HARBIN INSTITUTE OF TECHNOLOGY

hello 的一生

姓　　名：x　x　x

学　　号：xxxxxxxxxxx

学　　院：计算机科学与技术学院

2018 年 1 月 28 日

摘　要

Y86-64 指令集是 CSAPP 一书中给出的一个类似于 x86-64 的指令集，相比于 x86-64 指令集，它更加简单，数据类型、指令、寻址方式更少，实现起来也更为容易。SEQ 是计算机执行指令的一种模式，即每个时钟周期上，顺序的执行处理一条完整指令所需的步骤。本此实验从 Y86-64 指令集出发用 hcl 和 verilog 实现一个 SEQ 的 64 位小端模式的 CPU，并完成了波形仿真。

摘要：　Y86-64，　SEQ，　veriog，　处理器。

Abstract

Y86-64 instruction set, which is given in the CSAPP book, is similar to the x86-64 instruction set. Compared to the x86-64 instruction set, it is more simple, with less data types, instructions and addressing methods, so that easier to achieve. SEQ is a pattern for computer to execute instructions. That is to say, sequentially execute the steps required to process a complete instruction on each clock cycle. The experiment achieves a SEQ CPU with hcl and verilog, based on the Y86-64 instruction set and completed the waveform simulation.

keywords： Y86-64, SEQ, veriog, CPU 。

目录

1 Hello 程序的编译链接过程

1.1 基本变量

CPU 中共有 15 个 64 位的程序寄存器, 指示指令的 PC 的长度也为 64 位。CPU 还有三个标志位, 分别是零标志 zf、符号标志 sf、溢出标志 of, 这三个标志位统称为 CC。

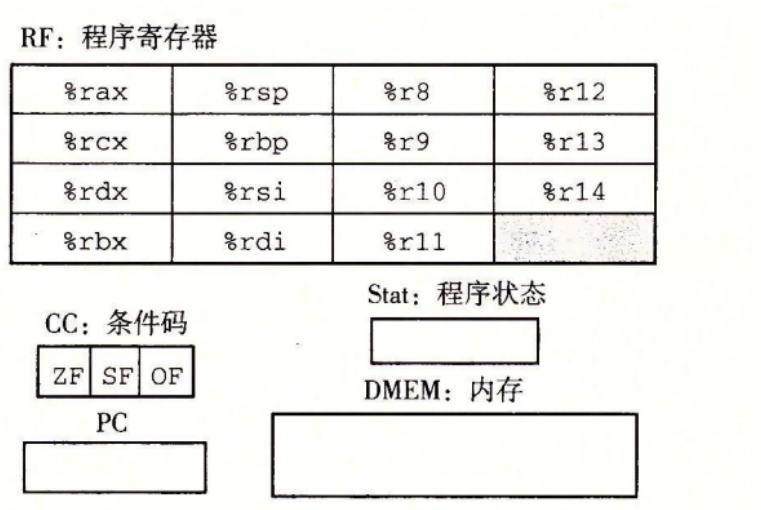

图 1.1: Y86-64 基本变量

1.2 ISA+ 的指令编码设计

Y86-64 指令集基本上是 x86-64 指令集的一个自己。它只包括 8 字节整数操作，寻址方式较少，操作也较少。图 1.2 中，左边是指令的汇编码，右边是字节编码。可以看出，Y86-64 指令每条需要 1 10 字节不等，这取决于需要哪些字段。每条指令的第一个字节表明指令的类型。这个字节分为两个部分，每部分 4 位：高 4 位是代码 (code) 部分，低 4 位是功能 (function) 部分。

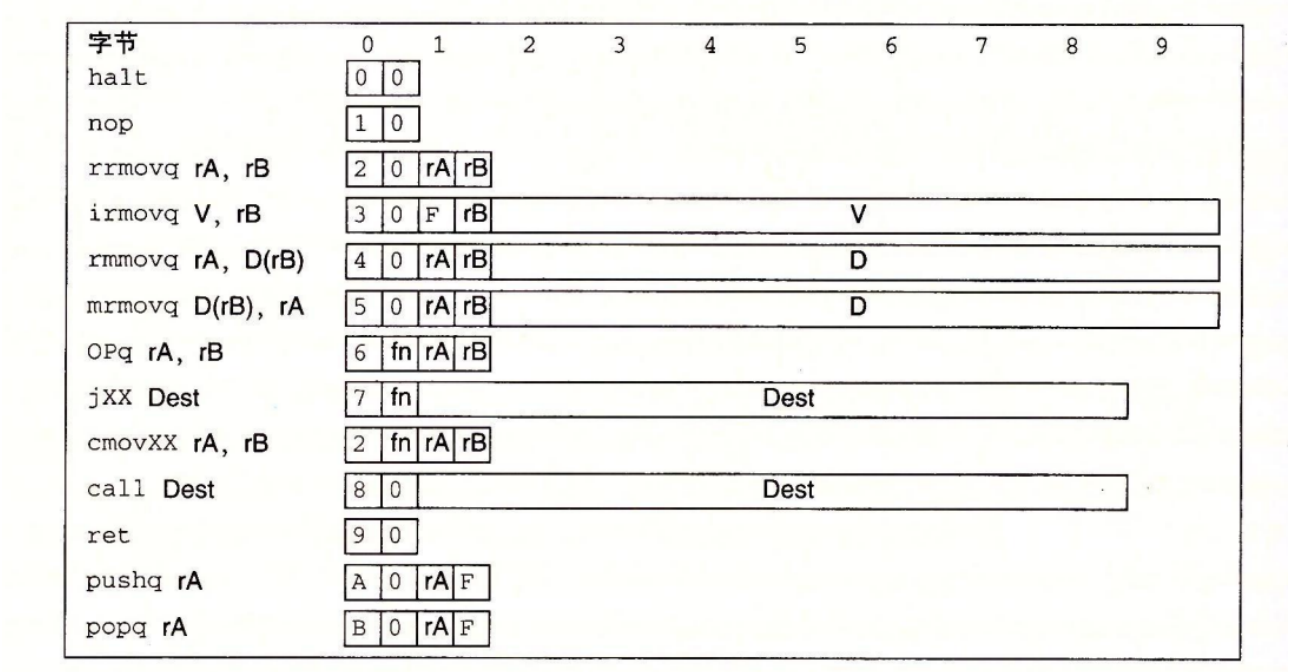

图 1.2: Y86-64 指令集

我们要实现的 CPU 还增加了一条指令 IADDQ，功能如下图 1.3：

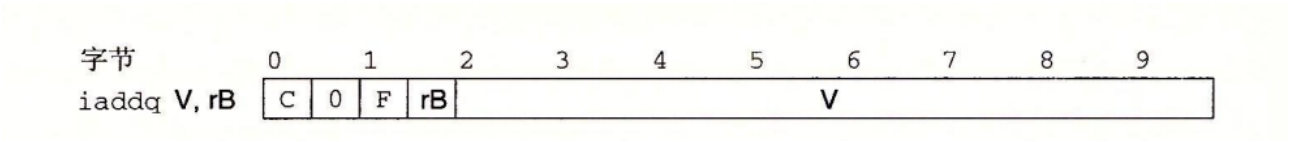

图 1.3: IADDQ 指令

1.3 指令编码

图 1.4 给出了整数操作、分支和条件传送指令的具体编码。

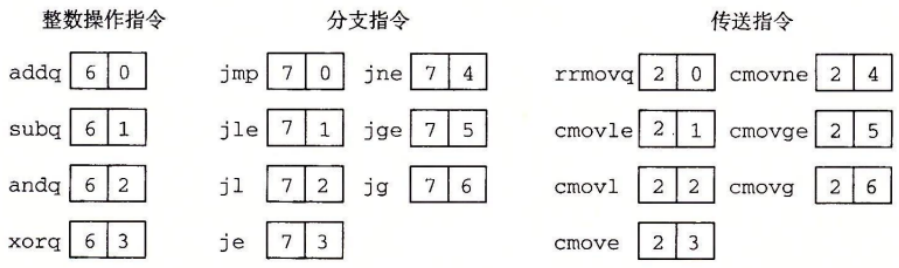

图 1.4: Y86-64 指令集的功能码

很多指令都会有一个字节用来表示所用到的寄存器。每个寄存器都用一个数字来表示, 使用 RNONE(0xF) 代表没有用到寄存器。有些指令会有四个字节的立即数, 立即数在指令中以小端模式存储。

