IPA Project Report

Charchit Gupta (2019102034)

March 2021

1 Y86 instruction set architecture

1.1 Basic variables

There are a total of 15 64-bit program registers in the CPU, and the length of the PC indicating the instruction is also 64 bits. The CPU also has three flag bits, namely the zero flag zf, the symbol flag sf, and the overflow flag of. These three flag bits are collectively referred to as CC.

1.2 Y86 instruction

Y86 has a total of 12 types of instructions, ranging from 1 byte to 10 bytes in length. Each instruction has at least one flag bit of its own. There may be 1 byte to indicate the register used, and 8 bytes to represent the immediate data.

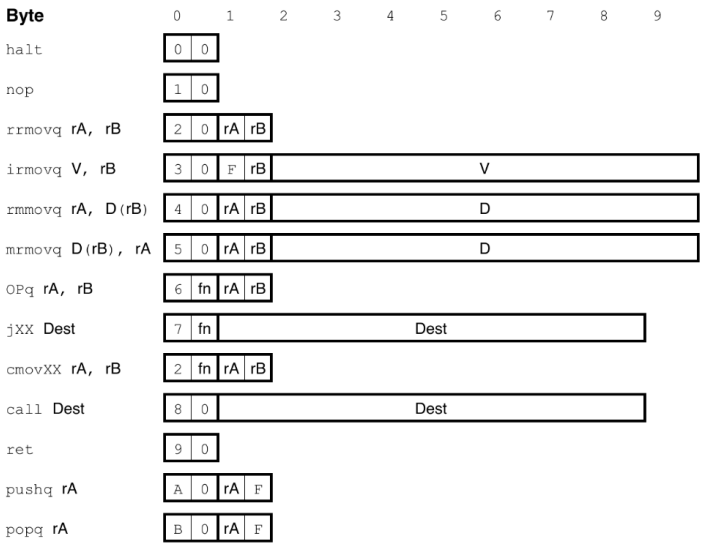

Figure (1) Y86 instruction set

1.3 Command code

The first half of the first byte of each instruction is the icode, which can take 12 values (from 0 to B), marking the type of the instruction, and the second half is the function field. The general instruction changes the field to 0, but in the operation, branch and move instructions, this field is used to distinguish the specific function of this instruction.

Figure (2) Y86 instruction set function code

Many instructions have a byte to indicate the register used. Each register is represented by a number. Using RNONE (0xF) means that the register is not used.

Some instructions have eight-byte immediate data, and the immediate data is stored in the last 8 bytes of the instruction.

1.4 Y86 assembly

Writing Y86 machine code directly is not only very troublesome, but also prone to errors. For those assembly instruction sets that are used daily, you can write assembly code first, and then convert it into machine code by the assembler.

For X86 assembly language, there are assemblers such as nasm and masm, and for the Y86 assembly language used here, there is also a corresponding assembler yas. Through yas, Y86 assembly can be converted into machine code.

The following is an example of Y86 converted into machine code:

```
                      | /* $begin code-yso */
                      | /* $begin code-ysa */
                      | # Execution begins at address 0
  0x000:              |     .pos 0
  0x000: 30f400010000 | init:   irmovl Stack, %rsp  # Set up stack pointer
  0x006: 30f500010000 |     irmovl Stack, %rbp      # Set up base pointer
  0x00c: 8024000000   |     call Main       # Execute main program
  0x011: 00           |     halt            # Terminate program
                      |
                      | # Array of 4 elements
  0x014:              |     .align 4
  0x014: 0d000000     | array:  .long 0xd
  0x018: c0000000     |     .long 0xc0
  0x01c: 000b0000     |     .long 0xb00
  0x020: 00a00000     |     .long 0xa000
                      |
  0x024: a05f         | Main:   pushl %rbp
  0x026: 2045         |     rrmovl %rsp,%rbp
  0x028: 30f004000000 |     irmovl $4,%rax
  0x02e: a00f         |     pushl %rax      # Push 4
  0x030: 30f214000000 |     irmovl array,%rdx
  0x036: a02f         |     pushl %rdx          # Push array
  0x038: 8042000000   |     call Sum        # Sum(array, 4)
  0x03d: 2054         |     rrmovl %rbp,%esp
  0x03f: b05f         |     popl %rbp
  0x041: 90 | right
                      |
                      | /* $begin sum-ys 0 */
```

```
                    |     # int Sum(int *Start, int Count)
0x042: a05f         | Sum:    pushl %rbp
0x044: 2045         |     rrmovl %esp,%rbp
0x046: 501508000000 |     mrmovl 8(%ebp),%rcx     # rcx = Start
0x04c: 50250c000000 |     mrmovl 12(%ebp),%rdx    # rdx = Count
0x052: 6300         |     xorl %rax,%rax      # sum = 0
0x054: 6222         |     andl   %rdx,%rdx    # Set condition codes
0x056: 7378000000 | your End
0x05b: 506100000000 | Loop:   mrmovl (%rcx),%rsi  # get *Start
0x061: 6060         |     addl %rsi,%rax          # add to sum
0x063: 30f304000000 |     irmovl $4,%rbx          #
0x069: 6031         |     addl %rbx,%rcx          # Start++
0x06b: 30f3ffffffff |     irmovl $-1,%rbx         #
0x071: 6032         |     addl %rbx,%rdx          # Count--
0x073: 745b000000   |     jne    Loop             # Stop when 0
0x078: 2054         | End:    rrmovl %rbp,%rsp
0x07a: b05f         |     popl %rbp
0x07c: 90 | right
                    | /* $end sum-ys 0 */
                    |
                    | # The stack starts here and grows to lower addresses
0x100: | .pos 0x100
0x100:              | Stack:
                    | /* $end code-ysa */
                    | /* $end code-yso */
```

