中山大学本科生实验报告

(2016 学年春季学期)

课程名称：嵌入式系统案例分析与设计　　任课教师：王军　　助教：杨涵铄

年级&班级	2014 级 8 班	专业（方向）	软件工程（嵌入式软件与系统）
学号	14331062	姓名	方铭
电话	18826072452	Email	fangm7@mail2.sysu.edu.cn
开始日期	2017.4.20	完成日期	2017.5.2

实验题目：Case 1:Pipeline Processor with Hazards

一、实验目的

在上一次的实验基础上实现带处理 Hazard 功能的流水线 CPU

二、实验内容

实验步骤

1. 分析并设计处理器的数据通路和控制通路。
2. 编写设计代码并编译。
3. 软件仿真。
4. 进行硬件配置。

实验原理

Hazard

1. Structure
2. Data
   (a) Arithmetic
   - Software solution
     - insert NOP
   - Hardware solution
     - Data forwarding

   (b) LOAD
3. Control
   - Software solution
     - NOP
     - Independent operation
   - Hardware solution
     - Flushing

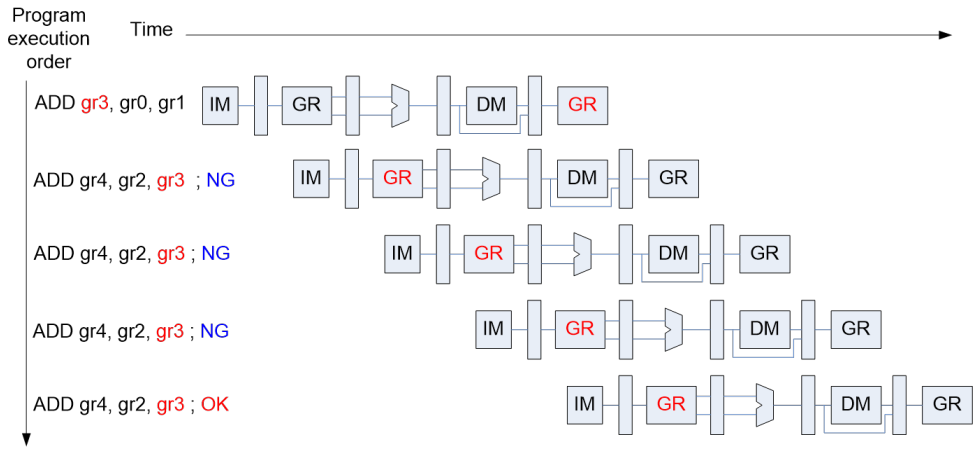

图 1: An instruction depends on completion of data access by a previous instruction

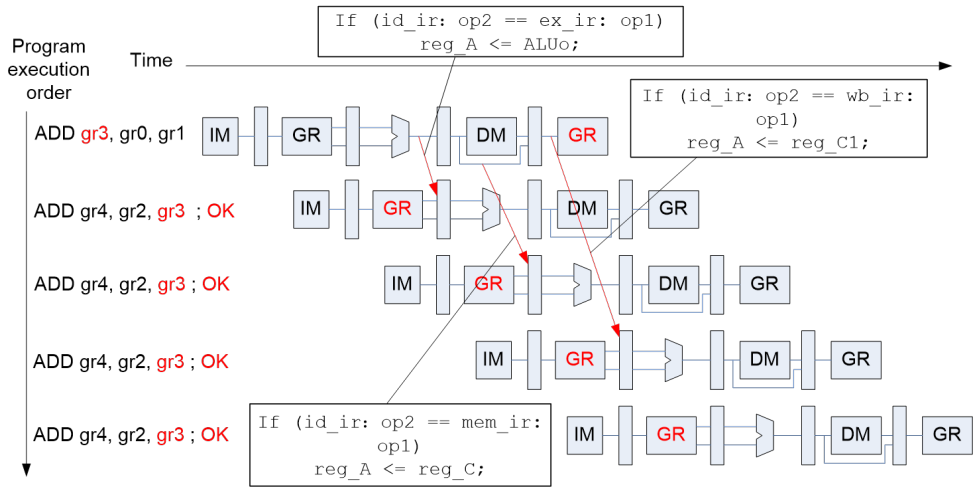

图 2: Data forwarding happens in ID stage, always check id_ir to decide reg_A/B

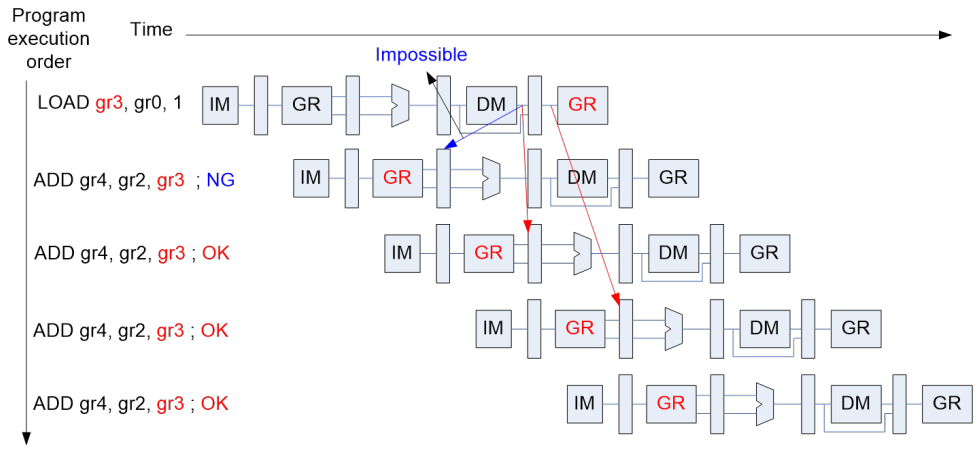

图 3: "LOAD" can still cause hazard

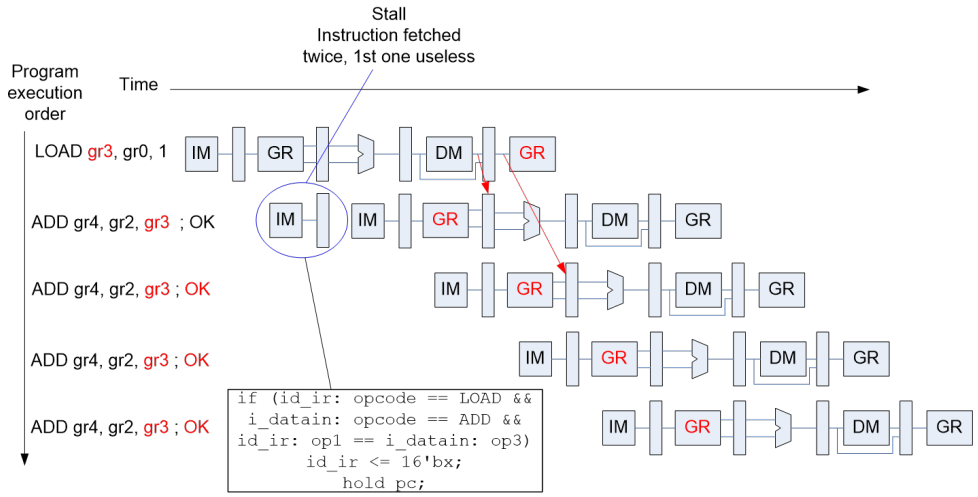

图 4: Stall the pipeline by keeping an instruction in the same stage

实验假设

1. 本实验不实现自启动的控制电路，也不提供进程切换所需引脚，不实现进程切换，读、写、内部寄存器的功能。
2. 开机后 pc 初始化为 0。
3. 指令和数据有两个独立的缓存，同时读写不会有冲突。
4. 指令地址和数据地址均已经被映射到缓存 (cache) 中，读写可在一个时钟周期内完成，从而满足流水线的要求。
5. 所有通用寄存器在开机或者复位后会清零。
6. 本实验运行的时钟频率较低，是为了在实验板 (Xlinx® Spartan6 XC6LX16-CS324) 上观察调试。可以在实验板上正常工作的最大时钟频率暂未测定。