1.4 Y86-64 异常

Y86-64 指令还包含一些状态码，它描述程序执行的总体状态。状态码可能的值如下图所示:

值	名字	含义
1	AOK	正常操作
2	HLT	遇到器执行 halt 指令
3	ADR	遇到非法地址
4	INS	遇到非法指令

图 1.5: Y86-64 状态码

1.5 Y86-64 程序

直接写 Y86-64 的机器代码, 不仅很麻烦, 而且容易出错。对日常使用的那些汇编指令集, 可以先写汇编代码, 然后由汇编器转成机器代码。对于 X86-64 汇编语言来说, 有 nasm、masm 等汇编器, 而对于这里用到的 Y86-64 汇编语言, 也有相应的汇编器 yas。通过 yas, 可以将 Y86-64 汇编转成机器代码。

下面是一个 Y86-64 转成机器码的实例，用 iaddq 指令重写的 sum 函数:

```
1                                  | # Execution begins at address 0
2     0x000:                       |     .pos 0
3     0x000: 30f40001000000000000 |     irmovq stack, %rsp      # Set up stack ...
        pointer
4     0x00a: 803800000000000000   |     call main       # Execute main program
5     0x013: 00                    |     halt            # Terminate program
```

```
6       |
7       | # Array of 4 elements
8       0x018:                      |     .align 8
9       0x018: 0d000d000d000000     | array:    .quad 0x000d000d000d
10      0x020: c000c000c0000000     |     .quad 0x00c000c000c0
11      0x028: 000b000b000b0000     |     .quad 0x0b000b000b00
12      0x030: 00a000a000a00000     |     .quad 0xa000a000a000
13      |
14      0x038: 30f71800000000000000 | main: irmovq array,%rdi
15      0x042: 30f60400000000000000 |     irmovq $4,%rsi
16      0x04c: 805600000000000000   |     call sum        # sum(array, 4)
17      0x055: 90                   |     ret
18      |
19      | /* $begin sumi-ys */
20      | # long sum(long *start, long count)
21      | # start in %rdi, count in %rsi
22      0x056:                      | sum:
23      0x056: 6300                 |     xorq %rax,%rax      # sum = 0
24      0x058: 6266                 |     andq %rsi,%rsi      # Set condition codes
25      0x05a: 708300000000000000   |     jmp    test
26      0x063:                      | loop:
27      0x063: 50a700000000000000   |     mrmovq (%rdi),%r10  # Get *start
28      0x06d: 60a0                 |     addq %r10,%rax          # Add to sum
29      0x06f: c0f708000000000000   |     iaddq $8,%rdi           # start++
30      0x079: c0f6ffffffffffffff   |     iaddq $-1,%rsi          # count--
31      0x083:                      | test:
32      0x083: 746300000000000000   |     jne    loop             # Stop when 0
33      0x08c: 90                   |     ret
34      | /* $end sumi-ys */
35      |
36      | # The stack starts here and grows to lower addresses
37      0x100:                      |     .pos 0x100
38      0x100:                      | stack:
```

2 Y86-64 的顺序实现

2.1 SEQ

SEQ 处理器将指令的执行过程分为了六个过程，分别是取指 (fetch)、译码 (decode)、执行 (execute)、访存 (memory)、写会 (wirte back)、更新 PC(PC update)。同时 CPU 只有一个算数/逻辑单元，根据所执行的指令类型的不同，而进行不同的运算。

2.2 指令在 SEQ 中的计算过程

有了上面 SEQ 的过程，我们就可以把指令划分为一条条的微指令，所有的微指令在一起构成了一次指令的执行。各个指令的微指令如下:

阶段	OPq rA, rB	rrmovq rA, rB	irmovqV, rB
取指	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC+10
译码	valA ← R[rA] valB ← R[rB]	valA ← R[rA]	
执行	valE ← valB OP valA Set CC	valE ← 0+valA	valE ← 0+valC
访存			
写回	R[rB]← valE	R[rB]← valE	R[rB]← valE
更新 PC	PC ← valP	PC ← valP	PC ← valP

图 2.1: 微指令 1

阶段	jXX Dest	call Dest	ret
取指	icode:ifun ← M_1[PC] valC ← M_8[PC+1] valP ← PC+9	icode:ifun ← M_1[PC] valC ← M_8[PC+1] valP ← PC+9	icode:ifun ← M_1[PC] valP ← PC+1
译码		valB ← R[%rsp]	valA ← R[%rsp] valB ← R[%rsp]
执行	Cnd ← Cond(CC, ifun)	valE ← valB+(−8)	valE ← valB+8
访存		M_8[valE] ← valP	valM ← M_8[valA]
写回		R[%rsp] ← valE	R[%rsp] ← valE
更新 PC	PC ← Cnd?valC:valP	PC ← valC	PC ← valM

图 2.2: 微指令 2

阶段	pushq rA	popq rA
取指	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2
译码	valA ← R[rA] valB ← R[%rsp]	valA ← R[%rsp] valB ← R[%rsp]
执行	valE ← valB+(−8)	valE ← valB+8
访存	M_8[valE] ← valA	valE ← M_8[valA]
写回	R[%rsp] ← valE	R[%rsp] ← valE R[rA] ← valM
更新 PC	PC ← valP	PC ← valP

图 2.3: 微指令 3

阶段	iaddq v rB
取指	icode:ifun ← M_1[PC] rA:rB ← M_1[PC] valC ← M_8[PC] valP ← PC + 10
译码	valB ← R[rB]
执行	valE ← valB + valC Set CC
访存	
写回	R[rB] ← valE
更新 PC	PC ← valP

图 2.4: iaddq 微指令

阶段	rmmovq rA, D(rB)	mrmovq D(rB), rA
取指	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC+10	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC+10
译码	valA ← R[rA] valB ← R[rB]	valB ← R[rB]
执行	valE ← valB+valC	valE ← valB+valC
访存	M_8[valE] ← valA	valE ← M_8[valE]
写回		
		R[rA] ← valM
更新 PC	PC ← valP	PC ← valP

图 2.5: 微指令 4

阶段	cmovXX rA, rB
取指	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2
译码	valA ← R[rA]
执行	valE ← 0+valA Cnd ← Cond(CC, ifun)
访存	
写回	if(Cnd) R[rB] ← valE
更新 PC	PC ← valP