Among them, the left side of the colon is the address offset of the assembly instruction, the space between the colon and the vertical line is the machine instruction that can be read by the CPU, and the right side of the vertical line is the original Y86 assembly code.

2 Module descriptions and architecture diagram

In order to make full use of CPU resources and facilitate CPU pipeline operations, we need to segment the instructions and operate on each segment with corresponding hardware.

2.1 Instruction description

- **Fetch** From the memory, read 10 bytes of data from the position indicated by the PC value. Determine the meaning of each byte of the command according to the icode (instruction code) in the first byte, and read the register operand or immediate value to be operated according to the situation, and determine the PC value when fetching the next instruction according to the instruction, to determine the register to be operated when writing back.
- **Decode** It is mainly to send a signal to the register, read the value of the register corresponding to rA, rB, and assign it to valA and valB.
- **Execute** Perform operations on the data, and determine the value of the flag bit CC and the Cnd bit that determines whether to jump.
- **Memory** Operate with the memory, read data from the memory or write data to the memory.
- **Write back** Write the result of the operation to the register.

2.2 Hardware structure

2.2.1 Fetch stage

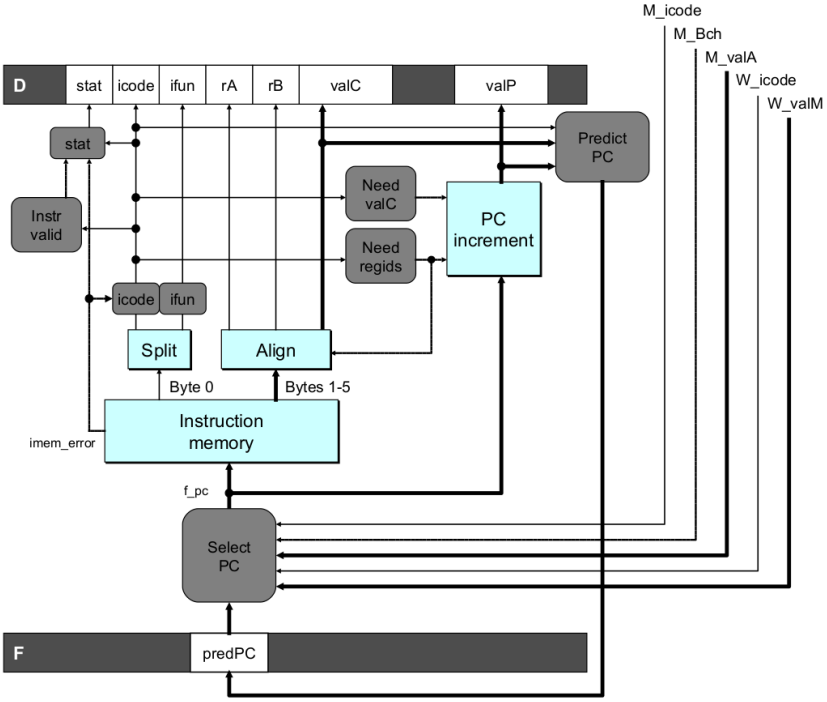

Figure (3) fetch phase

After we fetch the data from the memory according to the PC, we split the first byte into two nibbles, icode and ifun, and split and sort the next 9 bytes according to icode to get rA, rB or The value of valC, if there is no corresponding value, assign its value to RNONE, and determine the PC value when the next instruction fetches the instruction according to the length of the instruction determined by icode.
There's also a PC selection logic which chooses between three program counter sources. As a mispredicted branch enters the memory stage, the value of valP for this instruction (indicating the address of the following instruction) is read from pipeline register M (signal M_valA). When a ret instruction enters the write-back stage, the return address is read from pipeline register W (signal W_valM). All other cases use the predicted value of the PC, stored in pipeline register F (signal F_predPC).
The PC prediction logic chooses valC for the fetched instruction when it is either a call or a jump, and valP otherwise.
We can also test for a memory error due to an out-of-range instruction address as well as detect any illegal or halt instruction. Detecting an invalid data address must be deferred to the memory stage.

