Jin-Soo Kim
(jinsoo.kim@snu.ac.kr)

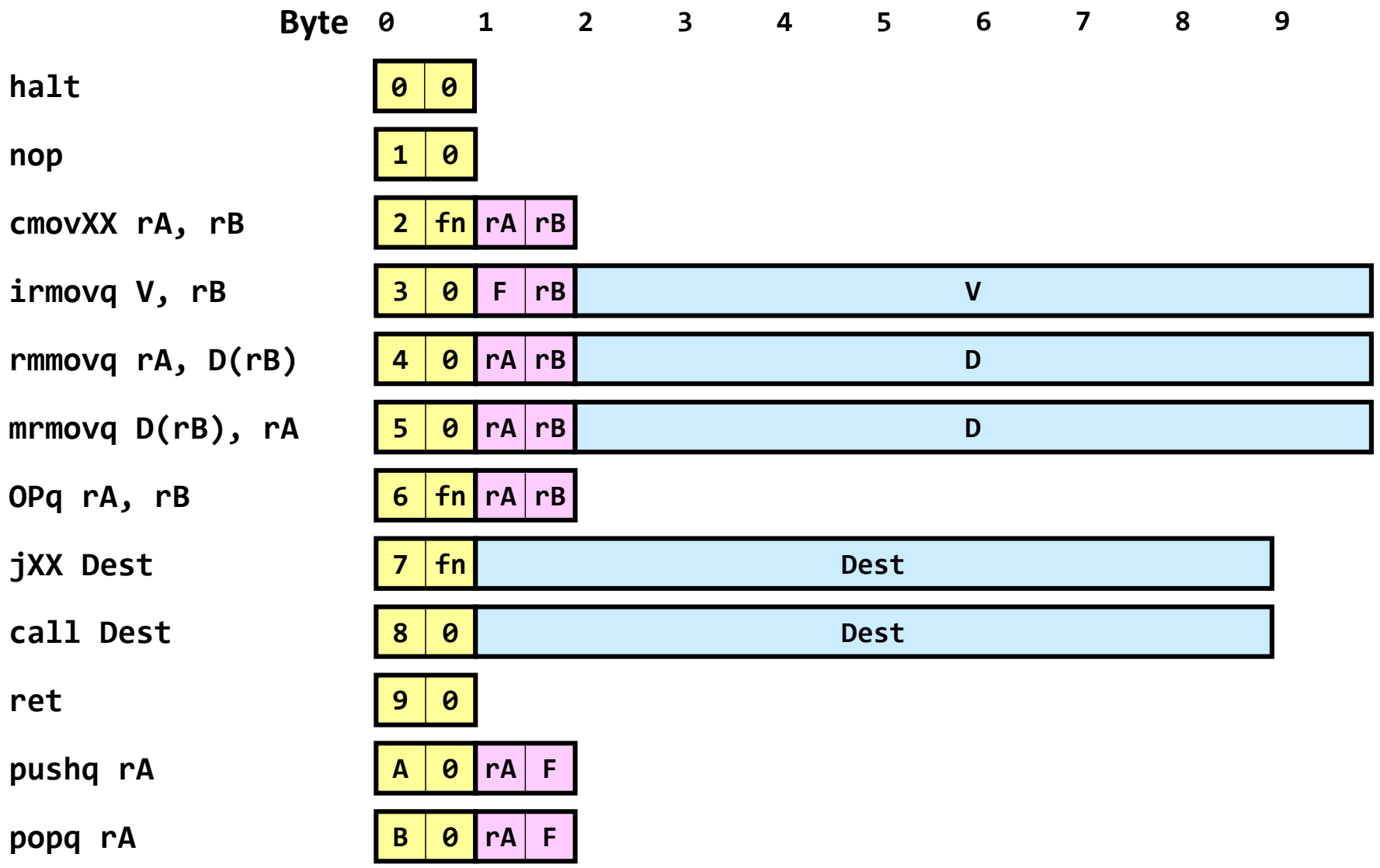
Systems Software &
Architecture Lab.

Seoul National University

Fall 2018

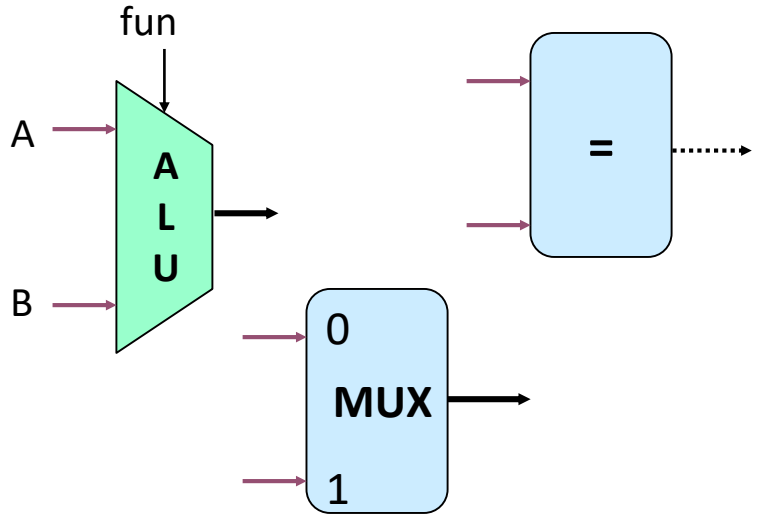
SEQ: Sequential Y86-64 Implementation

Recall: Y86-64 Instruction Set

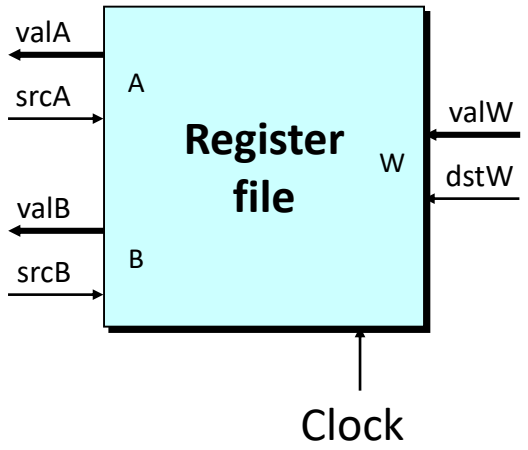


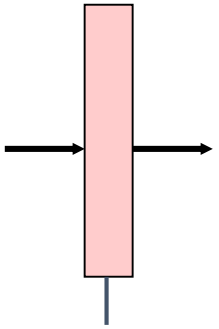
Building Blocks

- Combinational logic
  - Compute Boolean functions of inputs
  - Continuously respond to input changes
  - Operate on data and implement control



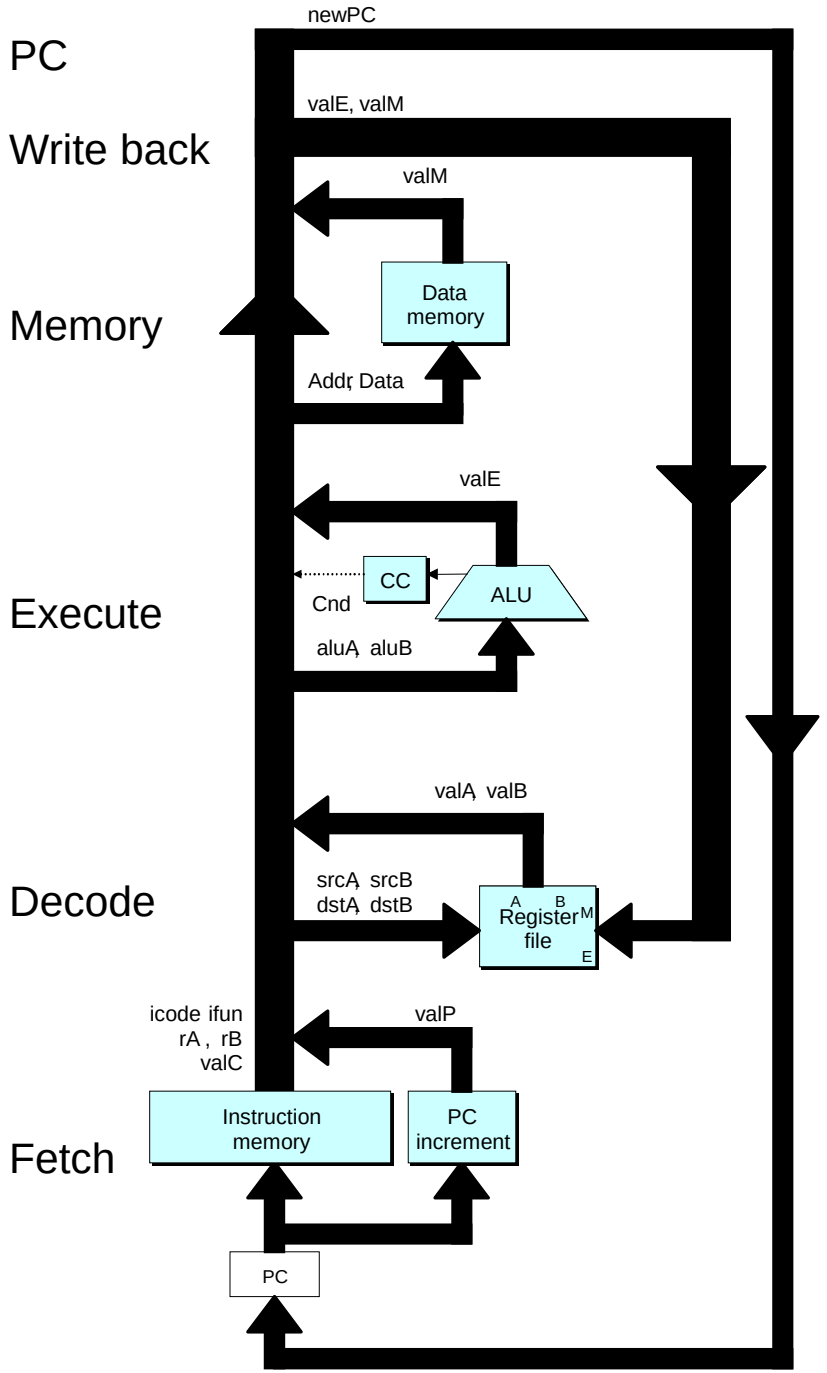
- Storage elements
  - Store bits (or states)
  - Addressable memories
  - Loaded only as clock rises





