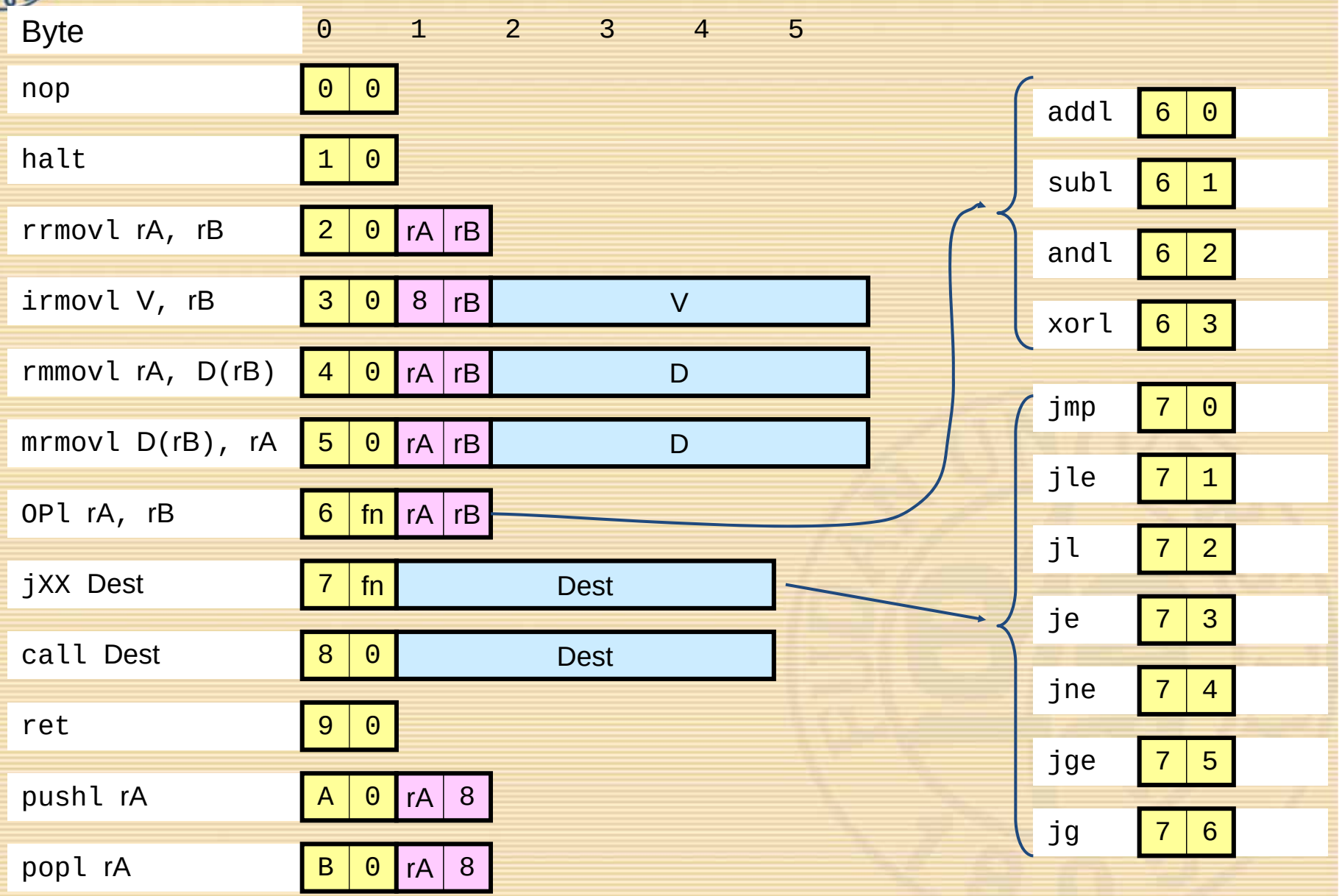
Sequential CPU Implementation

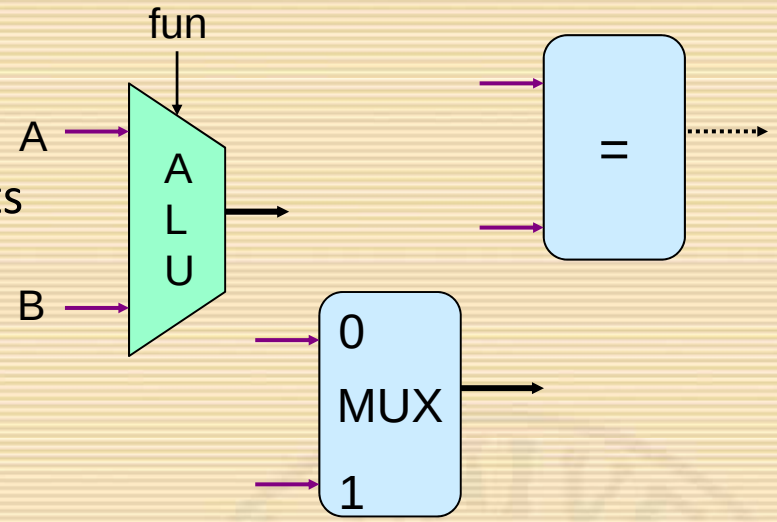
jc@fudan.edu.cn

Y86 Instruction Set P259

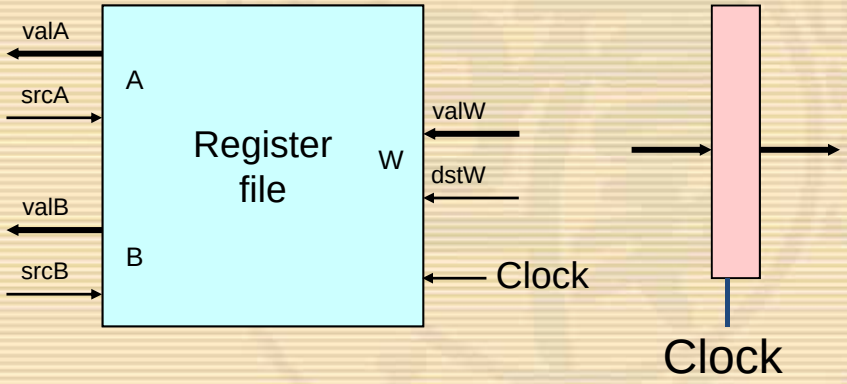


Building Blocks P278, P279, P280

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



- Storage Elements
  - Store bits
  - Addressable memories
  - Non-addressable registers
  - Loaded only as clock rises

