CS:APP Chapter 4 Computer Architecture Sequential Implementation 顺序实现

任课教师：

宿红毅　　张艳　　　黎有琦　　　　李秀星

原作者：

Randal E. **Bryant and** David R. O'Hallaron

Carnegie Mellon University

Y86-64 Instruction Set 指令集#1

字节 Byte	0	1	2	3	4	5	6	7	8	9
`halt`	0 0									
`nop`	1 0									
`cmovXX` rA, rB	2 fn	rA rB								
`irmovq` V, rB	3 0	F rB	V							
`rmmovq` rA, D(rB)	4 0	rA rB	D							
`mrmovq` D(rB), rA	5 0	rA rB	D							
`OPq` rA, rB	6 fn	rA rB								
`jXX` Dest	7 fn	Dest								
`call` Dest	8 0	Dest								
`ret`	9 0									
`pushq` rA	A 0	rA F								
`popq` rA	B 0	rA F								

Y86-64 Instruction Set 指令集#2

字节 Byte	0		1		2	3	4	5	6
halt	0	0							
nop	1	0							
cmovXX rA, rB	2	fn	rA	rB					
irmovq V, rB	3	0	F	rB	V				
rmmovq rA, D(rB)	4	0	rA	rB	D				
mrmovq D(rB), rA	5	0	rA	rB	D				
OPq rA, rB	6	fn	rA	rB					
jXX Dest	7	fn	Dest						
call Dest	8	0	Dest						
ret	9	0							
pushq rA	A	0	rA	F					
popq rA	B	0	rA	F					

rrmovq	7	0
cmovle	7	1
cmovl	7	2
cmove	7	3
cmovne	7	4
cmovge	7	5
cmovg	7	6

Y86-64 Instruction Set 指令集#3

字节 Byte	0	1	2	3	4	5	6	7	8	9
`halt`	0 0									
`nop`	1 0									
`cmovXX` rA, rB	2 fn	rA rB								
`irmovq` V, rB	3 0	F rB	V							
`rmmovq` rA, D(rB)	4 0	rA rB	D							
`mrmovq` D(rB), rA	5 0	rA rB	D							
`OPq` rA, rB	6 fn	rA rB								
`jXX` Dest	7 fn	Dest								
`call` Dest	8 0	Dest								
`ret`	9 0									
`pushq` rA	A 0	rA F								
`popq` rA	B 0	rA F								

`addq`	6 0
`subq`	6 1
`andq`	6 2
`xorq`	6 3

Y86-64 Instruction Set 指令集#4

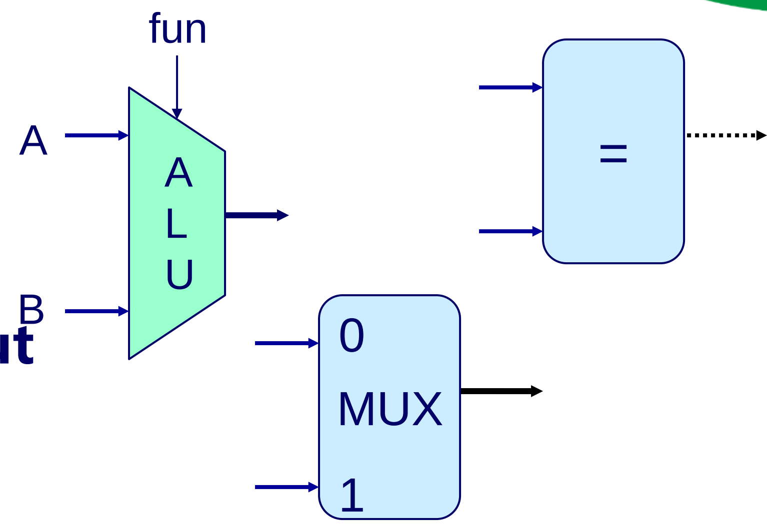
字节 Byte	0	1	2	3	4	5	6	7
`halt`	0 0							
`nop`	1 0							
`cmovXX rA, rB`	2 fn	rA rB						
`irmovq V, rB`	3 0	F rB	V					
`rmmovq rA, D(rB)`	4 0	rA rB	D					
`mrmovq D(rB), rA`	5 0	rA rB	D					
`OPq rA, rB`	6 fn	rA rB						
`jXX Dest`	7 fn	Dest						
`call Dest`	8 0	Dest						
`ret`	9 0							
`pushq rA`	A 0	rA F						
`popq rA`	B 0	rA F						

`jmp`	7 0
`jle`	7 1
`jl`	7 2
`je`	7 3
`jne`	7 4
`jge`	7 5
`jg`	7 6

构建块 Building Blocks

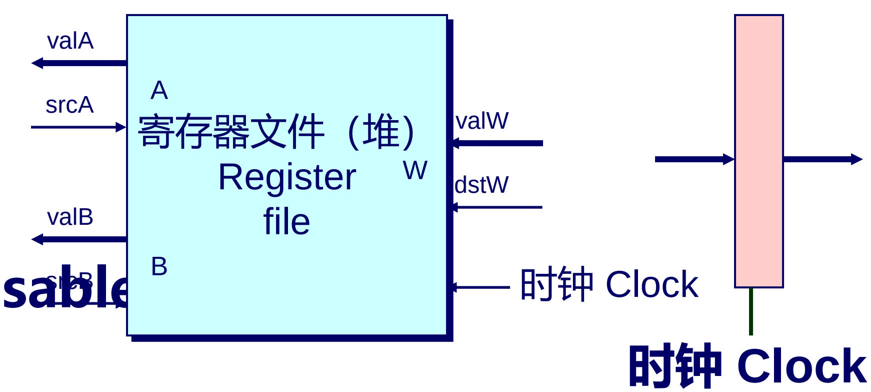
组合逻辑 Combinational Logic

- 计算输入的布尔函数 Compute Boolean functions of inputs
- 连续响应输入的变化 Continuously respond to input changes
- 对数据进行操作并实现控制 Operate on data and implement control



存储元素 Storage Elements

- 存储若干位 Store bits
- 可寻址的内存 Addressable memories
- 非寻址的寄存器 Non-addressable registers
- 仅在时钟上升时装载 Loaded only as clock rises



硬件控制语言 Hardware Control Language

- 非常简单的硬件描述语言 Very simple hardware description language
- 仅可以表达有限的硬件操作 Can only express limited aspects of hardware operation
  - 我们想要探索和修改部分 Parts we want to explore and modify

数据类型 Data Types

- `bool`: Boolean
  - `a`, `b`, `c`, …
- `int`: words
  - `A`, `B`, `C`, …
  - 不指定字长-字节还是32位字，。。。 Does not specify word size---bytes, 32-bit words, …

语句 Statements

- `bool a =` *`bool-expr`* `;`
- `int A =` *`int-expr`* `;`

HCL操作 HCL Operations

- 按照返回值类型进行分类 Classify by type of value returned

布尔表达式 Boolean Expressions

- 逻辑操作 Logic Operations
  - `a && b, a || b, !a`
- 字比较 Word Comparisons
  - `A == B, A != B, A < B, A <= B, A >= B, A > B`
- 集合成员关系 Set Membership
  - `A in { B, C, D }`
    - » 等同于 Same as `A == B || A == C || A == D`

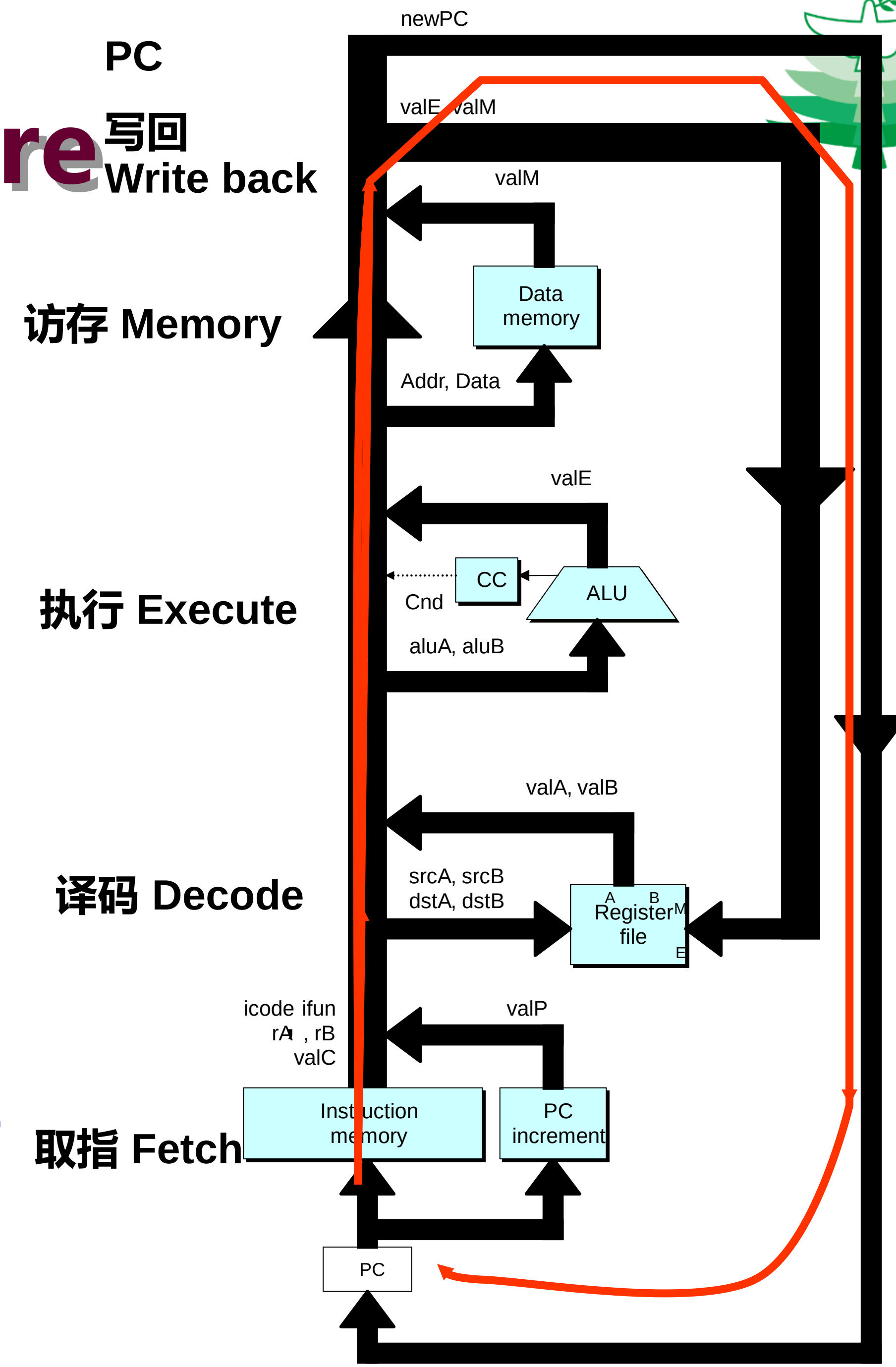
字表达式 Word Expressions

- Case表达式 Case expressions
  - `[ a : A; b : B; c : C ]`
  - 按顺序评估测试表达式 Evaluate test expressions `a`, `b`, `c`, … in sequence
  - 返回第一个成功测试的字表达式 Return word expression `A`, `B`, `C`, … for first successful test