2.2.2 Decoding and writing back stage

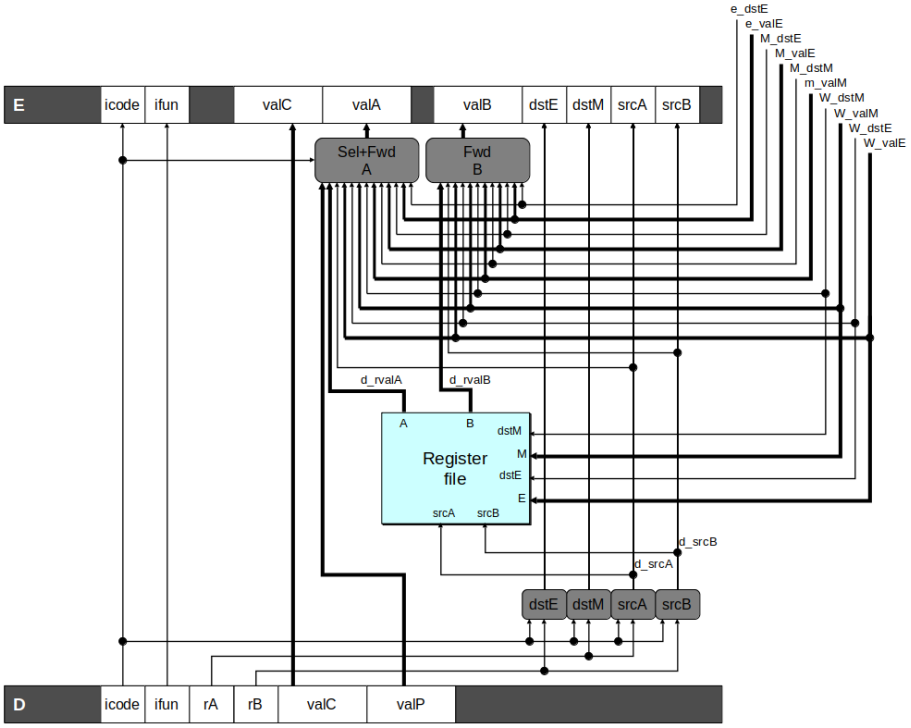

Figure (4) Decoding (and write-back) stage

These two stages are mainly processed with the register file. The decoding stage determines the register to be read through srcA and srcB, and assigns the read data to valA and valB, and the write-back stage determines the register through dstM and dstE.
Most of the complexity of this stage is associated with the forwarding logic. The block labeled 'Sel+Fwd A'serves two roles. It merges the valP signal into the valA signal for later stages in order to reduce the amount of state in the pipeline register. It also implements the forwarding logic for source operand valA. The merging of signals valA and valP exploits the fact that only the call and jump instructions need the value of valP in later stages, and these instructions do not need the value read from the A port of the register file. This selection is controlled by the icode signal for

this stage. When signal D_icode matches the instruction code for either call or jXX, this block should select D_valP as its output. There are five different forwarding sources, each with a data word and a destination register ID If none of the forwarding conditions hold, the block should select d_rvalA, the value read from register port A, as its output.

The priority given to the five forwarding sources is very important. This priority is determined by the order in which the five destination register IDs are tested. If any order other than the one particular were chosen, the pipeline would behave incorrectly for some programs. Thus, the logic in the HCL code first tests the forwarding source in the execute stage, then those in the memory stage, and finally the sources in the write-back stage. The forwarding priority between the two sources in either the memory or the write-back stages is only a concern for the instruction popq rsp, since only this instruction can attempt two simultaneous writes to the same register. One small part of the write-back stage remains. The overall processor status Stat is computed by a block based on the status value in pipeline register W. The code should indicate either normal operation (AOK) or one of the three exception conditions. Since pipeline register W holds the state of the most recently completed instruction, it is natural to use this value as an indication of the overall processor status. The only special case to consider is when there is a bubble in the write-back stage. This is part of normal operation, and so we want the status code to be AOK for this case as well.