三、实验结果

3.1 综合的 RTL 电路图

增加 hazard 处理后的 PCPU 模块的 RTL 图见附录。

3.2 texture simulation

3.2.1 简单指令测试

内存初始值：

```
00 000a
01 0004
02 0005
03 2369
04 69c3
05 0060
06 0fff
07 5555
08 6152
09 1057
0a 2895
```

Testing LOAD/STORE hazard

测试代码 load.asm

```
0 load  $1, $0, 3  #0x2369
1 srl   $1, $1, 12 #0x0002
2 load  $3, $0, 4  #0x69c3
3 store $3, $0, 0
4 load  $2, $0, 1  #0x0004
5 nop
6 sub   $2, $2, $1 #0x0002
7 load  $1, $0, 5  #0x0060
8 store $1, $2, 0
9 halt
```

```
pc:     id_ir      :reg_A:reg_B:reg_C:da:dd  :w:reC1:gr1 :gr2 :gr3
00:0000000000000000:xxxx :xxxx :xxxx :xx:xxxx:x:xxxx:0000:0000:0000
01:0001000100000011:0000 :0000 :xxxx :xx:xxxx:0:xxxx:0000:0000:0000
01:0000000000000000:0000 :0003 :0000 :00:xxxx:0:xxxx:0000:0000:0000
02:0010100100011100:0000 :0000 :0003 :03:xxxx:0:0000:0000:0000:0000
03:0001001100000100:2369 :000c :0000 :00:xxxx:0:2369:0000:0000:0000
04:0001101100000000:0000 :0004 :0002 :02:xxxx:0:0000:2369:0000:0000
05:0001001000000001:0000 :0000 :0004 :04:xxxx:0:0002:2369:0000:0000
06:0000000000000000:0000 :0001 :0000 :00:69c3:1:69c3:0002:0000:0000
07:0101001000100001:0000 :0000 :0001 :01:0004:0:0000:0002:0000:69c3
08:0001000100000101:0004 :0002 :0000 :00:0004:0:0004:0002:0000:69c3
09:0001100100100000:0000 :0005 :0002 :02:0004:0:0000:0002:0004:69c3
0a:0000100000000000:0002 :0000 :0005 :05:0004:0:0002:0002:0004:69c3
0b:0000000000000000:0000 :0000 :0002 :02:0060:1:0060:0002:0002:69c3
0c:0000000000000000:0000 :0000 :0000 :00:0005:0:0002:0060:0002:69c3
0d:0000000000000000:0000 :0000 :0000 :00:0005:0:0000:0060:0002:69c3
ISim>
```

图 5: pc is stalled when pc = 1 and data are correctly forwarded

这里测试了从内存读取数据后，马上使用该数据的情况。容易忽视 store 指令的 R1 寄存器也可能存在数据冒险情况。指令 8 测试了同时存在两个数据冒险的情况。

通过对比，可以看到添加 nop 指令（指令 5）对 pc 值的影响：pc=1 时维持了 2 个时钟，而 pc=5 时只维持了一个时钟。如果从内存 load 一个数，并立即通过 store 写回内存，那么不需要阻塞 pc，可以通过数据旁路获得数值。

Testing data hazard

测试代码 consecutive-data.asm

```
0  addi $1, 1       #0x0001
1  addi $2, 1       #0x0001
2  add  $3, $1, $2  #0x0002
3  add  $1, $2, $3  #0x0003
4  add  $2, $3, $1  #0x0005
5  add  $3, $1, $2  #0x0008
6  add  $3, $3, $1  #0x000b
7  add  $3, $1, $3  #0x000e
8  add  $3, $3, $3  #0x001c
9  add  $3, $3, $3  #0x0038
10 add  $3, $3, $3  #0x0070
11 add  $3, $3, $3  #0x00e0
12 halt
```

```
pc:     id_ir      :reg_A:reg_B:reg_C:da:dd  :w:reC1:gr1 :gr2 :gr3
00:0000000000000000:xxxx :xxxx :xxxx :xx:xxxx:x:xxxx:0000:0000:0000
01:0100100100000001:0000 :0000 :xxxx :xx:xxxx:0:xxxx:0000:0000:0000
02:0100101000000001:0000 :0001 :0000 :00:0000:0:xxxx:0000:0000:0000
03:0100001100010010:0000 :0001 :0001 :01:0000:0:0000:0000:0000:0000
04:0100000100100011:0001 :0001 :0001 :01:0000:0:0001:0000:0000:0000
05:0100001000110001:0001 :0002 :0002 :02:0000:0:0001:0001:0000:0000
06:0100001100010010:0002 :0003 :0003 :03:0000:0:0002:0001:0001:0000
07:0100001100110001:0003 :0005 :0005 :05:0000:0:0003:0001:0001:0002
08:0100001100010011:0008 :0003 :0008 :08:0000:0:0005:0003:0001:0002
09:0100001100110011:0003 :000b :000b :0b:0002:0:0008:0003:0005:0002
0a:0100001100110011:000e :000e :000e :0e:0002:0:000b:0003:0005:0008
0b:0100001100110011:001c :001c :001c :1c:0002:0:000e:0003:0005:000b
0c:0100001100110011:0038 :0038 :0038 :38:0008:0:001c:0003:0005:000e
0d:0000100000000000:0070 :0070 :0070 :70:000b:0:0038:0003:0005:001c
0e:0000000000000000:0000 :0000 :00e0 :e0:000e:0:0070:0003:0005:0038
0f:0000000000000000:0000 :0000 :0000 :00:0000:0:00e0:0003:0005:0070
10:0000000000000000:0000 :0000 :0000 :00:0000:0:0000:0003:0005:00e0
ISim>
```