顺序硬件结构 SEQ Hardware Structure

状态 State

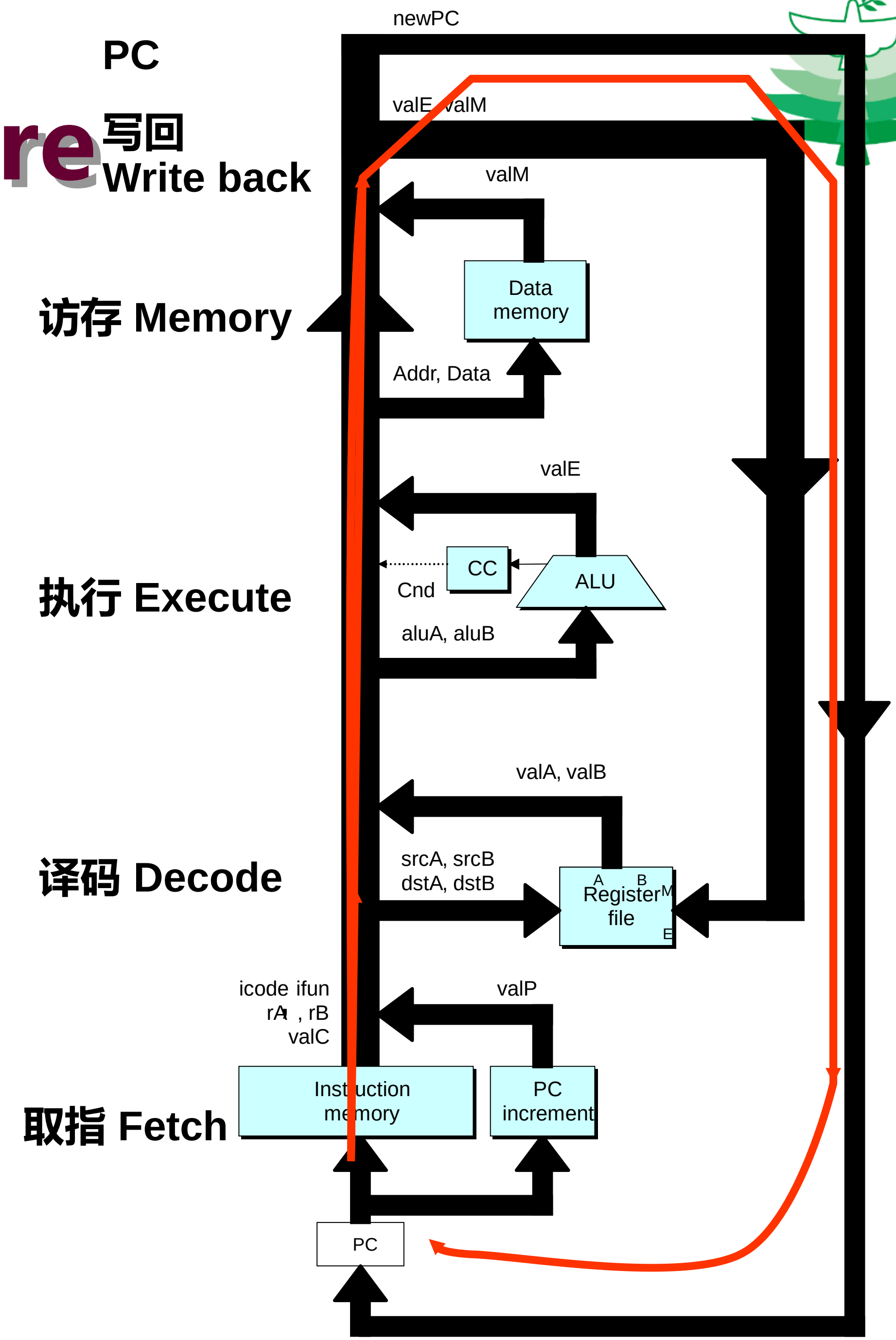
- 程序计数器寄存器 Program counter register (PC)
- 条件码寄存器 Condition code register (CC)
- 寄存器文件（堆） Register File
- 内存 Memories
  - 访问同样的内存空间 Access same memory space
  - 数据：读/写程序数据 Data: for reading/writing program data
  - 指令：读指令 Instruction: for reading instructions



顺序硬件结构 SEQ Hardware Structure

指令流 Instruction Flow

- 读PC指定地址处的指令 Read instruction at address specified by PC
- 分成阶段处理 Process through stages
- 更新程序计数器 Update program counter



顺序阶段 SEQ Stages

取指 Fetch

- 从指令内存读指令 Read instruction from instruction memory

译码 Decode

- 读程序寄存器 Read program registers

执行 Execute

- 计算值或地址 Compute value or address

访存 Memory

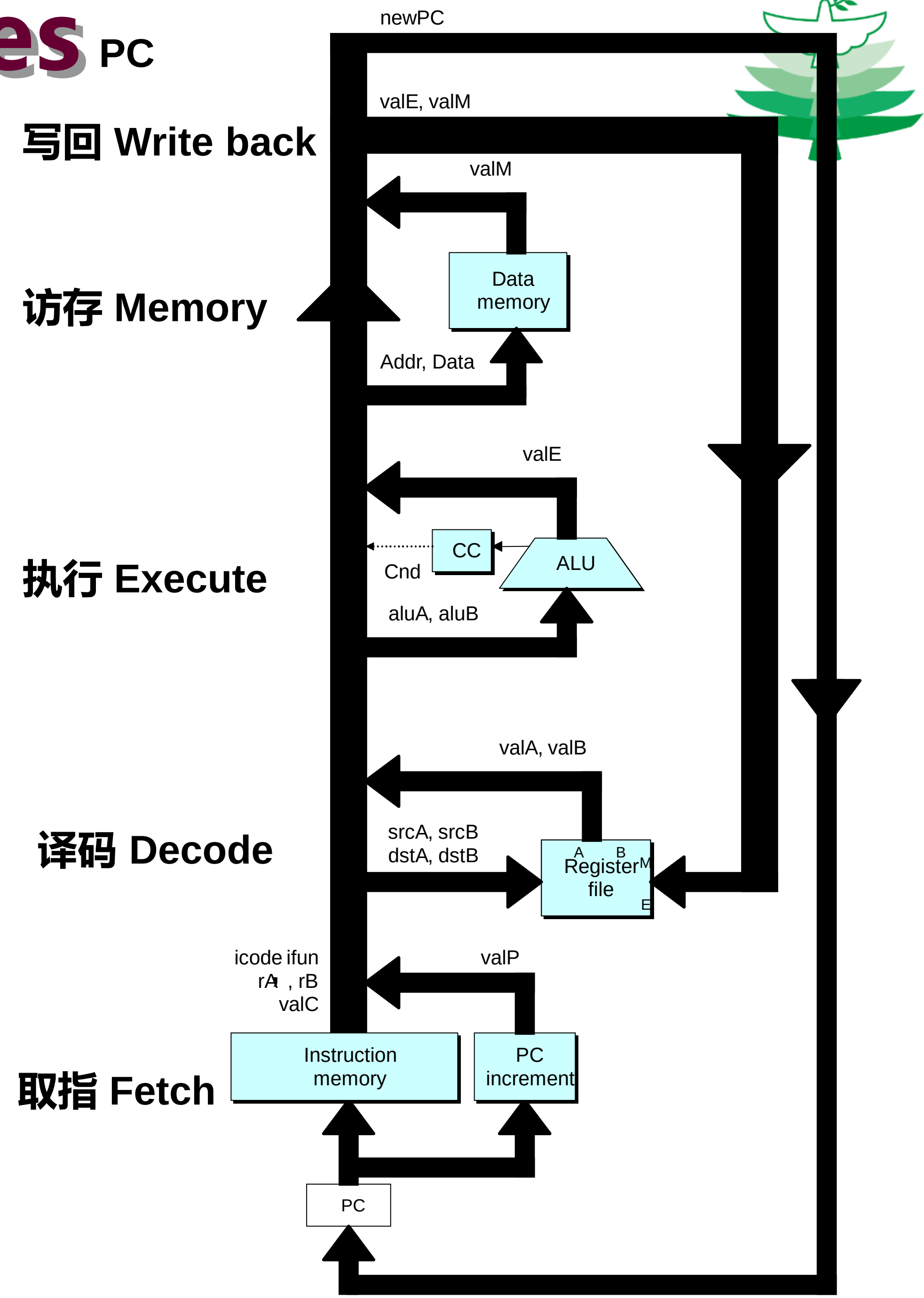
- 读或写数据 Read or write data

写回 Write Back

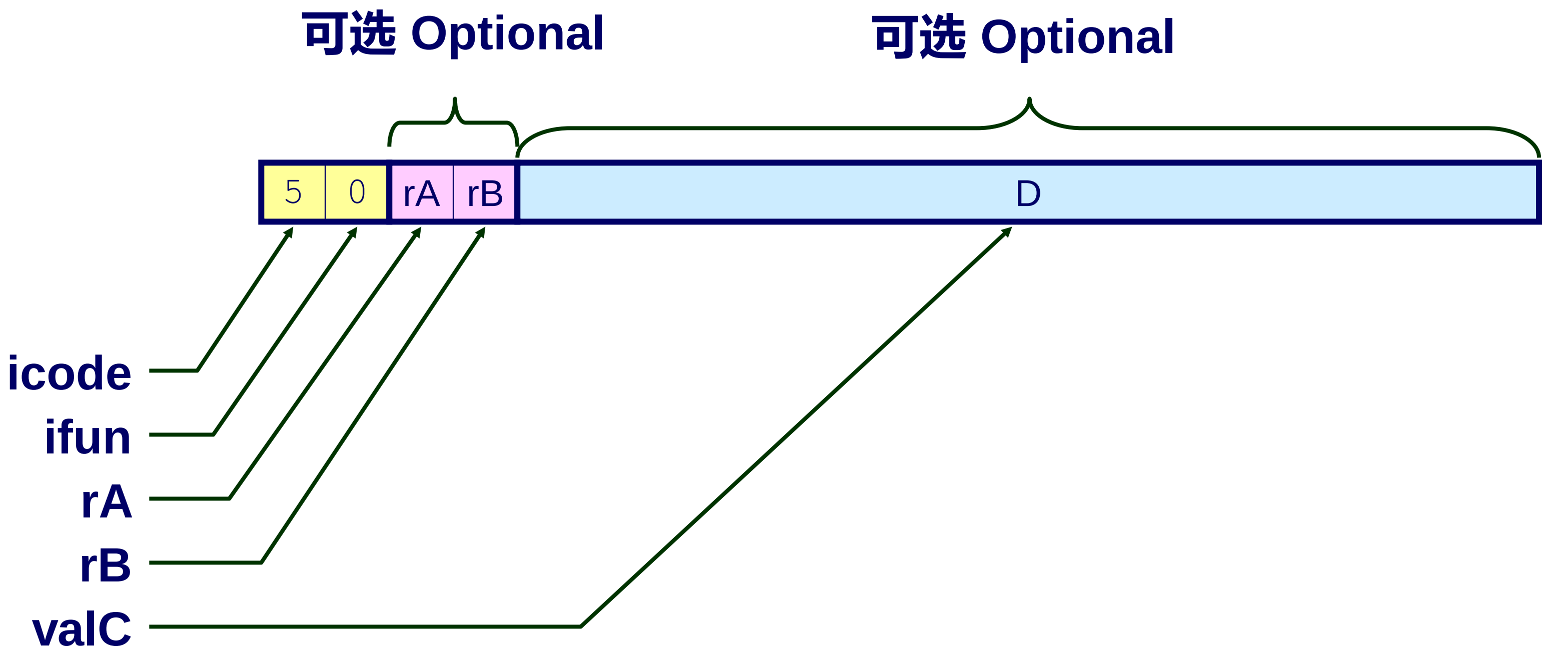
- 写程序寄存器 Write program registers

PC

- 更新程序计数器 Update



指令译码 Instruction Decoding



指令格式 Instruction Format

- 指令字节 Instruction byte
  icode:ifun
- 可选的寄存器字节 Optional register byte rA:rB
- 可选的常量字 Optional constant word valC