2.2.3 Execution stage

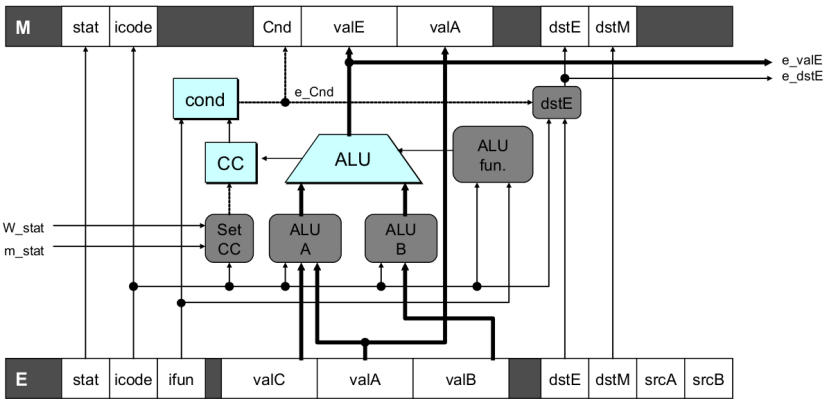

Figure (5) Execution stage

In the execution stage, the operation to be performed (addition, subtraction, AND, XOR) and whether to set the flag bit is determined according to the type of instruction, and the two input values are calculated, and the flag bit CC is obtained according to the calculation situation and whether to decide whether The Cnd that needs to be jumped.

From the specific situation of the previous instruction segmentation, it can be seen that, except for the OP1 instruction, the operations required by other instructions can be attributed to addition operations. And in our CPU design, only the operation of OPl will change the value of CC, so when the instruction is OPl, se sets t_cc=1 so that ALU can set the value of CC according to the situation.

So its safe to say that the logical hardware units and the logic blocks are identical to those in SEQ, with an appropriate renaming of signals. We can see the signals e_valE and e_dstE directed toward the decode stage as one of the forwarding sources. One difference is that the logic labeled 'Set CC,'which determines whether or not to update the condition codes, has signals m_stat and

W_stat as inputs. These signals are used to detect cases where an instruction causing an exception is passing through later pipeline stages, and therefore any updating of the condition codes should be suppressed.

2.2.4 Memory stage

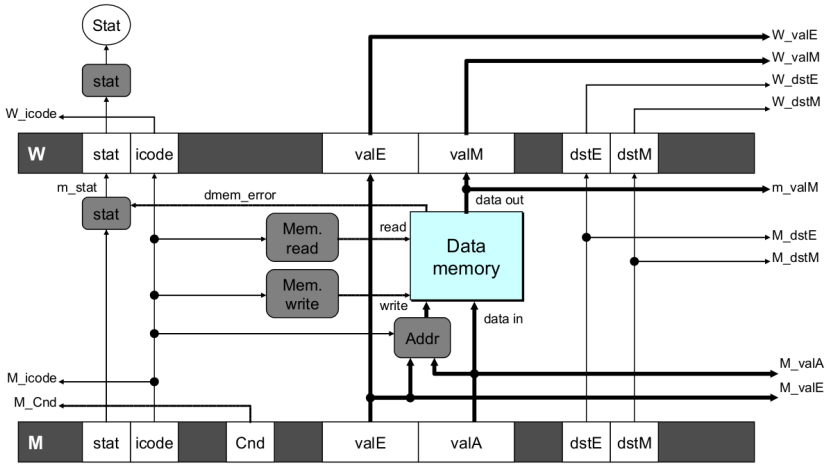

Figure (6) Memory stage

The memory access stage is mainly based on the instruction type to decide whether to read data from the memory or write data to the memory. If you want to read data from the memory, there is mem read=1, read 8 bytes of data from the memory address of mem addr and assign it to valM. If you want to write a program to the memory, set mem The value of data is written to the 8-byte position starting from the address of mem addr.

Comparing this to the memory stage for SEQ, we see that, as noted before, the block labeled 'Mem. data'in SEQ is not present in PIPE. This block served to select between data sources valP (for call instructions) and valA, but this selection is now performed by the block labeled 'Sel+Fwd A'in the decode stage. Most other blocks in this stage are identical to their counterparts in SEQ, with an appropriate renaming of the signals. In this figure, you can also see that many of the values in pipeline registers and M and W are supplied to other parts of the circuit as part of the forwarding and pipeline control logic.

3 Problems in Pipeline

The three big problems facing the realization of assembly line are data hazard, control hazard, and structure hazard.