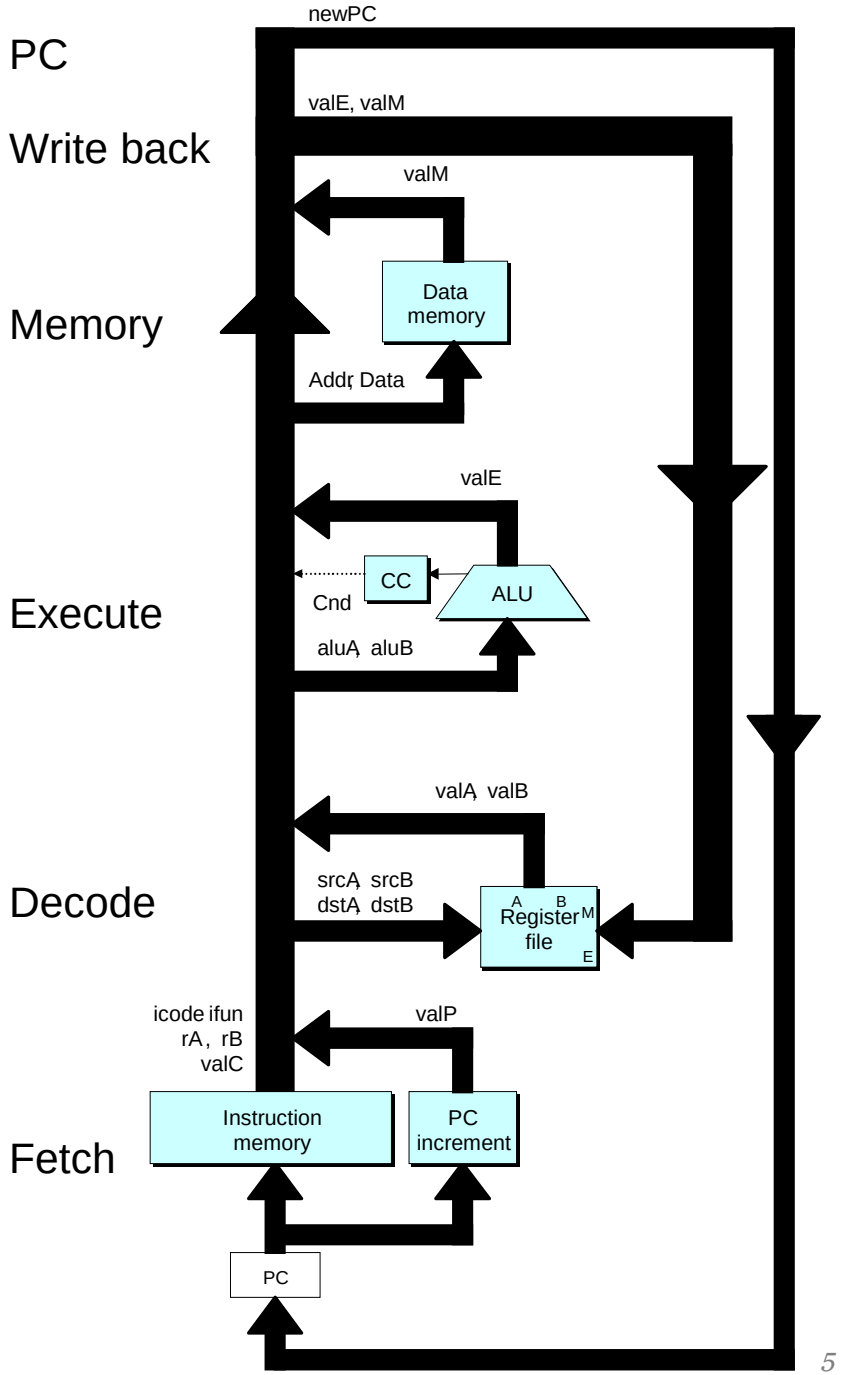
SEQ Hardware Structure

- State
  - Program counter registers (PC)
  - Condition code register (CC)
  - Register file
  - Memories
    - Access same memory space
    - Data: for reading/writing program data
    - Instruction: for reading instructions
- Instruction flow
  - Read instructions at address specified by PC
  - Process through stages
  - Update program counter



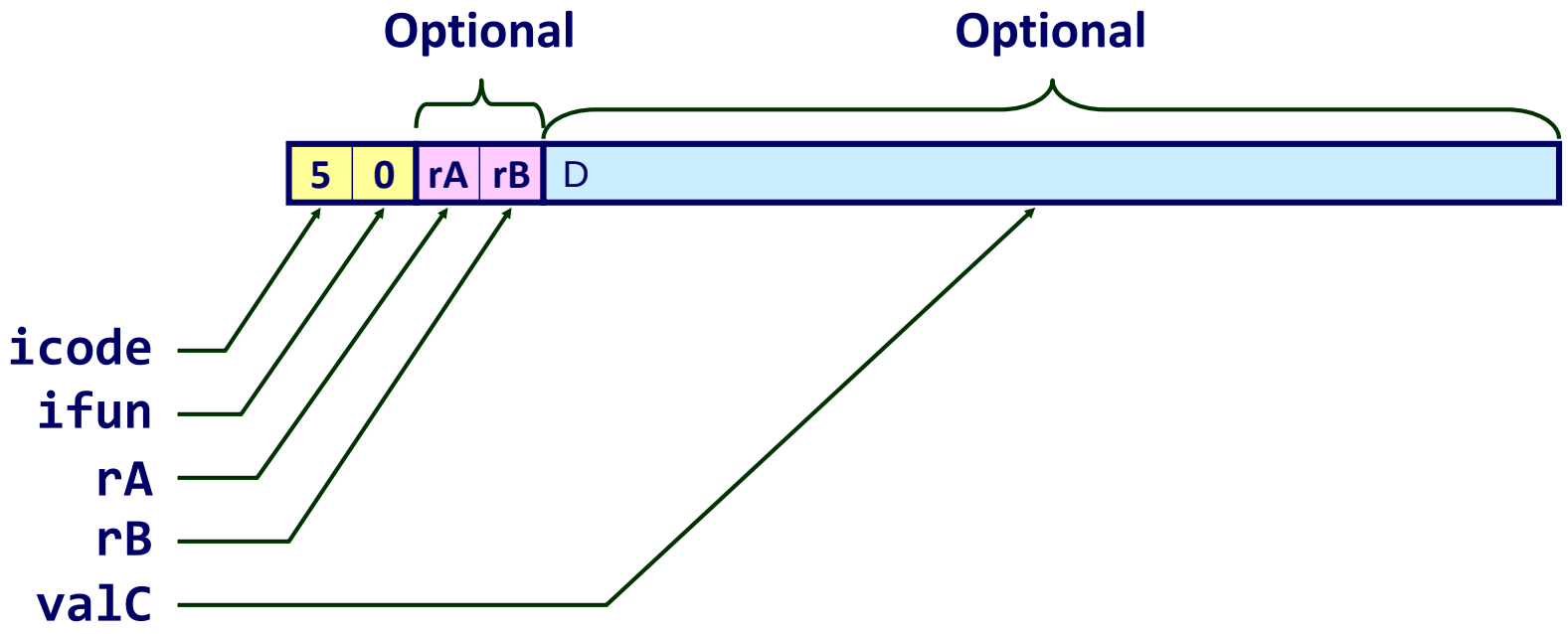
SEQ Stages

- Fetch: Read instruction from instruction memory
- Decode: Read program registers
- Execute: Compute value or address
- Memory: Read or write data
- Write Back: Write program registers
- PC Update: Update program counter

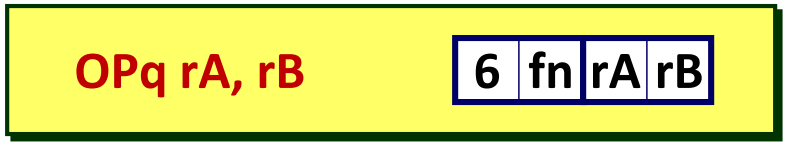


Instruction Decoding

- Instruction format
  - Instruction byte: `icode : ifun`
  - Optional register byte: `rA : rB`
  - Optional constant word: `valC`



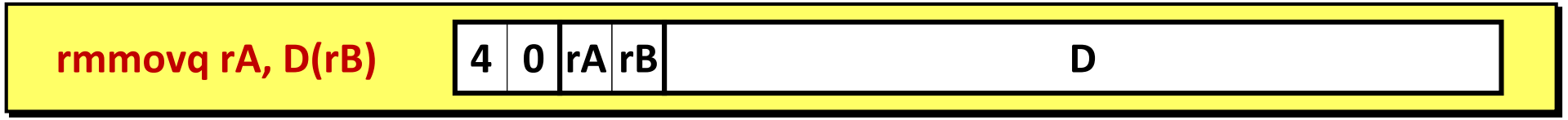
Executing ALU Operations



Fetch	`icode : ifun ← M1[PC]`
	`rA : rB ← M1[PC+1]`
	`valP ← PC+2`
		Read instruction byte
		Read register byte
		Compute next PC