执行算术/逻辑操作
Executing Arith./Logical Operation

```
OPq rA, rB
```

6	fn	rA	rB

取指 Fetch
- 读2个字节 Read 2 bytes

译码 Decode
- 读操作数寄存器 Read operand registers

执行 Execute
- 执行运算 Perform operation
- 设置条件码 Set condition codes

访存 Memory
- 无操作 Do nothing

写回 Write back
- 更新寄存器 Update register

PC更新 PC Update
- PC加2 Increment PC by 2

每个阶段的计算：算/逻操作
Stage Computation: Arith/Log. Ops

	OPq rA, rB	
取指 Fetch	icode:ifun ← M_1[PC]	读指令字节 Read instruction byte
	rA:rB ← M_1[PC+1]	读寄存器字节 Read register byte
	valP ← PC+2	计算下一个PC Compute next PC
译码 Decode	valA ← R[rA]	读操作数A Read operand A
	valB ← R[rB]	读操作数B Read operand B
执行 Execute	valE ← valB OP valA	执行ALU操作 Perform ALU operation
	Set CC	设置条件码寄存器 Set condition code register
访存 Memory		
写回 Write back	R[rB] ← valE	写回结果 Write back result
PC更新 PC update	PC ← valP	更新PC Update PC

- 指令的执行公式化为一系列简单的步骤 Formulate instruction execution as sequence of simple steps
- 对所有的指令使用同样的通用形式 Use same general form for all instructions

执行传送类指令 Executing `rmmovq`

`rmmovq rA, D(rB)`	4	0	rA	rB	D

取指 Fetch

- 读10个字节 Read 10 bytes

译码 Decode

- 读操作数寄存器 Read operand registers

执行 Execute

- 计算有效地址 Compute effective address

访存 Memory

- 写入内存 Write to memory

写回 Write back

- 无操作 Do nothing

PC更新 PC Update

- PC增加10 Increment PC by 10

每个阶段的计算：传送类指令
Stage Computation: rmmovq

	rmmovq rA, D(rB)	
Fetch	icode:ifun $\leftarrow$ M_1[PC]	Read instruction byte
	rA:rB $\leftarrow$ M_1[PC+1]	Read register byte
	valC $\leftarrow$ M_8[PC+2]	读变址值 Read displacement D
	valP $\leftarrow$ PC+10	Compute next PC
Decode	valA $\leftarrow$ R[rA]	Read operand A
	valB $\leftarrow$ R[rB]	Read operand B
Execute	valE $\leftarrow$ valB + valC	Compute effective address
Memory	M_8[valE] $\leftarrow$ valA	Write value to memory
Write back		
PC update	PC $\leftarrow$ valP	Update PC

- 使用ALU进行地址计算 Use ALU for address computation

执行出栈指令 Executing popq

popq rA	b	0	rA	F

取指 Fetch
- 读2个字节 Read 2 bytes

译码 Decode
- 读栈指针 Read stack pointer

执行 Execute
- 栈指针增加8 Increment stack pointer by 8

访存 Memory
- 从老的栈指针读 Read from old stack pointer

写回 Write back
- 更新栈指针 Update stack pointer
- 写结果到寄存器 Write result to register

PC更新 PC Update
- PC增加2 Increment PC by 2

每个阶段的计算：出栈
Stage Computation: popq

	popq rA	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	rA:rB ← M_1[PC+1]	Read register byte
	valP ← PC+2	Compute next PC
Decode	valA ← R[%rsp]	Read stack pointer
	valB ← R[%rsp]	Read stack pointer
Execute	valE ← valB + 8	Increment stack pointer
Memory	valM ← M_8[valA]	Read from stack
Write back	R[%rsp] ← valE	Update stack pointer
	R[rA] ← valM	Write back result
PC update	PC ← valP	Update PC

- 使用ALU增加栈指针 Use ALU to increment stack pointer
- 必须更新两个寄存器 Must update two registers
  - 弹出的值 Popped value
  - 新的栈指针 New stack pointer

执行条件传送 Executing Conditional Moves

cmovXX rA, rB	2	fn	rA	rB

取指 Fetch
- 读2个字节 Read 2 bytes

译码 Decode
- 读操作数寄存器 Read operand registers

执行 Execute
- 如果没有设置条件码，则将目的寄存器设置为0xF If !cnd, then set destination register to 0xF

访存 Memory
- 无操作 Do nothing

写回 Write back
- 更新寄存器（或不更新）Update register (or not)

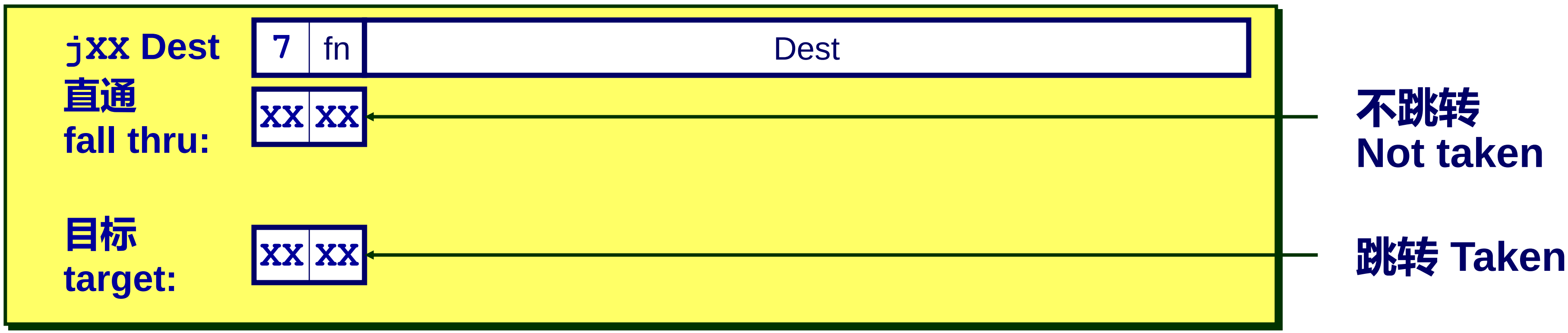
PC更新 PC Update
- PC增加2 Increment PC by 2

每个阶段的计算：条件传送
Stage Computation: Cond. Move

	cmovXX rA, rB	
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	Read instruction byte Read register byte Compute next PC
Decode	valA ← R[rA] valB ← 0	Read operand A
Execute	valE ← valB + valA If ! Cond(CC,ifun) rB ← 0xF	Pass valA through ALU (Disable register update)
Memory		
Write back	R[rB] ← valE	Write back result
PC update	PC ← valP	Update PC

- 读寄存器rA并通过ALU Read register rA and pass through ALU
- 取消传送通过设置目的寄存器为0xF Cancel move by setting destination register to 0xF
  - 如果条件码与传送条件指明不需传送 If condition codes & move condition indicate no move

执行跳转类指令 Executing Jumps



取指 Fetch

- 读9个字节 Read 9 bytes
- PC增加9 Increment PC by 9

译码 Decode

- 无操作 Do nothing

执行 Execute

- 根据跳转条件和条件码确定是否进行分支转移 Determine whether to take branch based on jump condition and condition codes

访存 Memory

- 无操作 Do nothing

写回 Write back

- 无操作 Do nothing

PC更新 PC Update

- 如果选择分支设置PC为目标地址，或者如果不选择分支则增加PC Set PC to Dest if branch taken or to incremented PC if not branch

每个阶段的计算：跳转 Stage Computation: Jumps

	jXX Dest	
Fetch	icode:ifun $\leftarrow$ M_1[PC]	Read instruction byte
	valC $\leftarrow$ M_8[PC+1]	Read destination address
	valP $\leftarrow$ PC+9	下一条指令地址 Fall through address
Decode		
Execute	Cnd $\leftarrow$ Cond(CC,ifun)	Take branch?
Memory		
Write back		
PC update	PC $\leftarrow$ Cnd ? valC : valP	Update PC

- 计算跳转和不跳转两个地址 Compute both addresses
- 根据条件码的设置和分支条件选择一个地址 Choose based on setting of condition codes and branch condition

执行过程调用 Executing `call`

`call` Dest	8	0	Dest
返回地址 return:	xx	xx	
目标地址 target:	xx	xx	

取指 Fetch

- 读9个字节 Read 9 bytes
- PC增加9 Increment PC by 9

译码 Decode

- 读栈指针 Read stack pointer

执行 Execute

- 栈指针减8 Decrement stack pointer by 8

访存 Memory

- 将增加后的PC值写到栈指针新值 Write incremented PC to new value of stack pointer

写回 Write back

- 更新栈指针 Update stack pointer

PC更新 PC Update

- 设置PC为目标地址 Set PC to Dest

每个阶段的计算：过程调用
Stage Computation: `call`

	`call` Dest	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	valC ← M_8[PC+1]	Read destination address
	valP ← PC+9	Compute return point
Decode	valB ← R[`%rsp`]	Read stack pointer
Execute	valE ← valB + –8	Decrement stack pointer
Memory	M_8[valE] ← valP	Write return value on stack
Write back	R[`%rsp`] ← valE	Update stack pointer
PC update	PC ← valC	Set PC to destination

- 使用ALU减栈指针 Use ALU to decrement stack pointer
- 存储增加后的PC Store incremented PC

执行过程返回 Executing `ret`

```
ret                  | 9 | 0 |

返回地址 return:      | xx | xx |
```

取指 Fetch
- 读1个字节 Read 1 byte

译码 Decode
- 读栈指针 Read stack pointer

执行 Execute
- 栈指针增加8 Increment stack pointer by 8

访存 Memory
- 从老的栈指针读返回地址 Read return address from old stack pointer

写回 Write back
- 更新栈指针 Update stack pointer

PC更新 PC Update
- 设置PC为返回地址 Set PC to return address

每个阶段的计算：过程返回 Stage Computation: `ret`

	`ret`	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
Decode	valA ← R[`%rsp`]	Read operand stack pointer
	valB ← R[`%rsp`]	Read operand stack pointer
Execute	valE ← valB + 8	Increment stack pointer
Memory	valM ← M_8[valA]	Read return address
Write back	R[`%rsp`] ← valE	Update stack pointer
PC update	PC ← valM	Set PC to return address