图 2.6: 微指令 5

2.3 SEQ 硬件结构

参照 CSAPP 书中给出的 SEQ 的硬件结构，如下图，我们可以得出各个阶段中我们具体需要做的事情。

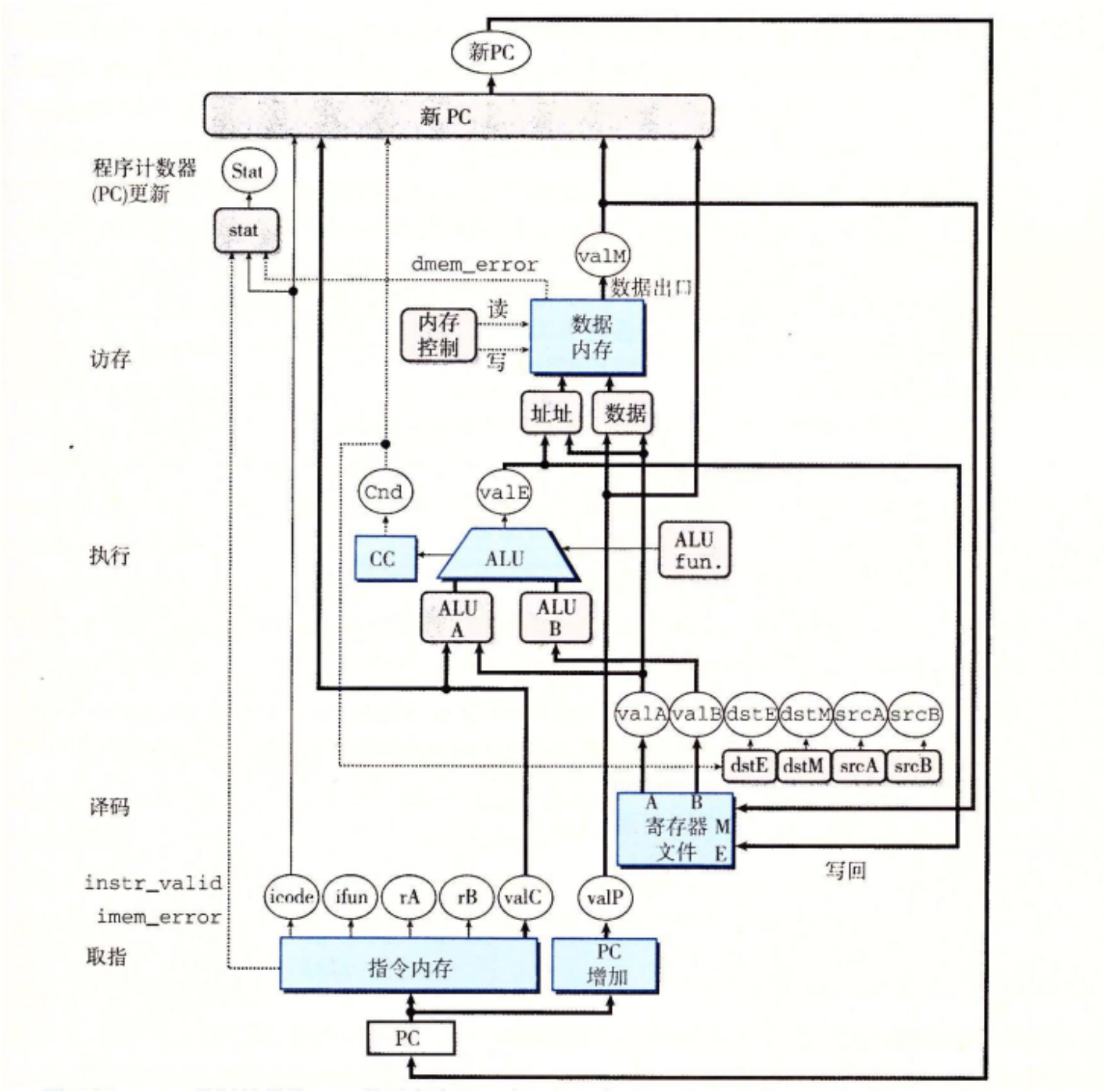

图 2.7: SEQ 硬件结构，一些控制信号和寄存器以及控制线路未画出

2.4 SEQ 的时序

SEQ 中大多数都是组合逻辑，不需要进行相应的时序控制，需要时序控制的硬件单元一共有 4 个，分别是程序计数器、条件码寄存器、数据内存和寄存器文件。可以采用边沿控制，只有在一个新周期开始时，对将上个周期内产生的值写到寄存器或内存中并更新 pc 和条件码。

3 SEQ 结构设计和 HCL 实现

3.1 取指阶段

取指阶段的硬件设计如下：

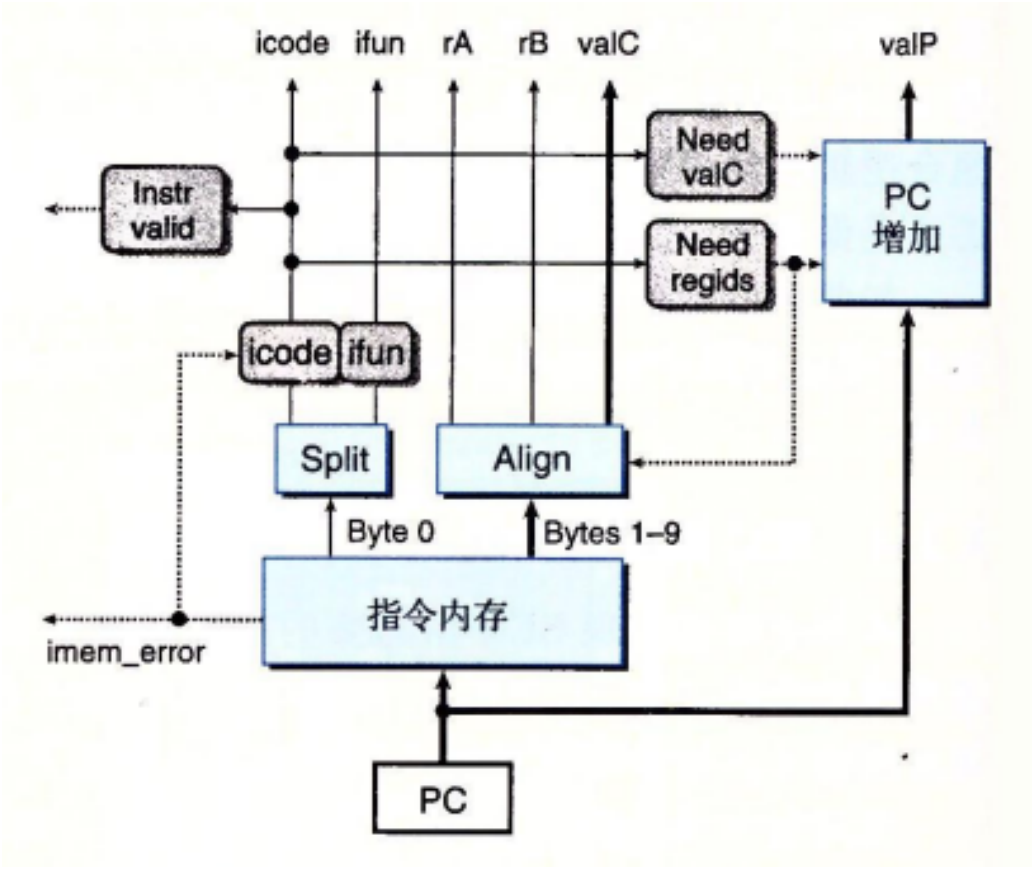

图 3.1: 取指阶段

这个单元一次从内存读出 10 个字节。第一个字节被解释成指令字节，也就是 Split 单元，同时将该字节分为两个四字节的部分，放入 icode 和 ifun 寄存器，同时判断指令是否正确。同时根据 icode 判断是否取出 rA，rB 和 valC，以及 pc 的增加量。其中 instr_valid 对应于该条指令是否合法，need_regids 对应于是否需要寄存器，need_valC 对应于是否取出 valC。具体实现代码如下：

```
    # Determine instruction code
    word icode = [
    imem_error: INOP;
    1: imem_icode;      # Default: get from instruction memory
    ];

    # Determine instruction function
    word ifun = [
    imem_error: FNONE;
    1: imem_ifun;       # Default: get from instruction memory
    ];

    bool instr_valid = icode in
    { INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
    IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ, IIADDQ };

    # Does fetched instruction require a regid byte?
    bool need_regids =
    icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
    IIRMOVQ, IRMMOVQ, IMRMOVQ, IIADDQ };

    # Does fetched instruction require a constant word?
    bool need_valC =
    icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ, IJXX, ICALL, IIADDQ };
```

