嵌入式系统设计

CPU Lab 实验报告

软件学院 嵌入式软件方向

王旸 11331300

2013-12-18

目录

指令集设计

指令集设计思路

在这个实验中，我除了实现实验要求中已经规定好的指令格式，以及实现了实验要求中给出要求但是没给定好格式的指令之外，增加了 5 条指令 NOT，NAND，NOR，XNOR，LDIL，分别实现取否、与非、或非、同或以及直接向寄存器存入立即数。

operation code 对于大多数操作数来说都是有规律的：在我的设计当中，NOP、HALT、LOAD、STORE 这四条与外部电路、控制电路比较有关的指令，占据的是 000xx 的指令范围，也就是说都是以 000 开头的，而与跳转有关的 JUMP、JMPR、BZ、BNZ、BN、BNN、BC、BNC 等指令，则占用了 11xxx 的范围，与逻辑运算有关的 CMP，AND，OR，XOR 则占用了 011xx，与移位运算有关的 SLL，SRL，SLA，SRA 则占用了 001xx，

因此，我的 NOT，NAND，NOR，XNOR 操作，采用了 101xx 的指令编码范围。而 LDIL 则使用了 10011 编码，LDIL 指令我觉得是非常有作用的，因为它可以直接在寄存器中存入立即数，而不像 ADDC 一样，只有在 R1 为 0 的时候才能真正存入立即数。使用 LDIL 搭配 LDIH，可以用 2 条指令就向寄存器中存入立即数，所以我采取了这个指令。

先前曾经设想增加乘法指令，但是采用乘法指令非常消耗时间、面积、能源等资源，而且实现起来非常不方便，采用流水线方式的话又很难与其他指令兼容。因此我就偷懒没有完成乘法器的设计。

具体的指令细节将在下一面给出。

指令集图表

具体指令集设计如下页图：

NOT 10100 r1 r2 —

NAND 10101

NOR 10110

XNOR 10111

LDIL 10011 [r1] = {0x0, 0x0, V2, V3}

ZF (zero flag) NF (negetive flag)

CF (carry flag)

		15 11	10 8	7 4	3 0	
002--	NOP	00000				
	HALT	00001				
	LOAD	00010	r1	r2	value3	[r1] = m([r2]+value3)
	STORE	00011	r1	r2	value3	m([r2]+value3) = [r1]
	LDIH	10000	r1	value2	value3	[r1] = [r1] + {V2, V3, 0x0, 0x0}
	ADD	01000	r1	r2	r3	[r1] = [r2] + [r3]
	ADDI	01001	r1	value2	value3	[r1] = [r1] + {0x0, 0x0, V2, V3}
	ADDC	10001	r1	r2	r3	[r1] = [r2] + [r3] + CF
	SUB	01010	r1	r2	r3	[r1] = [r2] - [r3]
	SUBI	01011	r1	value2	value3	[r1] = [r1] - {0x0, 0x0, V2, V3}
	SUBC	10010	r1	r2	r3	[r1] = [r2] + ~[r3] + CF - 1
011--	CMP	01100		r2	r3	[r2] - [r3], 不回写, 只设 FLAG
	AND	01101	r1	r2	r3	[r1] = [r2] & [r3]
	OR	01110	r1	r2	r3	[r1] = [r2] | [r3]
	XOR	01111	r1	r2	r3	[r1] = [r2] ^ [r3]
001--	SLL	00100	r1	r2	value3	[r1] = [r2] << value3 (逻辑)
	SRL	00110	r1	r2	value3	[r1] = [r2] >> value3 (逻辑)
	SLA	00101	r1	r2	value3	[r1] = [r2] << value3 (算术)
	SRA	00111	r1	r2	value3	[r1] = [r2] >>> value3 (算术)
11---	JUMP	11000		value2	value3	jump to {value2, value3}
	JUMPR	11001	r1	value2	value3	jump to r1 + {value2, value3}
	BZ	11010	r1	value2	value3	(ZF=1?) jump to r1 + {value2, value3}
	BNZ	11011	r1	value2	value3	(ZF=0?) jump to r1 + {value2, value3}
	BN	11100	r1	value2	value3	(NF=1?) jump to r1 + {value2, value3}
BNN	~~BP~~	11101	r1	value2	value3	(NF=0?) jump to r1 + {value2, value3}
BC	~~BNC~~	11110	r1	value2	value3	(CF=1?) jump to r1 + {value2, value3}
	BNC	11111	r1	value2	value3	(CF=0?) jump to r1 + {value2, value3}

效率分析

面积分析

因为最后我个人增加了 memory 模块（将在后续部分介绍），所以在这里仅对 cpu 模块进行分析，优化方面除了对于 IR 的流水线传送位数进行优化之外，也利用指令集编码的特点，为电路中判断状况作了优化。比如说，所有 11xxx 开头的代码就是跳转指令，这可以帮助节省判断的位数以节省面积。