图 6: all data are correctly forwarded

这里重点测试了数据旁路。

指令 2-6 测试了两个操作数同时需要旁路的情况。

指令 6-11 测试了多次更新某个通用寄存器时旁路的优先级，应该从最近被修改的寄存器那里获取数值。

Testing branch hazard

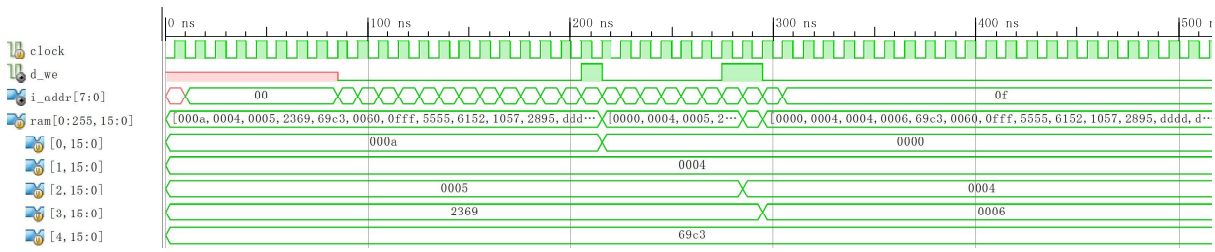

图 7: 前三个地址单元的内存读写波形图

测试代码 branch.asm

```
0  j      6
1  store  $1, $0, 0
2  add    $3, $3, $3 # 0x0006
3  add    $2, $3, $2 # 0x0004
4  j      9
5  j      1          # flush
6  addi   $3, 3      # 0x0003
7  subi   $2, 2      # 0xfffe
8  bn     $0, 1
9  store  $2, $0, 2
10 store  $3, $0, 3
11 halt
```

```
pc:     id_ir      :reg_A:reg_B:reg_C:da:dd  :w:reC1:gr1 :gr2 :gr3
00:0000000000000000:xxxx :xxxx :xxxx :xx:xxxx:x:xxxx:0000:0000:0000
01:1100000000000110:0000 :0000 :xxxx :xx:xxxx:0:xxxx:0000:0000:0000
02:0001100100000000:0000 :0006 :0000 :00:xxxx:0:xxxx:0000:0000:0000
03:0100001100110011:0000 :0000 :0006 :06:xxxx:0:0000:0000:0000:0000
06:0000000000000000:0000 :0000 :0000 :00:0000:0:0006:0000:0000:0000
07:0100101100000011:0000 :0000 :0000 :00:0000:0:0000:0000:0000:0000
08:0101101000000010:0000 :0003 :0000 :00:0000:0:0000:0000:0000:0000
09:1110000000000001:0000 :0002 :0003 :03:0000:0:0000:0000:0000:0000
0a:0001101000000010:0000 :0001 :fffe :fe:0000:0:0003:0000:0000:0000
0b:0001101100000011:0000 :0002 :0001 :01:0000:0:fffe:0000:0000:0003
01:0000000000000000:0000 :0003 :0002 :02:fffe:0:0001:0000:fffe:0003
02:0001100100000000:0000 :0000 :0003 :03:0003:0:0002:0000:fffe:0003
03:0100001100110011:0000 :0000 :0000 :00:0003:0:0003:0000:fffe:0003
04:0100001000110010:0003 :0003 :0000 :00:0000:1:0000:0000:fffe:0003
05:1100000000001001:0006 :fffe :0006 :06:0000:0:0000:0000:fffe:0003
06:1100000000000001:0000 :0009 :0004 :04:0000:0:0006:0000:fffe:0003
07:0100101100000011:0000 :0001 :0009 :09:0000:0:0004:0000:fffe:0006
09:0000000000000000:0006 :0003 :0001 :01:0000:0:0009:0000:0004:0006
0a:0001101000000010:0000 :0000 :0007 :07:0000:0:0001:0000:0004:0006
0b:0001101100000011:0000 :0002 :0000 :00:0000:0:0007:0000:0004:0006
0c:0000100000000000:0000 :0003 :0002 :02:0004:1:0000:0000:0004:0006
0d:0000000000000000:0000 :0000 :0003 :03:0006:1:0002:0000:0004:0006
0e:0000000000000000:0000 :0000 :0000 :00:0006:0:0003:0000:0004:0006
0f:0000000000000000:0000 :0000 :0000 :00:0006:0:0000:0000:0004:0006
ISim>
```

图 8: flush when branch predict is wrong

这里重点测试了跳转后是否存在对涉及通用寄存器和数据内存的非法写入，以及有没有非法的跳转（指令 5）。经测试，跳转后没有非法的写入操作，指令 5 也没有真正被执行。第二次执行指令 9 时，因为没有成功跳转，所以不需要清除流水线。内存读写波形图见图7。

Testing halt

测试代码 halt.asm

```
0  ldih   $1, 190    # $1 = 0xbe00
1  addi   $1, 239    # $1 = 0xbeef
2  halt
3  srl    $1, 1      # gr and mem
4  store  $1, $0, 0  # cannot be
5  store  $1, $0, 1  # written
6  store  $1, $0, 2  # after halt
7  store  $1, $0, 3  # instruction.
```

```
pc:     id_ir      :reg_A:reg_B:reg_C:da:dd  :w:reC1:gr1
00:0000000000000000:xxxx :xxxx :xxxx :xx:xxxx:x:xxxx:0000
01:1000000110111110:0000 :0000 :xxxx :xx:xxxx:0:xxxx:0000
02:0100100111101111:0000 :be00 :0000 :00:xxxx:0:xxxx:0000
03:0000100000000000:be00 :00ef :be00 :00:xxxx:0:0000:0000
04:0010100100010001:0000 :0000 :beef :ef:xxxx:0:be00:0000
05:0001100100000000:beef :0001 :0000 :00:xxxx:0:beef:be00
06:0001100100000001:0000 :0000 :5f77 :77:xxxx:0:0000:beef
06:0001100100000010:0000 :0001 :0000 :00:5f77:0:5f77:beef
06:0001100100000010:0000 :0002 :0001 :01:5f77:0:0000:beef
06:0001100100000010:0000 :0002 :0002 :02:beef:0:0001:beef
06:0001100100000010:0000 :0002 :0002 :02:beef:0:0002:beef
ISim>
```

图 9: flush when branch predict is wrong

对于流水线 CPU，一条指令需要分多个阶段执行，因此取指导停机指令后还不能立即停下。如果 halt 指令后没有写加上三个 nop，在装入指令寄存器之后，由于指令内存可能未初始化为 0，halt 后可能跟有 store，涉及到对数据内存的写入操作，破坏原始数据。因此要禁止在 halt 后对内存的写操作。这里重点测试了 halt 指令后，是否存在对数据内存的非法写入。由于写回操作是最后一个阶段，肯定在 halt 指令完全经过流水线之后，因此一般通用寄存器不会被修改。

3.2.2 复杂指令测试

sort

测试代码及初始内存见附录，运行结束后，数据内存的结果如下：

	0	1	2	3	4	5	6	7
0x0	DDDD	DDDD	0001	0003	0004	0005	0006	0009
0x8	000A	0011	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD
0x10	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD
0x18	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD

图 10: 数据内存的结果

inst_test

测试代码及初始内存见附录，运行结束后，数据内存的结果如下：

	0	1	2	3	4	5	6	7
0x0	FFFD	0004	0005	C369	69C3	0041	FFFF	0001
0x8	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD
0x10	B600	0001	0005	0001	FFFF	FFFE	4141	EBEB
0x18	AAAA	C369	86D2	3690	8000	C369	61B4	00C3
0x20	0001	C369	86D2	E900	8000	69C3	5386	4300
0x28	0000	C369	E1B4	FFC3	FFFF	69C3	34E1	0069
0x30	0000	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD
0x38	0030	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD
0x40	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD

图 11: 数据内存的结果

这里未定义的内存初始化为 0xdddd。运行结果除了地址为 0x5 的内存单元外，与复杂指令测试【origin】中得 dmem.mem.master 的结果完全一致。与《测试结果.doc》不一致的内存单元有 0x23,0x26,0x27,0x28，这部分涉及的指令是 sla，我实现的方式是 `ALUo = {operand[15], operand[14:0] << shift}`，估计《测试结果.doc》所采用的实现方式是 `ALUo = $signed(operand) <<< shift`，因此会有不同。实际上，`$signed(operand) <<< shift = operand << shift`。我更倾向于算术左移操作保留符号位，等价于算术上乘以 2，因此在这里保留不同结果。

gcd&lcm

测试代码及初始内存见附录，运行结束后，数据内存的结果如下：

	0	1	2	3	4	5	6	7
0x0	0000	0020	0018	0008	0060	0000	DDDD	DDDD
0x8	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD
0x10	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD
0x18	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD

图 12: 数据内存的结果

这个程序是求：数字 0x20 和数字 0x18 的最大公约数和最小公倍数。最大公约数求出是 0x0008，最小公倍数是 0x0060。

add64b

测试代码及初始内存见附录，运行结束后，数据内存的结果如下：

```
   0000 FFFE FFFE FFFE
+  0000 FFFF FFFF FFFF
   0001 FFFE FFFE FFFD
```

	0	1	2	3	4	5	6	7
0x0	FFFE	FFFE	FFFE	0000	FFFF	FFFF	FFFF	0000
0x8	FFFD	FFFE	FFFE	0001	DDDD	DDDD	DDDD	DDDD
0x10	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD
0x18	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD

图 13: 数据内存的结果

0x0-0x3 为被加数，0x4-0x7 为加数，0x8-0xb 为求和结果。三个 64 位整数都是按小端模式 (Little-Endian) 存放。

bubble

测试代码及初始内存见附录，运行结束后，数据内存的结果如下：

	0	1	2	3	4	5	6	7
0x0	000A	69C3	6152	5555	2895	2369	1057	0FFF
0x8	0060	0005	0004	DDDD	DDDD	DDDD	DDDD	DDDD
0x10	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD
0x18	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD	DDDD