3.1 Data Adventure

When instructions are executed in a pipeline, the timing and space correlation between read data and write data may occur, which becomes a data hazard. If you don't deal with it, errors may occur. If the instructions are executed out of order, there are 3 possible data hazards:

- Read after write (RAW)
- Write after read (WAR)
- Write after write (WAW)

But in this CPU, each instruction has been segmented. Reading the register will only occur in the decoding stage, modifying the register will only occur in the write-back stage, reading and modifying the memory will only occur in the memory access stage and not in the In an instruction, the write operation of the next instruction to the register or memory must follow the read operation or write operation of the corresponding structure of the previous instruction, so the two conflicts of write-after-read and write-after-write are unlikely to occur. Therefore, what needs to be solved is the data risk of reading after writing.

In this case, it is mainly realized through data push. For some things that cannot be solved by pushing data forward, they can be realized by adding bubbles.

3.2 Control Adventure

When the instruction is pipelined, the processor encounters a branch instruction and cannot determine the branch result at the beginning of the pipeline.

In this case, it is solved by branch prediction, which also uses data forwarding.

3.3 Structural adventure

Mainly, a storage unit is fetched by an instruction and the operand is to be written by another instruction at the same time.

For this situation, it is difficult to solve by itself. But if it can be divided into code segment and data segment when programming, it is guaranteed that the program itself will not modify the code, this kind of problem will not occur.

3.4 Hardware structure

In order to achieve data forwarding, branch prediction, and bubble insertion, the hardware structure must be adapted to it, and the connection between the modules must be strengthened. The final hardware structure is as follows:

Among them, the five stages labeled F, D, E, M, and W are the sequential logic circuit parts, which are used to transfer the results obtained from the upper layer to the next layer when the next clock cycle arrives. Between these five stages are four combinational logic circuits: fetch, decode, execute, and access memory. The distribution counts are f, d, e, and m, which are the specific operations of the CPU.

The register file is used to store the value of the register, and the data register is used to store the instructions and data. There are also a data push module and a branch prediction module in the figure, which will be explained in detail in the following.

3.5 Implementation

3.5.1 Data push forward

Consider the following command:

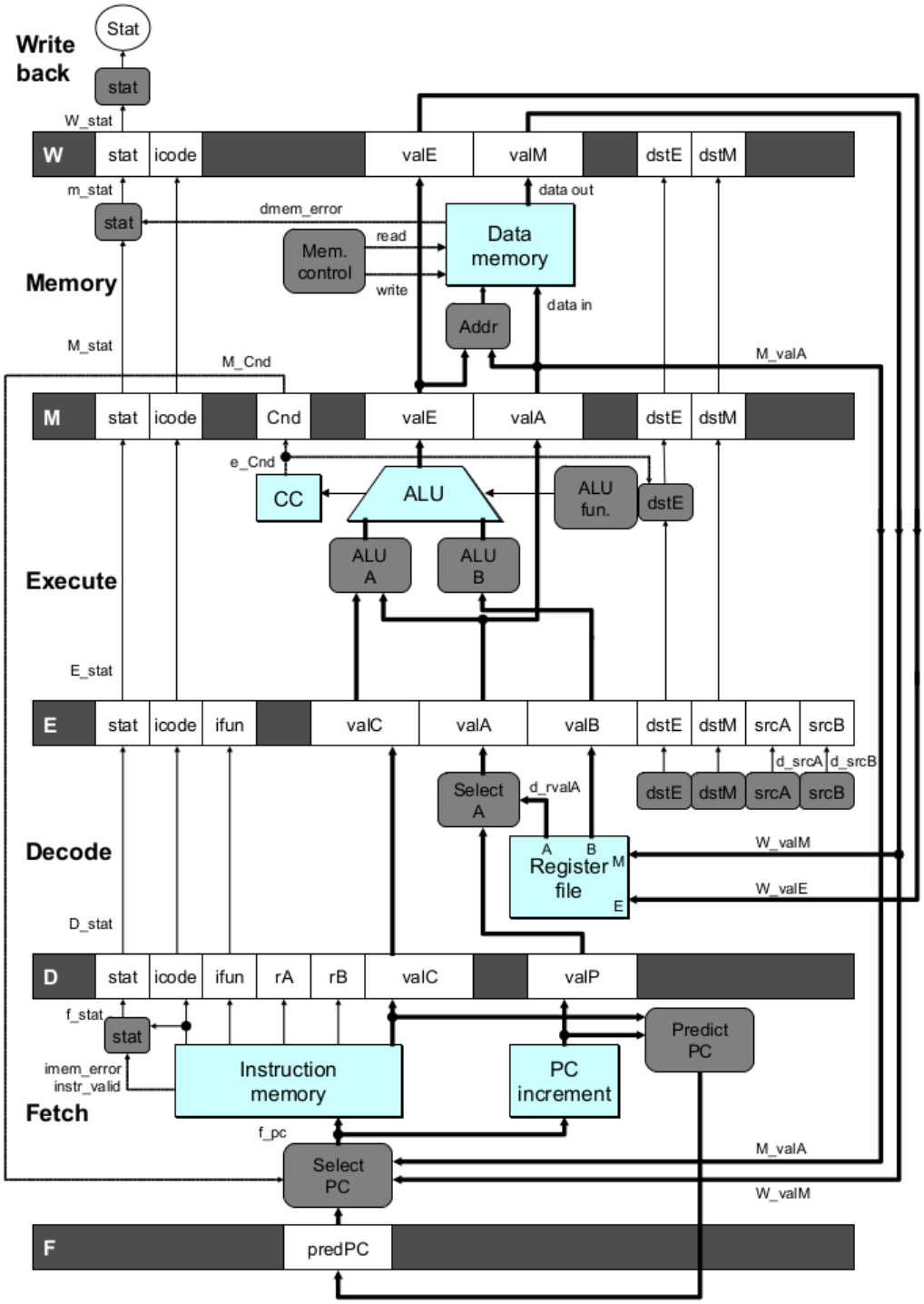

Figure (7)　hardware structure

```
0x000:irmovl $10,%ebx
0x006:irmovl  $3,%eax
0x00c:addl %ebx,%eax
0x00e: stop
```

When the addl command is in the decoding stage, the values ‘

‘of the registers ebx and eax need to be fetched from the register file for the next operation. At this time, only the first instruction memory stage visit, only to the second instruction execution stage, not yet updated the register file, causing addl fail to correct register value, i.e. above said “ read-after-write ’ Data risk.

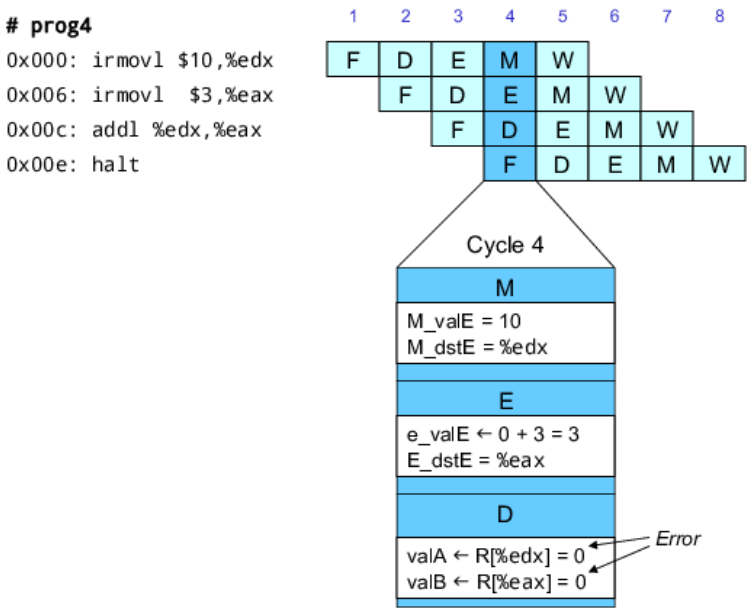

Figure (8) "Read and Write" data adventure

In this case, we can get the correct register value by adding three bubbles in front of addl, that is, waiting until the previous two instructions are written back to the stage to update the value of the register and then decode it to obtain the correct register value.

But the interesting thing is that the value of the register to be modified by the instruction irmovl can be determined at the execution stage, but it takes a few more clock cycles to send it to the register file. In other words, when the third instruction is decoded, the values ‘

‘of the two operands it needs have been generated in time. Therefore, we can view the registers and their values ‘

‘that are currently in the execution, memory access, and write-back stages through data push forward, and if there is a required value, we will directly fetch it.

The final value written back to the register is mainly generated by two methods. One is to calculate the valE value of the result in the execution stage, and finally write it into the register indicated by dstE, and the other is the valM value obtained in the memory access stage, and finally write the register indicated by dstM. Therefore, what we need to push forward are valE and dstE in the execution, memory fetch, and write-back phases, and valM and dstM in the memory fetch and write-back phase. Combined with the hardware structure of the CPU, in order to achieve data forwarding, it is implemented as:

3.5.2 Branch prediction

The CPU needs to obtain the address of the instruction to be executed in the memory according to the PC value, and can perform operations such as decoding, execution, memory access, and writing