3.2 译码和写回阶段

译码阶段的硬件设计如下:

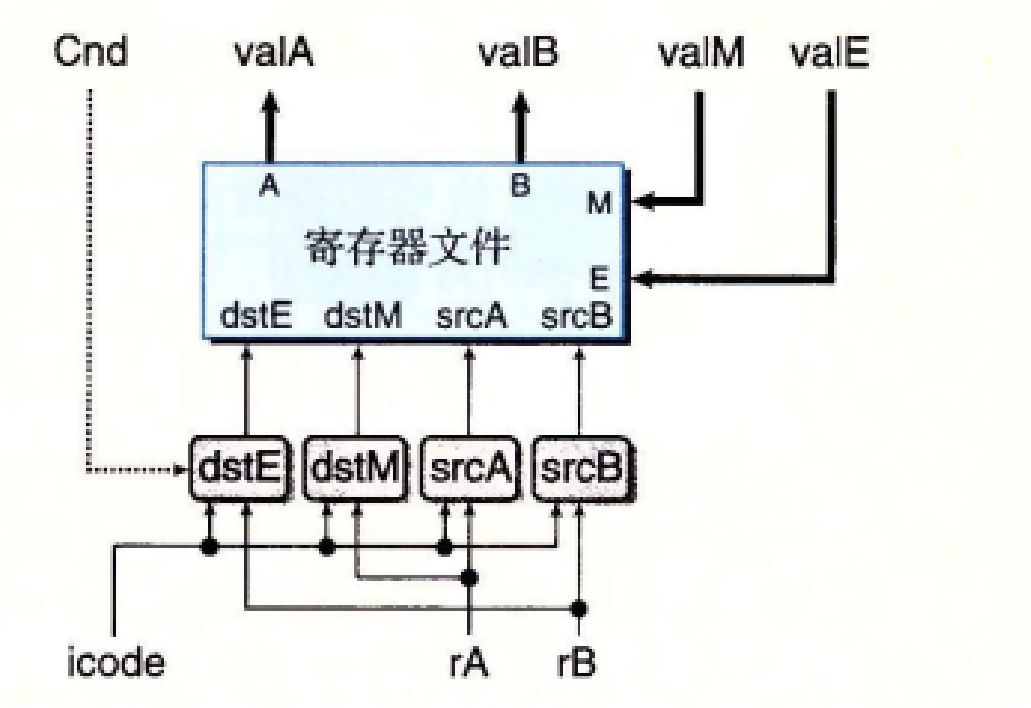

图 3.2: 译码和写回阶段

寄存器文件有四个端口，同时支持两个读和两个写操作，其中读的端口输出的是 srcA 和 srcB 对应的寄存器的值到 valA，valB，若为 0xf，则代表不需要从寄存器取出数。同理，在写回阶段，我们将 valM 和 valE 的值写进 dstM 和 dstE 对应的寄存器，若为 0xf 则不执行写操作。其中 srcA、srcB、dstM、dstE 的值是根据 icode 从 rA、rB 中取出的。具体 hcl 实现如下：

```
1     ## What register should be used as the A source?
2     word srcA = [
3     icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ  } : rA;
4     icode in { IPOPQ, IRET } : RRSP;
5     1 : RNONE; # Don't need register
6     ];
7
8     ## What register should be used as the B source?
9     word srcB = [
10    icode in { IOPQ, IRMMOVQ, IMRMOVQ, IIADDQ  } : rB;
11    icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
12    1 : RNONE;  # Don't need register
13    ];
```

```
14
15      ## What register should be used as the E destination?
16      word dstE = [
17      icode in { IRRMOVQ } && Cnd : rB;
18      icode in { IIRMOVQ, IOPQ, IIADDQ} : rB;
19      icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
20      1 : RNONE;  # Don't write any register
21      ];
22
23      ## What register should be used as the M destination?
24      word dstM = [
25      icode in { IMRMOVQ, IPOPQ } : rA;
26      1 : RNONE;  # Don't write any register
27      ];
```

3.3 执行阶段

执行阶段的硬件设计如下:

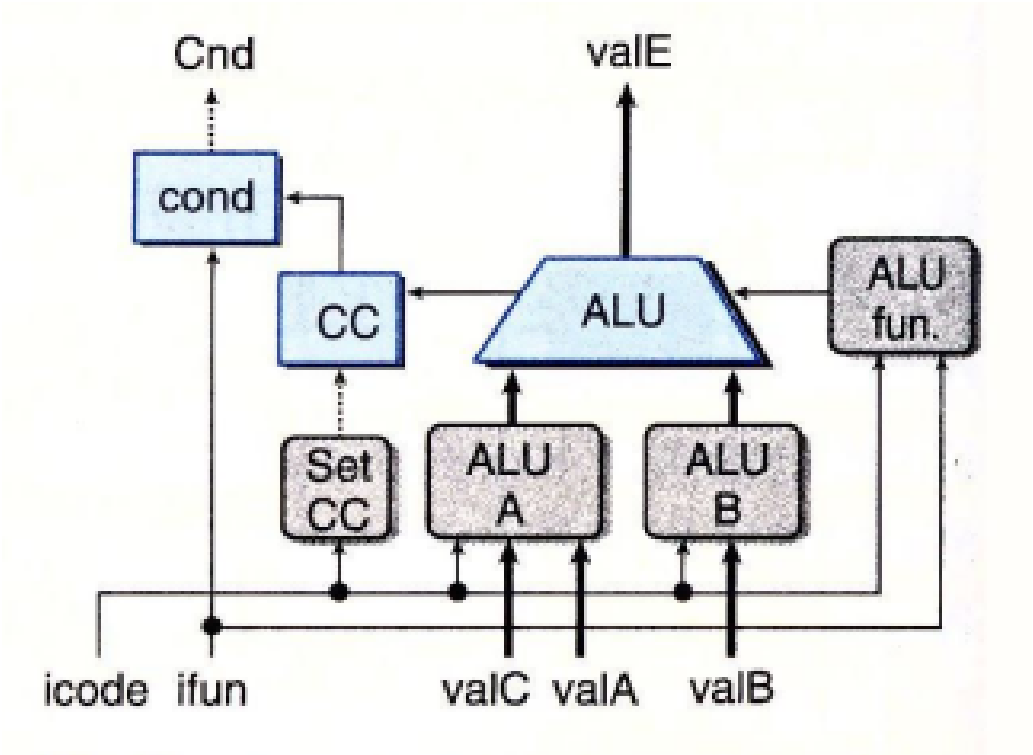

图 3.3: 执行阶段

执行阶段。算数/逻辑单元会根据 alufun 的值，对输入的 aluA 和 aluB 执行 ADD、SUBTRACT、AND 或 EXCLUSIVE-OR 运算。同时执行 OPQ 指令时会产生一个 setCC 信号，根据运算结果对状态码进行改变。aluA 的值由 icode 来进行决定，具体 hcl 实现如下：

```
## Select input A to ALU
word aluA = [
icode in { IRRMOVQ, IOPQ } : valA;
icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ, IIADDQ } : valC;
icode in { ICALL, IPUSHQ } : -8;
icode in { IRET, IPOPQ } : 8;
# Other instructions don't need ALU
];

## Select input B to ALU
word aluB = [
icode in { IRMMOVQ, IMRMOVQ, IOPQ, ICALL,
IPUSHQ, IRET, IPOPQ, IIADDQ } : valB;
icode in { IRRMOVQ, IIRMOVQ } : 0;
# Other instructions don't need ALU
];

## Set the ALU function
word alufun = [
icode == IOPQ : ifun;
1 : ALUADD;
];

## Should the condition codes be updated?
bool set_cc = icode in { IOPQ, IIADDQ };
```