图 14: 数据内存的结果

0x0 单元表示有 10 个数要排序。运行完后，数据已经有序的（降序）。

3.3 开发板的显示效果

- SW7 为 Debug，0 表示选择自动时钟，1 为选择手动时钟，按下并释放中间的 BTN(B8) 可以让手动时钟翻转一次；
- SW6 为 enable，SW5 为 start，若要使能并开始运行，需要把这两个开关打开；
- 按下 BTNU 可以重置 CPU，由于我使用 IP core 生成内存时没有加入 RESET 功能，所以内存数据不会重置。
- SW3-SW0 为 select[3:0] 用于选择在数码管上显示的数据，定义见附录 PCPU.v，
- SW4 为选择显示通用寄存器，当 SW4 打开时，结合 SW2-SW0 可以选择 0-7 号通用寄存器。

5 个复杂指令测试生成的 bit 文件以及仿真测试输出的 dmem.mem.master 文件均已提交至 ftp result 目录下，并且已经上传至我的 github：https://github.com/SimonFang1/Pipeline-RISC-CPU。烧入 add64b 对应的 bit 文件，运行程序，运行结束后，通用寄存器的结果显示如下：

图 15: 通用寄存器 gr[0:7] 的最终结果

运行结束后，pc 和 wb_ir 的结果显示如下：

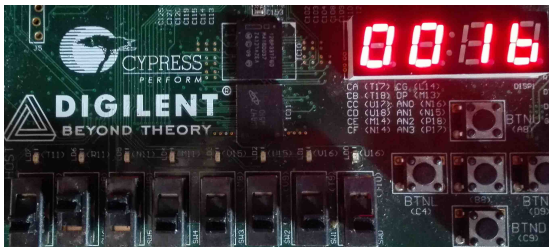

图 16: pc 的最终结果

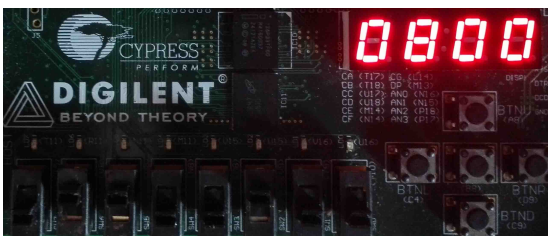

图 17: wb_ir 的最终结果

四、实验感想

本次实验实现了一个基于精简指令集的流水线 CPU，可以正确有序地执行输入指令，经仿真测试和板级综合验证，通过了 5 个复杂指令测试，进一步提高了我利用 verilog 语言设计并实现较复杂硬件系统的能力。

实验实现的 CPU 具备很弱的分支预测功能，即在发生分支跳转时，也能先把 pc+1，pc+2，pc+3 的指令先装入流水线，对 b 类型的指令具有一定的分支预测功能，而对 j 类型的指令则是 100% 错误预测率。

如果要进一步改进流水线 CPU，可以增强分支预测的功能。支持 j(jump) 指令的分支预测比较简单。j 指令可以在译码阶段获取立即数得到 pc 下个时钟周期的值。j 指令没有分支，能做到 100% 的正确预测率。jr(jmpr) 指令需要读取寄存器，也有可能有读取冒险，有不确定性，因此考虑和 b 类型的指令一起用预测跳转表的值进行预测。支持 b 类型的指令需要若干个预测跳转表，按照最近最少使用 (LRU) 的原则，为所有的 jr 指令，b 类型的指令分配预测跳转表。在取指阶段直接从预测跳转表跳中确定 pc 下个时钟周期的值，如果缺失，则使用 pc+1 作为默认值。

一般情况下，在 load 指令后需要立即使用寄存器值的话，需要延迟一个时钟周期，但是 store 指令除外，如果需要把某个内存单元的值写到另一个内存单元，那么两个连续的 load 和 store 指令执行起来不需要延迟一个时钟。但是，如果通过 load 指令更新了某个寄存器，而这个寄存器的值是 store 指令寻址的基地址，那么两个连续的 load 和 store 指令之间还是会有一个延迟。

本以为通过了 inst_test 这一复杂指令测试，其他复杂指令测试应该都能通过。但是通过 sort 测试，出现了不能停机的情况。这是由于更新 pc 时，没有充分考虑条件判断的优先级，在这个测试中，需要通过 bz 指令进行条件跳转，实际上判断跳转时，bz 指令已经进入了 mem_ir，而此时 id_ir 的指令是 load，代取指的指令和 load 之间存在冒险，需要延迟，pc 需要保持不变。如果判断时，后者（stall）的优先级比前者（pc 跳转）高，就会出现上述无法停机的情况。

所有涉及到立即数的指令，立即数的表示形式都是无符号整数，这是考虑到 addi 指令需要和 ldih 指令配合来实现一个 16 位立即数，如果实现成有符号整数，那么 addi 的时候就会在高位补 0xff，破坏 ldih 所读入的立即数。

经过实验验证并结合我的分析，我认为：使用 always @* 语句生成组合电路时，不完整的条件语句会导致 latch，但是 always @(posedge clk) 结合非阻塞赋值 <= 生成的触发器可以没有完整的条件分支。因为这些语句生成的是触发器，分支中赋值的语句的右值作为数据选择器的输入，判断条件通过编码变成一个个选择信号，如果有不完整的条件分支就需要为触发器加一个使能端，为以上条件使能；如果条件分支完整，那么这个触发器不需要使能端，或者令使能端一直处于使能状态。本实验中，我在设计触发器部分时，由于需要保持原来的数据，条件分支没有写完整，但最后也没有任何 warning 提示生成了 latch；相反在利用 always @* 设计的组合逻辑部分，如果在某个条件分支里写了类似 x = x 这样的代码，即使条件完整，也会有 warning 提示生成了 latch。

实验中使用了 Core_Gernerate 生成了 IP，使用了 coe 文件初始化内存。根据我的理解，IP 可以生成一些高级模块的代码，从而进一步完成更加复杂的设计，比如生成一些浮点运算的模块。这可以帮助设计人员减少开发周期，更加快速地设计一些硬件系统。本实验中指令内存被设计为只读的，而数据内存被设计为可读可写的。