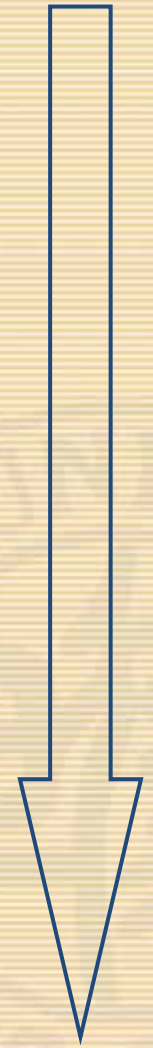


How function works?

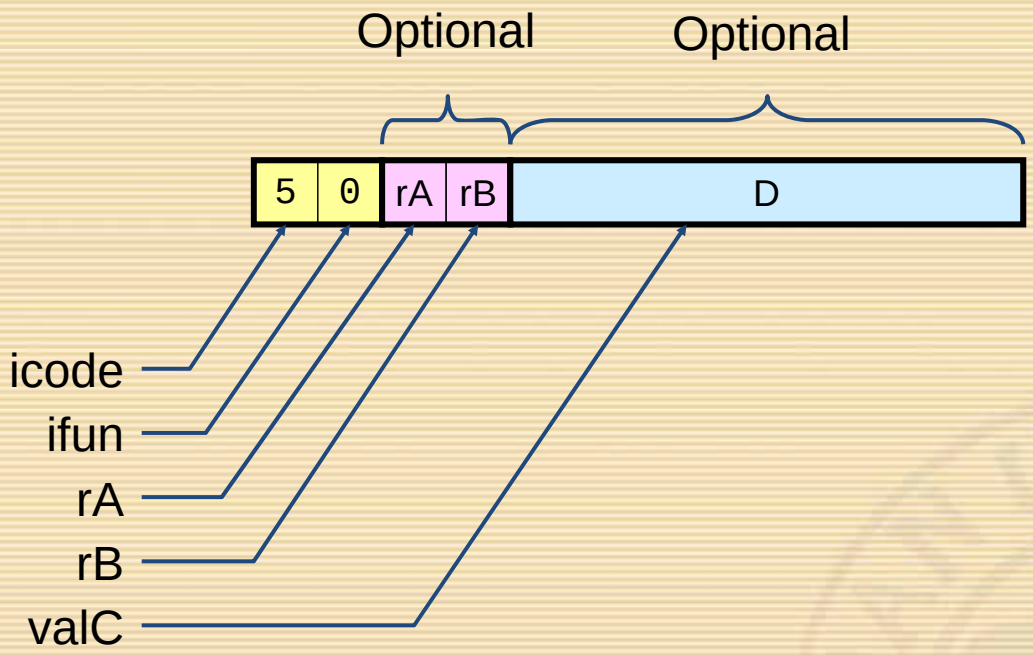
- Operations
- Movs
- Push/Pop
- Jump
- Call

Instruction Execution 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 program counter



Instruction Decoding



- Instruction Format
  - Instruction byte — icode:ifun
  - Optional register byte — rA:rB
  - Optional constant word — valC