- 使用ALU增加栈指针 Use ALU to increment stack pointer
- 从内存读返回地址 Read return address from memory

计算步骤 Computation Steps

		OPq rA, rB	
取指 Fetch	icode,ifun	icode:ifun ← M_1[PC]	读指令字节 Read instruction byte
	rA,rB	rA:rB ← M_1[PC+1]	读寄存器字节 Read register byte
	valC		读常量字 [Read constant word]
	valP	valP ← PC+2	计算下一个PC Compute next PC
译码 Decode	valA, srcA	valA ← R[rA]	读操作数A Read operand A
	valB, srcB	valB ← R[rB]	读操作数B Read operand B
执行 Execute	valE	valE ← valB OP valA	执行ALU操作 Perform ALU operation
	Cond code	Set CC	设置/使用条件码寄存器Set/use cond. code reg
访存 Memory	valM		内存读/写[Memory read/write]
写回 Write back	dstE	R[rB] ← valE	写回ALU结果 Write back ALU result
	dstM		写回内存结果 [Write back memory result]
更新 PC update	PC	PC ← valP	更新PC Update PC

- 所有指令遵循同样的通用模式 All instructions follow same general pattern
- 差别在于每步计算什么 Differ in what gets computed on each step

计算步骤 Computation Steps

		`call` Dest	
取指 Fetch	icode,ifun	icode:ifun ← M_1[PC]	读指令字节 Read instruction byte
	rA,rB		读寄存器字节 [Read register byte]
	valC	valC ← M_8[PC+1]	读常量字 Read constant word
	valP	valP ← PC+9	计算下一个PC Compute next PC
译码 Decode	valA, srcA		读操作数A [Read operand A]
	valB, srcB	valB ← R[`%rsp`]	读操作数B Read operand B
执行 Execute	valE	valE ← valB + –8	执行ALU操作 Perform ALU operation
	Cond code		设置/使用条件码寄存器 [Set /use cond. code reg]
访存 Memory	valM	M_8[valE] ← valP	内存读/写 Memory read/write
写回 Write back	dstE	R[`%rsp`] ← valE	写回ALU结果 Write back ALU result
	dstM		写回内存结果 [Write back memory result]
更新 PC update	PC	PC ← valC	更新PC Update PC

- 所有指令遵循同样的通用模式 All instructions follow same general pattern
- 差别在于每步计算什么 Differ in what gets computed on each step

计算的值 Computed Values

取指 Fetch

- icode　指令代码 Instruction code
- ifun　指令功能 Instruction function
- rA　指令寄存器A Instr. Register A
- rB　指令寄存器B Instr. Register B
- valC　指令常量 Instruction constant
- valP　增加后的PC Incremented PC

译码 Decode

- srcA　寄存器ID Register ID A
- srcB　寄存器ID Register ID B
- dstE　目的寄存器 Destination Register E
- dstM　目的寄存器 Destination Register M
- valA　寄存器A的值 Register value A
- valB　寄存器B的值 Register value B

执行 Execute

- valE　ALU结果 ALU result
- Cnd　分支/传送标志 Branch/move flag

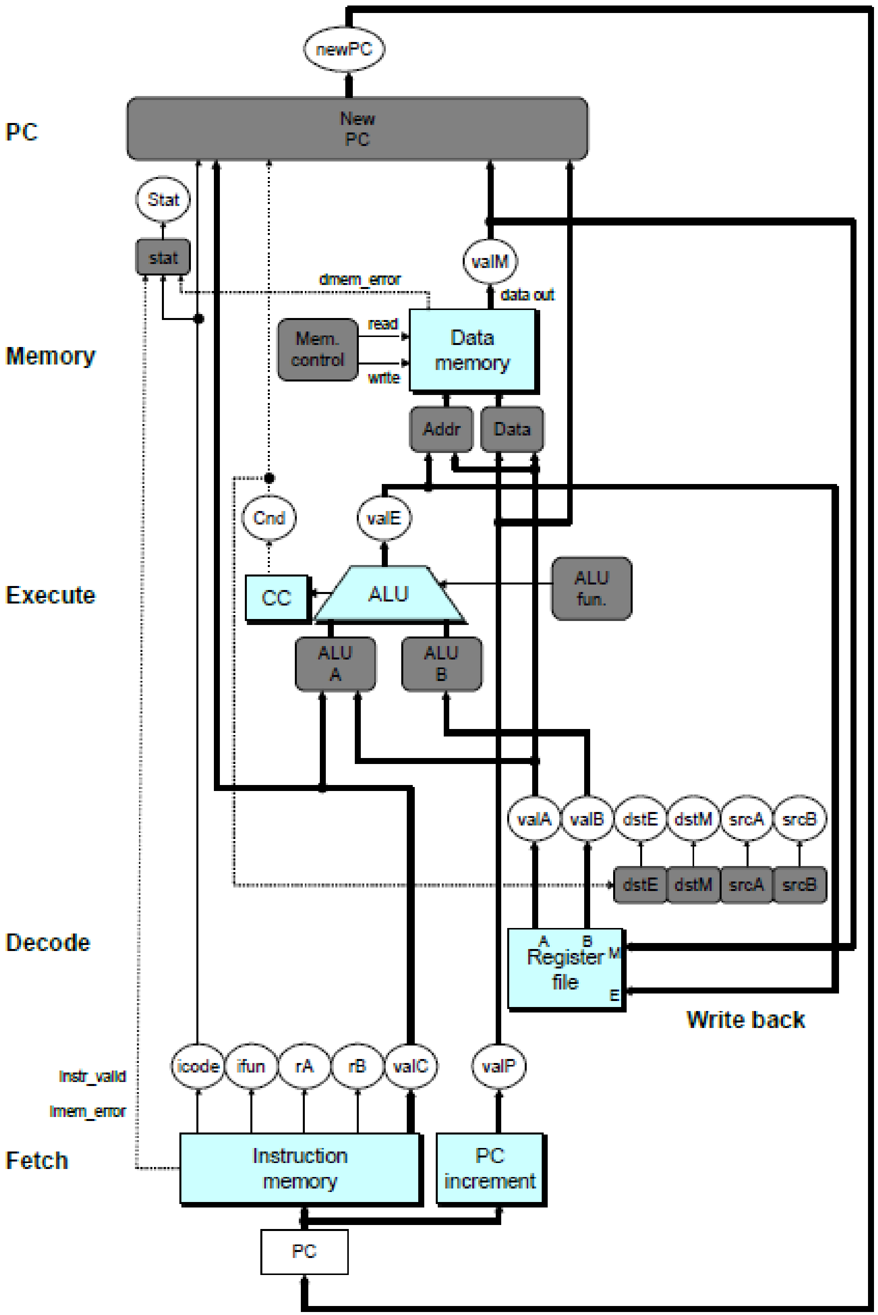
访存 Memory

- valM　内存的值 Value from memory

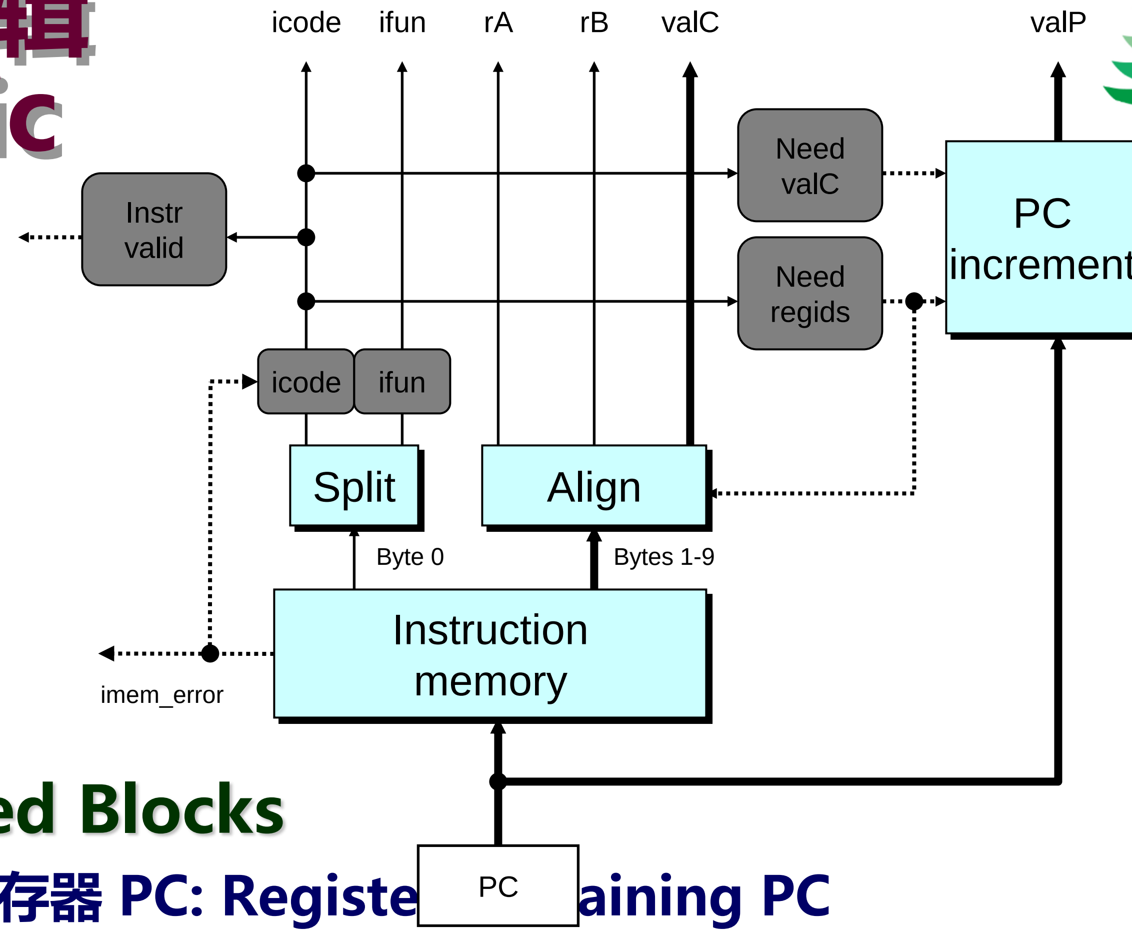
顺序处理器硬件 SEQ Hardware

关键 Key

- 蓝框：预设计的硬件块 Blue boxes: predesigned hardware blocks
  - 例如内存、ALU E.g., memories, ALU
- 灰框：控制逻辑 Gray boxes: control logic
  - HCL中描述 Describe in HCL
- 白椭圆框：信号标签 White ovals: labels for signals
- 粗线：64位字的值 Thick lines: 64-bit word values
- 细线：4-8位值 Thin lines: 4-8 bit values
- 虚线：1位值 Dotted lines: 1-bit values