但是其实我非常怀疑这种类似卡诺图化简的优化方式是否已经在综合过程中完成了 ,因此对于面积方面的优化可以说我做的还是比较少的。

Device Utilization Summary			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	301	18,224	1%
Number used as Flip Flops	283		
Number used as Latches	17		
Number used as Latch-thrus	1		
Number used as AND/OR logics	0		
Number of Slice LUTs	521	9,112	5%
Number used as logic	507	9,112	5%
Number using O6 output only	484		
Number using O5 output only	0		
Number using O5 and O6	23		
Number used as ROM	0		
Number used as Memory	0	2,176	0%
Number used exclusively as route-thrus	14		
Number with same-slice register load	14		
Number with same-slice carry load	0		
Number with other load	0		
Number of occupied Slices	192	2,278	8%
Number of LUT Flip Flop pairs used	551		
Number with an unused Flip Flop	276	551	50%
Number with an unused LUT	30	551	5%
Number of fully used LUT-FF pairs	245	551	44%
Number of unique control sets	6		
Number of slice register sites lost to control set restrictions	20	18,224	1%
Number of bonded IOBs	69	232	29%
Number of RAMB16BWERs	0	32	0%
Number of RAMB8BWERs	0	64	0%
Number of BUFIO2/BUFIO2_2CLKs	0	32	0%

时间分析

关于提升速率方面，我们需要找到的是耗时间最长的流水线级，根据分析我们可以知道，在 ALU 这一级，需要耗时比较长。因此我做的最大的优化，就是将 ALU 这一级的任务，尽量移入控制器这一级来完成。比如说：如果需要计算两个数相减，那么明显需要将被减数取反，那么我就将取反这一步移入控制器中完成，直接在 reg_B 计算的时候就取反完成，这样尽量只让 ALU 完成他应该完成的事情，应该有助于增加速率。

Clock to Setup on destination clock clock

Source Clock	Src:Rise|Dest:Rise	Src:Fall|Dest:Rise	Src:Rise|Dest:Fall	Src:Fall|Dest:Fall
clock	4.739			

功耗分析

功耗方面进行的优化，主要就是减少没有意义的 0，1 转换，也就是说，在寄存器的值不需要变化的时候，我优先生成一个锁存器来保存数据。

2. Summary

2.1.

