5 Processor Design

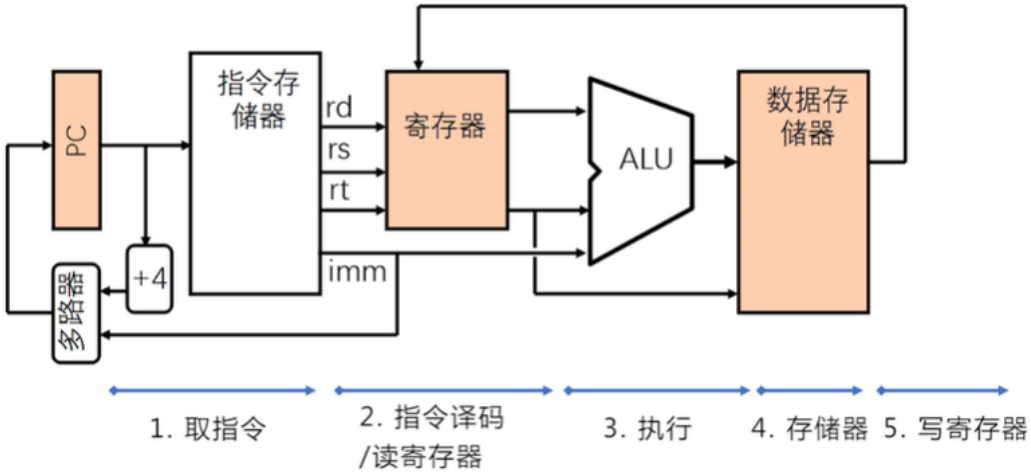
5.1 The Steps of Processor Design

1. Analyze the Instruction Set Architecture and get the requirements for the data path;
2. Select suitable components in the data path;
3. Connect the components to form data path;
4. Analyze the implementation of every instruction to get the control signals.
5. Collect control signals together and complete the control logic.

The Execution of An Instruction

- Instruction fetch;
- Instruction decode;
- Instruction Execute.

Therefore, our data path looks like the following picture.



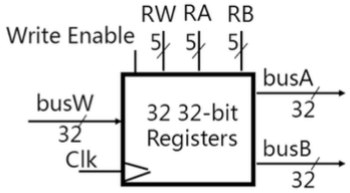
Single-Cycle Processor: Complete the execution of an instruction in a single clock period. In data path, the state change of some components need to be invoked by the clock edge (rising edge, falling edge), and the components are called state unit（状态单元）.

- Read the contents of state unit;
- Use circuits to implement the function of the instruction;
- Put the result into several state units.

The state unit is updated once in every period, and we need an extra explicit signal to tell the processor whether we should update the state unit. Only when the clock edge comes and the update signal is valid, we can update the state unit's data.

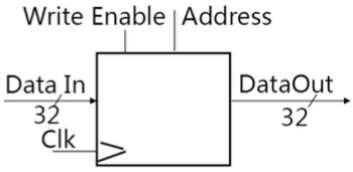
Here are some components in the data path.

- **State Unit: Register**

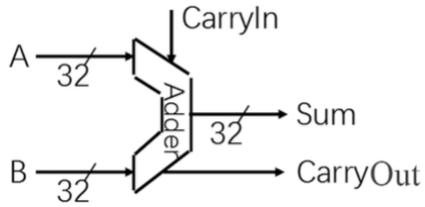


  - `CLK` is the clock's input;

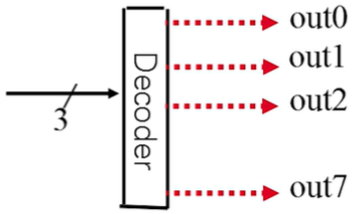
    - 32 registers. Two 32-bit output buses: `busA` and `busB`, and one 32-bit input but: `busW`.
    - Use `RA` and `RB` to control which register to read.
        - Input `RA` and `RB`;
        - Output `Reg[RA]` and `Reg[RB]` in `busA` and `busB` respectively.
    - Use `RW` to control which register to write (**sequential circuits, 时序电路**): Only when `CLK` comes to its edge and `Write Enable` is 1 (true), the data in `busW` can be written into `Reg[RW]`.
- **State Unit: Memory**