Executing Arith./Logical Operation

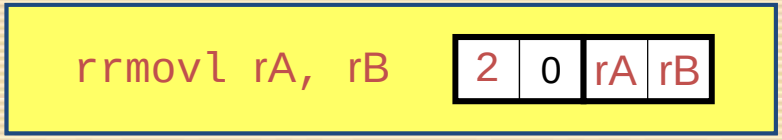
- Fetch
  - Read 2 bytes
- Decode
  - Read operand registers
- Execute
  - Perform operation
  - Set condition codes
- Memory
  - Do nothing
- Write back
  - Update register
- PC Update
  - Increment PC by 2

Stage Computation: Arith/Log. Ops

	OPl rA, rB	
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[rA]	Read operand A
	valB ← R[rB]	Read operand B
Execute	valE ← valB OP valA	Perform ALU operation
	Set CC	Set condition code register
Memory		
Write back	R[rB] ← valE	Write back result
PC update	PC ← valP	Update PC

- Formulate instruction execution as sequence of simple steps
- Use same general form for all instructions

Executing `rrmovl`



- Fetch
  - Read 2 bytes
- Decode
  - Read operand register rA
- Execute
  - Do nothing
- Memory
  - Do nothing
- Write back
  - Update register
- PC Update
  - Increment PC by 2

Stage Computation: `rrmovl`

	rrmovl 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]	Read operand A
Execute	valE ← 0 + valA	Perform ALU operation *valE
Memory		
Write back	R[rB] ← valE	Write back result
PC update	PC ← valP	Update PC

- Formulate instruction execution as sequence of simple steps
- Use same general form for all instructions

Executing `irmovl`

`irmovl V, rB`

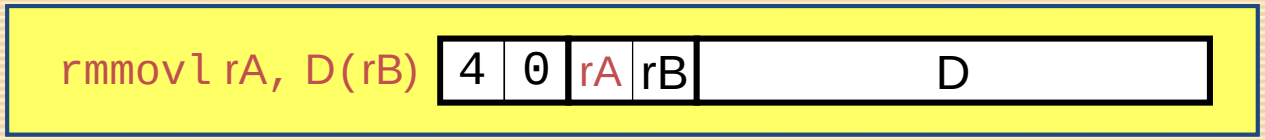
- Fetch
  - Read 6 bytes
- Decode
  - Do nothing
- Execute
  - Do nothing
- Memory
  - Do nothing
- Write back
  - Update register
- PC Update
  - Increment PC by 6

Stage Computation: `irmovl`

	irmovl V, rB	
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_4[PC+2] valP ← PC+6	Read instruction byte Read register byte Read constant value Compute next PC
Decode		
Execute	valE ← 0 + valC	Perform ALU operation
Memory		
Write back	R[rB] ← valE	Write back result
PC update	PC ← valP	Update PC

- Formulate instruction execution as sequence of simple steps
- Use same general form for all instructions

Executing `rmmovl`



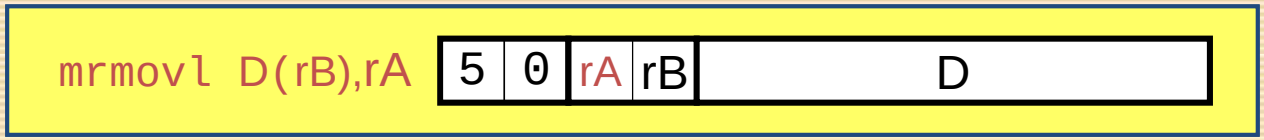
- Fetch
  - Read 6 bytes
- Decode
  - Read operand registers
- Execute
  - Compute effective address
- Memory
  - Write to memory
- Write back
  - Do nothing
- PC Update
  - Increment PC by 6

Stage Computation: `rmmovl`

	`rmmovl` rA, D(rB)	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	rA:rB ← M_1[PC+1]	Read register byte
	valC ← M_4[PC+2]	Read displacement D
	valP ← PC+6	Compute next PC
Decode	valA ← R[rA]	Read operand A
	valB ← R[rB]	Read operand B
Execute	valE ← valB + valC	Compute effective address (sum of the displacement and the base register value)
Memory	M_4[valE] ← valA	Write value to memory
Write back		
PC update	PC ← valP	Update PC

– Use ALU for address computation

Executing `mrmovl`



- Fetch
  - Read 6 bytes
- Decode
  - Read operand register rB
- Execute
  - Compute effective address
- Memory
  - Read from memory
- Write back
  - Update register rA
- PC Update
  - Increment PC by 6

Stage Computation: `mrmovl`

	`mrmovl` D(rB) , rA	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	rA:rB ← M_1[PC+1]	Read register byte
	valC ← M_4[PC+2]	Read displacement D
	valP ← PC+6	Compute next PC
Decode	valB ← R[rB]	Read operand B
Execute	valE ← valB + valC	Compute effective address
Memory	valM ← M_4[valE]	Read data from memory
Write back	R[rA] ← valM	Update register rA
PC update	PC ← valP	Update PC