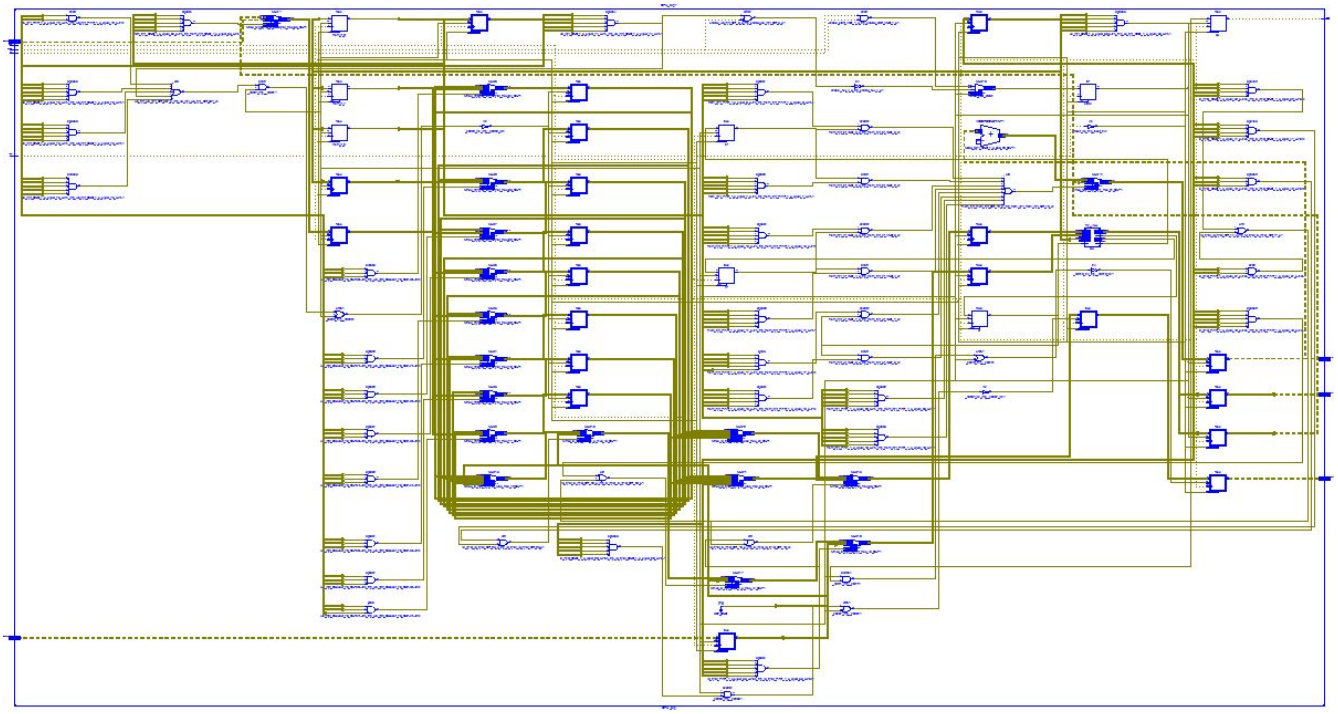
On-Chip Power Summary

On-Chip	Power (mW)	Used	Available	Utilization (%)
Clocks	0.03	3	---	---
Logic	0.00	521	9112	6
Signals	0.00	631	---	---
IOs	0.00	69	232	30
Quiescent	14.84			
Total	14.88			

RTL 电路图

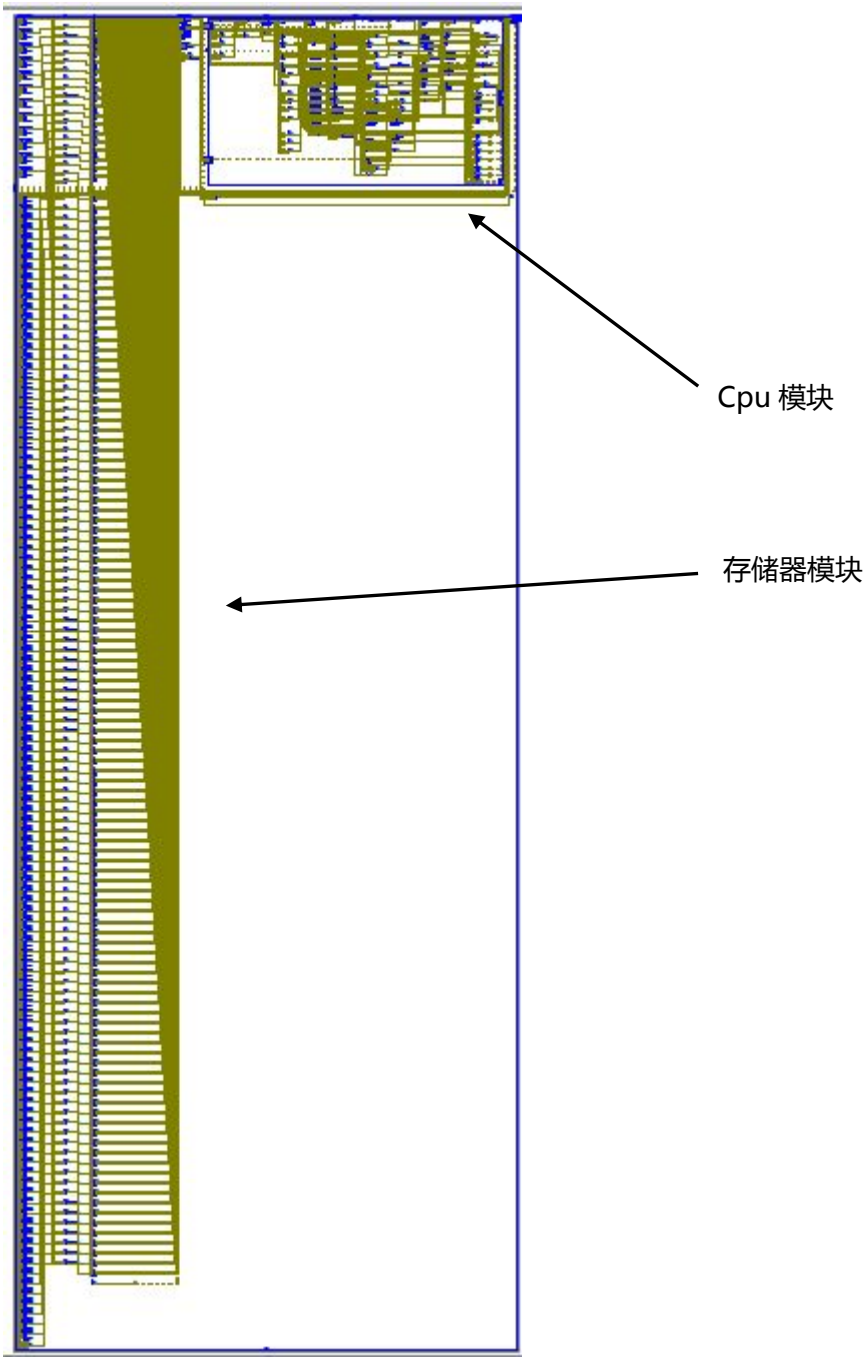
CPU 的 RTL 电路图

CPU 的电路图如下



Memory 模块的 RTL

在我的设计中，增加了一个模块，称为 Memory 模块，其作用是为 CPU 模块提供两个存储模块：指令存储器以及数据存储器。具体将在下一部分说明，其抽象层 RTL 电路图如下：