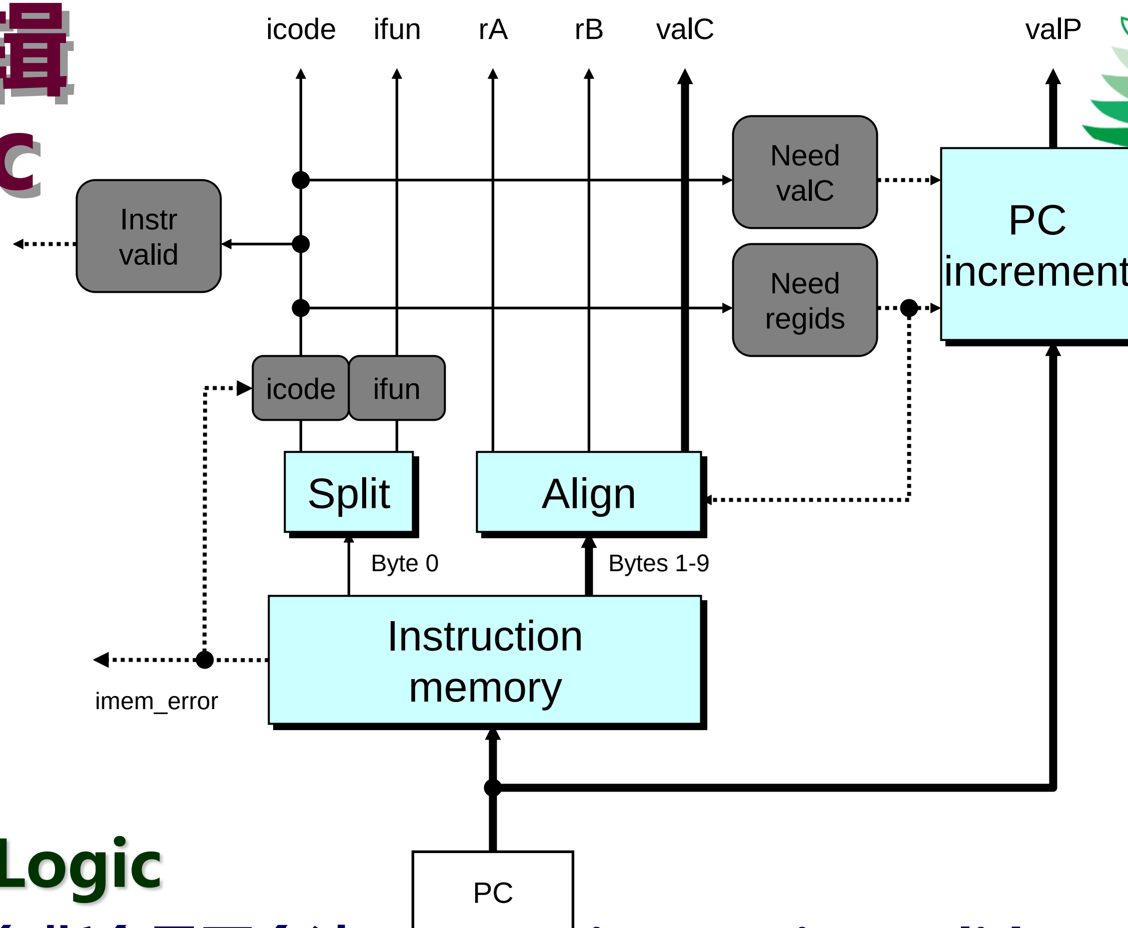
取指阶段逻辑 Fetch Logic



预定义块 Predefined Blocks

- PC：包含PC的寄存器 PC: Registe aining PC
- 指令内存：读10个字节 Instruction memory: Read 10 bytes (PC to PC+9)
  - 不合法地址给出信号指示 Signal invalid address
- 分离器：指令字节分成icode和ifun两部分 Split: Divide instruction byte into icode and ifun
- 对齐：得到rA、rB和valC字段 Align: Get fields for rA, rB, and valC

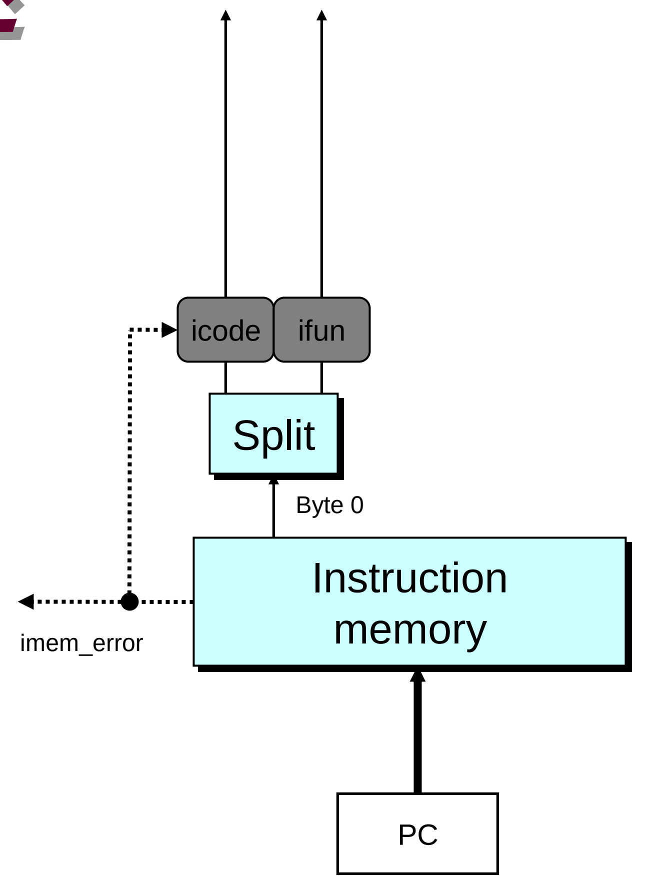
取指阶段逻辑 Fetch Logic



控制逻辑 Control Logic

- Instr. Valid: 这条指令是否合法? Is this instruction valid?
- icode, ifun: 如果地址不合法，产生空指令 Generate no-op if invalid address
- Need regids: 这条指令有寄存器字节吗? Does this instruction have a register byte?
- Need valC: 这条指令有常量字吗? Does this instruction have a constant word?

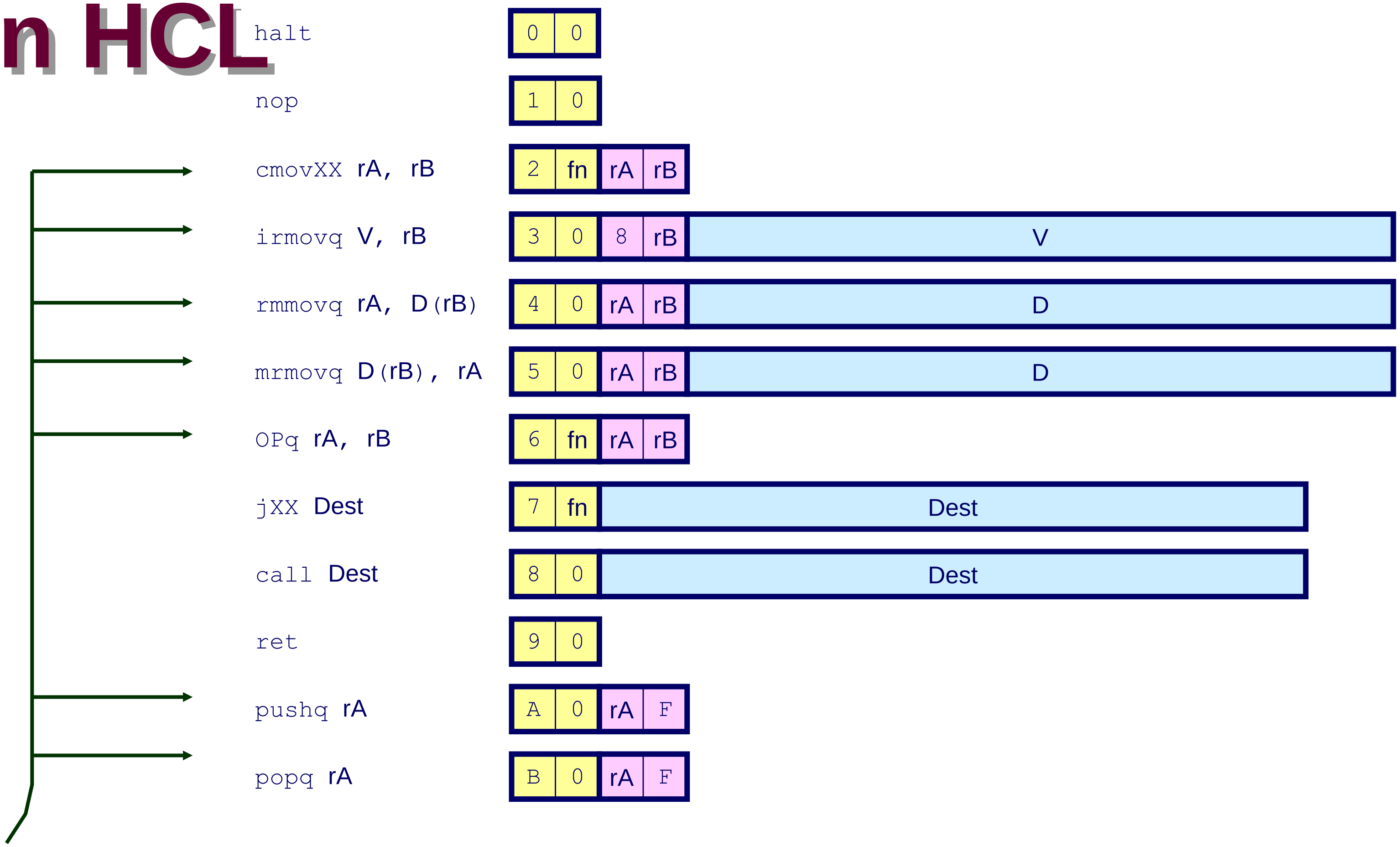
取指控制逻辑HCL描述 Fetch Control Logic in HCL




```
# Determine instruction code
int icode = [
        imem_error: INOP;
        1: imem_icode;
];

# Determine instruction function
int ifun = [
        imem_error: FNONE;
        1: imem_ifun;
];
```