– Use ALU for address computation

Executing pushl

- Fetch
  - Read 2 bytes
- Decode
  - Read stack pointer and rA
- Execute
  - Decrement stack pointer by 4
- Memory
  - Store valA at the address of new stack pointer
- Write back
  - Update stack pointer
- PC Update
  - Increment PC by 2

Stage Computation: `pushl`

	`pushl` rA	
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 ← R [%esp]	Read valA Read stack pointer
Execute	valE ← valB + (-4)	Decrement stack pointer
Memory	M_4[valE] ← valA	Store to stack
Write back	R[%esp] ← valE	Update stack pointer *在write back之前实际上写入的元素在堆栈外。
PC update	PC ← valP	Update PC

– Use ALU to Decrement stack pointer

Executing popl

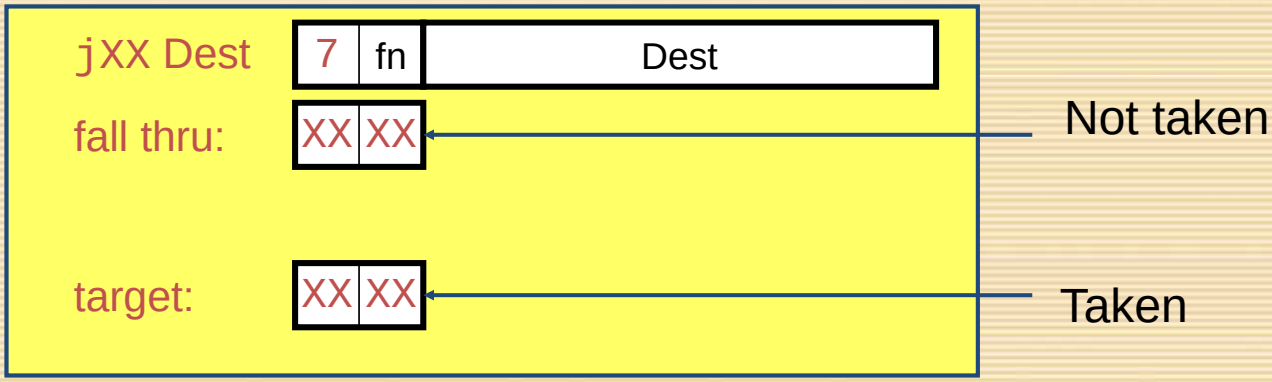
- Fetch
  - Read 2 bytes
- Decode
  - Read stack pointer
- Execute
  - Increment stack pointer by 4
- Memory
  - Read from old stack pointer
- Write back
  - Update stack pointer
  - Write result to register
- PC Update
  - Increment PC by 2

Stage Computation: popl

	popl rA	
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[%esp] valB ← R [%esp]	Read stack pointer Read stack pointer
Execute	valE ← valB + 4	Increment stack pointer
Memory	valM ← M_4[valA]	Read from stack
Write back	R[%esp] ← valE R[rA] ← valM	Update stack pointer Write back result
PC update	PC ← valP	Update PC

- Use ALU to increment stack pointer
- Must update two registers
  - Popped value
  - New stack pointer

Executing Jumps



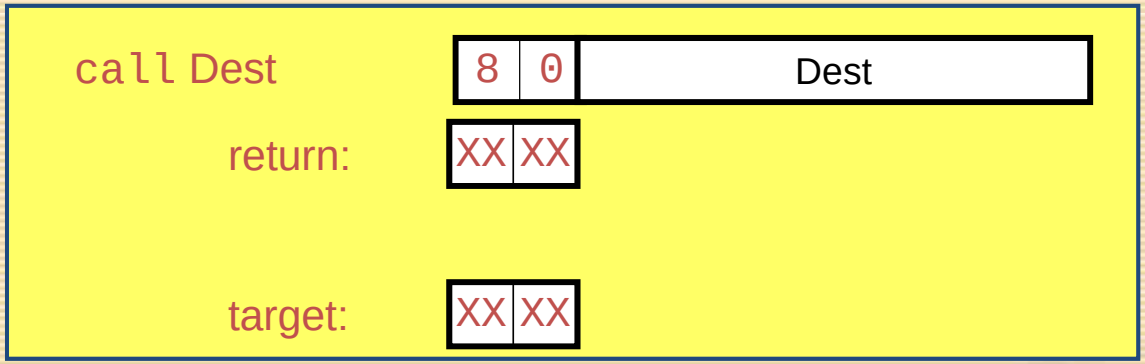
- Fetch
  - Read 5 bytes
  - Increment PC by 5
- Decode
  - Do nothing
- Execute
  - Determine whether to take branch based on jump condition and condition codes
- Memory
  - Do nothing
- Write back
  - Do nothing
- PC Update
  - Set PC to Dest if branch taken or to incremented PC if not branch