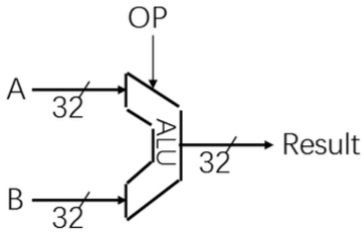
    - `CLK` is the clock's input;
    - A 32-bit input bus: `DataIn` and a 32-bit output bus: `DataOut`;
    - Use `Address` to control which memory unit to read: Input `Address` in bus `DataIn` and output `Mem[Address]` in bus `DataOut`;
    - Use `Address` to control which memory unit to write (**sequential circuits, 时序电路**): Only when `CLK` comes to its edge and `Write Enable` is 1 (true), the data in `DataIn` can be written in to `Mem[Address]`.
- **Adder**: To calculate `PC = PC+4` and so on.



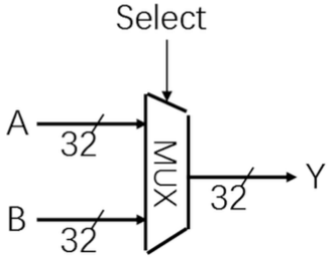
- **Decoder**: To select some specific data. The following figure is a 3 to 2^3 decoder.



- **ALU**: To implement the arithmetic / logic calculation.



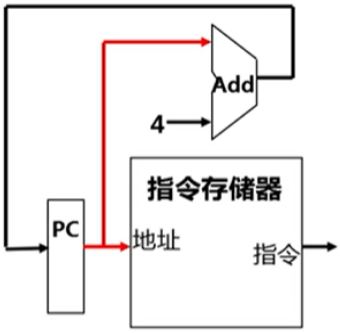
- **Selector**: To select data from several inputs. The following figure is a 2-way to 1-way selector.



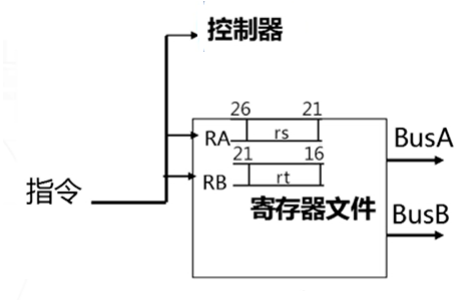
5.2 The Data Path of Processor

The following content shows the designing process of data path.

- **Instruction Module**: Read the instruction from instruction memory, then update `PC` to `PC + 4`, which is the next instruction sequentially. But `PC` does not update immediately, because the state update of `PC` is controlled by *clock edge*, so it needs `CLK` to invoke updating. Therefore, `PC` will update in the beginning of every clock cycle.

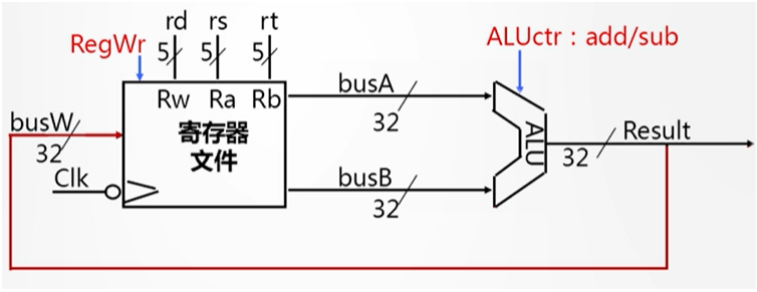


- **Register Module**
  - Put the `op` code and the `func` code in the instruction as the input to the controller.
  - Put the register address `rs` and `rt` in the instruction as the input to the register files to get the source operands in bus A and bus B.



- To support some of R-Type instructions, such as `add` and `sub`.

```
add rd, rs, rt   ; R[rd] <- R[rs] + R[rt]
```

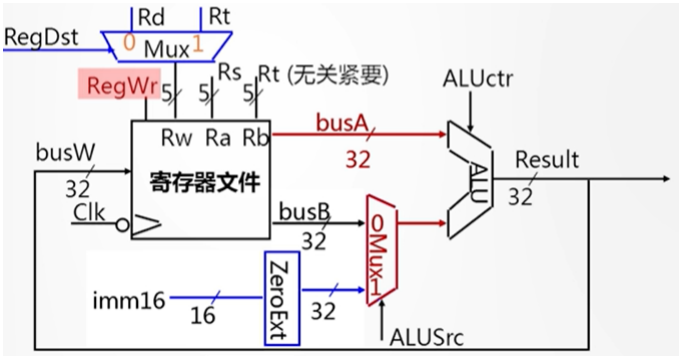


  - `ALUctr` used to control ALU to do different operations, and it is set based on the instruction function;
  - The result will be stored in register `Rw`. When the clock edge invokes the `CLK` pin in the register files and `RegWr = 1`, the result will be written in registers.