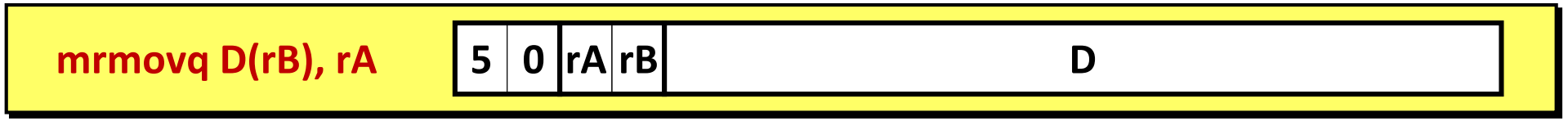
Decode	`valA ← R[rA]` `valB ← R[rB]`	Read operand A Read operand B
Execute	`valE ← valB OP valA` `Set CC`	Perform ALU operation Set condition code register
Memory		<Do nothing>
Write Back	`R[rB] ← valE`	Write back result
PC Update	`PC ← valP`	Increment PC by 2

Executing rmmovq



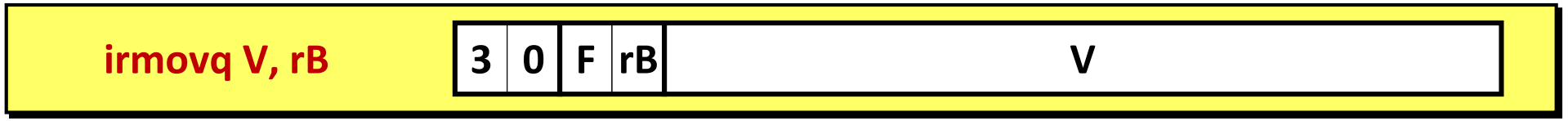
Fetch	`icode : ifun ← M1[PC]`	Read instruction byte
	`rA : rB ← M1[PC+1]`	Read register byte
	`valC ← M8[PC+2]`	Read displacement D
	`valP ← PC+10`	Compute next PC
Decode	`valA ← R[rA]`	Read operand A
	`valB ← R[rB]`	Read operand B
Execute	`valE ← valB + valC`	Compute effective address
Memory	`M8[valE] ← valA`	Write value to memory
Write Back		<Do nothing>
PC Update	`PC ← valP`	Increment PC by 10

Executing mrmovq



Fetch	`icode : ifun ← M1[PC]`	Read instruction byte
	`rA : rB ← M1[PC+1]`	Read register byte
	`valC ← M8[PC+2]`	Read displacement D
	`valP ← PC+10`	Compute next PC
Decode	`valB ← R[rB]`	Read operand B
Execute	`valE ← valB + valC`	Compute effective address
Memory	`valM ← M8[valE]`	Read value from memory
Write Back	`R[rA] ← valM`	Write back result
PC Update	`PC ← valP`	Increment PC by 10

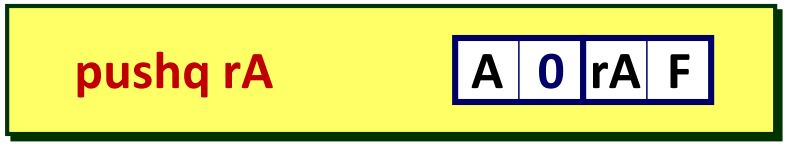
Executing irmovq



Fetch	`icode : ifun` ← M_1`[PC]`
	`rA : rB` ← M_1`[PC+1]`
	`valC` ← M_8`[PC+2]`
	`valP` ← `PC+10`
		Read instruction byte
		Read register byte
		Read immediate value V
		Compute next PC

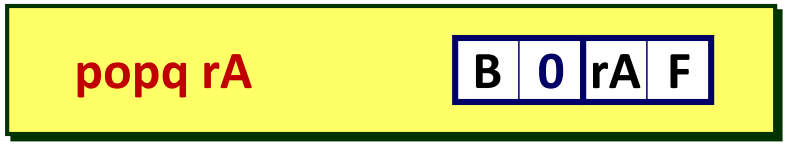
Decode	`valB` ← `0`	
Execute	`valE` ← `valB + valC`	Pass **`valC`** through ALU
Memory		<Do nothing>
Write Back	`R[rB]` ← `valE`	Write back result
PC Update	`PC` ← `valP`	Increment PC by 10

Executing pushq



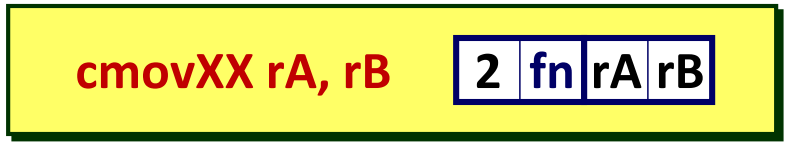
Fetch	`icode : ifun ← M1[PC]`	Read instruction byte
	`rA : rB ← M1[PC+1]`	Read register byte
	`valP ← PC+2`	Compute next PC
Decode	`valA ← R[rA]`	Read operand A
	`valB ← R[%rsp]`	Read stack pointer
Execute	`valE ← valB + (-8)`	Decrement stack pointer
Memory	`M8[valE] ← valA`	Write to stack
Write Back	`R[%rsp] ← valE`	Update stack pointer
PC Update	`PC ← valP`	Increment PC by 2

Executing popq



Fetch	`icode : ifun ← M1[PC]`	Read instruction byte
	`rA : rB ← M1[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 ← M8[valA]`	Read from stack
Write Back	`R[%rsp] ← valE`	Update stack pointer
	`R[rA] ← valM`	Write back result
PC Update	`PC ← valP`	Increment PC by 2

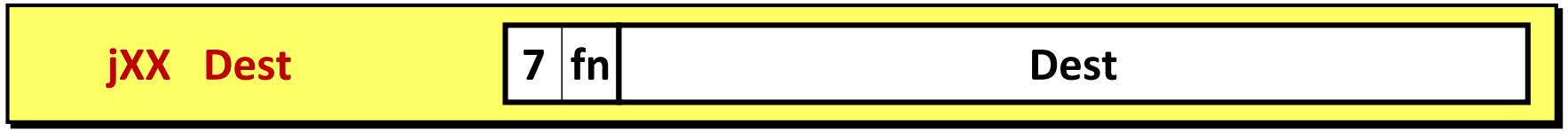
Executing Conditional Moves



Fetch	`icode : ifun ← M1[PC]`
	`rA : rB ← M1[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		<Do nothing>
Write Back	`R[rB] ← valE`	Write back result
PC Update	`PC ← valP`	Increment PC by 2

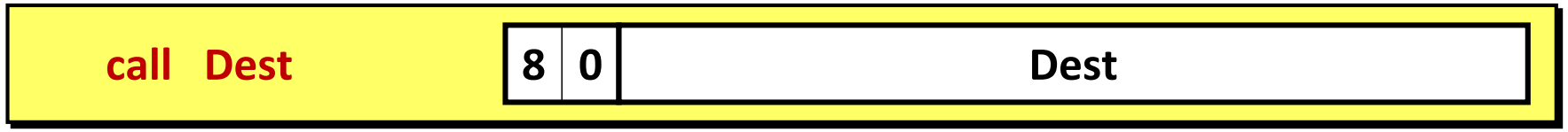
Executing Jumps



Fetch	`icode : ifun ← M1[PC]`
	`valC ← M8[PC+1]`
	`valP ← PC+9`
		Read instruction byte
		Read destination address
		Compute next PC

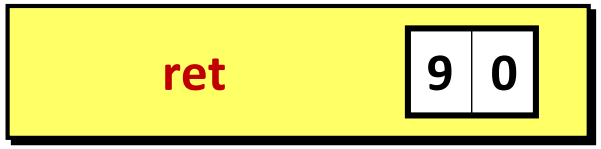
Decode		<Do nothing>
Execute	`Cnd ← Cond(CC,ifun)`	Take branch?
Memory		<Do nothing>
Write Back		<Do nothing>
PC Update	`PC ← Cnd? valC : valP`	Update PC

Executing `call`



Fetch	`icode : ifun ← M1[PC]`	Read instruction byte
	`valC ← M8[PC+1]`	Read destination address
	`valP ← PC+9`	Compute return address
Decode	`valB ← R[%rsp]`	Read stack pointer
Execute	`valE ← valB + (-8)`	Decrement stack pointer
Memory	`M8[valE] ← valP`	Write return address on stack
Write Back	`R[%rsp] ← valE`	Update stack pointer
PC Update	`PC ← valC`	Set PC to destination

Executing ret



Fetch	`icode : ifun ← M1[PC]` `valP ← PC+1`	Read instruction byte Compute next PC (not used)
Decode	`valA ← R[%rsp]` `valB ← R[%rsp]`	Read stack pointer Read stack pointer
Execute	`valE ← valB + 8`	Increment stack pointer
Memory	`valM ← M8[valA]`	Read return address from stack
Write Back	`R[%rsp] ← valE`	Update stack pointer
PC Update	`PC ← valM`	Set PC to return address