3.4 访存阶段

访存阶段的硬件结构如下：

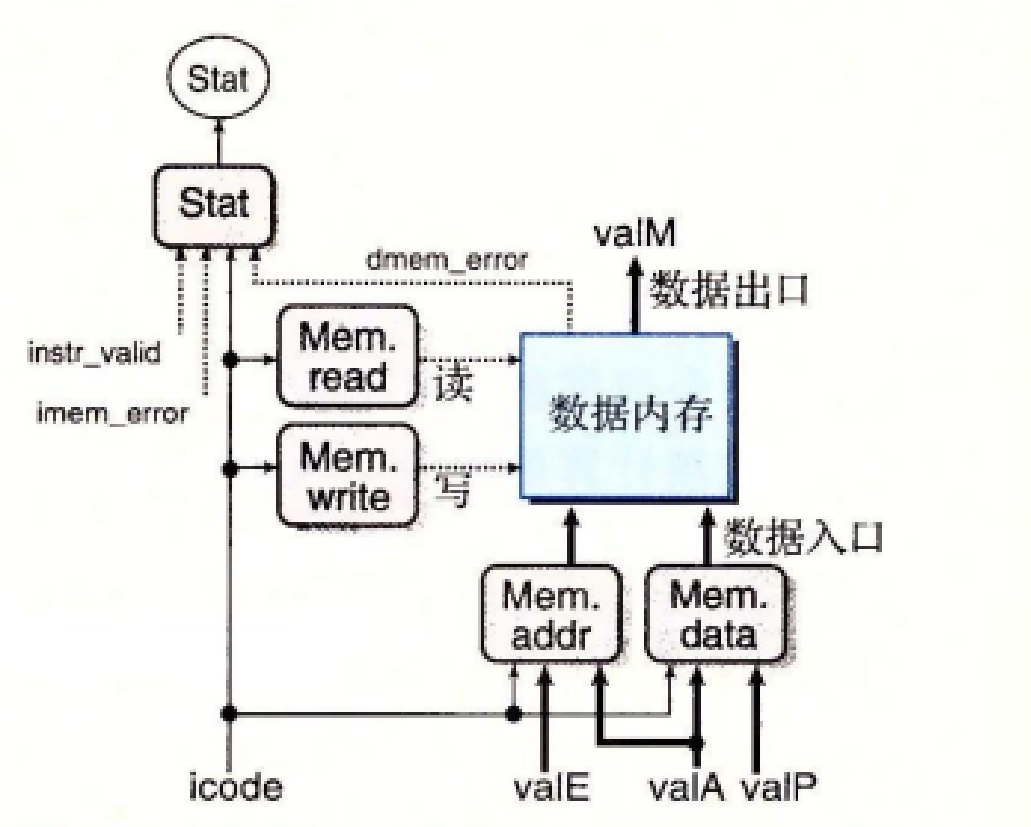

图 3.4: 访存阶段

访存阶段会对内存进行读或写，根据 icode 来确定是否需要进行内存操作，以及具体读写的地址, 同时会更新状态码。hcl 实现逻辑如下:

```
1     ## Set read control signal
2     bool mem_read = icode in { IMRMOVQ, IPOPQ, IRET };
3
4     ## Set write control signal
5     bool mem_write = icode in { IRMMOVQ, IPUSHQ, ICALL };
6
7     ## Select memory address
8     word mem_addr = [
9     icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : valE;
10    icode in { IPOPQ, IRET } : valA;
11    # Other instructions don't need address
12    ];
13
```

```
    ## Select memory input data
    word mem_data = [
    # Value from register
    icode in { IRMMOVQ, IPUSHQ } : valA;
    # Return PC
    icode == ICALL : valP;
    # Default: Don't write anything
    ];

    ## Determine instruction status
    word Stat = [
    imem_error || dmem_error : SADR;
    !instr_valid: SINS;
    icode == IHALT : SHLT;
    1 : SAOK;
    ];
```

3.5 更新 PC 阶段

更新 PC 阶段的硬件结构如下:

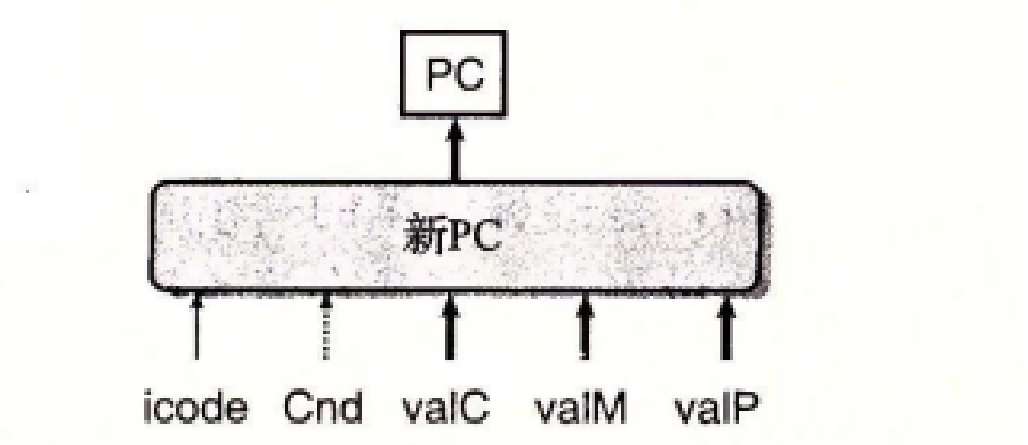

图 3.5: 更新 PC 阶段

SEQ 的最后一个阶段，会产生程序计数器的新值，新值的生成逻辑如下:

```
    word new_pc = [
```

```
    # Call.  Use instruction constant
    icode == ICALL : valC;
    # Taken branch.  Use instruction constant
    icode == IJXX && Cnd : valC;
    # Completion of RET instruction.  Use value from stack
    icode == IRET : valM;
    # Default: Use incremented PC
    1 : valP;
    ];
```

4 Verilog 实现

4.1 实现过程

4.1.1 顶层模块设计

CPU 的不同阶段分别写在了不同的模块中，最终在顶层模块中进行组装串联起来。按照我们之前对 SEQ 的划分，CPU 一共有 8 个子模块，其中 6 个对应着 6 个阶段，还有两个分别用来模拟寄存器组和内存。实现起来如下：

```
1   module y86cpu(...);
2
3   Fetch f(clk, rst, pc, rom_data_i, icode, ifun, rA, rB, valP, valC);
4   Decode d(clk, rst, icode, ifun, rA, rB, srcA, srcB);
5   regfile r(clk, rst, srcA, srcB, dstM, dstE, valM, valE, valA, ...
        valB,rax,rcx,rdx,rbx,rsp,rbp,rsi,rdi,r8,r9,r10,r11,r12,r13,r14,r15);
6   Execute e(clk, rst, icode, ifun, valA, valB, valC, valE, Cnd);
7   Write w(clk, rst, Cnd, icode, rA, rB, dstM, dstE);
8   Memory m(clk, rst, icode, valE, valA, valP, mem_read_o, mem_write_o, ...
        mem_addr_o, mem_data_o);
9   inst_rom i(clk,valM, pc, mem_read_o, mem_write_o, mem_addr_o, ...
        mem_data_o, valM, inst);
10  Update p(clk, rst, icode, Cnd, valP, valM, valC, pc);
11  assign rom_addr_o = pc;
12
13  endmodule
```