仿真测试

直接仿真

首先我按照实验要求进行了一轮仿真，具体效果大概如下图：

```
pc: id_ir:              reg_A:  reg_B:  reg_C:  da: dd:     dw: reC1:   gr1:    gr2:    gr3
Finished circuit initialization process.
xx: xxxxxxxxxxxxxxxx:   xxxx:   xxxx:   xxxx:   xx: xxxx:   x:  xxxx:   xxxx:   xxxx:   xxxx
00: 0000000000000000:   0000:   0000:   0000:   00: 0000:   0:  0000:   0000:   0000:   0000
01: 0001000100000000:   0000:   0000:   0000:   00: 0000:   0:  0000:   0000:   0000:   0000
02: 0001001000000001:   0000:   0000:   0000:   00: 0000:   0:  0000:   0000:   0000:   0000
03: 0000000000000000:   0000:   0001:   0000:   00: 0000:   0:  0000:   0000:   0000:   0000
04: 0000000000000000:   0000:   0000:   0001:   01: 0000:   0:  00ab:   0000:   0000:   0000
05: 0000000000000000:   0000:   0000:   0000:   00: 0000:   0:  3c00:   00ab:   0000:   0000
06: 0100001100010010:   0000:   0000:   0000:   00: 0000:   0:  0000:   00ab:   3c00:   0000
07: 0000000000000000:   00ab:   3c00:   0000:   00: 0000:   0:  0000:   00ab:   3c00:   0000
08: 0000000000000000:   0000:   0000:   3cab:   ab: 0000:   0:  0000:   00ab:   3c00:   0000
09: 0000000000000000:   0000:   0000:   0000:   00: 0000:   0:  3cab:   00ab:   3c00:   0000
0a: 0001101100000010:   0000:   0000:   0000:   00: 0000:   0:  0000:   00ab:   3c00:   3cab
0b: 0000100000000000:   0000:   0002:   0000:   00: 0000:   0:  0000:   00ab:   3c00:   3cab
0c: 0000100000000000:   0000:   0000:   0002:   02: 3cab:   1:  0000:   00ab:   3c00:   3cab
0d: 0000100000000000:   0000:   0000:   0000:   00: 3cab:   0:  0002:   00ab:   3c00:   3cab
0e: 0000100000000000:   0000:   0000:   0000:   00: 3cab:   0:  0000:   00ab:   3c00:   3cab
0f: 0000100000000000:   0000:   0000:   0000:   00: 3cab:   0:  0000:   00ab:   3c00:   3cab
```

这是我将 ISM 中的内容复制之后排版得到的，可以清楚看到流水线的运行以及各种计算的进行。

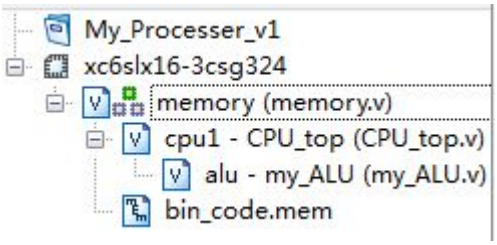
但是这种测试方式非常麻烦。主要在于需要手动对内存进行模拟，需要什么数据、什么指令的时候，都需要手动进行赋值。

因此，我对大多数操作正确性的验证，采用了即将介绍的外一种方式。

使用 Memory 模块仿真

Memory 模块结构介绍

新增模块 memory 在整个电路中的是处于包装 CPU 模块的，具体结构如下：



其中，memory 模块包含了指令存储器，以及数据存储器，并且调用了 CPU 模块，将它们连接在一起。而 cpu 模块调用了 ALU 模块，也就是算数逻辑单元，进行数学计算。这符合哈佛结构的计算机组成方式，也就是将指令与数据分开存储。在 bin_conde.mem 中，存储的是在初始化阶段将写入指令寄存器的初始值，也就是二进制代码。

在 memroy 模块中，存储器是使用 register 来实现的，是两个 128 个 16 位寄存器组成的阵列，具体实现代码如下：

```
11  reg [15:0] i_memory[0:127];
12  reg [15:0] d_memory[0:127];
```

并且在初始化阶段，采用$readmemb 方法初始化数据，具体代码如下：

```
26  initial
27     $readmemb("bin_code.mem", i_memory);
```

这样一来，就能将 bin_code.mem 中的内容直接加载到寄存器阵列中了。要进行编程只需要对 bin_code.mem 中的内容进行编辑，系统就能自动的运行，而且因为有模拟生成的数据内存，也可以直接对内存进行 LOAD，STORE 操作，不需要再进行人工模拟。

使用高级语言实现的汇编器

为了更加方便代码的书写，我使用 python 实现了一个非常简单的汇编器，其功能非常简单，对错误也没有任何处理，所有错误导致的结果都是未定义的，而且对语法也有非常严格的要求（比如只能使用一个空格，不能有空行之类的），但是它可以显著的增加我们书写代码的效率。

