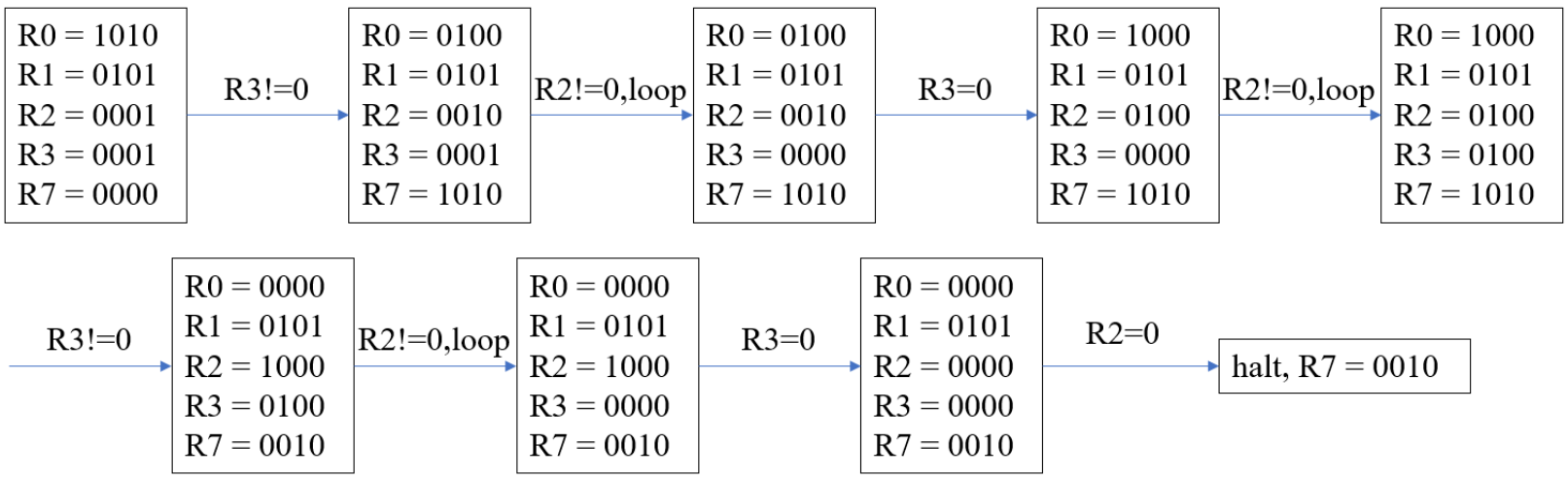
$$
\begin{array}{r}
1\ 0\ 1\ 0 \\
\times\ 0\ 1\ 0\ 1 \\
\hline
1\ 0\ 1\ 0 \\
0\ 0\ 0\ 0 \\
1\ 0\ 1\ 0 \\
0\ 0\ 0\ 0 \\
\hline
1\ 1\ 0\ 0\ 1\ 0
\end{array}
$$

Using one variable as mask to get each bit of R1. If the current bit is zero, just left-shift R0. If the current bit is one, then add R0 to R7 and left-shift R0. Once the mask overflowed, it will be zero. Examining the condition code after left-shifting mask to determine whether the task finished or not.

```
0001 010 010 1 00001 ; x3000, R2 <- 1, R2 is the mask

0101 011 001 000 010 ; x3001, R3 <- mask(R2) & R1
0000 010 000000001   ; x3002, if R3 = 0, goto x3004
0001 111 111 000 000 ; x3003, R7 <- R7 + R0
0001 000 000 000 000 ; x3004, R0 <- R0 + R0, left-shifting
0001 010 010 000 010 ; x3005, R2 <- R2 + R2, left-shifting
0000 101 111111010   ; x3006, if R2 != 0, goto x3001
```

I use the example (1010 × 0101, assuming the width is 4-bit) above to examine the correction of my code.



Evaluation 1

Times of loop is a constant - the width of instruction - 16. One loop contains 6 instructions. The instruction x3003 will be carried 0.5 times in average. So the average number of instructions is

$$1 + 16 \times (5 + 0.5) = 89$$

Optimization 1

Since there's no limitation of lines of instructions, and times of loop is a constant, we can directly expand the loop to reduce the additional instruction consumption caused by examining whether R2 is zero.

```
0001 010 010 1 00001 ; R2 <- 1, R2 is the mask

0101 011 001 000 010 ; R3 <- mask(R2) & R1
0000 010 000000001   ; if R3 = 0, ignore next instruction
0001 111 111 000 000 ; R7 <- R7 + R0
0001 000 000 000 000 ; R0 <- R0 + R0, left-shifting
0001 010 010 000 010 ; R2 <- R2 + R2, left-shifting

0101 011 001 000 010 ; R3 <- mask(R2) & R1
0000 010 000000001   ; if R3 = 0, ignore next instruction
0001 111 111 000 000 ; R7 <- R7 + R0
0001 000 000 000 000 ; R0 <- R0 + R0, left-shifting
0001 010 010 000 010 ; R2 <- R2 + R2, left-shifting

;... Omit 14 identical loop codes
```

Evaluation 2

The average number of instructions is

$$1 + 16 \times (4 + 0.5) = 73$$

Optimization 2

I use R2 to act as mask at the beginning for the consideration of the ADD instruction's only having 5 bits for using immediate number. However, since I abandoned using loop, I can use immediate number for first three parts of the code, which can save 3 lines.