这里省去了顶层模块的输入输出端口以及中间网线和寄存器的定义，其中 regfile 用来模拟寄存器组，inst_rom 模拟内存，用于访存阶段和指令读取。

4.1.2 Fetch 阶段设计

分析 3.1 中的硬件结构可知，Fetch 阶段需要的输入有 clk，rst，pc，inst(指令)，输出有 icode，ifun，rA，rB，valP，valC。指令一次取 10 字节，再根据需要取出相应部分。同时指令在内存中以小端模式存储，value 采用 64 位，如下

```
module Fetch(
input clk,
input rst,
input [63:0]pc,
input [79:0]inst,
output reg [3:0]icode,
output reg [3:0]ifun,
output reg [3:0]rA,
output reg [3:0]rB,
output reg [63:0]valP,
output reg [63:0]valC
);
```

首先取出 icode 和 ifun，取出的命令为：

```
icode = inst[79:76];
ifun = inst[75: 72];
```

之后根据 icode 的值，确定 valP，valC 以及 rA，rB。以 IMRMOVQ 为例：

```
valP = pc + 64'hA;
rA = inst[`RA];
rB = inst[`RB];
valN = inst[`D]; //辅助转为大端
valC = {valN[7:0], valN[15:8], valN[23:16], valN [31:24], valN [39:32], ...
    valN [47:40], valN[55:48], valN[63:56]};
```

4.1.3 Decode 阶段设计

分析 3.2 中的硬件结构可知，Decode 阶段需要的输入有 clk，rst，icode，ifun，rA，rB，输出有 srcA，srcB。本阶段主要根据 icode 来确定 srcA 和 srcB 的值，最终从寄存器取值是在 regfile 模块进行的。

```
module  Decode(
input wire clk,
input wire rst,
input wire [3:0]icode,
input wire [3:0]ifun,
input wire [3:0]rA,
input wire [3:0]rB,
output reg [3:0]srcA,
output reg [3:0]srcB
);
```

具体取值逻辑是由 3.2 中的 hcl 逻辑转变来的，这里以 IMRMOVQ 为例：

```
`IMRMOVQ:    begin
        srcA = `RNONE; // `RNONE = 0xf, 即不需要寄存器
        srcB =  rB;
    end
```

4.1.4 Execute 阶段设计

分析 3.3 中的硬件结构可知，Execute 阶段需要的输入有 clk，fun，icode，ifun，valA，valB，valC，输出有 valE 和状态码 Cnd，其中 valA，valB，valC，valE 都是有符号数。本阶段若执行 IOPQ 或 IIADDQ 指令，会设置状态寄存器 (of,zf,sf)，其余阶段则是获得状态码，以及输出 valE。

```
module Execute(
input wire clk,
input wire rst,
```

```
    input wire [3:0] icode,
    input wire [3:0] ifun,
    input wire signed [63:0] valA, //有符号数
    input wire signed [63:0] valB,
    input wire signed [63:0] valC,
    output reg signed [63:0] valE,
    output reg Cnd
    );
```

这里在运算时会用到 ifun 的值，会根据某个具体的命令计算 valE 的值和 Cnd 的值。以 IJXX 为例：

```
    `IJXX:  begin
        case (ifun )
            `FJMP:  begin
                Cnd = 1;
            end
                `FJLE:  begin
                Cnd = (sf ^ of) | zf;
            end
            `FJL:   begin
                Cnd = (sf ^ of);
            end
            `FJE:   begin
                Cnd = zf;
            end
            `FJNE:  begin
                Cnd = ¬zf;
            end
            `FJGE:  begin
                Cnd = ¬(sf ^ of);
            end
            `FJG:   begin
                Cnd = ¬(sf ^ of) & ¬zf;
            end
        endcase
    end
```

同时这个阶段中，我们要将 IOPQ 和 IIADDQ 命令的执行写在下降沿触发的 always 块中，这里是为了确保其在一个周期内只执行一次，以免状态寄存器更新错误。以 IIADDQ 为例：

```
1     always @(negedge clk) begin
2         case(icode)
3             `IIADDQ:     begin
4                 aluA = valC;
5                 aluB = valB;
6                 valE = aluA + aluB;
7                 /* set CC */
8                 zf = (valE == 0) ;
9                 sf = (aluB < aluA) ;
10                of = (valA<0 == valB<0) && (valE<0 != valA<0);
11            end
12        endcase
13    end
```

4.1.5 Memory 阶段设计

分析 3.4 中的硬件结构可知，Memory 阶段需要的输入有 clk,rst,icode,valE,valA,valP，输出有 mem_read(读信号),mem_write(写信号),mem_addr(读/写地址),mem_data(写数据),Stat(状态码)。这个阶段操作较少，只需根据 icode 确定 mem_read,mem_addr,mem_write,mem_data 即可。实现如下：

```
1     always @(*) begin
2         mem_read = 0;
3         mem_write = 0;
4         case(icode)
5             `IMRMOVQ:         begin
6                 mem_addr = valE;
7                 mem_read = 1;
8             end
9             `IRMMOVQ:         begin
```

```
10                  mem_addr = valE;
11                  mem_data = valA;
12                  mem_write = 1;
13              end
14              `IPUSHQ:          begin
15                  mem_addr = valE;
16                  mem_data = valA;
17                  mem_write = 1;
18              end
19              `ICALL:       begin
20                  mem_addr = valE;
21                  mem_data = valP;
22                  mem_write = 1;
23              end
24              `IPOPQ: begin
25                  mem_addr = valA;
26                  mem_read = 1;
27              end
28              `IRET:        begin
29                  mem_addr = valA;
30                  mem_read = 1;
31              end
32          endcase
33      end
```

4.1.6 Write back 阶段实现

Write back 阶段的硬件结构与译码阶段相同，但该阶段的工作是向寄存器中写入值，应有的输入有 clk,rst,Cnd,icode,rA,rB, 输出有 dstM,dstE，该输出会作为 regfile 的控制逻辑。dstM 和 dstE 是根据 icode 和 Cnd 的值从 rA，rB 中选择的。以 IRRMOVQ 为例：

```
1       module Write(
2       input wire clk,
3       input wire rst,
4       input wire Cnd,
5       input wire [3:0] icode,
```

```
    input wire [3:0] rA,
    input wire [3:0] rB,
    output reg [3:0] dstM,
    output reg [3:0] dstE
    );
```

```
    `IRRMOVQ:    begin   // 也是CMOV
        dstE = Cnd == 1 ? rB : `RNONE;
        dstM = `RNONE;
    end
```

4.1.7 PC update 阶段设计

PC update 阶段的硬件设计很简单，只是一个带 clk 时钟端的多路选择器，我们从之前产生的 valP,valM,valC 中选择一个值作为我们的新的 pc，选择要根据我们的 icode 和 Cnd 状态码来进行。

```
    module Update(
    input wire clk,
    input wire rst,
    input wire [3:0]icode,
    input wire Cnd,
    input wire [`WORD]valP,
    input wire [`WORD]valM,
    input wire [`WORD]valC,
    output reg [`WORD]pc
    );

    always @(posedge clk)   begin
        case(icode)
            `ICALL: pc = valC;
            `IJXX:  pc = Cnd == 1 ? valC : valP;
            `IRET:  pc = valM;
            default:    begin
```

```
18              if(valP > 0)
19                  pc = valP;
20              else
21                  pc = 0;   //这里要对pc做一个初始化，默认从指令第一条执行
22              end
23          endcase
24      end
25
26      endmodule
```

实际上需要我们单独处理的指令只有 ICALL,IJXX,IRET 三个，其余的都是根据我们一开始计算出的 valP 来跳转。同时，这个阶段的触发是在 clk 的上升沿进行的，保证了每个周期结束后更新 pc。

4.1.8 寄存器模块设计

regfile 模块用于模拟寄存器组，里面我们定义了 16 个 64 位的寄存器, 其中第 16 个不用:

```
1       reg [`WORD] regs [0:15];
```