  Signals: `ALUctr = add/sub/...`, `RegWr = 1`.

- To support I-Type Instructions, such as `ori` and `andi`.

```
ori rt, rs, imm16    ; R[rt] <- R[rs] or ZeroExt[imm16]
```



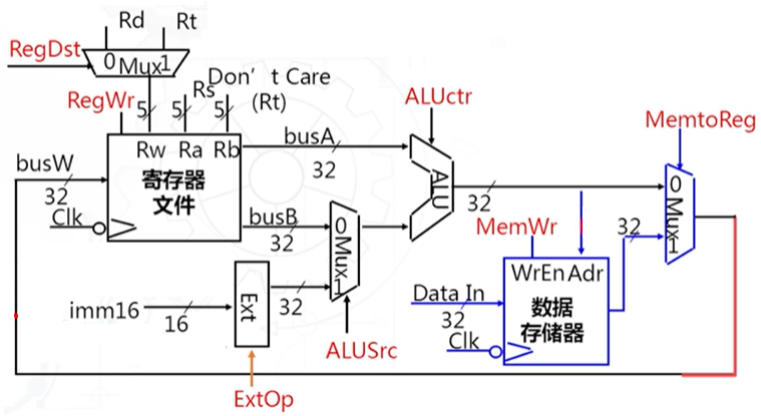
- Add two selectors to choose different output destinations. In this case we need to write result to register `Rt`, therefore, `RegDst = 1`;
- Add an zero-extension module and a selector in another input of ALU. In this case, we need another operand to be an immediate number, therefore `ALUSrc = 1`.

Signals: `ALUctr = and/or/not/...`, `RegWr = 1`, `RegDst = 1`, `ALUSrc = 1`.

- To support previous instructions, we only need to set `RegDst = 0`, `ALUSrc = 0`.

- To support load instruction like `lw`.

```
lw rt, rs, imm16    ; R[rt] <- M[R[rs] + SignExt[imm16]]
```



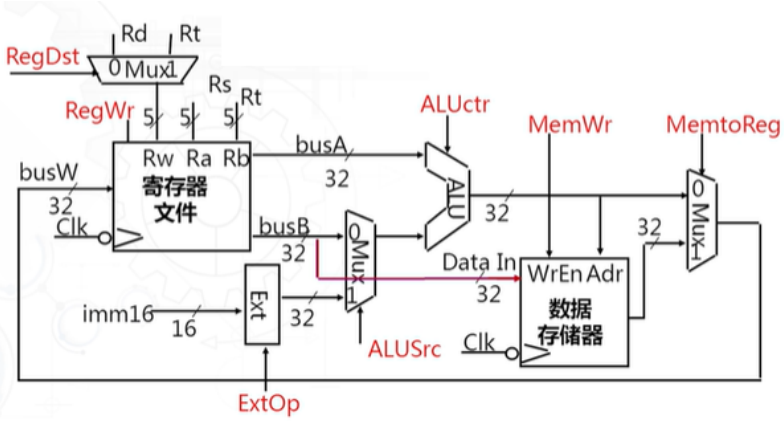
- Add an `ExtOp` in the extension module to represent zero-extension or sign-extension, where `ExtOp = 0` means zero-extension and `ExtOp = 1` means sign-extension. In this case we use `ExtOp = 1` to perform sign-extension;
- Add an memory module and set `MemWr = 0` since we only need to read memory. Add an selector to choose whether we use the result from ALU or from memory. Here we need result from memory, so `MemtoReg = 1`.

Signals: `ALUctr = add` (we need to perform addition to get the address of data), `RegWr = 1`, `RegDst = 1`, `ALUSrc = 1`, `ExtOp = 1`, `MemWr = 0`, `MemtoReg = 1`.