And at the end of the program, last 2 instructions is unnecessary. They were supposed to prepare for the next cycle in the loop-used version.

```
0001 010 010 1 01000 ; R2 <- 01000, R2 is the mask

0101 011 001 1 00001 ; R3 <- 1 & R1
0000 010 000000001   ; if R3 = 0, ignore next instruction
0001 111 111 000 000 ; R7 <- R7 + R0
0001 000 000 000 000 ; R0 <- R0 + R0, left-shifting

0101 011 001 1 00010 ; R3 <- 10 & R1
0000 010 000000001   ; if R3 = 0, ignore next instruction
0001 111 111 000 000 ; R7 <- R7 + R0
```

```
0001 000 000 000 000 ; R0 <- R0 + R0, left-shifting

0101 011 001 1 00100 ; R3 <- 100 & R1
0000 010 000000001   ; if R3 = 0, ignore next instruction
0001 111 111 000 000 ; R7 <- R7 + R0
0001 000 000 000 000 ; R0 <- R0 + R0, left-shifting

0101 011 001 000 010 ; R3 <- mask(R2) & R1
0000 010 000000001   ; if R3 = 0, ignore next instruction
0001 111 111 000 000 ; R7 <- R7 + R0
0001 000 000 000 000 ; R0 <- R0 + R0, left-shifting
0001 010 010 000 010 ; R2 <- R2 + R2, left-shifting

;... Omit 11 identical loop codes

0101 011 001 000 010 ; R3 <- mask(R2) & R1
0000 010 000000001   ; if R3 = 0, ignore next instruction
0001 111 111 000 000 ; R7 <- R7 + R0
```

Evaluation 3

The average number of instructions is

$$1 + 3 \times (3 + 0.5) + 12 \times (4 + 0.5) + 1 \times (2 + 0.5) = 68$$

Verify the samples in LC-3tools

Registers		
R0	x0001	1
R1	x0001	1
R2	x0000	0
R3	x0000	0
R4	x0000	0
R5	x0000	0
R6	x0000	0
R7	x0000	0

1 * 1
Step over →

Registers		
R0	x0000	0
R1	x7FFF	32767
R2	x8000	32768
R3	x0000	0
R4	x0000	0
R5	x0000	0
R6	x2FFE	12286
R7	x0001	1

Registers		
R0	x0FA0	4000
R1	x0005	5
R2	x0000	0
R3	x0000	0
R4	x0000	0
R5	x0000	0
R6	x0000	0
R7	x0000	0

4000 * 5
Step over →

Registers		
R0	x0000	0
R1	x7FFF	32767
R2	x8000	32768
R3	x0000	0
R4	x0000	0
R5	x0000	0
R6	x2FFE	12286
R7	x4E20	20000

Registers		
R0	x0005	5
R1	x0FA0	4000
R2	x0000	0
R3	x0000	0
R4	x0000	0
R5	x0000	0
R6	x0000	0
R7	x0000	0

5 * 4000
Step over →

Registers		
R0	x0000	0
R1	x7FFF	32767
R2	x8000	32768
R3	x0000	0
R4	x0000	0
R5	x0000	0
R6	x2FFE	12286
R7	x4E20	20000

Registers		
R0	xFE0C	65036
R1	x01B1	433
R2	x0000	0
R3	x0000	0
R4	x0000	0
R5	x0000	0
R6	x0000	0
R7	x0000	0

-500 * 433
Step over →

Registers		
R0	x0000	0
R1	x7FFF	32767
R2	x0000	0
R3	x0000	0
R4	x0000	0
R5	x0000	0
R6	x2FFE	12286
R7	xB24C	45644

Registers		
R0	xFF8E	65422
R1	xFF17	65303
R2	x0000	0
R3	x0000	0
R4	x0000	0
R5	x0000	0
R6	x0000	0
R7	x0000	0

-114 * -233
Step over →

Registers		
R0	x0000	0
R1	x7FFF	32767
R2	x0000	0
R3	x8000	32768
R4	x0000	0
R5	x0000	0
R6	x2FFE	12286
R7	x67C2	26562