取指控制逻辑HCL描述 Fetch Control Logic in HCL




```
bool need_regids =
        icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
                   IIRMOVQ, IRMMOVQ, IMRMOVQ };

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

译码阶段逻辑 Decode Logic

寄存器文件（堆）Register File

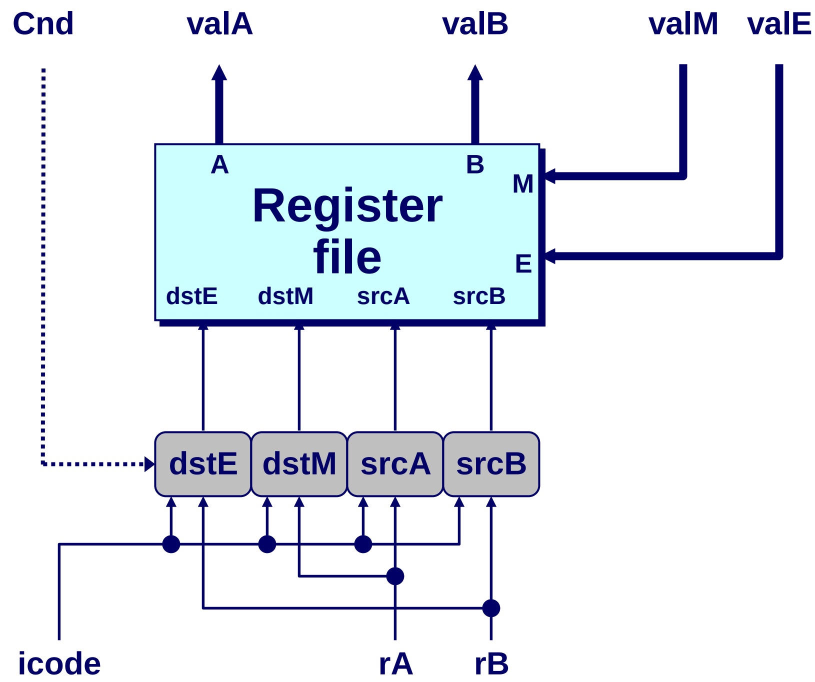
- 读端口A和B Read ports A, B
- 写端口E和M Write ports E, M
- 地址是寄存器ID或15（0xF）（不访问）Addresses are register IDs or 15 (0xF) (no access)

控制逻辑 Control Logic

- srcA, srcB: 读端口地址 read port addresses
- dstE, dstM: 写端口地址 write port addresses

信号 Signals

- Cnd: 指明是否执行条件传送 Indicate whether or not to perform conditional move
  - 执行阶段计算 Computed in Execute stage



端口A的来源
A Source

OPq rA, rB	Decode	valA ← R[rA]	读操作数A Read operand A
cmovXX rA, rB	Decode	valA ← R[rA]	读操作数A Read operand A
rmmovq rA, D(rB)	Decode	valA ← R[rA]	读操作数A Read operand A
popq rA	Decode	valA ← R[%rsp]	读栈指针 Read stack pointer
jXX Dest	Decode		无操作数 No operand
call Dest	Decode		无操作数 No operand
ret	Decode	valA ← R[%rsp]	读栈指针 Read stack pointer

```
int srcA = [
        icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ  } : rA;
        icode in { IPOPQ, IRET } : RRSP;
        1 : RNONE; # Don't need register
];
```

端口E的目的地址 E Destination

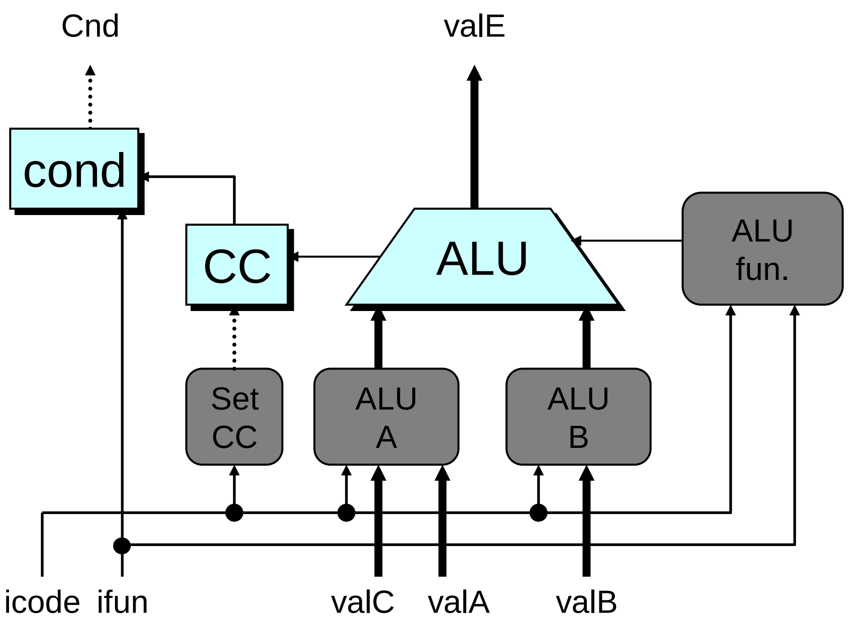
	Write-back	
OPq rA, rB	R[rB] ← valE	写回结果 Write back result
cmovXX rA, rB	R[rB] ← valE	条件写回结果 Conditionally write back result
rmmovq rA, D(rB)		无 None
popq rA	R[%rsp] ← valE	更新栈指针 Update stack pointer
jXX Dest		无 None
call Dest	R[%rsp] ← valE	更新栈指针 Update stack pointer
ret	R[%rsp] ← valE	更新栈指针 Update stack pointer

```
int dstE = [
        icode in { IRRMOVQ } && Cnd : rB;
        icode in { IIRMOVQ, IOPQ} : rB;
        icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
        1 : RNONE;  # Don't write any register
];
```

执行阶段逻辑 Execute Logic

单元 Units

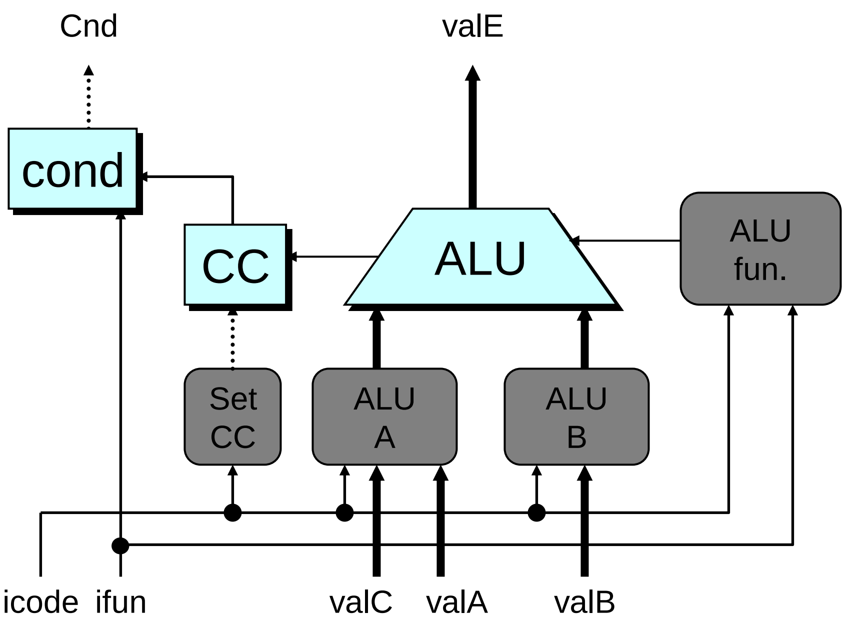
- 算术逻辑单元 ALU
  - 实现4个需要的功能 Implements 4 required functions
  - 产生条件码的值 Generates condition code values
- 条件码 CC
  - 有3个条件代码位的寄存器 Register with 3 condition code bits
- cond
  - 计算条件跳转/传送标志 Computes conditional jump/move flag



执行阶段逻辑 Execute Logic

控制逻辑 Control Logic

- **Set CC: 装载条件码寄存器吗? Should condition code register be loaded?**
- **ALU A: ALU输入端A Input A to ALU**
- **ALU B: ALU输入端B Input B to ALU**
- **ALU fun: ALU的计算功能 What function should ALU compute?**



ALU A 输入端 ALU A Input

Instruction	Stage	Computation	说明 / Description
OPq rA, rB	Execute	valE ← valB OP valA	执行ALU操作 Perform ALU operation
cmovXX rA, rB	Execute	valE ← 0 + valA	传递valA直通ALU Pass valA through ALU
rmmovq rA, D(rB)	Execute	valE ← valB + valC	计算有效地址 Compute effective address
popq rA	Execute	valE ← valB + 8	增加栈指针 Increment stack pointer
jXX Dest	Execute		无操作 No operation
call Dest	Execute	valE ← valB + –8	减少栈指针 Decrement stack pointer
ret	Execute	valE ← valB + 8	增加栈指针 Increment stack pointer

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

ALU 运算 ALU Oper-ation

	OPq rA, rB
Execute	valE ← valB OP valA

执行ALU操作

Perform ALU operation

	cmovXX rA, rB
Execute	valE ← 0 + valA

传递valA直通ALU

Pass valA through ALU

	`rmmovl` rA, D(rB)
Execute	valE ← valB + valC

计算有效地址

Compute effective address

	`popq` rA
Execute	valE ← valB + 8

增加栈指针

Increment stack pointer

	jXX Dest
Execute	

无操作 No operation

	`call` Dest
Execute	valE ← valB + –8

减少栈指针

Decrement stack pointer

	`ret`
Execute	valE ← valB + 8

增加栈指针

Increment stack pointer