Stage Computation: Jumps

	jXX Dest	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	valC ← M_4[PC+1]	Read destination address
	valP ← PC+5	Fall through address
Decode		
Execute	Bch ← Cond(CC,ifun)	Take branch?
Memory		
Write back		
PC update	PC ← Bch ? valC : valP	Update PC

- Compute both addresses
- Choose based on setting of condition codes and branch condition

Executing `call`



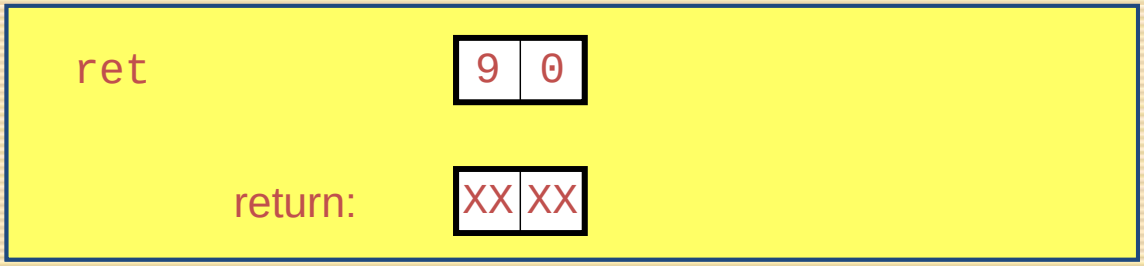
- Fetch
  - Read 5 bytes
  - Increment PC by 5
- Decode
  - Read stack pointer
- Execute
  - Decrement stack pointer by 4
- Memory
  - Write incremented PC to new value of stack pointer
- Write back
  - Update stack pointer
- PC Update
  - Set PC to Dest

Stage Computation: `call`

	`call` Dest	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	valC ← M_4[PC+1]	Read destination address
	valP ← PC+5	Compute return point
Decode	valB ← R[%esp]	Read stack pointer
Execute	valE ← valB + –4	Decrement stack pointer
Memory	M_4[valE] ← valP	Write return value on stack
Write back	R[%esp] ← valE	Update stack pointer
PC update	PC ← valC	Set PC to destination

- Use ALU to decrement stack pointer
- Store incremented PC

Executing `ret`



- Fetch
  - Read 1 byte
- Decode
  - Read stack pointer
- Execute
  - Increment stack pointer by 4
- Memory
  - Read return address from old stack pointer
- Write back
  - Update stack pointer
- PC Update
  - Set PC to return address

Stage Computation: `ret`

	`ret`	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
Decode	valA ← R[%esp] valB ← R[%esp]	Read operand stack pointer Read operand stack pointer
Execute	valE ← valB + 4	Increment stack pointer
Memory	valM ← M_4[valA]	Read return address
Write back	R[%esp] ← valE	Update stack pointer
PC update	PC ← valM	Set PC to return address

- Use ALU to increment stack pointer
- Read return address from memory

Computation Steps

		OPl rA, rB	
Fetch	icode,ifun	icode:ifun ← M_1[PC]	Read instruction byte
	rA,rB	rA:rB ← M_1[PC+1]	Read register byte
	valC		[Read constant word]
	valP	valP ← PC+2	Compute next PC
Decode	valA, srcA	valA ← R[rA]	Read operand A
	valB, srcB	valB ← R[rB]	Read operand B
Execute	valE	valE ← valB OP valA	Perform ALU operation
	Cond code	Set CC	Set condition code register
Memory	valM		[Memory read/write]
Write back	dstE	R[rB] ← valE	Write back ALU result
	dstM		[Write back memory result]
PC update	PC	PC ← valP	Update PC

- All instructions follow same general pattern
- Differ in what gets computed on each step