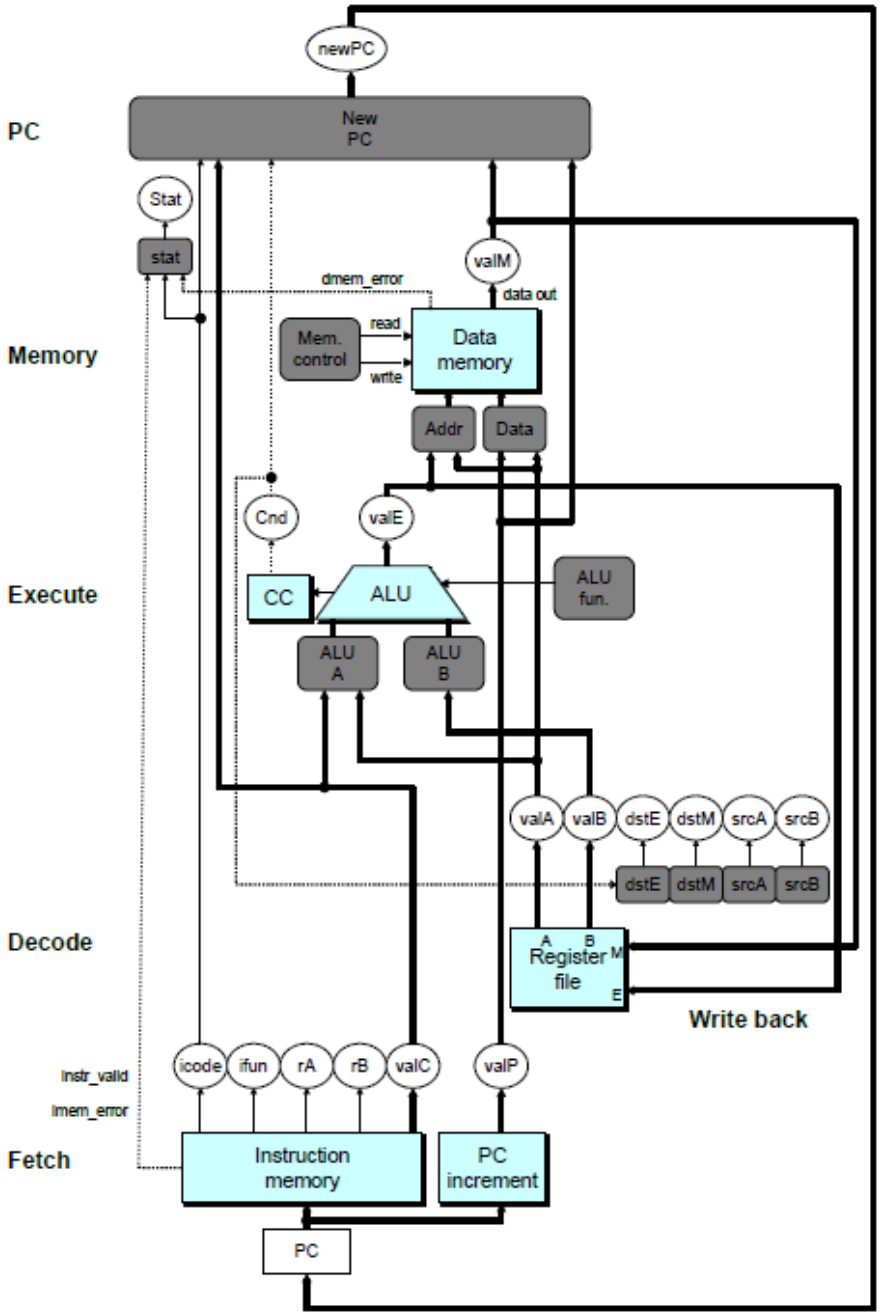
Computed Values

Fetch	`icode, ifun`
	`rA, rB`
	`valC`
	`valP`
		Instruction code / Instruction function
		Instruction register A / B
		Instruction constant
		Incremented PC

Decode	`srcA, valA` `srcB, valB`	Register ID A / Register value A Register ID B / Register value B
Execute	`valE` `Cnd`	ALU result Branch/move flag
Memory	`valM`	Value from memory
Write Back	`dstE, dstM`	Destination register E / M
PC Update	`PC`	PC register

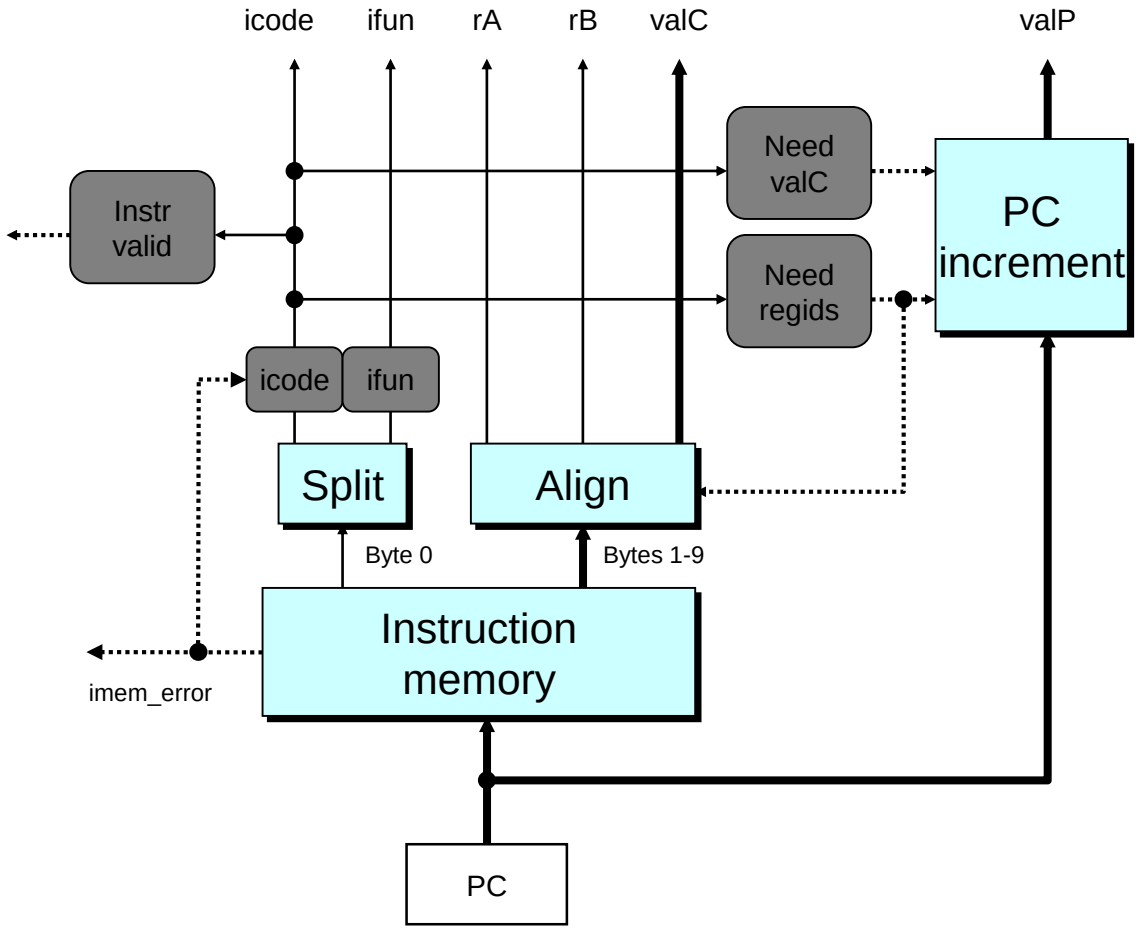
SEQ Hardware

- Blue boxes
  - Predesigned hardware blocks
  - e.g. memories, ALU
- Gray boxes
  - Control logic (described in HCL)
- Write ovals
  - Labels for signals
- Thick lines: 64-bit word values
- Thin lines: 4-8 bit values
- Dotted lines: 1-bit values



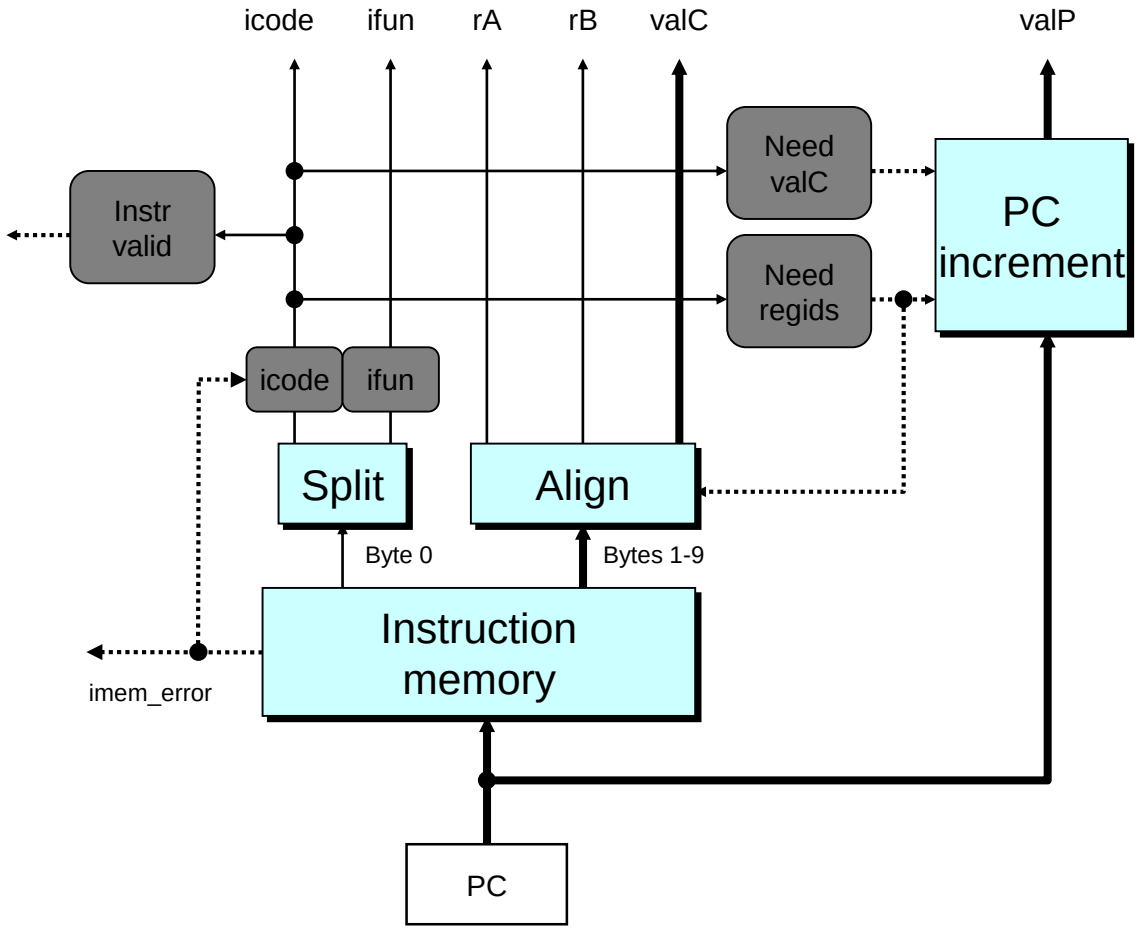
Fetch Logic: Data Path

- PC
  - Register containing PC
- Instruction memory
  - Read 10 bytes (PC to PC+9)
  - Signal invalid address
- Split
  - Divide instruction byte into `icode` and `ifun`
- Align
  - Get fields for `rA`, `rB`, and `valC`