- To support previous instructions, we only need to set `ExtOp = 0`, `MemWr 0` and `MemtoReg = 0`.

- To support save instruction like `sw`.

```
sw rt, rs, imm16    ; M[R[rs] + SignExt[imm16]] <- R[rt]
```



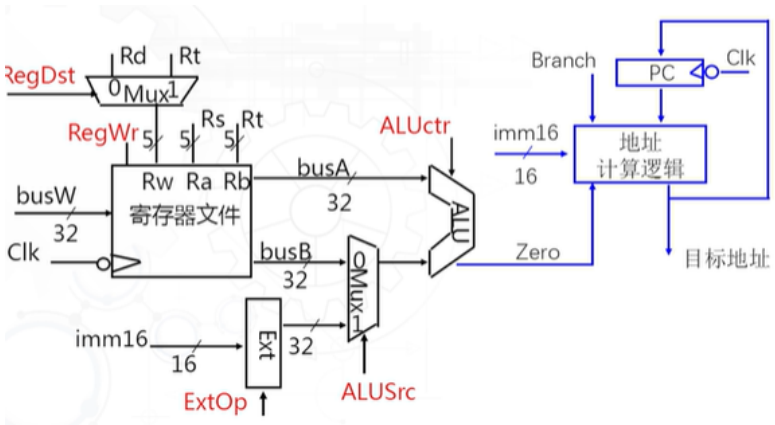
- Only need to link `DataIn` pin in data memory to `busB`. If we want to write memory, set `MemWr = 1` and we can complete the operation.

Signals: `ALUctr = add`, `ALUSrc = 1`, `ExtOp = 1`, `MemWr = 1`, `RegWr = 0`, `MemtoReg = x`, `RegDst = x`.

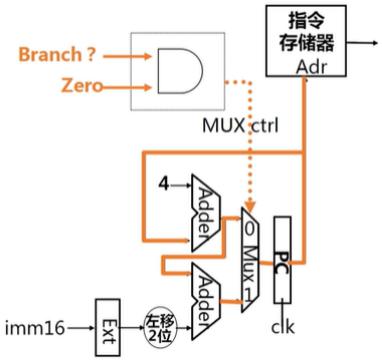
- It's obvious that we can support the previous instructions by setting `MemWr` to `0`.

- To support branch instructions like `beq`.

```
beq rs, rt, imm16  ; if R[rs] == R[rt] then PC <- PC + 4 + SignExt[imm16] *
4
                   ; else PC <- PC + 4
```



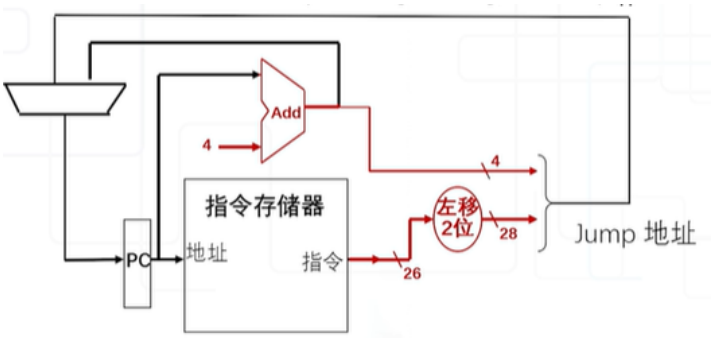
- Add a module to support `PC` operations.



Add an and-gate and a selector. Only when `R[Rs] == R[Rt]` (the result of `ALU` is zero) and the branch instruction is true, `MUX ctrl = 1`, we use the result `PC + 4 + SignExt[imm16]`, or we just use the result `PC + 4`.