程序从系统读入写有汇编代码的文件，并且直接生成对应的机器码

```
assemble code:
sub r0 r0 r0
ldih r0 0 8
addi r0 3 0
addi r0 0 4
store r0 r1 0
addi r0 0 1
store r0 r1 1
store r0 r1 4
store r1 r1 0
load r1 r1 1
store r1 r2 6

assembleing...

01010_000_0000_0000
10000_000_0000_1000
01001_000_0011_0000
01001_000_0000_0100
00011_000_0001_0000
01001_000_0000_0001
00011_000_0001_0001
00011_000_0001_0100
00011_001_0001_0000
00010_001_0001_0001
00011_001_0010_0110
[Finished in 0.6s]
```

直接将生成的代码存入 bin_code.mem 即可使用 ISM 进行模拟。

仿真结果

因为在测试中，我们对 data memory 的观察，仅限于当 data memory 的内容变更的时候，因此我将每个运算结果得到的值存入内存之后再观察。

LDIH，LDIL，STORE，NOP

首先对 LDIH，LDIL，STORE 3 个指令进行仿真：

```
assemble code

ldil r0 9 3
ldil r1 0 0
nop
nop
ldih r0 3 5
ldih r1 0 0
nop
nop
nop
store r0 r1 0
```

采用的汇编代码

```
10011_000_1001_0011
10011_001_0000_0000
00000_000_0000_0000
00000_000_0000_0000
10000_000_0011_0101
10000_001_0000_0000
00000_000_0000_0000
00000_000_0000_0000
00000_000_0000_0000
00011_000_0001_0000
```

对应的机器码

```
ISim M.70d (signature 0x36e8438f)
This is a Full version of ISim.
Time resolution is 1 ps
Simulator is doing circuit initialization process.
Finished circuit initialization process.
DATA    :xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx
DATA    :3593 xxxx xxxx xxxx xxxx xxxx xxxx xxxx
ADDRESS:m[0]:m[1]:m[2]:m[3]:m[4]:m[5]:m[6]:m[7]
ISim>
```

内存状态

其中 DATA 代表数据内存的数据变化，而 ADDRESS 代表对应的地址，我只让其显示出了前 8 位内存的状态。

可以看出，这条汇编指令的目的，是将 R0 置为 0x3593，然后将 R1 置为 0x0000，随后使用 R1+0=0 这个地址值来保存 R0 的值，可以采用如下的伪代码来表示：

R[0] = 0x0093

R[1] = 0x0000

R[0] = R[0] + 0x3500

R[1] = R[1] + 0x0000

M [R[1] + 0] = R[0]

可从内存状态中看出，该代码执行的结果是正确的。

ADD，ADDI，LOAD，HALT

具体如下：

```
ldil r1 9 3
ldil r0 0 0
nop
nop
ldih r1 3 5
ldih r0 0 0
nop
nop
nop
store r1 r0 0
addi r1 1 1
load r2 r0 0
nop
nop
nop
add r3 r1 r2
store r1 r0 0
store r2 r0 1
nop
store r3 r0 2
halt
```

```
10011_001_1001_0011
10011_000_0000_0000
00000_000_0000_0000
00000_000_0000_0000
10000_001_0011_0101
10000_000_0000_0000
00000_000_0000_0000
00000_000_0000_0000
00000_000_0000_0000
00011_001_0000_0000
01001_001_0001_0001
00010_010_0000_0000
00000_000_0000_0000
00000_000_0000_0000
00000_000_0000_0000
01000_011_0001_0010
00011_001_0000_0000
00011_010_0000_0001
00000_000_0000_0000
00011_011_0000_0010
00001_000_0000_0000
```

```
DATA    :xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx
DATA    :3593 xxxx xxxx xxxx xxxx xxxx xxxx xxxx
DATA    :35a4 xxxx xxxx xxxx xxxx xxxx xxxx xxxx
DATA    :35a4 3593 xxxx xxxx xxxx xxxx xxxx xxxx
DATA    :35a4 3593 6b37 xxxx xxxx xxxx xxxx xxxx
ADDRESS:m[0]:m[1]:m[2]:m[3]:m[4]:m[5]:m[6]:m[7]
ISim>
```

伪代码：

R[1] = 3593

R[0] = 0000

M [R[0] + 0] = R[1]

R[1] = R[1] + 0011

R[2] = M [R[0] + 0]

R[3] = R[1] + R[2]

M [R[0] + 0] = R[1]

M [R[0] + 1] = R[2]

M [R[0] + 2] = R[3]