附录

A 复杂指令测试

A.1 init_test

测试指令

```
0  addi  $7, 16        # $7 = 0x10
1  ldih  $1, 182       # $1 = 0xb600
2  store $1, $7, 0     # str to mem10
3  load  $1, $0, 0     # $1 = 0xfffd = -3
4  load  $2, $0, 1     # $2 = 4
5  addc  $3, $1, $2    # $3 = 1, cf = 1
6  store $3, $7, 1     # str to mem11
7  addc  $3, $0, $2    # $3 = 5, cf = 0
8  store $3, $7, 2     # str to mem12
9  load  $1, $0, 2     # $1 = 5
10 subc  $3, $1, $2    # $3 = 1, cf = 0
11 store $3, $7, 3     # str to mem13
12 sub   $3, $2, $1    # $3 = -1, cf = 1
13 store $3, $7, 4     # str to mem14
14 subc  $3, $2, $1    # $3 = -2, cf = 1
15 store $3, $7, 5     # str to mem15
16 load  $1, $0, 3     # $1 = 0xc369
17 load  $2, $0, 4     # $2 = 0x69c3
18 and   $3, $1, $2    # $3 = 0x4141
19 store $3, $7, 6     # str to mem16
20 or    $3, $1, $2    # $3 = 0xebeb
21 store $3, $7, 7     # str to mem17
22 xor   $3, $1, $2    # $3 = 0xaaaa
23 store $3, $7, 8     # str to mem18
24 sll   $3, $1, 0     # $3 = 0xc369
25 store $3, $7, 9     # str to mem19
26 sll   $3, $1, 1     # $3 = 0x86d2
27 store $3, $7, 10    # str to mem1a
28 sll   $3, $1, 4     # $3 = 0x3690
29 store $3, $7, 11    # str to mem1b
30 sll   $3, $1, 15    # $3 = 0x8000
31 store $3, $7, 12    # str to mem1c
32 srl   $3, $1, 0     # $3 = 0xc369
33 store $3, $7, 13    # str to mem1d
34 srl   $3, $1, 1     # $3 = 0x61b4
35 store $3, $7, 14    # str to mem1e
36 srl   $3, $1, 8     # $3 = 0x00c3
37 store $3, $7, 15    # str to mem1f
38 srl   $3, $1, 15    # $3 = 0x0001
39 addi  $7, 16        # $7 = 0x20
40 store $3, $7, 0     # str to mem20
41 sla   $3, $1, 0     # $3 = 0xc369
42 store $3, $7, 1     # str to mem21
43 sla   $3, $1, 1     # $3 = 0x86d2
44 store $3, $7, 2     # str to mem22
45 sla   $3, $1, 8     # $3 = 0xe900
46 store $3, $7, 3     # str to mem23
47 sla   $3, $1, 15    # $3 = 0x8000
48 store $3, $7, 4     # str to mem24
49 sla   $3, $2, 0     # $3 = 0x69c3
50 store $3, $7, 5     # str to mem25
51 sla   $3, $2, 1     # $3 = 0x5386
52 store $3, $7, 6     # str to mem26
53 sla   $3, $2, 8     # $3 = 0x4300
54 store $3, $7, 7     # str to mem27
55 sla   $3, $2, 15    # $3 = 0x0000
56 store $3, $7, 8     # str to mem28
57 sra   $3, $1, 0     # $3 = 0xc369
58 store $3, $7, 9     # str to mem29
59 sra   $3, $1, 1     # $3 = 0xe1b4
60 store $3, $7, 10    # str to mem2a
61 sra   $3, $1, 8     # $3 = 0xffc3
62 store $3, $7, 11    # str to mem2b
63 sra   $3, $1, 15    # $3 = 0xffff
64 store $3, $7, 12    # str to mem2c
65 sra   $3, $2, 0     # $3 = 0x69c3
66 store $3, $7, 13    # str to mem2d
67 sra   $3, $2, 1     # $3 = 0xe1b4
68 store $3, $7, 14    # str to mem2e
69 sra   $3, $2, 8     # $3 = 0xffc3
70 store $3, $7, 15    # str to mem2f
71 addi  $7, 16        # $7 = 0x30
72 sra   $3, $2, 15    # $3 = 0xffff
73 store $3, $7, 0     # str to mem30
74 load  $1, $0, 5     # $1 = 0x41
75 load  $2, $0, 6     # $2 = 0xffff
76 load  $3, $0, 7     # $3 = 0x1
77 jump  79            # j to 0x4f
78 store $7, $7, 1     # flush
79 jmpr  $1, 16        # j to 0x51
80 store $7, $7, 2     # flush
81 add   $4, $2, $3    # $4 = 0x0, cf = 1
82 bnc   $1, 40        # !j to 0x69
83 bc    $1, 20        # j to 0x55
84 store $7, $7, 3     # flush
85 add   $4, $3, $3    # $4 = 0x2, cf = 0
86 bc    $1, 40        # !j to 0x69
87 bnc   $1, 24        # j to 0x59
88 store $7, $7, 4     # flush
89 cmp   $3, $3        # zf = 1, nf = 0
90 bnz   $1, 40        # !j to 0x69
91 bz    $1, 28        # j to 0x5d
92 store $7, $7, 5     # flush
93 cmp   $4, $3        # zf = 0, nf = 0
94 bz    $1, 40        # !j to 0x69
95 bnz   $1, 32        # j to 0x61
96 store $7, $7, 6     # flush
97 cmp   $3, $4        # zf = 0, nf = 1
98 bnn   $1, 40        # !j to 0x69
99 bn    $1, 36        # j to 0x65
100 store $7, $7, 7    # flush
101 cmp  $4, $3        # zf = 0, nf = 0
102 bn   $1, 40        # !j to 0x69
103 bnn  $1, 39        # j to 0x68
104 store $7, $7, 8    # str to mem38
105 halt
```

初始内存，未指定地址的默认值为 0xdddd

```
1 00 fffd
2 01 0004
3 02 0005
4 03 c369
5 04 69c3
6 05 0041
7 06 ffff
8 07 0001
```

A.2 gcd & lcm

测试指令

```
load  $1, $0, 1  # $1 = 0x0020
load  $2, $0, 2  # $2 = 0x0018
add   $3, $0, $1 # (1)
sub   $1, $1, $2 #
bz    $0, 9      # jump to (2)
bnn   $0, 2      # jump to (1)
add   $1, $0, $2
add   $2, $0, $3
jump  2          # jump to (1)
store $2, $0, 3  # (2)
load  $1, $0, 1
load  $2, $0, 2
addi  $4, 1      # (3)
sub   $2, $2, $3
bz    $0, 16     # jump to (4)
jump  12         # jump to (3)
subi  $4, 1      # (4)
bn    $0, 20     # jump to (5)
add   $5, $5, $1
jump  16         # jump to (4)
store $5, $0, 4  # (5)
load  $1, $0, 3  # gcd
load  $2, $0, 4  # lcm
halt
```

初始内存，未指定地址的默认值为 0xdddd

```
00 0000
01 0020
02 0018
03 0000
04 0000
05 0000
```

A.3 add64b

测试指令

```
addi  $4, 4
load  $1, $0, 0  # (1)
load  $2, $0, 4
add   $3, $1, $2
bnc   $5, 6      # jump to (2)
addi  $6, 1      # carry
add   $3, $3, $7 # (2)
bnc   $5, 11     # jump to (3)
subi  $6, 0      #
bnz   $5, 11     # jump to (3)
addi  $6, 1      #
sub   $7, $7, $7 # (3)
add   $7, $7, $6 #
sub   $6, $6, $6 #
store $3, $0, 8  #
addi  $0, 1      #
cmp   $0, $4     #
bn    $5, 1      # jump to (1)
halt
```

初始内存，未指定地址的默认值为 0xdddd

```
00 fffe
01 fffe
02 fffe
03 0000
04 ffff
05 ffff
06 ffff
07 0000
```

A.4 bubble

测试指令

```
load  $3, $0, 0  # $3 = 10
subi  $3, 2      # $3 = 8
add   $1, $0, $0
add   $2, $3, $0 # (1)
load  $4, $2, 1  # (2)
load  $5, $2, 2
cmp   $5, $4
bn    $0, 10     # jump to (3)
store $4, $2, 2
store $5, $2, 1
subi  $2, 1      # (3)
cmp   $2, $1
bnn   $0, 4      # jump to (2)
addi  $1, 1
cmp   $3, $1
bnn   $0, 3      # jump to (1)
halt
```

初始内存，未指定地址的默认值为 0xdddd

```
00 000a
01 0004
02 0005
03 2369
04 69c3
05 0060
06 0fff
07 5555
08 6152
09 1057
0a 2895
```

A.5 sort

测试指令

```
addi  $1, 9
addi  $2, 9
jump  5          # jump to start
subi  $1, 1      # new round
bz    $7, 18     # jump to end
load  $3, $0, 0  # start
load  $4, $0, 1
cmp   $3, $4
bn    $7, 11     # jump to NO_op
store $3, $0, 1
store $4, $0, 0
addi  $0, 1      # NO_op
cmp   $0, $2
bn    $7, 17     # jump to continue
subi  $2, 1
sub   $0, $0, $0
jump  3          # jump to new round
jump  5          # continue, jump to start
halt
```

初始内存，未指定地址的默认值为 0xdddd

```
00 000a
01 0009
02 0006
03 0005
04 0001
05 0004
06 0003
07 0011
```