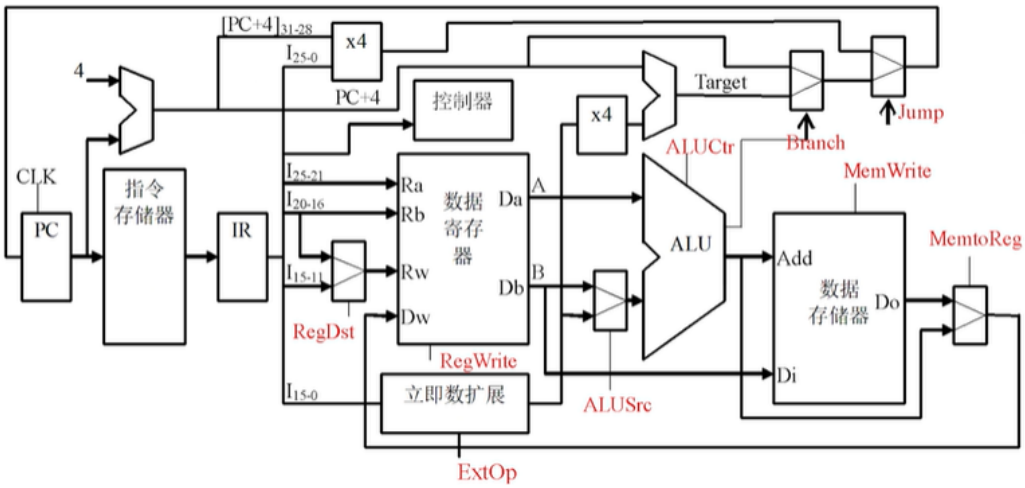
- Obviously, we add a module and do not change any of the original module, so we can support all the previous instructions.

- To support jump instruction: simple, only need to add an adder.



We don't need to change other module, therefore we can support all the previous instructions.

- **Summary**: Our Single-Cycle Data Path is as follows.



The controller set the signals `ALUSrc`, `RegDst`, `RegWrite`, `ExtOp`, `ALUCtr`, `Branch`, `Jump`, `MemWrite`, `MemtoReg`; and the data path can perform the correct operations.

Note: Separate the instruction memory with the data memory because we cannot perform two read operations from a memory in one cycle.

5.3 Control Signals

We need these control signals: `ALUSrc`, `RegDst`, `RegWrite`, `ExtOp`, `ALUCtr`, `Branch`, `Jump`, `MemWrite`, `MemtoReg`. (`... = x` means this signal does not matter)

ADD instruction: `Branch = 0`, `Jump = 0`, `ALUSrc = 0`, `ALUCtr = add`, `ExtOp = x`, `MemWrite = 0`, `MemtoReg = 0`, `RegDst = 1`, `RegWrite = 1`;

ORI instruction: `Branch = 0`, `Jump = 0`, `ALUSrc = 1`, `ALUCtr = or`, `ExtOp = 0`, `MemWrite = 0`, `MemtoReg = 0`, `RegDst = 0`, `RegWrite = 1`;

LOAD instruction: `Brach = 0`, `Jump = 0`, `ALUSrc = 1`, `ALUCtr = add`, `ExtOp = 1`, `MemWrite = 0`, `MemtoReg = 1`, `RegDst = 0`, `RegWrite = 1`;

STORE instruction: `Brach = 0`, `Jump = 0`, `ALUSrc = 1`, `ALUCtr = add`, `ExtOp = 1`, `MemWrite = 1`, `MemtoReg = x`, `RegDst = x`, `RegWrite = 0`;

BEQ instruction: `Branch = 1`, `Jump = 0`, `ALUSrc = 0`, `ALUCtr = sub`, `ExtOp = 1`, `MemWrite = 0`, `MemtoReg = x`, `RegDst = x`, `RegWrite = 0`;

JUMP instruction: `Branch = 0`, `Jump = 1`, `MemWrite = 0`, `RegWrite = 0`, other signals do not matter.

Here is a brief introduction to the control signals we have discussed about (in Chinese).