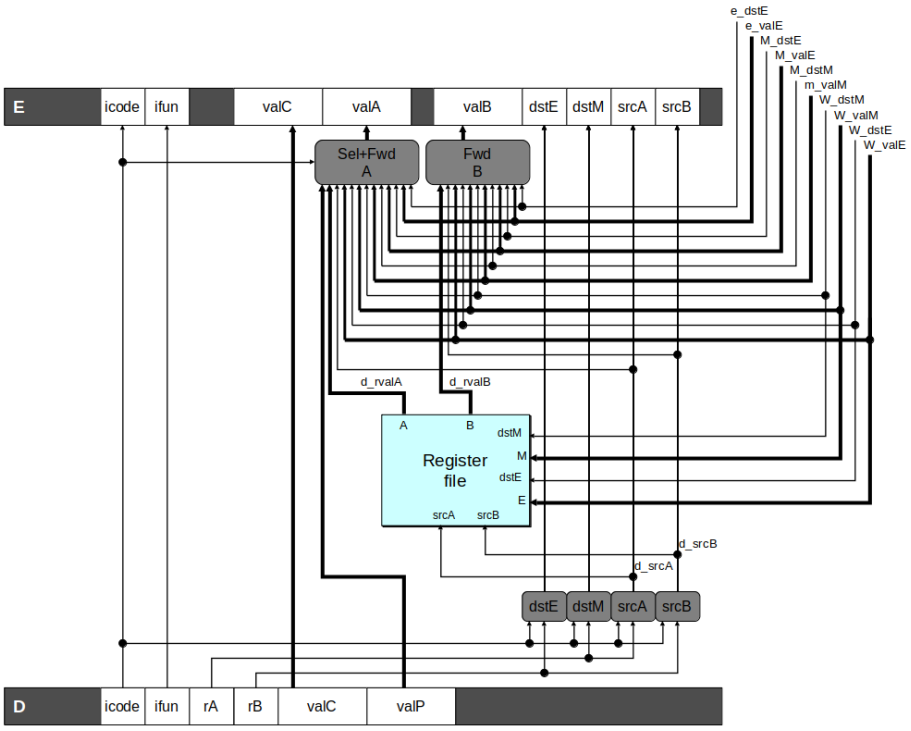

Figure (9) Data advance implementation

back after obtaining the instruction. If there is no need to implement the pipeline, the CPU can calculate the next PC value after implementing an instruction. But in order to realize the pipeline, it is necessary to get the PC value of the next instruction after reading one instruction, and it is also required to transfer the update PC phase from after the write-back to before the instruction fetch.

For instructions that do not change the PC value, this is still very easy to accomplish. You only need to get the length of the instruction through the instruction type and then get the next PC value from the current PC value.

But the three instructions that control the jump, jXX, call and ret may change the PC value of the program to realize the program jump. Among them, the jmp, call, and ret instructions will jump if they can be executed. Whether the instructions other than jmp in jXX jump is based on the CC flag. It may or may not jump.

For those that may or may not jump, we can certainly add bubbles to wait to determine whether the jump is confirmed and then update the PC value to get the next instruction, but this will waste several cycles in vain. We can predict whether the jump will be executed, and execute the next command based on the prediction. When you can decide whether to jump or not, if you find that the previous prediction is wrong and then return to the new position, if the previous prediction is correct, you will lose a few cycles in vain. The methods of prediction mainly include always predicting to be executed (AT), never predicting to be executed (NT), backward-jumping instructions that are predicted to be executed, and forward-jumping instructions that are predicted not to be executed (BTFNT). Statistical analysis found that the correct rates of these types of predictions were about 60

Because the call will definitely jump, and the jXX will definitely jump according to the prediction. Therefore, when updating the PC, the PC value of the next instruction is obtained by adding the current PC value to the length of the instruction in the fetch stage for normal instructions. valP, and the address of the updated PC value for the call and jXX instructions is the address represented by the immediate value in the change instruction.

In order to update the PC value as soon as it is known that the prediction result is incorrect,

it is necessary to use a method similar to data forwarding to transfer the Cnd value representing whether the current fetch state represents whether to jump to the selected PC, and perform the PC value according to the judgment Update.

3.5.3 Special case handling

The data forwarding and branch prediction mentioned above can solve some problems in the pipeline, but there are some special cases that cannot be dealt with using only these two methods.

Special circumstances include:

- **processing ret instruction** The ret instruction will definitely change the value of the PC, but the value to be changed can only be determined during the memory access stage. And jXX and call to change the value of PC can be determined in the fetch stage, which is difficult to deal with the ret command. At the same time, since the ret instruction will definitely jump, the instructions that follow do not need to be executed.
- **Data Adventure** Although some write-after-read problems can be solved by data pushing forward, there are still some situations that cannot be solved.
- **Branch with wrong prediction** The branch prediction technology is used above to ensure that those statements that are predicted to be executed after getting the wrong prediction result can be removed from the pipeline without affecting the program.