```
int alufun = [
        icode == IOPQ : ifun;
        1 : ALUADD;
];
```

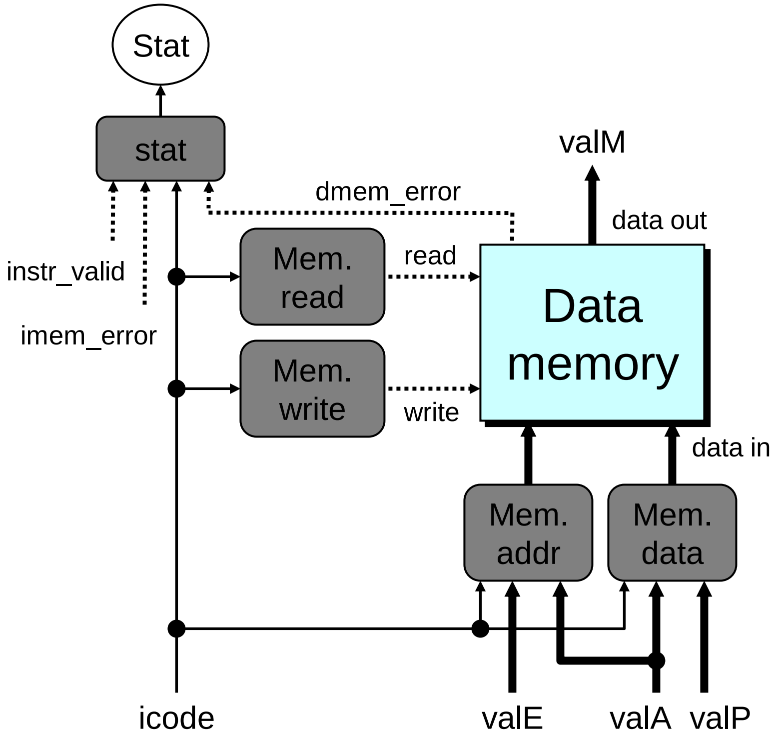
访存阶段逻辑 Memory Logic

访存 Memory

- 读或写内存字 Reads or writes memory word

控制逻辑 Control Logic

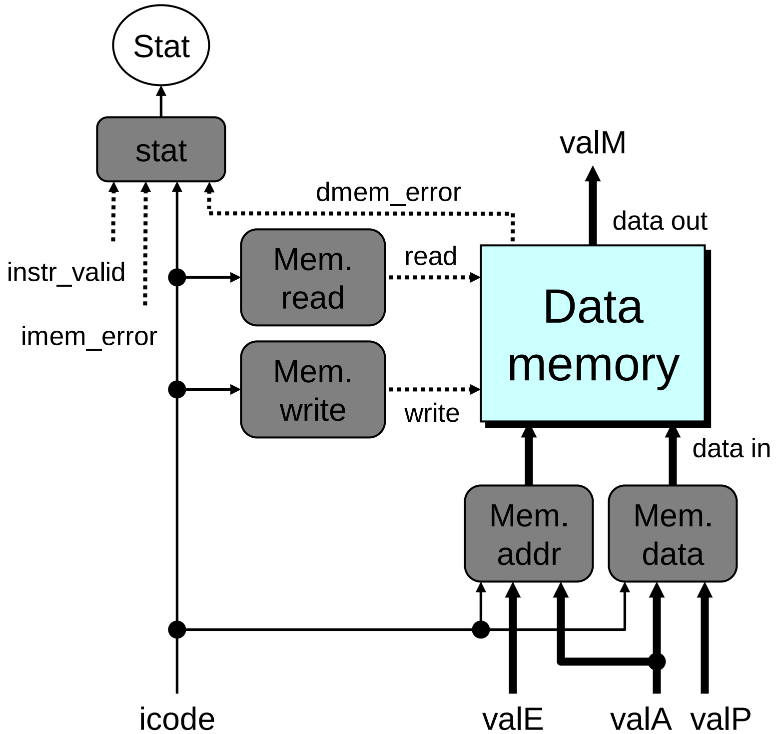
- stat: 指令状态 What is instruction status?
- Mem. read: 读字 should word be read?
- Mem. write: 写字 should word be written?
- Mem. addr.: 选择地址 Select address
- Mem. data.: 选择数据 Select data



指令状态 Instruction Status

控制逻辑 Control Logic

- **stat: 指令状态 What is instruction status?**




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

内存地址 Memory Address

Instruction	Memory	
OPq rA, rB		无操作 No operation
rmmovq rA, D(rB)	M_8[valE] ← valA	写数据到内存 Write value to memory
popq rA	valM ← M_8[valA]	从栈读数据 Read from stack
jXX Dest		无操作 No operation
call Dest	M_8[valE] ← valP	写返回地址到栈 Write return value on stack
ret	valM ← M_8[valA]	读返回地址 Read return address

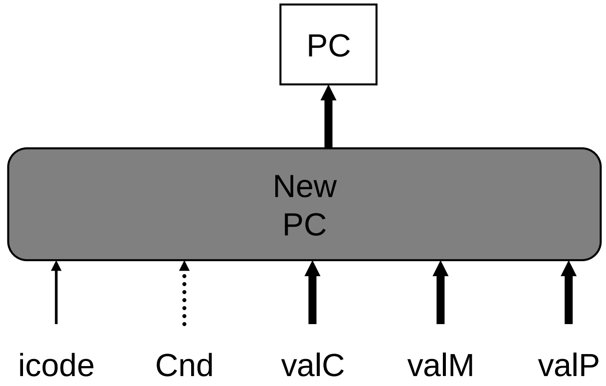
```
int mem_addr = [
        icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : valE;
        icode in { IPOPQ, IRET } : valA;
        # Other instructions don't need address
];
```

内存读 Memory Read

OPq rA, rB	Memory		无操作 No operation
`rmmovq` rA, D(rB)	Memory	$M_8[\text{valE}] \leftarrow \text{valA}$	写数据到内存 Write value to memory
`popq` rA	Memory	$\text{valM} \leftarrow M_8[\text{valA}]$	从栈读数据 Read from stack
jXX Dest	Memory		无操作 No operation
`call` Dest	Memory	$M_8[\text{valE}] \leftarrow \text{valP}$	写返回地址到栈 Write return value on stack
`ret`	Memory	$\text{valM} \leftarrow M_8[\text{valA}]$	读返回地址 Read return address

```
bool mem_read = icode in { IMRMOVQ, IPOPQ, IRET };
```

PC更新阶段逻辑 PC Update Logic



New PC

- 选择PC的下一个值 Select next value of PC

PC更新 PC Update

	OPq rA, rB
PC update	PC ← valP

更新PC Update PC

	`rmmovq` rA, D(rB)
PC update	PC ← valP

更新PC Update PC

	`popq` rA
PC update	PC ← valP

更新PC Update PC

	jXX Dest
PC update	PC ← Cnd ? valC : valP

更新PC Update PC

	`call` Dest
PC update	PC ← valC

设置PC为目的地址

Set PC to destination

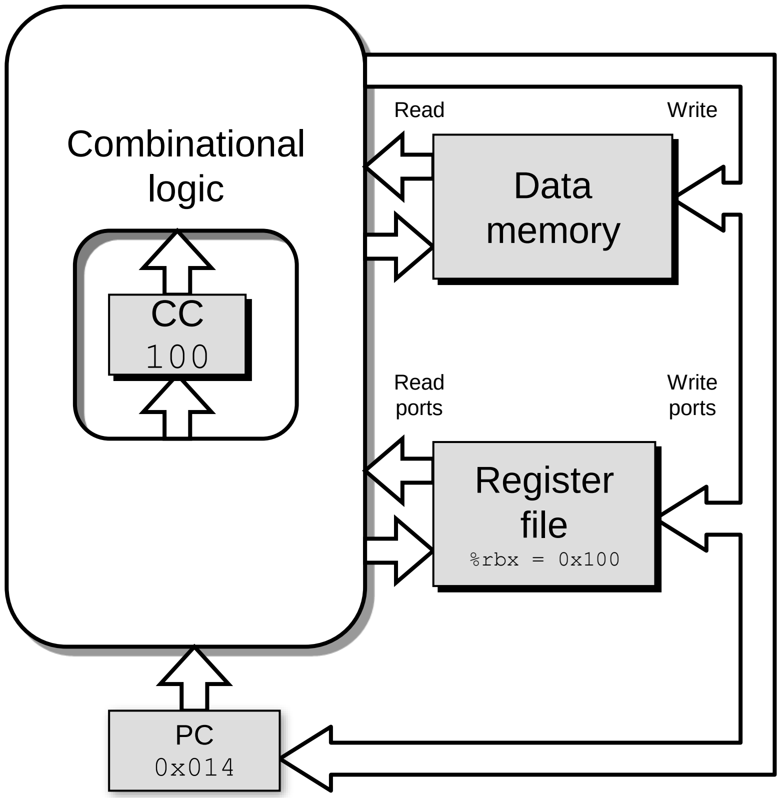
	`ret`
PC update	PC ← valM

设置PC为返回地址

Set PC to return address