Computation Steps

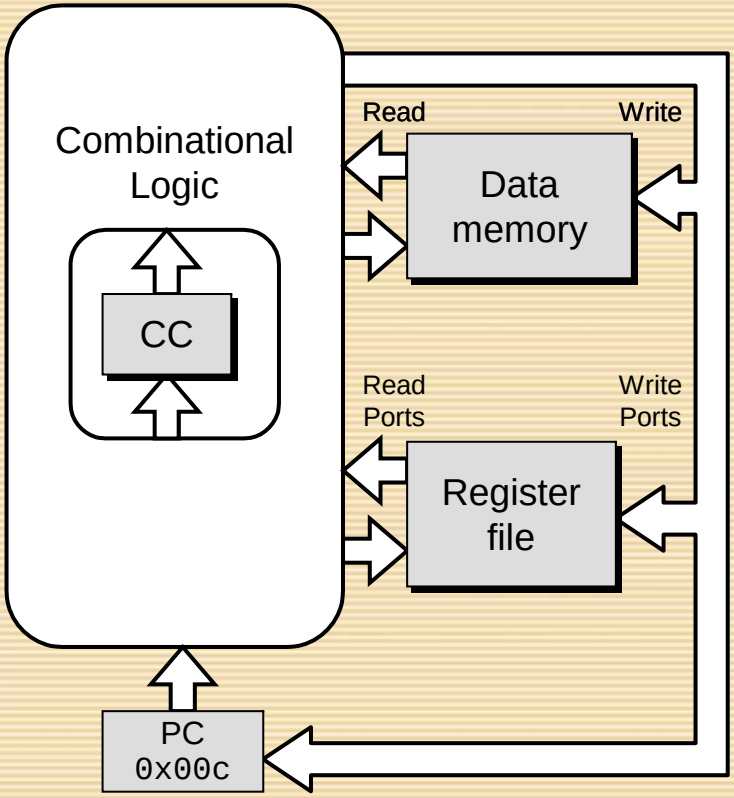
		`call` Dest	
Fetch	icode,ifun	icode:ifun ← M_1[PC]	Read instruction byte
	rA,rB		[Read register byte]
	valC	valC ← M_4[PC+1]	Read constant word
	valP	valP ← PC+5	Compute next PC
Decode	valA, srcA		[Read operand A]
	valB, srcB	valB ← R[%esp]	Read operand B
Execute	valE	valE ← valB + –4	Perform ALU operation
	Cond code		[Set condition code reg.]
Memory	valM	M_4[valE] ← valP	Memory read/write
Write back	dstE	R[%esp] ← valE	Write back ALU result
	dstM		[Write back memory result]
PC update	PC	PC ← valC	Update PC

- All instructions follow same general pattern
- Differ in what gets computed on each step

Computed Values

- Fetch
  - icode Instruction code
  - ifun Instruction function
  - rA Instr. Register A
  - rB Instr. Register B
  - valC Instruction constant
  - valP Incremented PC
- Decode § Write back
  - valA Register value A
  - valB Register value B
- Execute
  - valE ALU result
  - Bch Branch flag
- Memory
  - valM Value from memory