B 设计代码

其他模块见上次的实验报告，这次改动不大，可以到项目主页 https://github.com/SimonFang1/Pipeline-RISC-CPU去查看，这里只贴出 PCPU.v 文件：

```
`timescale 1ns / 1ps
`include "header.v"
module PCPU(
    input clock,
    input [15:0] d_datain,
    input enable,
    input [15:0] i_datain,
    input reset,
    input [3:0] select_y,
    input start,
    output [7:0] d_addr,
    output [15:0] d_dataout,
    output d_we,
    output [7:0] i_addr,
    output reg [15:0] y,
    input show_gr
);
    reg [7:0] pc;
    reg [15:0] id_ir, ex_ir, mem_ir, wb_ir;
    reg [15:0] reg_A, reg_B, reg_C, reg_C1;
    reg [15:0] smdr, smdr1;
    reg [2:0] flags; // flag[0]-CF, flag[1]-ZF, flag[2]-NF
    `define CF      flags[0]
    `define ZF      flags[1]
    `define NF      flags[2]
    reg dw;
    wire [15:0] w_ALUo;
    wire [2:0] w_flags;

    assign d_dataout = smdr1;
    assign d_we = dw;
    assign d_addr = reg_C[7:0];
    assign i_addr = pc;

    //************* CPU control *************//
    reg state, next_state;
    always @(posedge clock) begin
        if (reset)
            state <= `idle;
```

```
40          else
41              state <= next_state;
42      end
43
44      always @(*) begin
45          case (state)
46              `idle :
47                  if ((enable == 1'b1)
48                  && (start == 1'b1))
49                      next_state = `exec;
50                  else
51                      next_state = `idle;
52              `exec :
53                  if ((enable == 1'b0)
54                  || wb_ir[`I_OP] == `HALT)
55                      next_state = `idle;
56                  else
57                      next_state = `exec;
58          endcase
59      end
60
61      //********** General Register **********//
62      reg [15:0] gr[0:7];
63      //************ WB ************//
64      always @(posedge clock or posedge reset) begin
65          if (reset) begin
66              gr[0] <= 0; gr[1] <= 0; gr[2] <= 0; gr[3] <= 0;
67              gr[4] <= 0; gr[5] <= 0; gr[6] <= 0; gr[7] <= 0;
68          end else if (state ==`exec) begin
69              case(wb_ir[`I_OP])
70                  `LOAD, `LDIH, `MOV,
71                  `ADD, `ADDI, `ADDC,
72                  `SUB, `SUBI, `SUBC,
73                  `NOT, `AND, `OR, `XOR,
74                  `SLL, `SLA, `SRL, `SRA:
75                      gr[wb_ir[`I_R1]] <= reg_C1;
76              endcase
77          end
78      end
79
80      //************ IF ************//
81      wire w_data_miss;
82      DetectLoadDataMiss pcpu_dldm_id(
83          .peek_ir(i_datain),
84          .prev_ir(id_ir),
85          .miss(w_data_miss)
86      );
87      reg pc_jump;
88      always @(*) begin
89          case(mem_ir[`I_OP])
90              `JUMP,
91              `JMPR: pc_jump = 1'b1;
92              `BZ: pc_jump = (`ZF == 1'b1) ? 1'b1 : 1'b0;
93              `BNZ: pc_jump = (`ZF == 1'b0) ? 1'b1 : 1'b0;
94              `BN: pc_jump = (`NF == 1'b1) ? 1'b1 : 1'b0;
95              `BNN: pc_jump = (`NF == 1'b0) ? 1'b1 : 1'b0;
96              `BC: pc_jump = (`CF == 1'b1) ? 1'b1 : 1'b0;
97              `BNC: pc_jump = (`CF == 1'b0) ? 1'b1 : 1'b0;
98              default: pc_jump = 1'b0;
99          endcase
100     end
101     always @(posedge clock or posedge reset) begin
102         if (reset) begin
103             id_ir <= 16'b0000_0000_0000_0000;
104             pc <= 8'b0000_0000;
105         end else if (state ==`exec) begin
```

```
106             //imcomplete else is ok, pc and id_ir are flip-flops
107              id_ir <= (pc_jump || w_data_miss) ? {`NOP, 11'd0} : i_datain;
108              if (pc_jump) begin
109                  pc <= reg_C[7:0];
110              end else begin
111                  if (wb_ir[`I_OP] != `HALT && !w_data_miss)
112                      pc <= pc + 1'b1;
113              end
114          end
115      end
116
117      //************* ID *************//
118      // store[R1] - 0, INST[R2] - 1, INST[R3] - 2
119      wire [3:0] w_rgra [1:2];
120      wire [2:0] w_dirty [0:2];
121
122      DetectWBData pcpu_dwbd_ex_0(
123          .gra(id_ir[`I_R1]),
124          .prev_ir(ex_ir),
125          .dirty(w_dirty[0][0])
126      );
127      DetectWBData pcpu_dwbd_mem_0(
128          .gra(id_ir[`I_R1]),
129          .prev_ir(mem_ir),
130          .dirty(w_dirty[0][1])
131      );
132      DetectWBData pcpu_dwbd_wb_0(
133          .gra(id_ir[`I_R1]),
134          .prev_ir(wb_ir),
135          .dirty(w_dirty[0][2])
136      );
137
138      ParseReadGR pcpu_parseRGR(
139          .ir(id_ir),
140          .gra1(w_rgra[1]),
141          .gra2(w_rgra[2])
142      );
143      DetectWBData pcpu_dwbd_ex_1(
144          .gra(w_rgra[1][2:0]),
145          .prev_ir(ex_ir),
146          .dirty(w_dirty[1][0])
147      );
148      DetectWBData pcpu_dwbd_mem_1(
149          .gra(w_rgra[1][2:0]),
150          .prev_ir(mem_ir),
151          .dirty(w_dirty[1][1])
152      );
153      DetectWBData pcpu_dwbd_wb_1(
154          .gra(w_rgra[1][2:0]),
155          .prev_ir(wb_ir),
156          .dirty(w_dirty[1][2])
157      );
158      DetectWBData pcpu_dwbd_ex_2(
159          .gra(w_rgra[2][2:0]),
160          .prev_ir(ex_ir),
161          .dirty(w_dirty[2][0])
162      );
163      DetectWBData pcpu_dwbd_mem_2(
164          .gra(w_rgra[2][2:0]),
165          .prev_ir(mem_ir),
166          .dirty(w_dirty[2][1])
167      );
168      DetectWBData pcpu_dwbd_wb_2(
169          .gra(w_rgra[2][2:0]),
170          .prev_ir(wb_ir),
171          .dirty(w_dirty[2][2])
```

```
172     );
173
174     always @(posedge clock or posedge reset) begin
175         if (reset) begin
176             ex_ir <= 16'd0;
177         end else if (state ==`exec) begin
178             ex_ir <= pc_jump ? {`NOP, 11'd0} : id_ir;
179             if (id_ir[`I_OP] == `STORE) begin
180                if (w_dirty[0][0])
181                     smdr <= w_ALUo;
182                 else if (w_dirty[0][1])
183                     smdr <= reg_C;
184                 else if (w_dirty[0][2])
185                     smdr <= reg_C1;
186                 else
187                     smdr <= gr[id_ir[`I_R1]];
188             end
189             // incomplete else, default is okay,
190             // because smdr, reg_A and reg_B are flip-flops
191             if (w_rgra[1][3]) begin
192                 case(w_rgra[1][2:0])
193                     `P_NULL: reg_A <= 16'd0;
194                     `P_VAL3: reg_A <= id_ir[`I_VAL3];
195                     `P_VAL2: reg_A <= id_ir[`I_VAL2];
196                     `P_IMDT: reg_A <= id_ir[`I_IMDT];
197                     `P_HIMDT: reg_A <= {id_ir[`I_IMDT], 8'd0};
198                 endcase
199             end else begin
200                 if (w_dirty[1][0])
201                     reg_A <= w_ALUo;
202                 else if (w_dirty[1][1])
203                     reg_A <= mem_ir[`I_OP] == `LOAD ? d_datain : reg_C;
204                 else if (w_dirty[1][2])
205                     reg_A <= reg_C1;
206                 else
207                     reg_A <= gr[w_rgra[1][2:0]];
208             end
209             if (w_rgra[2][3]) begin
210                 case(w_rgra[2][2:0])
211                     `P_NULL: reg_B <= 16'd0;
212                     `P_VAL3: reg_B <= id_ir[`I_VAL3];
213                     `P_VAL2: reg_B <= id_ir[`I_VAL2];
214                     `P_IMDT: reg_B <= id_ir[`I_IMDT];
215                     `P_HIMDT: reg_B <= {id_ir[`I_IMDT], 8'd0};
216                 endcase
217             end else begin