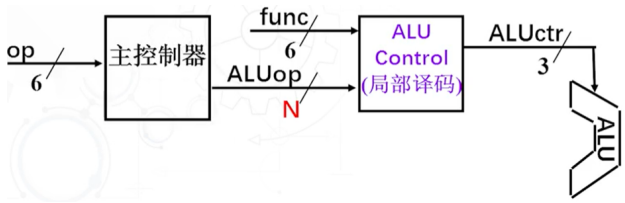
信号名称	无效(=0)时的作用	有效(=1)时的作用
RegDst	写寄存器在寄存器文件中的目标寄存器的编号来自于rt字段	写寄存器在寄存器文件中的目标寄存器的编号来自于rd字段
RegWrite	无	写数据输入的值，写入到目标寄存器编号（rt或者rd）选择的通用寄存器
ALUSrc	ALU的第二个操作数来自于寄存器	ALU的第二个操作数来自于立即数
MemWrite	无	将地址输入指定位置的存储器内容替换为写数据输入的值
MemtoReg	送往寄存器文件写数据输入的值来自于ALU的输出	送往寄存器文件写数据输入的值来自于数据存储器的输出
ExtOP	16位立即数零扩展到32位	16位立即数带符号扩展到32位
Branch	选择PC+4作为PC输入端	是一条应该转移的分支指令，
Jump	不选择Jump目标地址，而是选择PC+4或者条件转移目标地址，作为PC输入端	是一条应该转移的分支指令，选择Jump目标地址作为PC输入端

5.4 Control Logic

Design a controller with an input of a 32-bit instruction and output of 9 control signals above.

Design `ALUCtr` signals to be an 3-bit signals, representing different operations.

Use two-level decode to extract the `ALUCtr` signal from the original instructions.



Therefore, the main controller only need to generate the other signals, and ALU controller only need to generate `ALUCtr` signals.

Encode `ALUop` (3-bit)

	R-type	ori	lw	sw	beq	jump
ALUop (Symbolic)	“R-type”	Or	Add	Add	Subtr	xxx
ALUop<2:0>	1 xx	0 10	0 00	0 00	0x1	xxx

In this processor, 2-bit is enough, but in real MIPS processor, we need 3-bit. Therefore, we continue to use 3-bit `ALUop` here.

The truth table of `ALUop`, `func` and `ALUctr`

ALUop			func				ALU 运算操作	ALUctr		
bit2	bit1	bit0	bit<3>	bit<2>	bit<1>	bit<0>		bit<2>	bit<1>	bit<0>
0	0	0	x	x	x	x	Add	0	0	0
0	x	1	x	x	x	x	Subtract	0	0	1
0	1	0	x	x	x	x	Or	1	1	0
1	x	x	0	0	0	0	Add	0	0	0
1	x	x	0	0	1	0	Subtract	0	0	1
1	x	x	0	1	0	0	And	0	1	0
1	x	x	0	1	0	1	Or	1	1	0
1	x	x	1	0	1	0	Subtract	0	0	1

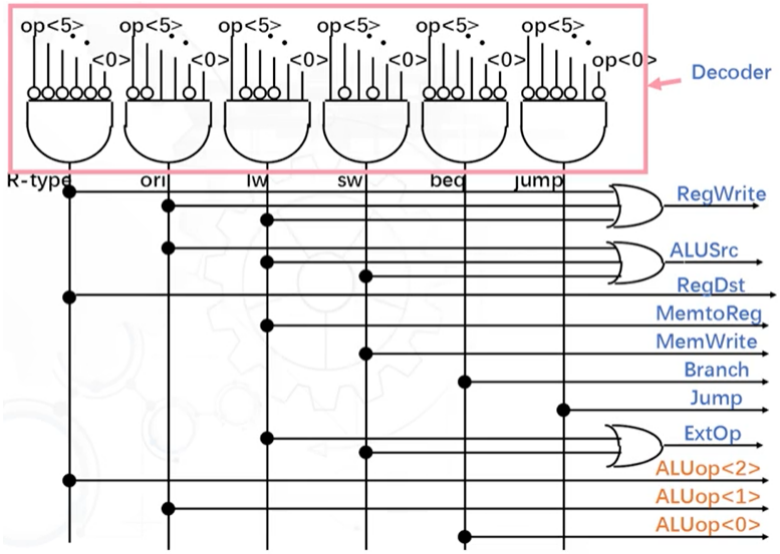
Analyze the truth table and then we can find out the expressions of `ALUctr` signal.

The truth table of main controller

op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop (符号)	"R-型"	Or	Add	Add	Subtr	xxx
ALUop <2>	1	0	0	0	0	x
ALUop <1>	x	1	0	0	x	x
ALUop <0>	x	0	0	0	1	x

Analyze the truth table and then we can find out the expressions of all the signals.

The design of the main controller is as follows.