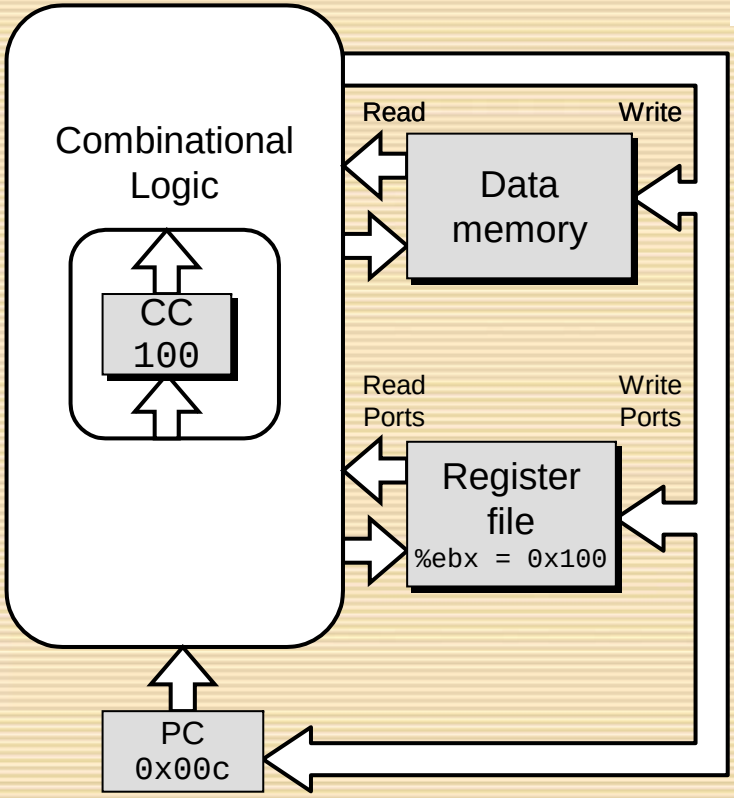
SEQ Operation P297

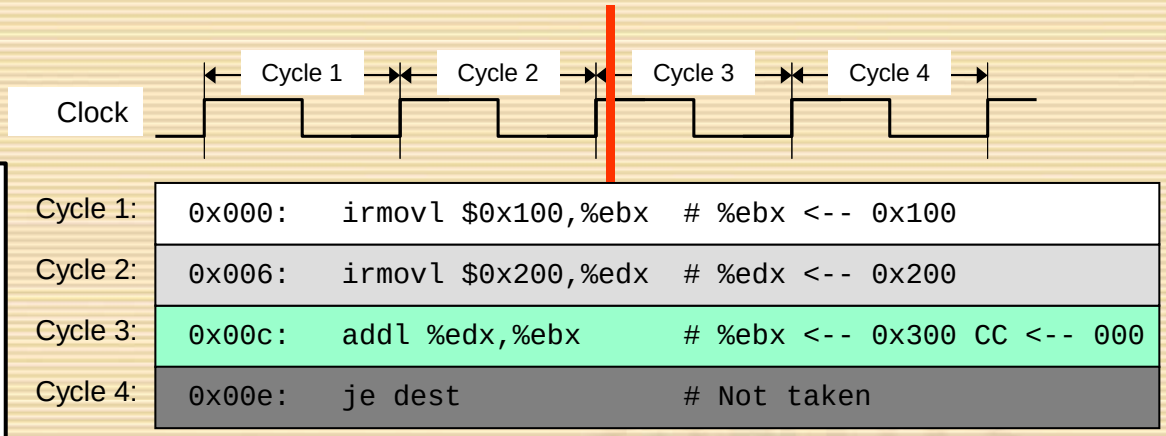


- State
  - Program counter register (PC)
  - Condition code register (CC)
  - Register File
  - Memories
    - Access same memory space
    - Data: for reading/writing program data
    - Instruction: for reading instructions

  All updated as clock rises
- Combinational Logic
  - ALU
  - Control logic
  - Memory reads
    - Instruction memory
    - Register file
    - Data memory

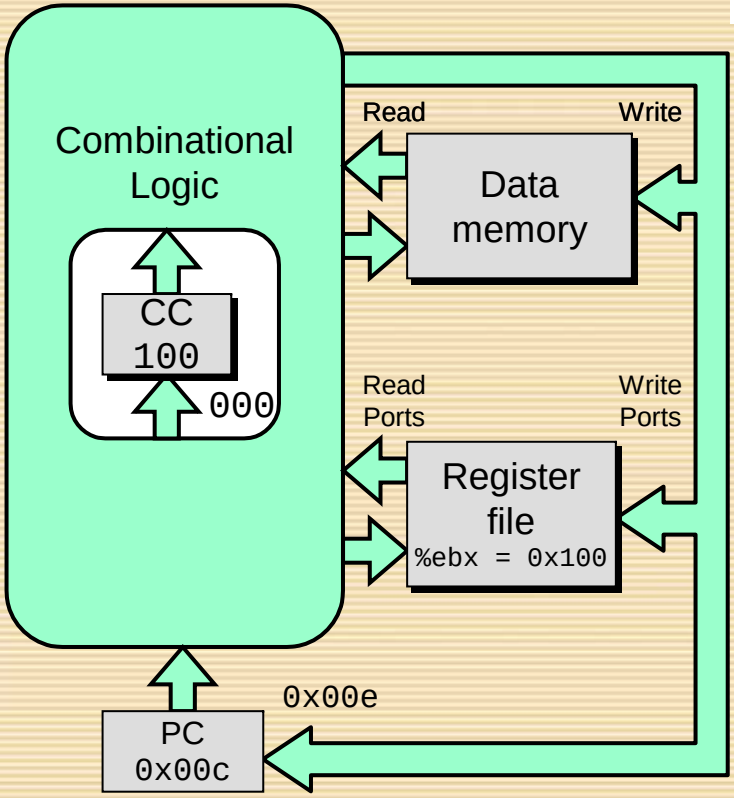
SEQ Operation

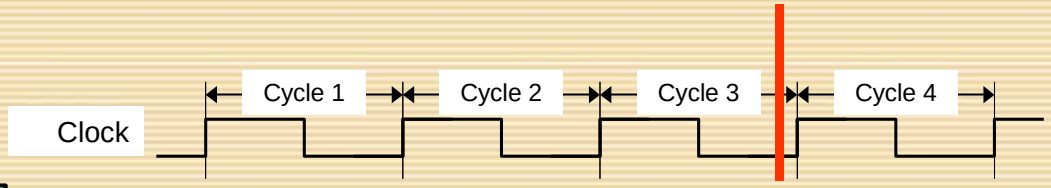




- state set according to second `irmovl` instruction
- combinational logic starting to react to state changes