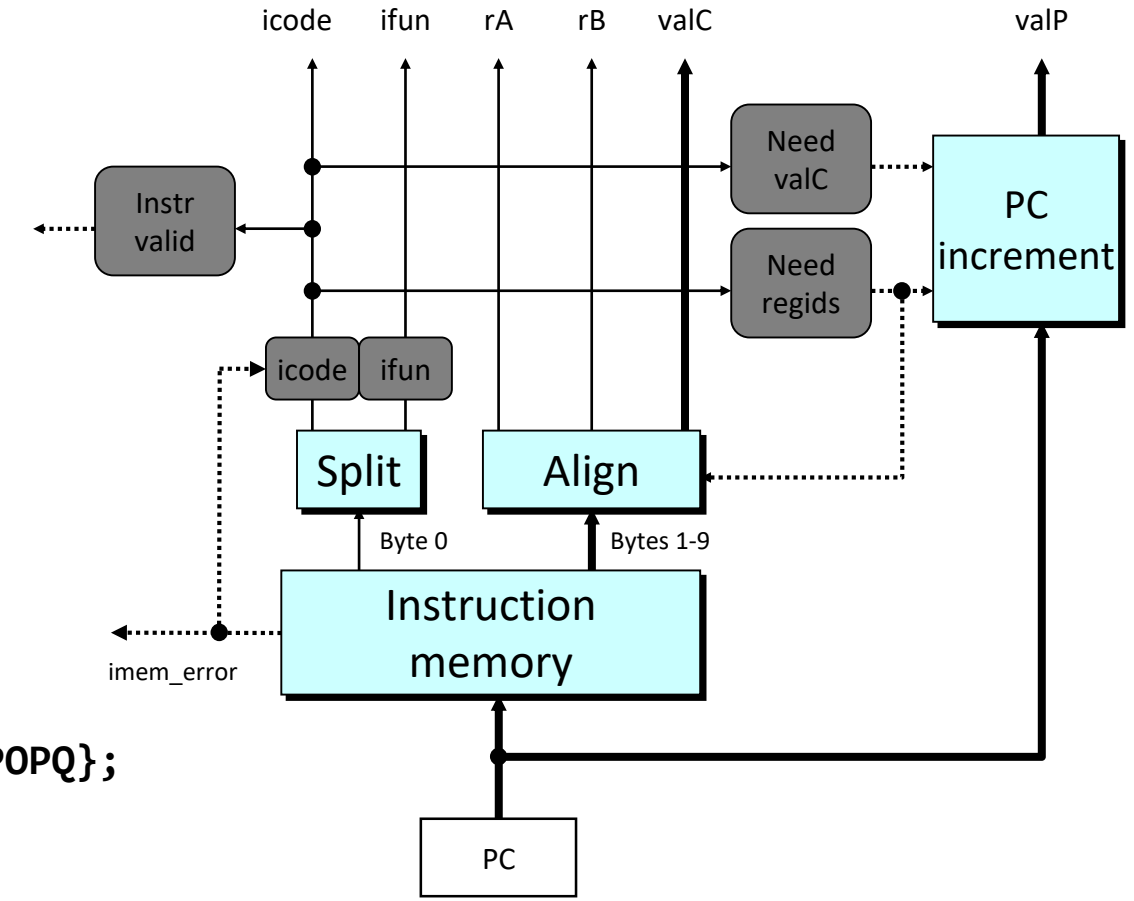
Fetch Logic: Control

- `instr_valid`
  - Is this instruction valid?
- `icode & ifun`
  - Generate no-op if invalid address
- `need_regids`
  - Does this instruction have a register byte?
- `need_valC`
  - Does this instruction have a constant word?



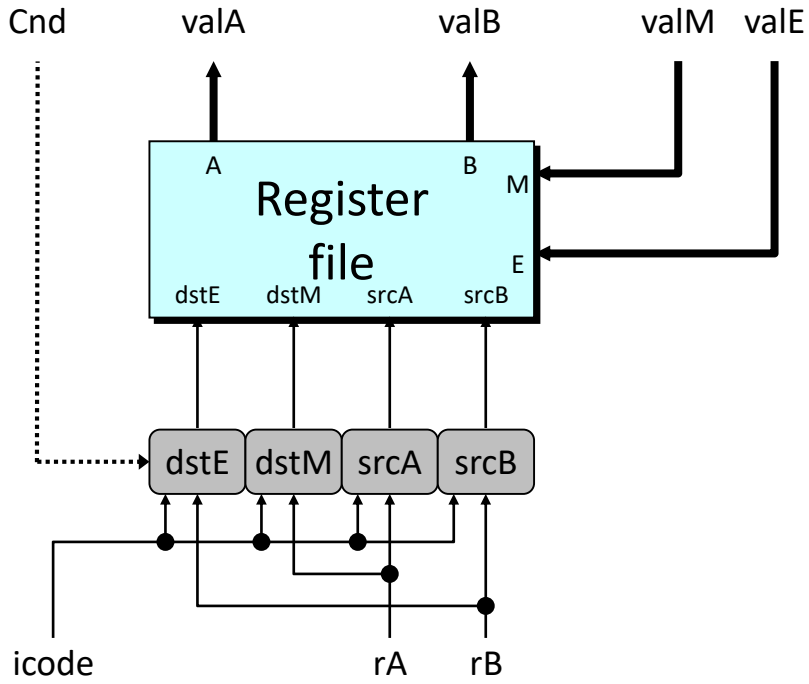
Fetch Logic: Control (in HCL)

```
int icode = [
    imem_error:  INOP;
    1: imem_icode;
];

int ifun = [
    imem_error:  FNONE;
    1:  imem_ifun;
];

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

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

bool need_valC = icode in
    { IIRMOVQ, IRMMOVQ, IMRMOVQ, IJXX, ICALL };
```



Decode Logic

- Register file
  - Read ports A, B
  - Write ports E, M
  - Addresses are register IDs or 15 (0xF – no access)
- Control
  - `srcA`, `srcB`: read port addresses
  - `dstE`, `dstM`: write port addresses
- Signals
  - Cnd: indicate whether or not to perform conditional move (computed in Execute stage)



Decode Logic: srcA in HCL

OPq	**Decode**	**valA ← R[rA]**	Read operand A
rmmovq	**Decode**	**valA ← R[rA]**	Read operand A
mrmovq	**Decode**		
irmovq	**Decode**		
pushq	**Decode**	**valA ← R[rA]**	Read operand A
popq	**Decode**	**valA ← R[%rsp]**	Read stack pointer
cmovXX	**Decode**	**valA ← R[rA]**	Read operand A
jXX	**Decode**		
call	**Decode**		
ret	**Decode**	**valA ← R[%rsp]**	Read stack pointer