5.5 Multi-Cycle Processor

The Pros and Cons of Single-Cycle Processor

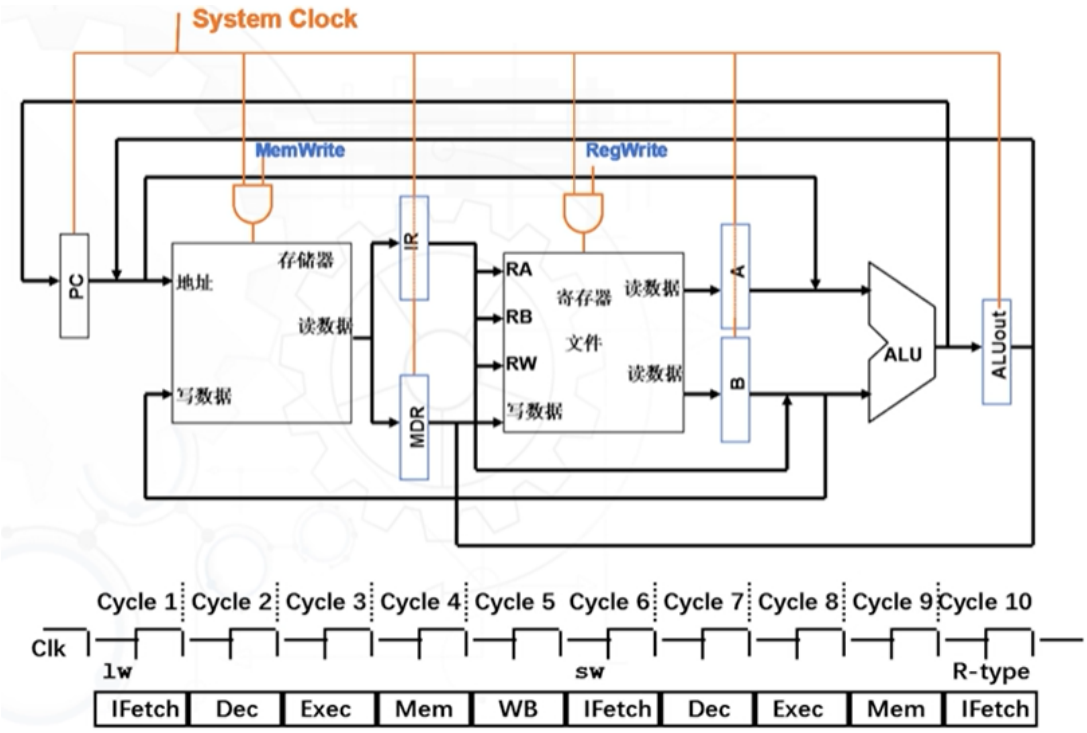
- The cycle is restricted to the slowest instruction. (waste time).

- Some module can only be accessed once in one cycle (such as ALU, adder), so we need multiple modules with the same function. (waste resources).
- But it is simple and easy to understand.

Solution: Multi-Cycle

- Every instruction takes different number of cycles;
- The time of a cycle becomes shorter. (save time).
- One module can be used in two different cycles, do not need to add multiple modules with the same function. (save resources).

The Multi-Cycle Design



Note: the execution time of `lw` instruction in multi-cycle processor is actually longer than the execution time in single-cycle processor, because we add new registers and some new designs.

Instruction Cycle

- **Clock Cycle**（时钟周期）: T, the smallest unit of time in the computer.
- **Machine Cycle**（机器周期）: the shortest time to read an instruction from memory.
- **Instruction Cycle**（指令周期）: the total time of executing an instruction, including instruction fetch (from memory), instruction decode and instruction execution and etc.
- An instruction cycle consists of several machine cycles. A machine cycle consists of several clock cycles.

5.6 Different Methods of Controller

Hard-wired controller（硬连线控制器）: The logic expression of each signal is like:

$$C_i = T_1 \times (Ins_{1,1} + Ins_{1,2} + \dots) + T_2 \times (Ins_{2,1} + Ins_{2,2} + \dots) + \dots$$

Here, $T_1, T_2, \dots$ represent different stages, and $Ins_{i,1}, Ins_{i,2}, \dots$ represent the instructions that use this signal in the stage T_i.