在这个模块中，读寄存器部分是组合逻辑，也就是输出的 valA，valB 会随时根据 srcA 和 srcB 的值发生变化，写的部分则是时序逻辑，写的操作只会发生在 clk 的上升沿部分。

```
1       always @(posedge clk)   begin
2           if(dstE != `RNONE)  begin
3               regs[dstE] = valE;
4           end
5           if(dstM != `RNONE)  begin
6               regs[dstM] = valM;
7           end
8       end
9
10      always @(*) begin
11          if(srcA != `RNONE)  begin
12              valA = regs[srcA];
13          end
```

```
    end

    always @(*) begin
        if(srcB != `RNONE)  begin
            valB = regs[srcB];
        end
    end
```

通过 assign 语句将 regs 的值赋给输出端口的寄存器:

```
    assign rax = regs[0];
    assign rcx = regs[1];
    assign rdx = regs[2];
    assign rbx = regs[3];
    assign rsp = regs[4];
    assign rbp = regs[5];
    assign rsi = regs[6];
    assign rdi = regs[7];
    assign r8 = regs[8];
    assign r9 = regs[9];
    assign r10 = regs[10];
    assign r11 = regs[11];
    assign r12 = regs[12];
    assign r13 = regs[13];
    assign r14 = regs[14];
    assign r15 = regs[15];
```

4.1.9 内存模块设计

inst_rom 模块用于模拟内存。在其中我们定义了一个 600 个的 64 位的寄存器组，当做我们的内存空间，初始化时，从文件中读取我们的指令 (文件是事先准备好的二进制文件，8 位放在一起，以空格隔开) 到寄存器组。

```
    reg [7:0] inst_mem[0:599];
    initial $readmemh ( "D:/Xilinx/Projects/seq/inst_rom.data", inst_mem);
```

取指令时一次取 10 字节，然后之后再根据具体的指令确定需要取出哪部分的值，这是 Fetch 中的工作，而取指逻辑只需要一条 assign 语句即可。

```
1    assign inst = ...
       {inst_mem[addr],inst_mem[addr+1],inst_mem[addr+2],inst_mem[addr+3],
2    inst_mem[addr+4],inst_mem[addr+5],inst_mem[addr+6],inst_mem[addr+7],
3    inst_mem[addr+8],inst_mem[addr+9]};
```

内存读写时，读是组合逻辑，随时都在进行，根据 valM(mem_addr) 来确定，而写则是时序逻辑，发生在 clk 的上升沿。

```
1    assign mem_data_o = mem_read_i ? ...
       {inst_mem[mem_addr_i+7],inst_mem[mem_addr_i+6],inst_mem[mem_addr_i+5],
2    inst_mem[mem_addr_i+4],inst_mem[mem_addr_i+3],inst_mem[mem_addr_i+2],
3    inst_mem[mem_addr_i+1],inst_mem[mem_addr_i]} : valM;
4
5    always @(posedge clk) begin
6        if( mem_write_i == `ENABLE ) begin
7            {inst_mem[mem_addr_i+7],inst_mem[mem_addr_i+6],
8            inst_mem[mem_addr_i+5],inst_mem[mem_addr_i+4],
9            inst_mem[mem_addr_i+3],inst_mem[mem_addr_i+2],
10           inst_mem[mem_addr_i+1],inst_mem[mem_addr_i]} =  mem_data_i;
11       end
12   end
```

由于指令和数据在内存中都是以小端方式存储的，所以在读和写的过程中，需要做一个转换，如上。

4.2 仿真结果

4.2.1 asumi 函数仿真

以上面的 asumi 函数为例，给出仿真结果，图 4.2 给出了 asumi 的汇编代码：

y86cpu.v | inst_rom.data | y86cpu_a.v

```
30 f4 00 01 00 00 00 00 00 00
80 38 00 00 00 00 00 00 00
00
00 00 00 00
0d 00 0d 00 0d 00 00 00
c0 00 c0 00 c0 00 00 00
00 0b 00 0b 00 0b 00 00
00 a0 00 a0 00 a0 00 00
30 f7 18 00 00 00 00 00 00 00
30 f6 04 00 00 00 00 00 00 00
80 56 00 00 00 00 00 00 00
90
63 00
62 66
70 83 00 00 00 00 00 00 00
50 a7 00 00 00 00 00 00 00 00
60 a0
c0 f7 08 00 00 00 00 00 00 00
c0 f6 ff ff ff ff ff ff ff ff
74 63 00 00 00 00 00 00 00
90
```

图 4.1: 输入的 asumi 机器码

```
                        | # Execution begins at address 0
0x000:                  |   .pos 0
0x000: 30f4000100000000000 |   irmovq stack, %rsp      # Set up stack pointer
0x00a: 803800000000000000  |   call main       # Execute main program
0x013: 00               |   halt            # Terminate program
                        |
                        | # Array of 4 elements
0x018:                  |   .align 8
0x018: 0d000d000d000000 | array:    .quad 0x000d000d000d
0x020: c000c000c0000000 |   .quad 0x00c000c000c0
0x028: 000b000b000b0000 |   .quad 0x0b000b000b00
0x030: 00a000a000a00000 |   .quad 0xa000a000a000
                        |
0x038: 30f718000000000000 | main: irmovq array,%rdi
0x042: 30f604000000000000 |   irmovq $4,%rsi
0x04c: 805600000000000000  |   call sum        # sum(array, 4)
0x055: 90               |   ret
                        |
                        | /* $begin sumi-ys */
                        | # long sum(long *start, long count)
                        | # start in %rdi, count in %rsi
0x056:                  | sum:
0x056: 6300             |   xorq %rax,%rax      # sum = 0
0x058: 6266             |   andq %rsi,%rsi      # Set condition codes
0x05a: 708300000000000000 |   jmp    test
0x063:                  | loop:
0x063: 50a700000000000000 |   mrmovq (%rdi),%r10  # Get *start
0x06d: 60a0             |   addq %r10,%rax          # Add to sum
0x06f: c0f708000000000000 |   iaddq $8,%rdi           # start++
0x079: c0f6ffffffffffffff |   iaddq $-1,%rsi          # count--
0x083:                  | test:
0x083: 746300000000000000 |   jne    loop             # Stop when 0
0x08c: 90               |   ret
                        | /* $end sumi-ys */
                        |
                        | # The stack starts here and grows to lower addresses
0x100:                  |   .pos 0x100
0x100:                  | stack:
```

图 4.2: asumi 函数的汇编代码

编写波形仿真文件，如下：

```
module y86_sim();

reg clk;
reg rst;
wire [63:0]rax;
wire [63:0]rcx;
wire [63:0]rdx;
wire [63:0]rbx;
wire [63:0]rsp;
wire [63:0]rbp;
wire [63:0]rsi;
wire [63:0]rdi;
wire [63:0]r8,r9,r10,r11,r12,r13,r14,r15;

initial begin
clk = 0;
rst = 0;
end
```

```
19
20      always #10 clk = ¬clk;
21
22      y86cpu_a sim1(clk,rst,rax,rcx,rdx,rbx,rsp,rbp,rsi,rdi,r8,r9,r10,r11,r12,
23      r13,r14,r15);
24
25      endmodule
```

运行后查看各寄存器的状态:

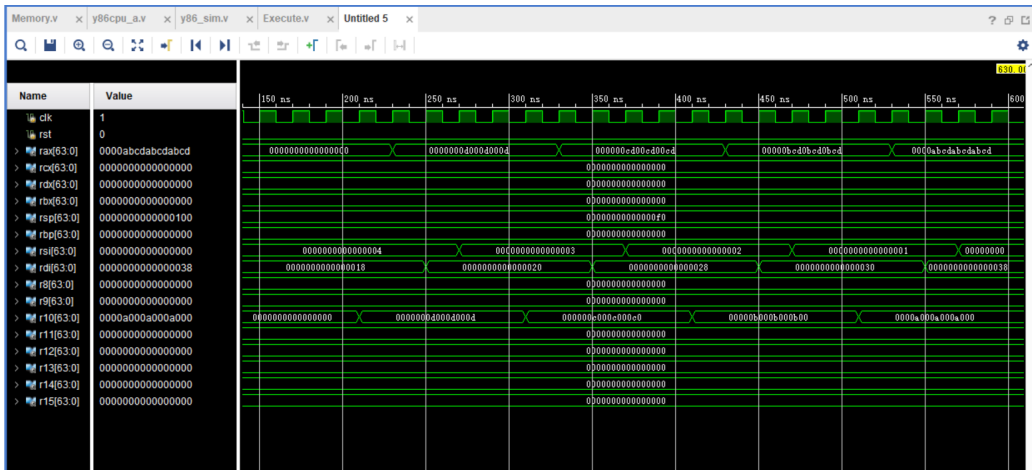

图 4.3: 程序中各寄存器的状态

可以看到汇编代码中,我们将 0x000d000d000d、0x00c000c000c0、0x0b000b000b00、0xa000a000a000 这四个值依次相加赋给 rax,rax 寄存器的状态一次为 0x0000000000000000、0x0000000d000d000d、0x000000cd00cd00cd、0x00000bcd0bcd0bcd、0x0000abcdabcdabcd, 如下图:

图 4.4: rax 的状态变化

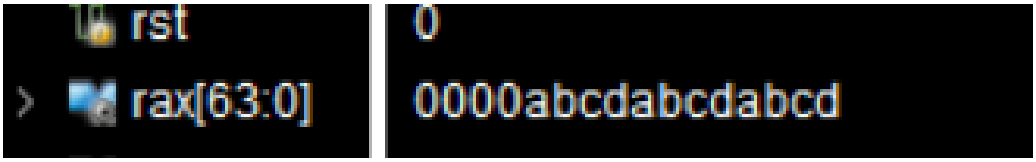

图 4.5: rax 的最终状态

4.2.2 irmovq 指令仿真

选取 asumi 函数中的 irmovq 指令单独分析，如下:

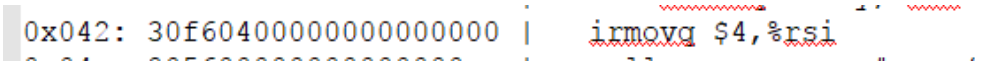

图 4.6: irmovq 指令

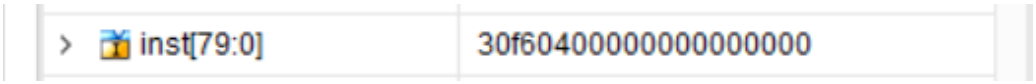

图 4.7: irmovq 指令执行前的指令寄存器的值

图 4.8: irmovq 指令执行前的 rsi 寄存器的值

图 4.9: irmovq 指令执行后的 rsi 寄存器的值

4.2.3 iaddq 指令仿真

选取 asumi 函数中的 iaddq 指令单独分析如下:

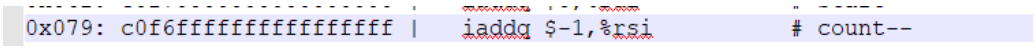

图 4.10: iaddq 指令

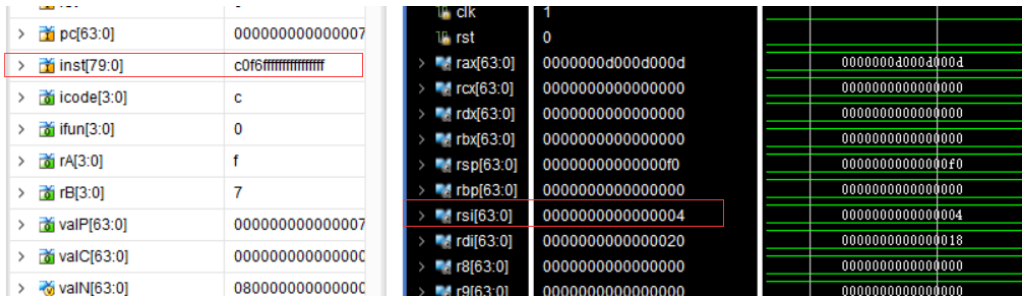

图 4.11: iaddq 指令执行前的 rsi 寄存器的值

图 4.12: iaddq 指令执行后的 rsi 寄存器的值

4.2.4 subq 指令仿真

选取 asumi 函数中的 subq 指令单独分析如下：

```
0x0c3: 6178                    |    subq %rdi, %r8
```

图 4.13: subq 指令

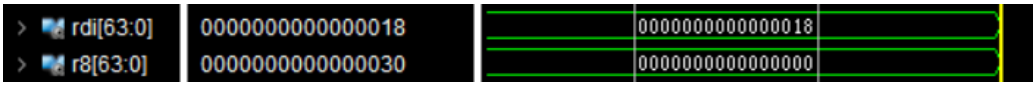

图 4.14: subq 指令执行前的 rsi 寄存器的值

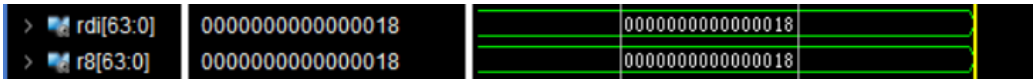

图 4.15: subq 指令执行后的 rsi 寄存器的值

4.2.5 addq 指令仿真

选取 asumi 函数中的 addq 指令单独分析如下：

```
0x09b: 6080                    |    addq %r8, %rax
```

图 4.16: iaddq 指令

图 4.17: addq 指令执行前的 rax 寄存器的值

图 4.18: addq 指令执行前的 r8 寄存器的值

图 4.19: addq 指令执行后的 rax 寄存器的值

4.2.6 rrmovq 指令仿真

选取 asumi 函数中的 rrmovq 指令单独分析如下：

```
0x073: 2098                  |    rrmovq %r9, %r8
```

图 4.20: rrmovq 指令

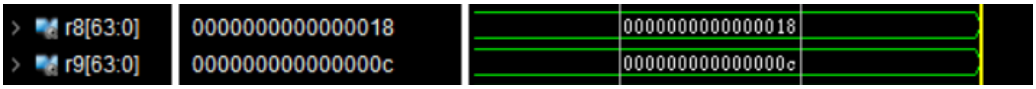

图 4.21: rrmovq 指令执行前的 r8,r9 寄存器的值

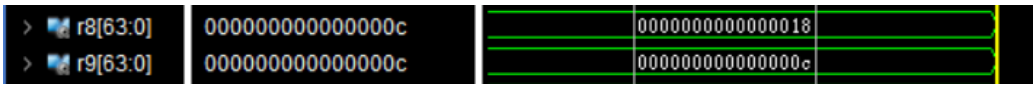

图 4.22: rrmovq 指令执行后的 r8,r9 寄存器的值

结　论

本次实验从《深入理解计算机系统》一书上给出的 Y86-64 指令集出发，加入了 IADDQ 指令构成新的指令集，设计并实现了以该指令集为基础的顺序处理器。实验首先分析 SEQ 的各个阶段各指令的执行过程，给出了指令在 SEQ 中的微指令，并以此完成了 SEQ 各阶段的硬件设计及其 hcl 逻辑实现。最后使用 verilog 硬件设计语言完成了 CPU 的编写，给出了自己编写的 testbanch 文件跑出的波形。CPU 可以在给出连续的指令后顺序执行，直到遇到 halt 指令或错误指令停止。

参 考 文 献

[1] 兰德尔 E. 布莱恩特等著，龚奕利，贺莲译. 深入理解计算机系统 (第三版)[M]. 北京：机械工业出版社，2016

[2] Palnitkar 著，夏宇闻等译，Verilog HDL 数字设计与综合 (第二版)[M]. 北京: 电子工业出版社，2004.11