```
int new_pc = [
        icode == ICALL : valC;
        icode == IJXX && Cnd : valC;
        icode == IRET : valM;
        1 : valP;
];
```

顺序处理器操作 SEQ Operation

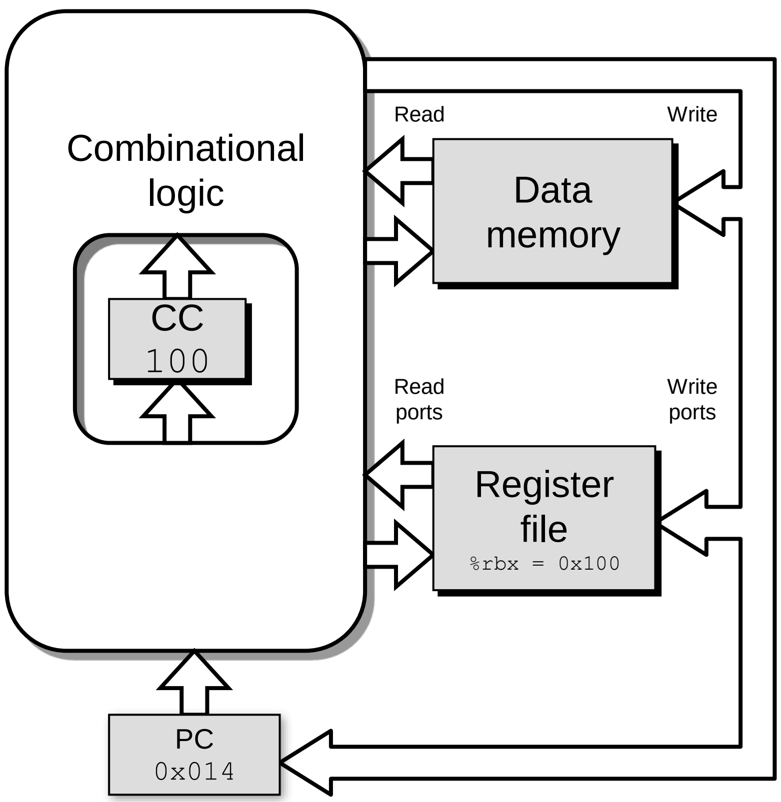


状态 State

- PC寄存器 PC register
- 条件码寄存器 Cond. Code register
- 数据内存 Data memory
- 寄存器文件（堆） Register file

所有更新在时钟上升沿 All updated as clock rises

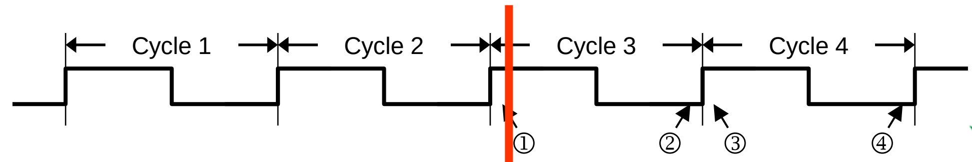
顺序处理器操作 SEQ Operation



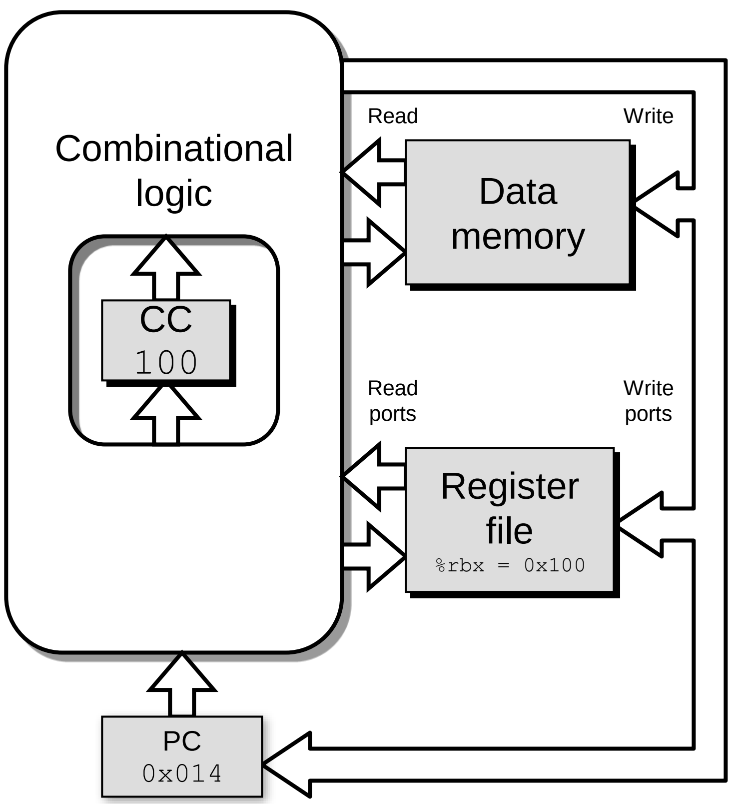
组合逻辑 Combinational Logic

- 算术/逻辑单元 ALU
- 控制逻辑 Control logic
- 内存读 Memory reads
  - 指令内存 Instruction memory
  - 寄存器文件（堆）Register file
  - 数据内存 Data memory

SEQ操作 SEQ Operation #2

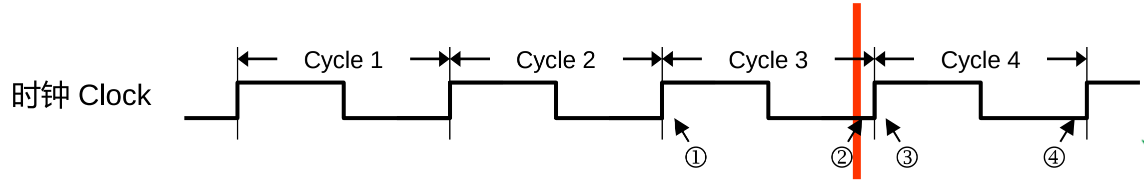


周期1 Cycle 1:	0x000:	irmovq $0x100,%rbx	# %rbx <-- 0x100
周期2 Cycle 2:	0x00a:	irmovq $0x200,%rdx	# %rdx <-- 0x200
周期3 Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
周期4 Cycle 4:	0x016:	je dest	# Not taken
周期5 Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300

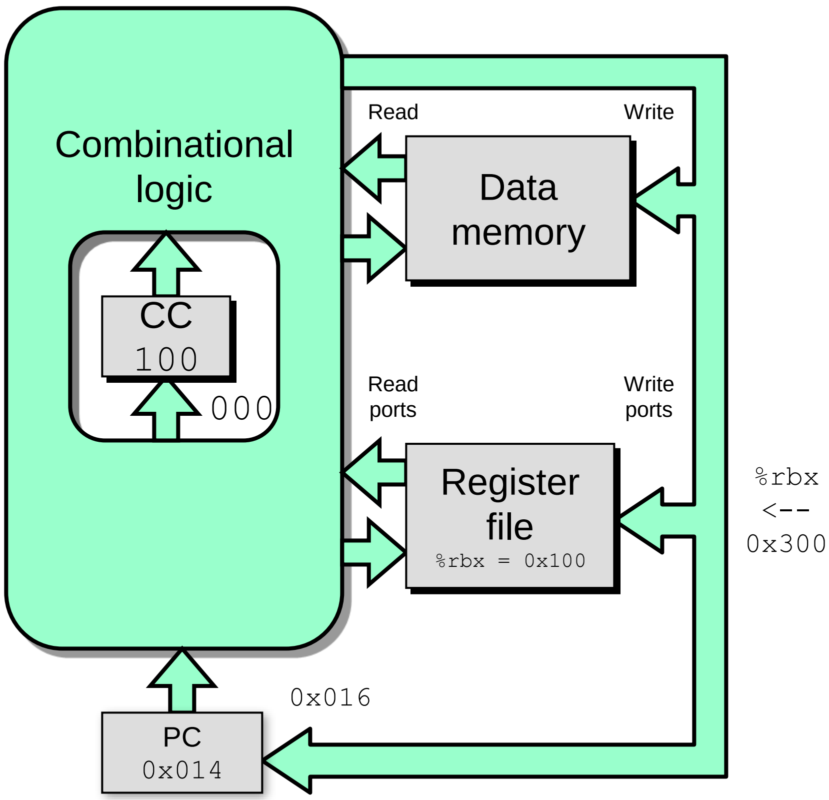


- 状态设置按照第二条irmovq指令 state set according to second `irmovq` instruction
- 组合逻辑开始反应状态改变 combinational logic starting to react to state changes

SEQ操作 SEQ Operation #3

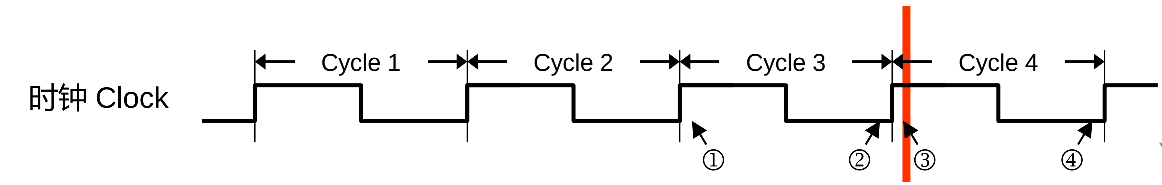


周期1 Cycle 1:	0x000:	irmovq $0x100,%rbx	# %rbx <-- 0x100
周期2 Cycle 2:	0x00a:	irmovq $0x200,%rdx	# %rdx <-- 0x200
周期3 Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
周期4 Cycle 4:	0x016:	je dest	# Not taken
周期5 Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300

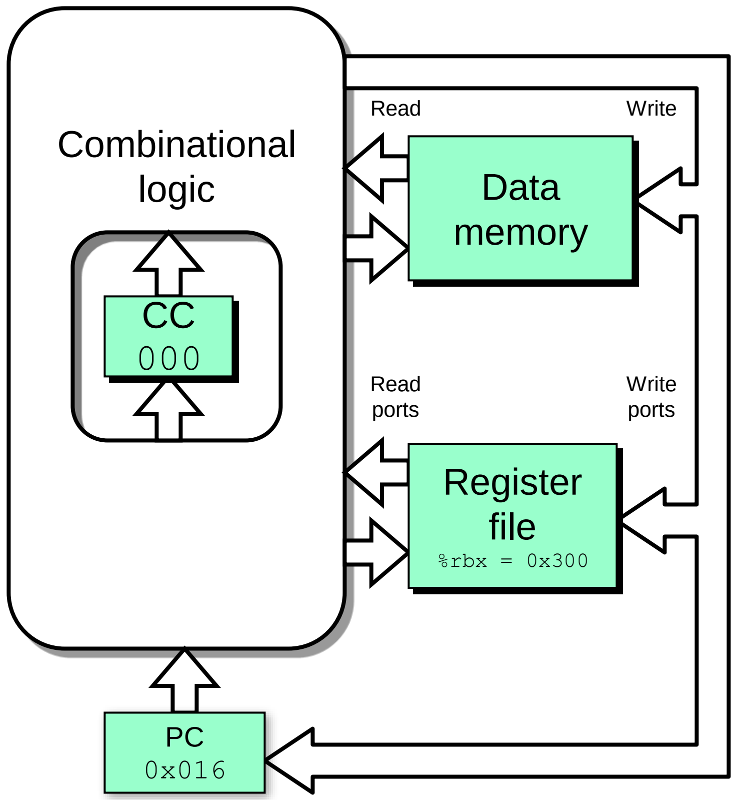


- **状态设置按照第二条irmovq指令state set according to second `irmovq` instruction**
- **组合逻辑产生addq指令的结果 combinational logic generates results for `addq` instruction**

SEQ操作 SEQ Operation #4

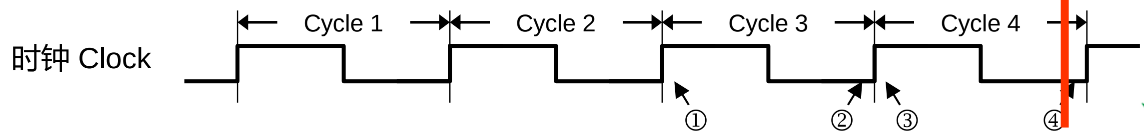


周期1 Cycle 1:	0x000:	irmovq $0x100,%rbx	# %rbx <-- 0x100
周期2 Cycle 2:	0x00a:	irmovq $0x200,%rdx	# %rdx <-- 0x200
周期3 Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
周期4 Cycle 4:	0x016:	je dest	# Not taken
周期5 Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300

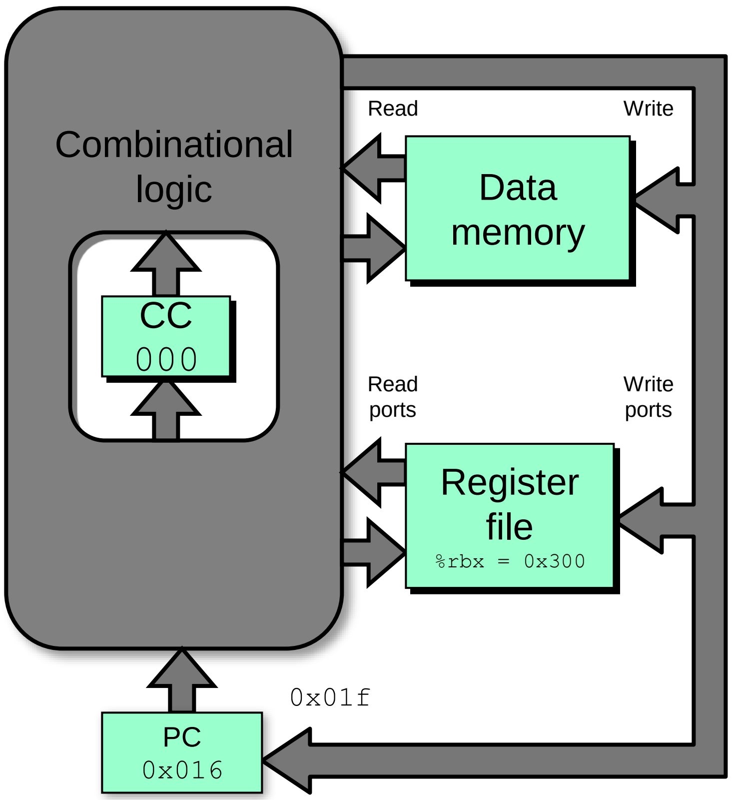


- 状态设置按照addq指令 state set according to addq instruction
- 组合逻辑开始反应状态改变 combinational logic starting to react to state changes

SEQ 操作 SEQ Operation #5



周期1 Cycle 1:	0x000:	irmovq $0x100,%rbx	# %rbx <-- 0x100
周期2 Cycle 2:	0x00a:	irmovq $0x200,%rdx	# %rdx <-- 0x200
周期3 Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
周期4 Cycle 4:	0x016:	je dest	# Not taken
周期5 Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300



- 状态设置按照addq指令 state set according to `addq` instruction
- 组合逻辑产生je指令的结果 combinational logic generates results for `je` instruction

顺序处理器小结 SEQ Summary

实现 Implementation

- 表达每条指令为一系列简单的步骤 Express every instruction as series of simple steps
- 每个指令类型遵循同样通用流程 Follow same general flow for each instruction type
- 装配寄存器、内存和预设计的组合逻辑块 Assemble registers, memories, predesigned combinational blocks
- 用控制逻辑进行连接 Connect with control logic

限制 Limitations

- 太慢不实用 Too slow to be practical
- 一个周期内必须传播通过指令内存、寄存器文件（堆）、ALU和数据内存 In one cycle, must propagate through instruction memory, register file, ALU, and data memory
- 需要运行时钟非常慢 Would need to run clock very slowly
- 硬件单元仅在时钟周期的一部分时间内是活动的 Hardware units only active for fraction of clock cycle