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

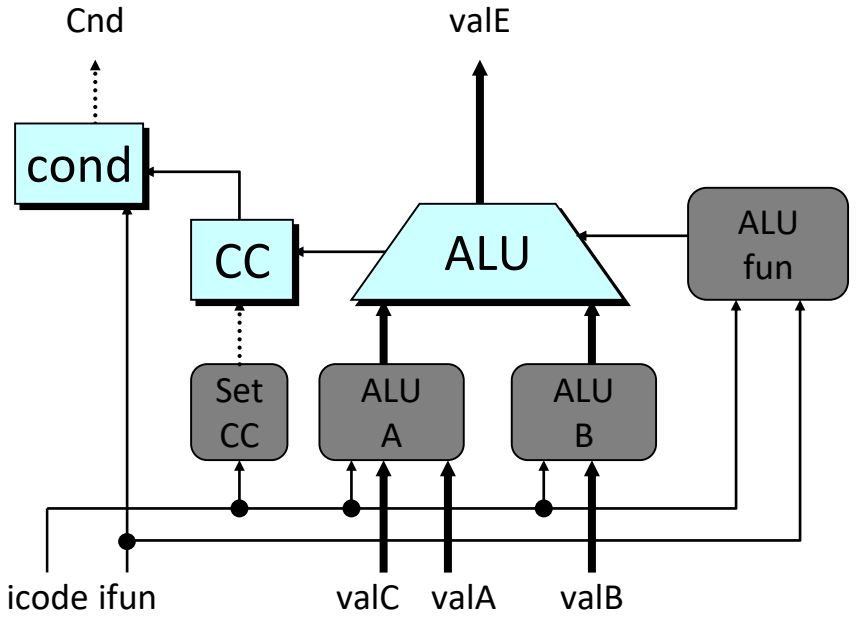
Decode Logic: dstE in HCL

OPq	Write Back	R[rB] ← valE	Write back result
rmmovq	Write Back		
mrmovq	Write Back		
irmovq	Write Back	R[rB] ← valE	Write back result
pushq	Write Back	R[%rsp] ← valE	Update stack pointer
popq	Write Back	R[%rsp] ← valE	Update stack pointer
cmovXX	Write Back	R[rB] ← valE	Write back result
jXX	Write Back		
call	Write Back	R[%rsp] ← valE	Update stack pointer
ret	Write Back	R[%rsp] ← valE	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
];
```

Execute Logic

- Data path
  - ALU
    - Implements 4 required functions
    - Generates condition code values
  - CC: Register with 3 condition code bits
  - Cond: Computes conditional jump/move flag
- Control logic
  - ALU A: Input A to ALU
  - ALU B: Input B to ALU
  - ALU fun: What function should ALU compute?
  - SetCC: Should condition code register be loaded?



Execute Logic: aluA in HCL

OPq	Execute	valE ← valB OP valA	Perform ALU operation
rmmovq	Execute	valE ← valB + valC	Compute effective address
mrmovq	Execute	valE ← valB + valC	Compute effective address
irmovq	Execute	valE ← valB + valC	Pass valC through ALU
pushq	Execute	valE ← valB + (-8)	Decrement stack pointer
popq	Execute	valE ← valB + 8	Increment stack pointer
cmovXX	Execute	valE ← valB + valA	Pass valA through ALU
jXX	Execute		
call	Execute	valE ← valB + (-8)	Decrement stack pointer
ret	Execute	valE ← valB + 8	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 instruction don't need ALU
];
```

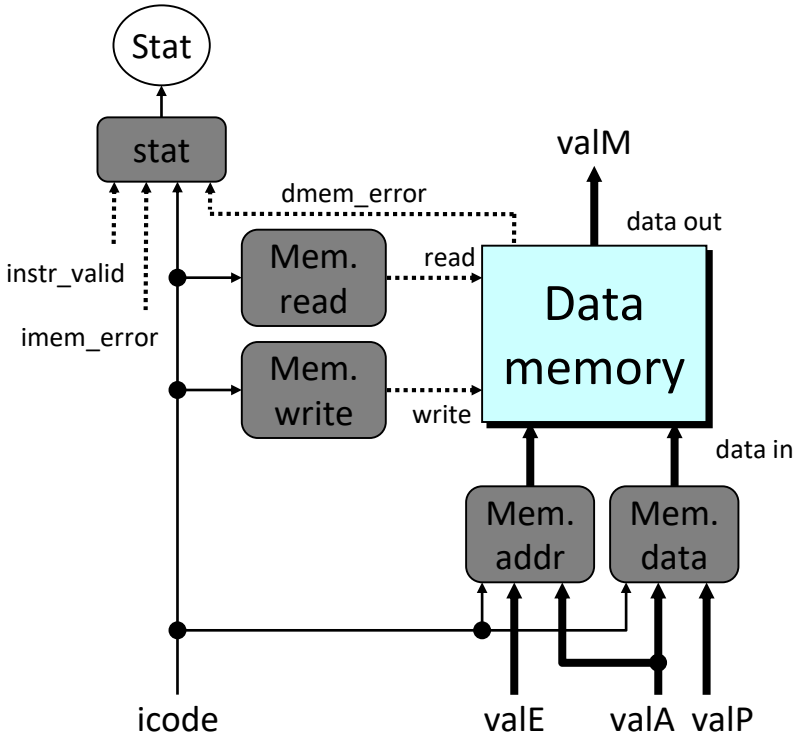
Execute Logic: alufun in HCL

OPq	Execute	valE ← valB OP valA	Perform ALU operation
rmmovq	Execute	valE ← valB + valC	Compute effective address
mrmovq	Execute	valE ← valB + valC	Compute effective address
irmovq	Execute	valE ← valB + valC	Pass valC through ALU
pushq	Execute	valE ← valB + -8	Decrement stack pointer
popq	Execute	valE ← valB + 8	Increment stack pointer
cmovXX	Execute	valE ← valB + valA	Pass valA through ALU
jXX	Execute		
call	Execute	valE ← valB + -8	Decrement stack pointer
ret	Execute	valE ← valB + 8	Increment stack pointer

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

Memory Logic

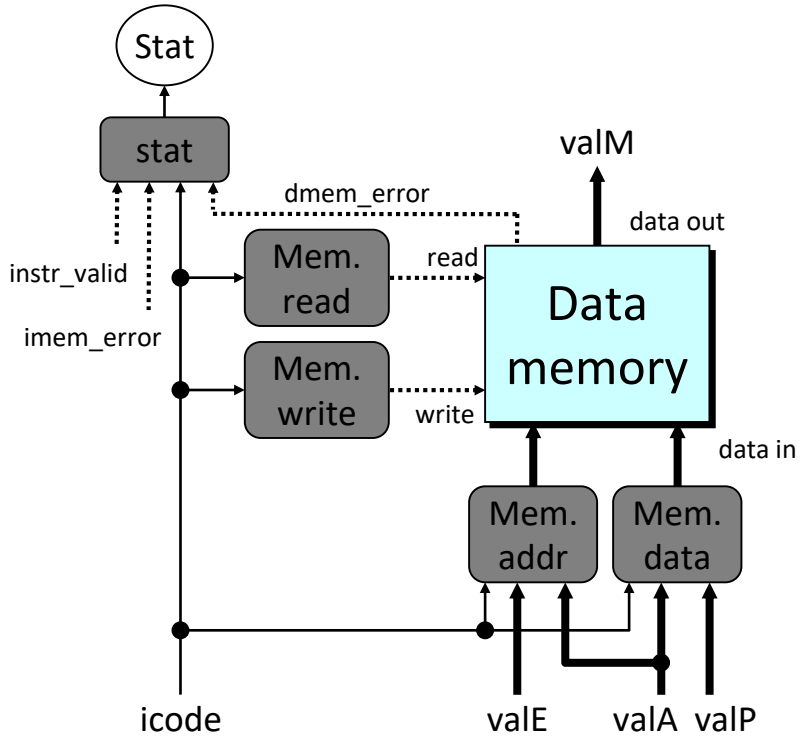
- Memory
  - Reads or writes memory word
- Control logic
  - `stat`: What is instruction status?
  - `Mem. read`: Should word be read?
  - `mem. write`: Should word be written?
  - `Mem. addr`: Select address
  - `Mem. data`: Select data



Memory Logic: stat in HCL

- Stat
  - What is instruction status?

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



Memory Logic: mem_addr in HCL

OPq	Memory		
rmmovq	Memory	M_8[valE] ← valA	Write value to memory
mrmovq	Memory	valM ← M_8[valE]	Read value from memory
irmovq	Memory		
pushq	Memory	M_8[valE] ← valA	Write to stack
popq	Memory	valM ← M_8[valA]	Read from stack
cmovXX	Memory		
jXX	Memory		
call	Memory	M_8[valE] ← valP	Update stack pointer
ret	Memory	valM ← M_8[valA]	Update stack pointer

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

Memory Logic: mem_read in HCL

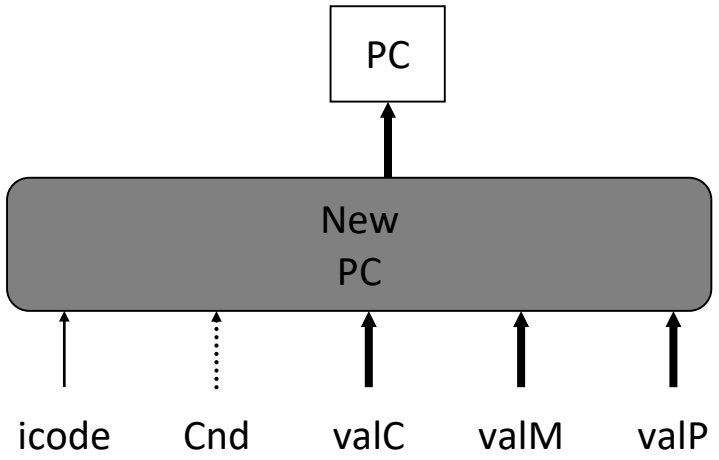
OPq	Memory		
rmmovq	Memory	M_8[valE] ← valA	Write value to memory
mrmovq	Memory	valM ← M_8[valE]	Read value from memory
irmovq	Memory		
pushq	Memory	M_8[valE] ← valA	Write to stack
popq	Memory	valM ← M_8[valA]	Read from stack
cmovXX	Memory		
jXX	Memory		
call	Memory	M_8[valE] ← valP	Update stack pointer
ret	Memory	valM ← M_8[valA]	Update stack pointer

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

PC Update Logic

- New PC
  - Select next value of PC
  - One of `valC`, `valM`, or `valP`

  - `valC`: from Instruction constant
    - `call`, `jXX`
  - `valM`: from Data memory
    - `ret`
  - `valP`: from next PC computed