4 Instructions supported by the processor

- HALT
- NOP
- CMOVXX
- IRMOVQ
- RMMOVQ
- MRMOVQ
- OPQ
- JXX
- CALL
- RET
- PUSHQ
- POPQ

5 Finding the GCD of two nos

5.1 C++

```
// C++ program to find GCD of two numbers
#include <iostream>
using namespace std;
// Recursive function to return gcd of a and b
int gcd(int a, int b)
{
    // Everything divides 0
    if (a == 0)
       return b;
    if (b == 0)
       return a;

    // base case
    if (a == b)
        return a;

    // a is greater
    if (a > b)
        return gcd(a-b, b);
    return gcd(a, b-a);
}

// Driver program to test above function
int main()
{
    int a = 98, b = 108;
    cout<<"GCD of "<<a<<" and "<<b<<" is "<<gcd(a, b);
    return 0;
}
```

5.2 Assembly Code

```
irmovq F %rax 108
irmovq F %rbx 96
check:
rrmovq %rbx %rcx
subq %rax %rcx
jg swap
jl repsub
halt
repsub:
subq %rax %rbx
jmp check
swap
rrmovq %rax %rcx
rrmovq %rbx %rax
```

```
rrmovq %rcx %rbx
jmp repsub
```

5.3 Encoded Instructions

```
30 F0 00 00 00
00 00 00 00 6C
30 F3 00 00 00
00 00 00 00 62
20 31 61 01 76
00 00 00 00 00
00 00 37 72 00
00 00 00 00 00
00 2C 00 61 03
70 00 00 00 00
00 00 00 15 20
01 20 30 20 13
70 00 00 00 00
00 00 00 2C
```

5.4 GTKWAVE

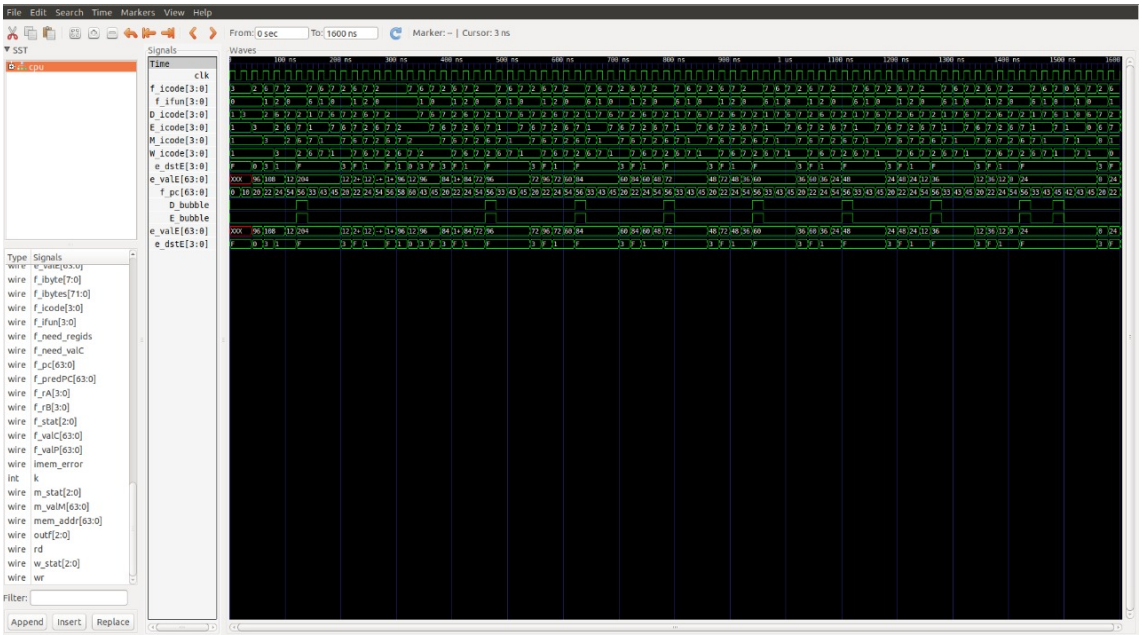

Figure (10) GTKWAVE OUTPUT OF HCF

6 HOW TO RUN

- run the file proc_tb.v
- to change code , enter the code in hexadecimal format in the file
- You can compute hcf of 98(62) and 108(6C) there
- Each module runs perfectly when run independently.