最终 DATA 值与预想相同，这几条指令的测试成功。

SUB，SUBI

与前一个验证基本相同，只是将所有加法改为减法：

```
ldil r1 9 3
ldil r0 0 0
nop
nop
ldih r1 3 5
ldih r0 0 0
nop
nop
nop
store r1 r0 0
subi r1 1 1
load r2 r0 0
nop
nop
nop
sub r3 r1 r2
store r1 r0 0
store r2 r0 1
nop
store r3 r0 2
halt
```

```
10011_001_1001_0011
10011_000_0000_0000
00000_000_0000_0000
00000_000_0000_0000
10000_001_0011_0101
10000_000_0000_0000
00000_000_0000_0000
00000_000_0000_0000
00000_000_0000_0000
00011_001_0000_0000
01011_001_0001_0001
00010_010_0000_0000
00000_000_0000_0000
00000_000_0000_0000
00000_000_0000_0000
01010_011_0001_0010
00011_001_0000_0000
00011_010_0000_0001
00000_000_0000_0000
00011_011_0000_0010
00001_000_0000_0000
```

```
DATA    :xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx
DATA    :3593 xxxx xxxx xxxx xxxx xxxx xxxx xxxx
DATA    :3582 xxxx xxxx xxxx xxxx xxxx xxxx xxxx
DATA    :3582 3593 xxxx xxxx xxxx xxxx xxxx xxxx
DATA    :3582 3593 ffef xxxx xxxx xxxx xxxx xxxx
ADDRESS:m[0]:m[1]:m[2]:m[3]:m[4]:m[5]:m[6]:m[7]
ISim>
```

因为与上一个测试流程基本相同，不再贴出伪代码。

根据手工计算验证，这些指令工作正确。

ADDC，SUBC

```
ldil r1 0 0
ldil r0 0 0
ldil r2 0 1
nop
ldih r1 9 0
ldih r0 0 0
ldih r2 0 0
nop
addc r3 r2 r2
add r7 r1 r1
addc r4 r2 r2
sub r7 r1 r2
subc r5 r1 r2
sub r7 r2 r1
subc r6 r1 r2
store r3 r0 0
store r4 r0 1
store r5 r0 3
store r6 r0 4
```

```
10011_001_0000_0000
10011_000_0000_0000
10011_010_0000_0001
00000_000_0000_0000
10000_001_1001_0000
10000_000_0000_0000
10000_010_0000_0000
00000_000_0000_0000
10001_011_0010_0010
01000_111_0001_0001
10001_100_0010_0010
01010_111_0001_0010
10010_101_0001_0010
01010_111_0010_0001
10010_110_0001_0010
00011_011_0000_0000
00011_100_0000_0001
00011_101_0000_0011
00011_110_0000_0100
```

```
DATA    :xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx
DATA    :0002 xxxx xxxx xxxx xxxx xxxx xxxx xxxx
DATA    :0002 0003 xxxx xxxx xxxx xxxx xxxx xxxx
DATA    :0002 0003 xxxx 8fff xxxx xxxx xxxx xxxx
DATA    :0002 0003 xxxx 8fff 8ffe xxxx xxxx xxxx
ADDRESS:m[0]:m[1]:m[2]:m[3]:m[4]:m[5]:m[6]:m[7]
ISim>
```

通过控制变量的方法，比较刚刚执行过得到进位的运算，与刚刚执行过不得到进位的运算，进行比较，得到的结论是，这两个指令工作正常。

AND，OR，XOR，NOT，NAND，NOR，XNOR，SLL，SRL，SLA，SRA

这几个指令都是执行比较直接的计算语句，所以我用一个代码来进行检验。

```
ldil r0 0 0
ldil r1 2 0
ldil r2 3 1
nop
ldih r0 0 0
ldih r1 8 4
ldih r2 9 5
nop
nop
nop
and r3 r1 r2
or r4 r1 r2
xor r5 r1 r2
nop
store r3 r0 0
store r4 r0 1
store r5 r0 2
nand r3 r1 r2
nor r4 r1 r2
nxor r5 r1 r2
not r6 r1
store r3 r0 0
store r4 r0 1
store r5 r0 2
store r6 r0 3
sll r3 r1 1
srl r4 r1 2
sla r5 r1 12
sra r6 r1 15
store r3 r0 0
store r4 r0 1
store r5 r0 2
store r6 r0 4
halt
```

```
10011_000_0000_0000
10011_001_0010_0000
10011_010_0011_0001
00000_000_0000_0000
10000_000_0000_0000
10000_001_1000_0100
10000_010_1001_0101
00000_000_0000_0000
00000_000_0000_0000
00000_000_0000_0000
01101_011_0001_0010
01110_100_0001_0010
01111_101_0001_0010
00000_000_0000_0000
00011_011_0000_0000
00011_100_0000_0001
00011_101_0000_0010
10101_011_0001_0010
10110_100_0001_0010
10111_101_0001_0010
10100_110_0001_0000
00011_011_0000_0000
00011_100_0000_0001
00011_101_0000_0010
00011_110_0000_0011
00100_011_0001_0001
00110_100_0001_0010
00101_101_0001_1100
00111_110_0001_1111
00011_011_0000_0000
00011_100_0000_0001
00011_101_0000_0010
00011_110_0000_0100
00001_000_0000_0000
```

```
DATA    :xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx
DATA    :8420 xxxx xxxx xxxx xxxx xxxx xxxx xxxx
DATA    :8420 9531 xxxx xxxx xxxx xxxx xxxx xxxx
DATA    :8420 9531 1111 xxxx xxxx xxxx xxxx xxxx
DATA    :7bdf 9531 1111 xxxx xxxx xxxx xxxx xxxx
DATA    :7bdf 6ace 1111 xxxx xxxx xxxx xxxx xxxx
DATA    :7bdf 6ace eeee xxxx xxxx xxxx xxxx xxxx
DATA    :7bdf 6ace eeee 7bdf xxxx xxxx xxxx xxxx
DATA    :0840 6ace eeee 7bdf xxxx xxxx xxxx xxxx
DATA    :0840 2108 eeee 7bdf xxxx xxxx xxxx xxxx
DATA    :0840 2108 0000 7bdf xxxx xxxx xxxx xxxx
DATA    :0840 2108 0000 7bdf 0001 xxxx xxxx xxxx
ADDRESS:m[0]:m[1]:m[2]:m[3]:m[4]:m[5]:m[6]:m[7]
ISim>
```