218                 if (w_dirty[2][0])
219                     reg_B <= w_ALUo;
220                 else if (w_dirty[2][1])
221                     reg_B <= mem_ir[`I_OP] == `LOAD ? d_datain : reg_C;
222                 else if (w_dirty[2][2])
223                     reg_B <= reg_C1;
224                 else
225                     reg_B <= gr[w_rgra[2][2:0]];
226             end
227         end
228     end
229
230     //************* EX *************//
231     wire strAfterLd =
232         ex_ir[`I_OP] == `STORE &&
233         mem_ir[`I_OP] == `LOAD &&
234         ex_ir[`I_R1] == mem_ir[`I_R1];
235     wire updateZFNF =
236         !(ex_ir[`I_OP] == `BN || ex_ir[`I_OP] == `BNN ||
237             ex_ir[`I_OP] == `BZ || ex_ir[`I_OP] == `BNZ ||
```

```
238             ex_ir['I_OP] == 'BC || ex_ir['I_OP] == 'BNC);
239     wire updateCF =
240         ex_ir['I_OP] == 'ADD  || ex_ir['I_OP] == 'SUB ||
241         ex_ir['I_OP] == 'ADDI || ex_ir['I_OP] == 'SUBI ||
242         ex_ir['I_OP] == 'ADDC || ex_ir['I_OP] == 'SUBC;
243     wire dwCondition =
244         mem_ir['I_OP] != 'HALT &&
245         wb_ir['I_OP] != 'HALT &&
246         !pc_jump && ex_ir['I_OP] == 'STORE;
247
248     always @(posedge clock or posedge reset) begin
249         if (reset) begin
250             mem_ir <= 16'd0;
251         end else if (state =='exec) begin
252             mem_ir <= pc_jump ? {'NOP, 11'd0} : ex_ir;
253             reg_C <= w_ALUo;
254             if (updateZFNF) begin
255                 flags[2:1] <= w_flags[2:1];
256                 if (updateCF)
257                     flags[0] <= w_flags[0];
258             end
259             smdr1 <= strAfterLd ? d_datain : smdr;
260             if (dwCondition)
261                 dw <= 1'b1;
262             else
263                 dw <= 1'b0;
264         end
265     end
266
267     //************* MEM *************//
268     always @(posedge clock or posedge reset) begin
269         if (reset) begin
270             wb_ir <= 16'd0;
271         end else if (state =='exec) begin
272             if (wb_ir['I_OP] != 'HALT)
273                 wb_ir <= pc_jump ? {'NOP, 11'd0} : mem_ir;
274             if (mem_ir['I_OP] == 'LOAD)
275                 reg_C1 <= d_datain;
276             else
277                 reg_C1 <= reg_C;
278         end
279     end
280
281     //ALU
282     reg [3:0] ALUop;
283     always @(ex_ir or flags) begin
284         case(ex_ir['I_OP])
285             'LOAD,'STORE,'LDIH,'BZ,'BNZ,
286             'BN,'BNN,'BC,'BNC,'ADD,
287             'ADDI: ALUop = 'A_ADD;
288             'ADDC: ALUop = 'CF == 1'b1 ? 'A_ADDPLS : 'A_ADD;
289             'SUB, 'SUBI,
290             'CMP: ALUop = 'A_SUB;
291             'SUBC: ALUop = 'CF == 1'b1 ? 'A_SUBMNS : 'A_SUB;
292             'AND: ALUop = 'A_AND;
293             'XOR: ALUop = 'A_XOR;
294             'NOT: ALUop = 'A_NOT;
295             'SLL: ALUop = 'A_SLL;
296             'SLA: ALUop = 'A_SLA;
297             'SRL: ALUop = 'A_SRL;
298             'SRA: ALUop = 'A_SRA;
299 //          'OR,'JUMP,'MOV: ALUop = 'A_OR;
300              default: ALUop = 'A_OR;
301         endcase
302     end
303
```

```
304     ALU alu(
305         .opcode(ALUop),
306         .operandA(reg_A),
307         .operandB(reg_B),
308         .ALUo(w_ALUo),
309         .flags(w_flags)
310     );
311 
312     //*********** Output ***********//
313     always @(*) begin
314         if (show_gr == 1'b1) begin
315             case (select_y[2:0])
316                 3'b000: y = gr[0];
317                 3'b001: y = gr[1];
318                 3'b010: y = gr[2];
319                 3'b011: y = gr[3];
320                 3'b100: y = gr[4];
321                 3'b101: y = gr[5];
322                 3'b110: y = gr[6];
323                 3'b111: y = gr[7];
324             endcase
325         end else begin
326             case (select_y)
327                 4'd0: y = pc;     //{8'b0000_0000, pc};
328                 4'd1: y = id_ir;
329                 4'd2: y = ex_ir;
330                 4'd3: y = mem_ir;
331                 4'd4: y = wb_ir;
332                 4'd5: y = reg_A;
333                 4'd6: y = reg_B;
334                 4'd7: y = reg_C;
335                 4'd8: y = flags;
336                 4'd9: y = dw;
337                 4'd10: y = smdr;
338                 4'd11: y = smdr1;
339                 4'd12: y = reg_C1;
340                 default: y = pc;
341             endcase
342         end
343     end
344 endmodule
345 
346 module ParseReadGR(
347     input [15:0] ir,
348     output reg [3:0] gra1,
349     output reg [3:0] gra2
350 );
351     // gra[3] - read neg enable      gra[2:0] - read address
352     always @(*) begin
353         case(ir[`I_OP])
354             `NOP,
355             `HALT: {gra1, gra2} = {1'b1, 3'd0, 1'b1, 3'd0};
356             `LDIH: {gra1, gra2} = {1'b0, ir[`I_R1], 1'b1, `P_HIMDT};
357             `JUMP,`MOV: {gra1, gra2} = {1'b1, `P_NULL, 1'b1, `P_IMDT};
358             `NOT: {gra1, gra2} = {1'b0, ir[`I_R2], 1'b1, `P_NULL};
359             `ADD,`ADDC,`SUB,`SUBC,`CMP,`AND,`OR,
360             `XOR: {gra1, gra2} = {1'b0, ir[`I_R2], 1'b0, ir[`I_R3]};
361             `ADDI,`SUBI,`JMPR,`BZ,`BNZ,`BN,`BNN,`BC,
362             `BNC: {gra1, gra2} = {1'b0, ir[`I_R1], 1'b1, `P_IMDT};
363             `SLL, `SLA, `SRL,`SRA,`LOAD,
364             `STORE: {gra1, gra2} = {1'b0, ir[`I_R2], 1'b1, `P_VAL3};
365             default: {gra1, gra2} = {1'b1, `P_NULL, 1'b1, `P_NULL};
366         endcase
367     end
368 endmodule
369 
```

```
370 // The following modules will cause some warnings:
371 // WARNING:Xst:647 - Input <xxx> is never used.
372 // e.g. ir[15:0] is connected but ir[7:0] is useless.
373 // Warinings can be eliminated by rewriting the modules
374 // and the caller of the modules, but it will make the code hard to read. STUPID
    ISE!
375 module ParseWriteGR(
376     input [15:0] ir,
377     output reg [3:0] gra
378 );
379     always @(*) begin
380         case(ir['I_OP])
381             'LOAD,'LDIH,'ADD,'ADDI,'ADDC,'SUB,'SUBI,'SUBC,
382             'AND,'XOR,'OR,'NOT,'SLL,'SLA,'SRL,'SRA:
383                 gra = {1'b0, ir['I_R1]};
384             default: gra = 4'b1000;
385         endcase
386     end
387 endmodule
388
389 module DetectLoadDataMiss(
390     input [15:0] peek_ir,
391     input [15:0] prev_ir,
392     output miss
393 );
394     // In this module, the case that
395     // prev_ir['I_OP] == 'LOAD && peek_ir['I_OP]== 'STORE &&
396     // prev_ir['I_R1] == peek_ir['I_R1]
397     // is ignored, since data can be forwarded
398     // from d_datain, nop needn't be inserted.
399     // See the code in EX part where smdr1 get the right data.
400     wire nre1, nre2;
401     wire [2:0] rgr1, rgr2;
402     ParseReadGR l_parseRGR(
403         .ir(peek_ir),
404         .gra1({nre1, rgr1}),
405         .gra2({nre2, rgr2})
406     );
407     assign miss =
408     (prev_ir['I_OP] == 'LOAD &&
409         ((!nre1 && prev_ir['I_R1] == rgr1) ||
410          (!nre2 && prev_ir['I_R1] == rgr2) /*||
411             (peek_ir['I_OP]== 'STORE &&
412              prev_ir['I_R1] == peek_ir['I_R1])*/ ) ) ?
413     1'b1 : 1'b0;
414 endmodule
415
416 module DetectWBData(
417     input [2:0] gra,
418     input [15:0] prev_ir,
419     output dirty
420 );
421     wire nwe;
422     wire [2:0] wgra;
423     assign dirty = !nwe && gra == wgra;
424     ParseWriteGR wb_parseWGR(
425         .ir(prev_ir),
426         .gra({nwe, wgra})
427     );
428 endmodule
```