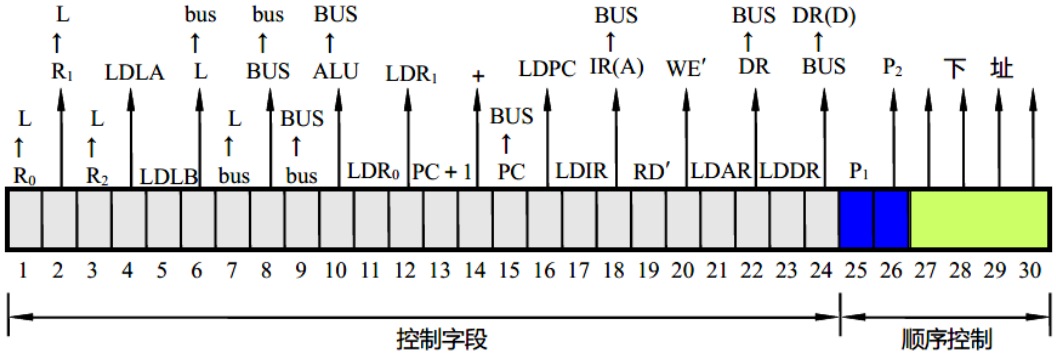
Features:

- Complex circuit network;
- No rules;

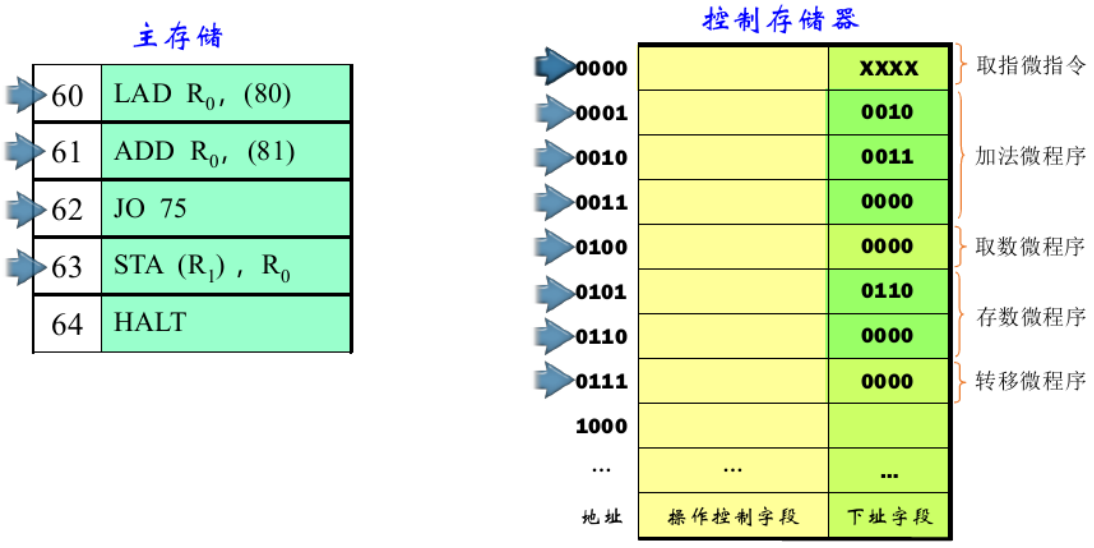
- Difficult to design an debug;
- Cannot change the instruction set and instruction function;
- Often used in VLSI (Very Large Scale Integration);
- Fast speed.

Microprogramming control（微程序控制）: Use memory to store the micro-instructions (control signals).

- The execution of an instruction contains a series of micro-operations, and represent the control signals in some micro-instructions.
- Execute a micro-instruction will give a set of control signals of micro-operations
- Execute an instruction is executing a micro-program containing several micro-instructions.
- Every bit of micro-instruction is a control signal and has a separate line.



- Store the micro-program in the control memory, and execute the micro-instructions in the micro-program when executing the instruction.



Features

- Need extra memory;
- Be able to represent complexer instructions without changing the data path (cheap).

Evolution

- 1980s, micro-programming plays an important role in controller design;
- Nowadays, micro-programming is an auxiliary method in modern micro-processor.
  - Most of instructions use the hard-wired controller methods;
  - Uncommon-used instructions and complex instructions are still controlled by micro-programming methods.

- Nowadays, still needs debugging of chips (patches in bootup process).