在这个测试中，我将不同的计算的值存储在内存中。因为内存的每一次变动都会被显示出来，因此我利用了这一点，用 8 位内存显示出了所有的结果，它们在不同的时间被计算出来。

通过分析可以知道，这几条指令的动作都是正确的。

BZ，BNZ

关于这 2 个控制跳转的寄存器，我编写了实用性比较强的汇编代码，完成软件实现指令集中不存在的基本运算。也就是运用 BNZ ,BZ 实现的乘法器 ,目前这个例子示范的乘法计算 ,是 :0x0035 * 0x0026 = 0x07DE 多次测验得数证明乘法器的运算是正确的，那么 BN，BNZ 的运作就没有问题了：

```
ldil r0 3 5
ldil r1 2 6
ldil r5 0 0
ldil r7 0 1
ldil r4 0 0
sll r3 r1 0
sll r2 r0 0
store r0 r5 0
store r1 r5 1
nop
nop
nop
and r6 r3 r7
bz r5 1 2
nop
nop
nop
add r4 r2 r4
sll r2 r2 1
srl r3 r3 1
bnz r5 0 9
nop
nop
nop
store r4 r5 2
halt
```

```
10011_000_0011_0101
10011_001_0010_0110
10011_101_0000_0000
10011_111_0000_0001
10011_100_0000_0000
00100_011_0001_0000
00100_010_0000_0000
00011_000_0101_0000
00011_001_0101_0001
00000_000_0000_0000
00000_000_0000_0000
00000_000_0000_0000
01101_110_0011_0111
11010_101_0001_0010
00000_000_0000_0000
00000_000_0000_0000
00000_000_0000_0000
01000_100_0010_0100
00100_010_0010_0001
00110_011_0011_0001
11011_101_0000_1001
00000_000_0000_0000
00000_000_0000_0000
00000_000_0000_0000
00011_100_0101_0010
00001_000_0000_0000
```

```
DATA    :xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx
DATA    :0035 xxxx xxxx xxxx xxxx xxxx xxxx xxxx
DATA    :0035 0026 xxxx xxxx xxxx xxxx xxxx xxxx
DATA    :0035 0026 07de xxxx xxxx xxxx xxxx xxxx
ADDRESS:m[0]:m[1]:m[2]:m[3]:m[4]:m[5]:m[6]:m[7]
ISim>
```

可以看到，两个操作数被存储在 R0，R1 两位，然后会进行乘法计算，答案会存储在 R4 中。两个操作数将被存储在 M[0]，M[1]当中（主要是便于观察），并且将答案存储在 M[2]中。

具体伪代码为：

R[0] = 0035;　　R[1] = 0026;　　R[5] = 0000;　　R[7] = 0001;　　R[4] = 0000;

R[3] = R[1];　　R[2] = R[0];

LABEL1:　　R[6] = R[3] & R[7]

　　IF R[6]==0 GOTO LABEL2

　　R[4] = R[4] + R[2]

LABEL2:　　R[2] = R[2] << 1

　　R[3] = R[3] >> 1

　　IF R[3]!=0 GOTO LABEL2

　　M[answer_addr] = R[4]