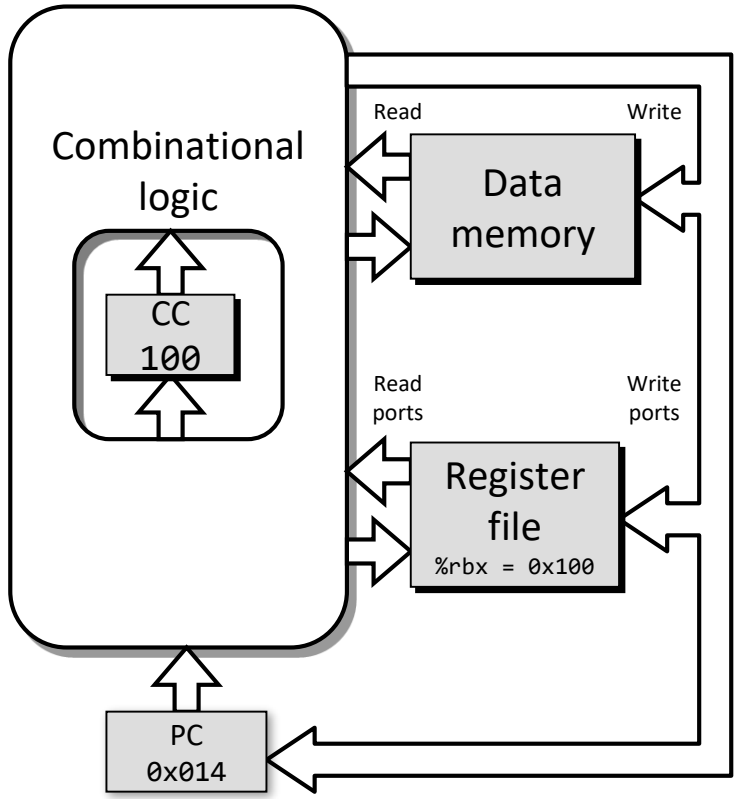
PC Update Logic: new_pc in HCL

OPq	PC Update	PC ← valP	Update PC
rmmovq	PC Update	PC ← valP	Update PC
mrmovq	PC Update	PC ← valP	Update PC
irmovq	PC Update	PC ← valP	Update PC
pushq	PC Update	PC ← valP	Update PC
popq	PC Update	PC ← valP	Update PC
cmovXX	PC Update	PC ← valP	Update PC
jXX	PC Update	PC ← Cnd? valC : valP	Update PC
call	PC Update	PC ← valC	Update PC
ret	PC Update	PC ← valM	Update PC

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

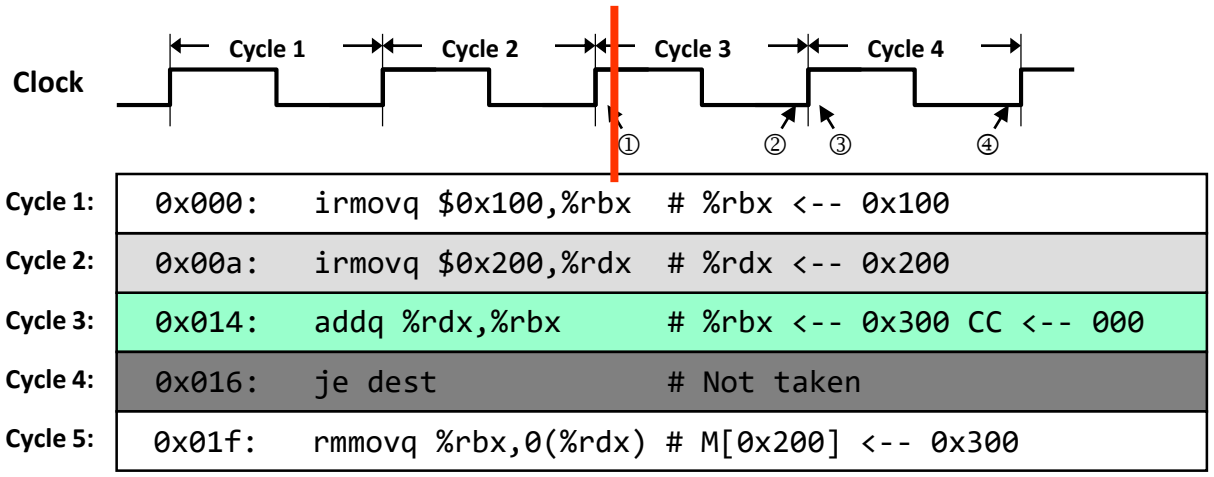
SEQ Operation (1)

- State (all updated as clock rises)
  - PC register
  - Cond. Code register
  - Data memory
  - Register file
- Combinational logic
  - ALU
  - Control logic
  - Memory reads
    - Instruction memory
    - Register file
    - Data memory

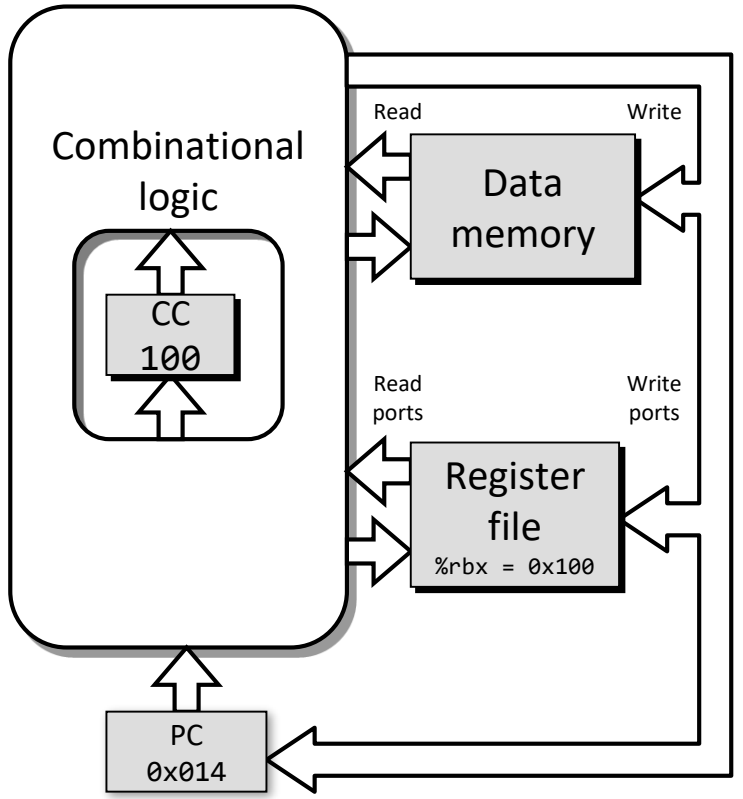


SEQ Operation (2)

- State set according to second `irmovq` instruction
- Combinational logic starting to react to state changes

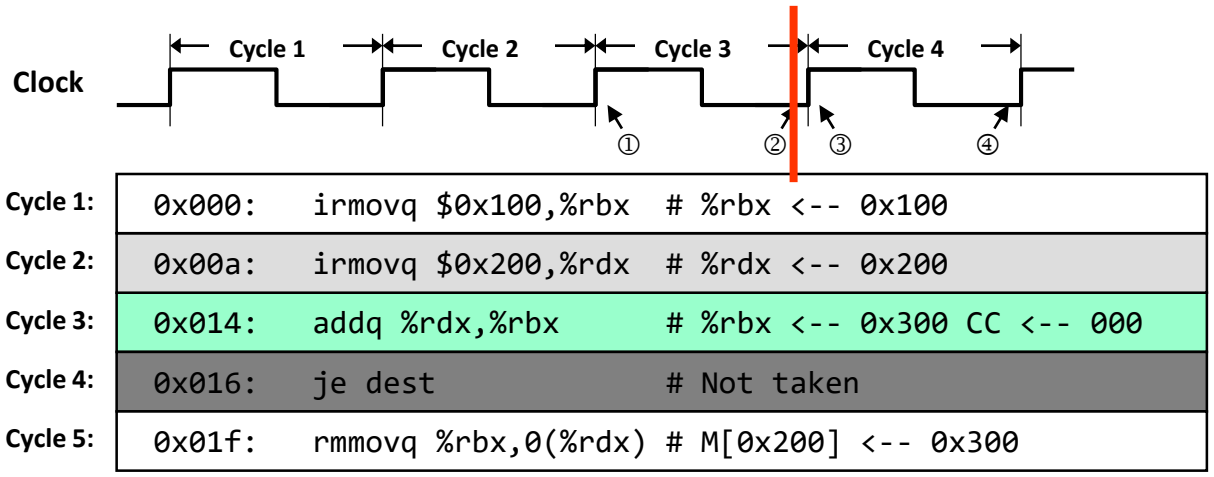


Cycle 1:	0x000:	irmovq $0x100,%rbx	# %rbx <-- 0x100
Cycle 2:	0x00a:	irmovq $0x200,%rdx	# %rdx <-- 0x200
Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	je dest	# Not taken
Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300

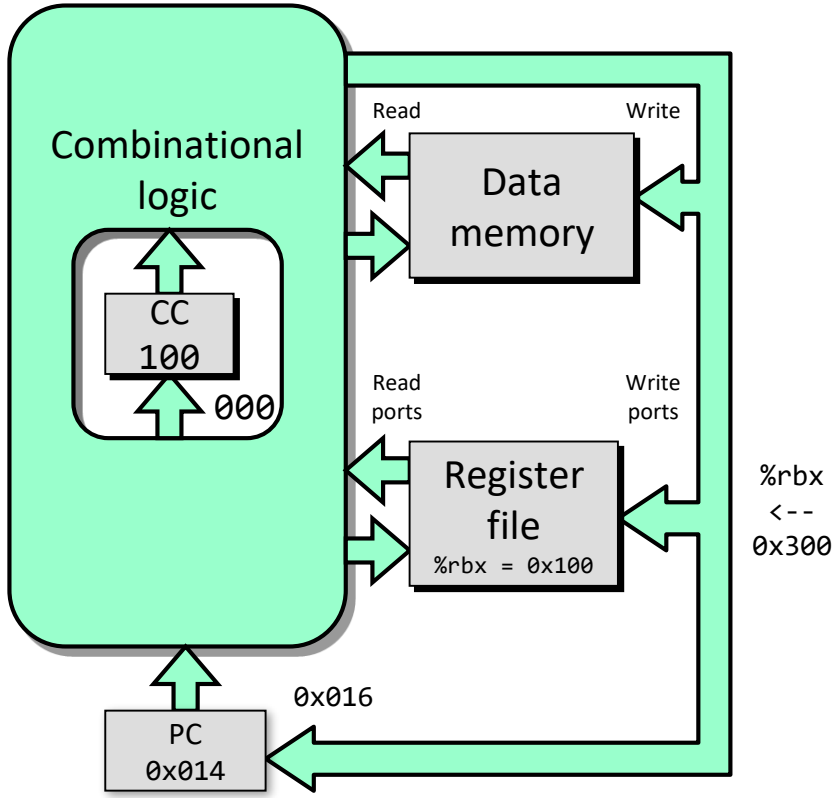


SEQ Operation (3)