C RTL 图

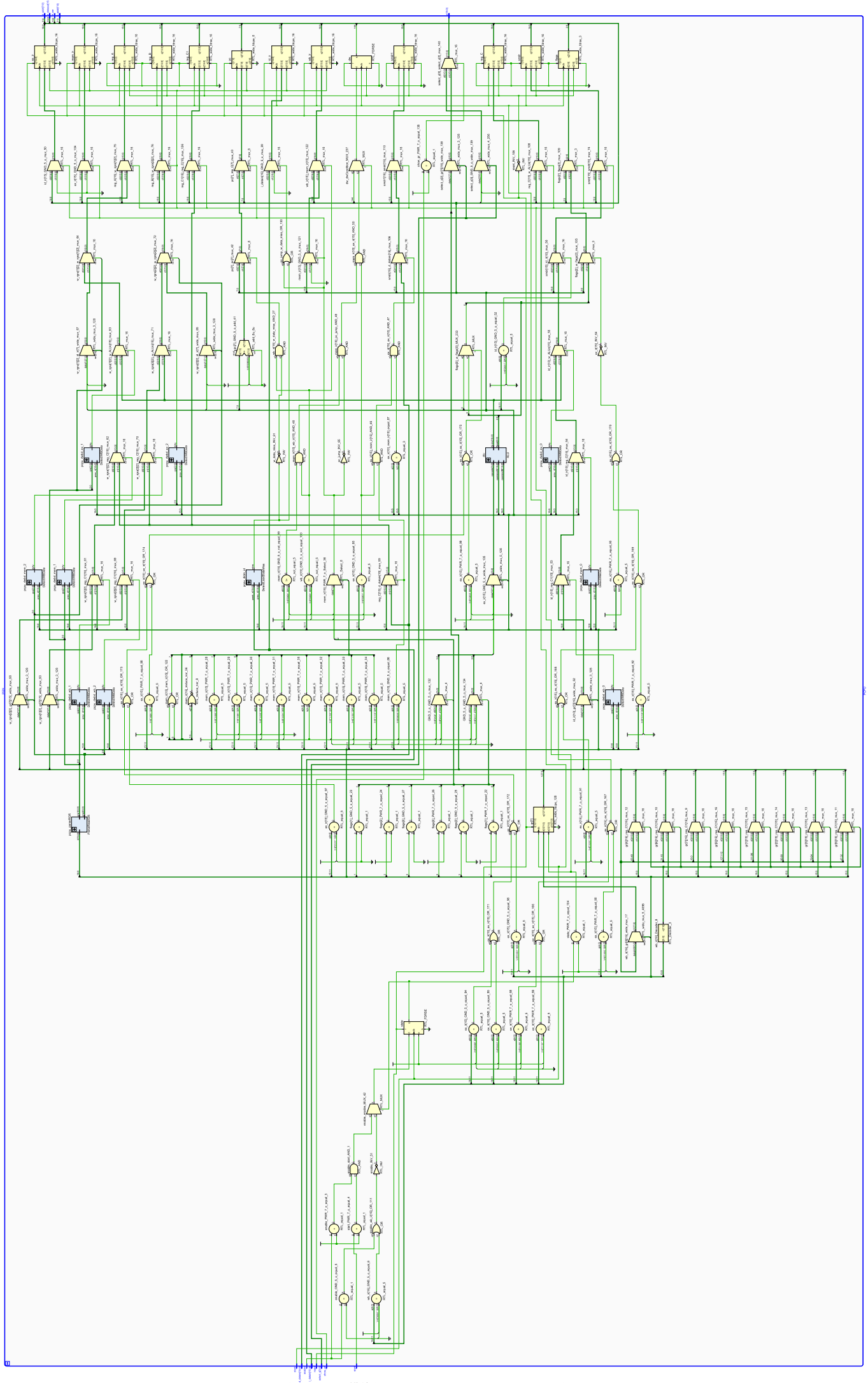

图 18: PCPU 模块的 RTL 图（Xilinx PlanAhead 14.7 导出）

top Project Status (05/02/2017 - 15:54:25)			
Project File:	PCPU.xise	**Parser Errors:**	No Errors
Module Name:	top	**Implementation State:**	Programming File Generated
Target Device:	xc6slx16-3csg324	• **Errors:**	
Product Version:	ISE 14.7	• **Warnings:**	
Design Goal:	Balanced	• **Routing Results:**	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	• **Timing Constraints:**	All Constraints Met
Environment:	System Settings	• **Final Timing Score:**	0 (Timing Report)

Device Utilization Summary				[-]
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	379	18,224	2%	
Number used as Flip Flops	378			
Number used as Latches	0			
Number used as Latch-thrus	0			
Number used as AND/OR logics	1			
Number of Slice LUTs	938	9,112	10%	
Number used as logic	937	9,112	10%	
Number using O6 output only	770			
Number using O5 output only	16			
Number using O5 and O6	151			
Number used as ROM	0			
Number used as Memory	0	2,176	0%	
Number used exclusively as route-thrus	1			
Number with same-slice register load	0			
Number with same-slice carry load	1			
Number with other load	0			
Number of occupied Slices	321	2,278	14%	
Number of MUXCYs used	168	4,556	3%	
Number of LUT Flip Flop pairs used	947			
Number with an unused Flip Flop	601	947	63%	
Number with an unused LUT	9	947	1%	
Number of fully used LUT-FF pairs	337	947	35%	
Number of unique control sets	13			
Number of slice register sites lost to control set restrictions	46	18,224	1%	
Number of bonded IOBs	23	232	9%	

Number of LOCed IOBs	23	23	100%	
Number of RAMB16BWERs	0	32	0%	
Number of RAMB8BWERs	2	64	3%	
Number of BUFIO2/BUFIO2_2CLKs	0	32	0%	
Number of BUFIO2FB/BUFIO2FB_2CLKs	0	32	0%	
Number of BUFG/BUFGMUXs	2	16	12%	
Number used as BUFGs	2			
Number used as BUFGMUX	0			
Number of DCM/DCM_CLKGENs	0	4	0%	
Number of ILOGIC2/ISERDES2s	0	248	0%	
Number of IODELAY2/IODRP2/IODRP2_MCBs	0	248	0%	
Number of OLOGIC2/OSERDES2s	0	248	0%	
Number of BSCANs	0	4	0%	
Number of BUFHs	0	128	0%	
Number of BUFPLLs	0	8	0%	
Number of BUFPLL_MCBs	0	4	0%	
Number of DSP48A1s	0	32	0%	
Number of ICAPs	0	1	0%	
Number of MCBs	0	2	0%	
Number of PCILOGICSEs	0	2	0%	
Number of PLL_ADVs	0	2	0%	
Number of PMVs	0	1	0%	
Number of STARTUPs	0	1	0%	
Number of SUSPEND_SYNCs	0	1	0%	
Average Fanout of Non-Clock Nets	5.29			