JUMP，JMPR

```
ldil r7 0 0
ldil r0 0 10
jump 0 7
nop
nop
nop
ldil r0 0 1
nop
nop
nop
store r0 r7 0
jmpr r0 0 6
nop
nop
nop
ldil r0 0 2
nop
nop
nop
store r0 r7 1
halt
```

```
10011_111_0000_0000
10011_000_0000_1010
11000_000_0000_0111
00000_000_0000_0000
00000_000_0000_0000
00000_000_0000_0000
10011_000_0000_0001
00000_000_0000_0000
00000_000_0000_0000
00000_000_0000_0000
00011_000_0111_0000
11001_000_0000_0110
00000_000_0000_0000
00000_000_0000_0000
00000_000_0000_0000
10011_000_0000_0010
00000_000_0000_0000
00000_000_0000_0000
00000_000_0000_0000
00011_000_0111_0001
00001_000_0000_0000
```

```
DATA    :xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx
DATA    :000a xxxx xxxx xxxx xxxx xxxx xxxx xxxx
DATA    :000a 000a xxxx xxxx xxxx xxxx xxxx xxxx
ADDRESS:m[0]:m[1]:m[2]:m[3]:m[4]:m[5]:m[6]:m[7]
ISim>
```

这是一个简单的分支语句的测试，测试结果显示，这两条指令的工作都正常。

BN，BNN，BC，BNC

```
ldil r1 0 1
ldil r0 0 0
ldil r2 3 3
nop
nop
nop
nop
ldih r2 9 0
cmp r1 r0
bnn r0 0 14
nop
nop
nop
store r2 r0 0
cmp r0 r1
bn r0 1 4
nop
nop
nop
store r2 r0 1
add r7 r2 r2
bc r0 1 10
```

```
nop
nop
nop
store r2 r0 2
add r7 r0 r0
bnc r0 2 1
nop
nop
nop
store r2 r0 3
halt
nop
nop
nop
store r2 r0 4
store r2 r0 5
halt
```

```
10011_001_0000_0001
10011_000_0000_0000
10011_010_0011_0011
00000_000_0000_0000
00000_000_0000_0000
00000_000_0000_0000
00000_000_0000_0000
10000_010_1001_0000
01100_000_0001_0000
11101_000_0000_1110
00000_000_0000_0000
00000_000_0000_0000
00000_000_0000_0000
00011_010_0000_0000
01100_000_0000_0001
11100_000_0001_0100
00000_000_0000_0000
00000_000_0000_0000
00000_000_0000_0000
00011_010_0000_0001
01000_111_0010_0010
11110_000_0001_1010
```

```
00000_000_0000_0000
00000_000_0000_0000
00000_000_0000_0000
00011_010_0000_0010
01000_111_0000_0000
11111_000_0010_0001
00000_000_0000_0000
00000_000_0000_0000
00000_000_0000_0000
00000_000_0000_0000
00011_010_0000_0011
00001_000_0000_0000
00000_000_0000_0000
00000_000_0000_0000
00000_000_0000_0000
00011_010_0000_0100
00011_010_0000_0101
00001_000_0000_0000
```

```
DATA    :xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx
DATA    :xxxx xxxx xxxx xxxx 9033 xxxx xxxx xxxx
DATA    :xxxx xxxx xxxx xxxx 9033 9033 xxxx xxxx
ADDRESS:m[0]:m[1]:m[2]:m[3]:m[4]:m[5]:m[6]:m[7]
ISim>
```

在该代码中，同样验证了这四条指令的正确性。

实验总结

实验中遇到的问题

本次实验中，遇到的最大的问题就是：有许多问题有一种无从下手的感觉！不过还好有预先给我们的实验大概框架，让我们可以更快速的入门。

实验中有几个小细节，我一直觉得在实验要求的图表中似乎不是很合理：首先是关于进位的问题，进位这个寄存器，它的作用应该是要给紧接着进行的 ADDC 或者 SUBC 或者 BC 或者 BNC 使用，那么，这个位的寄存器我觉得按理来说应该是 reg_C 的下一级，而不是与 reg_C 同级，但是这一点在实验图表中却显得有点奇怪：实验流程图中的 reg_C 与 FLAG 是同级的，这个设定确实让我晕头转向，但是最后我还是将它们归为了下一级流水线的，我觉得这样才是比较合理的设置。

对于 nf，cf，zf，它们的原理都是一样的，我觉得应该放入下一级流水线。

实验心得

本次实验让我深刻的理解了流水线处理器的工作原理。从取指令到回写的每一步，每一个细节是如何完成的。从指令集设计的合理性，到流水线抽象的流程图，再到每一个具体寄存器如何赋值如何跳转，都有了非常深刻的了解。

但是经过了解，这种指令集距离真正的实际应用的 CPU 来说还是有一些差距的，比如说，有些指令集支持非常长的指令，而且是不定长的，这又需要更加复杂的编码系统，而有些处理器的流水线级数非常的多，那么就需要更加精巧的设计以及更加复杂的冲突处理。

这样说来，我们的处理器目前为止还没有处理数据冲突的方法，只能通过软件上汇编程序员添加许多 NOP 命令来解决，这依旧是我们需要改进的地方。