- State set according to second `irmovq` instruction
- Combinational logic generates results for addq instruction

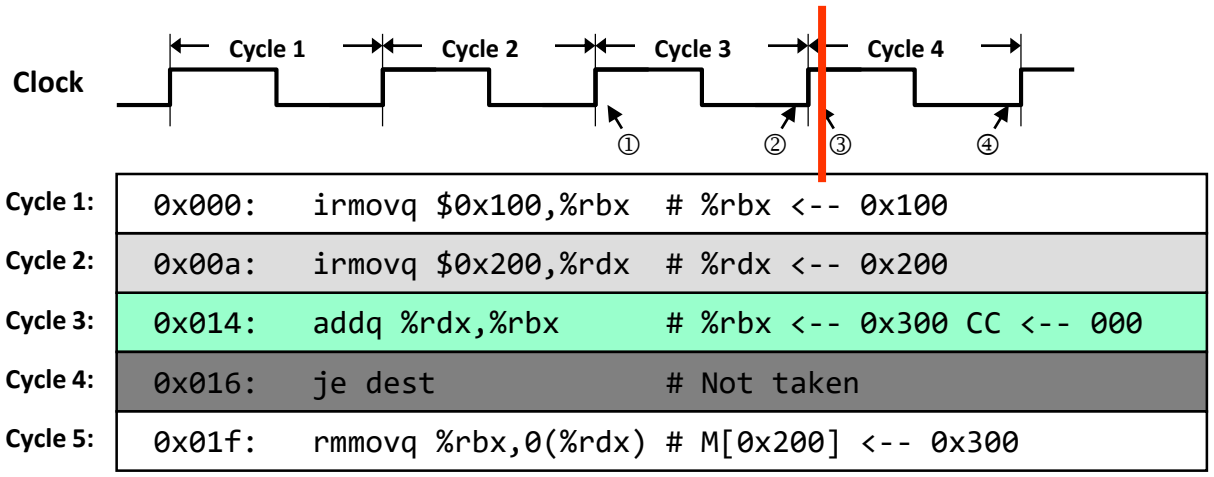


Cycle 1:	0x000:	irmovq $0x100,%rbx	# %rbx <-- 0x100
Cycle 2:	0x00a:	irmovq $0x200,%rdx	# %rdx <-- 0x200
Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	je dest	# Not taken
Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300

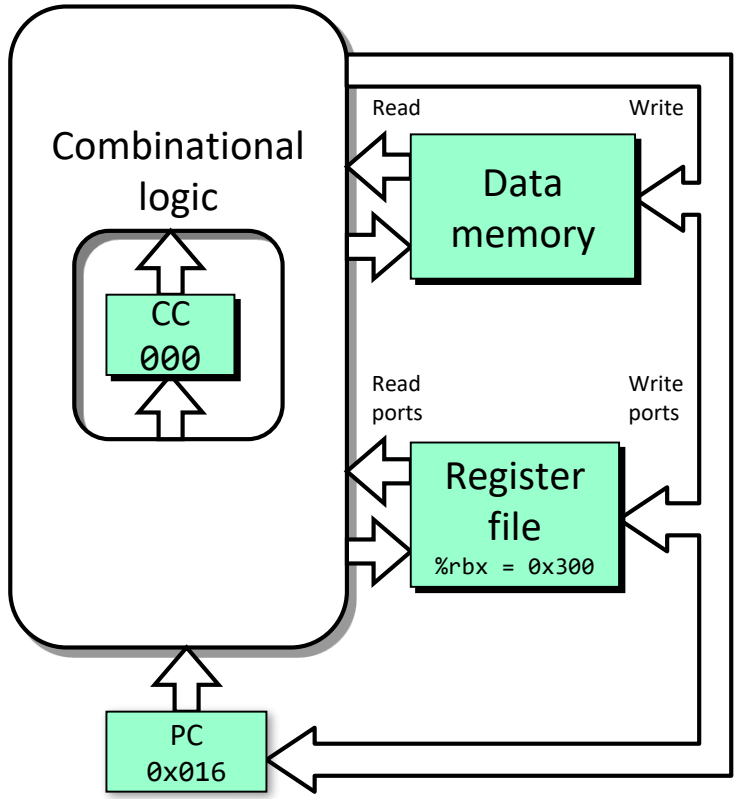


SEQ Operation (4)

- State set according to addq instruction
- Combinational logic starting to react to state changes

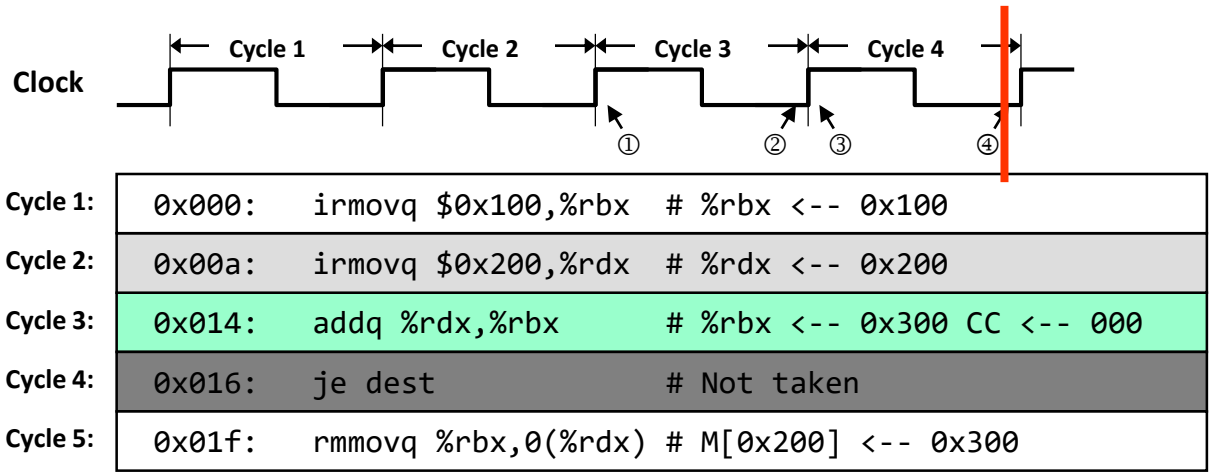


Cycle 1:	0x000:	irmovq $0x100,%rbx	# %rbx <-- 0x100
Cycle 2:	0x00a:	irmovq $0x200,%rdx	# %rdx <-- 0x200
Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	je dest	# Not taken
Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300

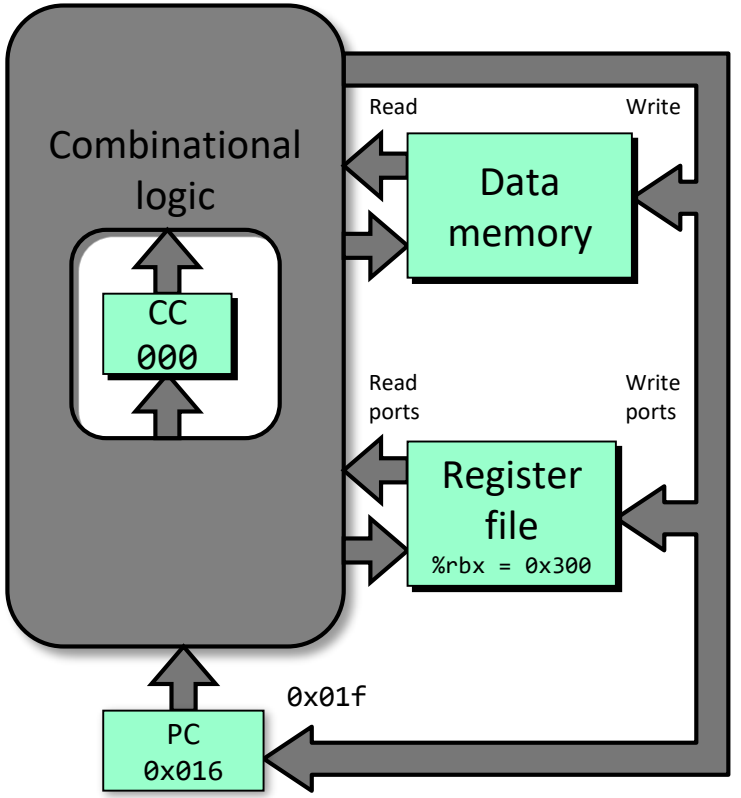


SEQ Operation (5)

- State set according to addq instruction
- Combinational logic generates results for `je` instruction



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



SEQ Summary

- Implementation
  - Express every instruction as series of simple steps
  - Follow same general flow for each instruction type
  - Assemble registers, memories, predesigned combinational blocks
  - Connect with control logic
- Limitations
  - Too slow to be practical
  - In one cycle, must propagate through instruction memory, register file, ALU, and data memory
  - Would need to run clock very slowly
  - Hardware units only active for fraction of clock cycle