SEQ Operation

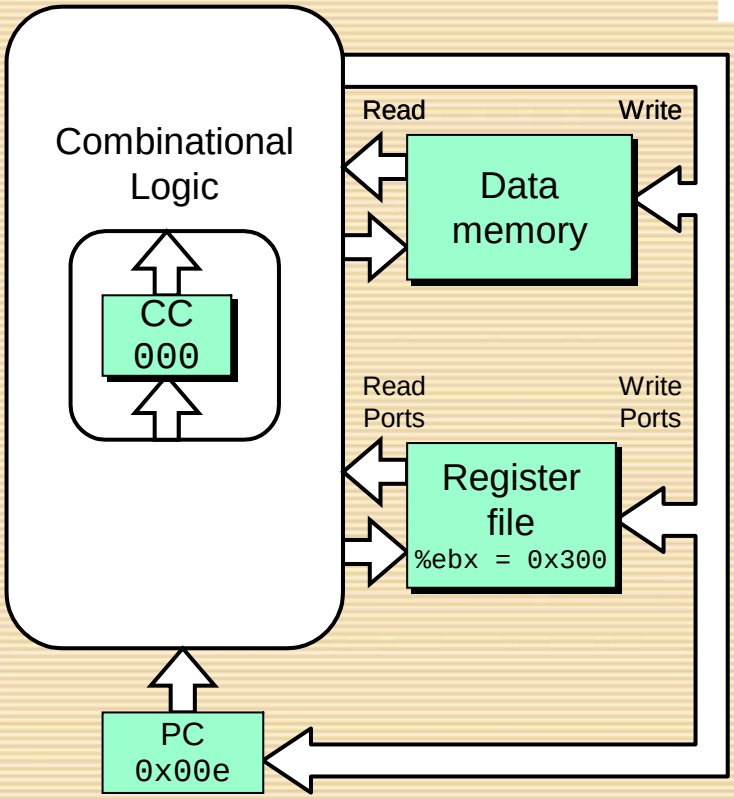


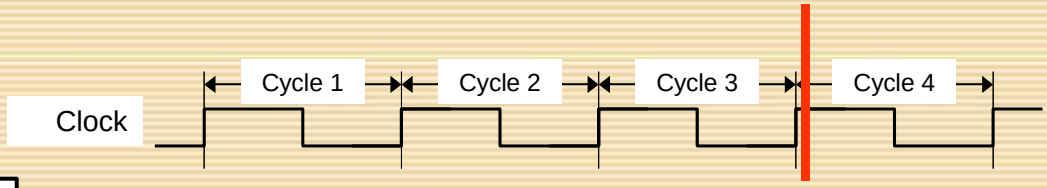


Cycle 1:	0x000:	irmovl $0x100,%ebx	# %ebx <-- 0x100
Cycle 2:	0x006:	irmovl $0x200,%edx	# %edx <-- 0x200
Cycle 3:	0x00c:	addl %edx,%ebx	# %ebx <-- 0x300 CC <-- 000
Cycle 4:	0x00e:	je dest	# Not taken

- state set according to second `irmovl` instruction
- combinational logic generates results for `addl` instruction

SEQ Operation

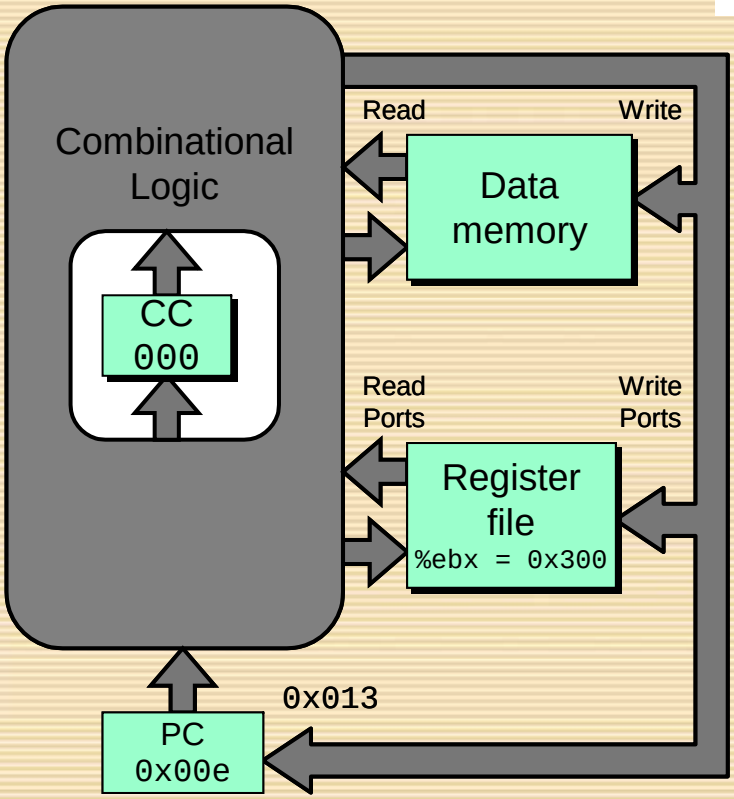


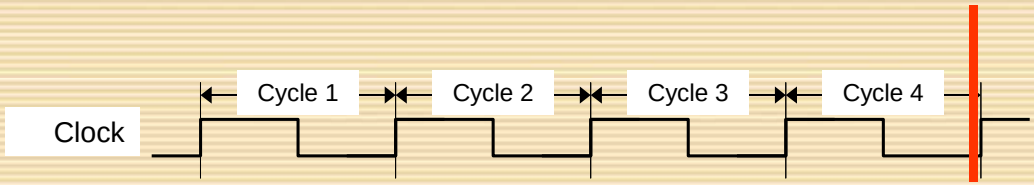


Cycle 1:	0x000:	irmovl $0x100,%ebx	# %ebx <-- 0x100
Cycle 2:	0x006:	irmovl $0x200,%edx	# %edx <-- 0x200
Cycle 3:	0x00c:	addl %edx,%ebx	# %ebx <-- 0x300 CC <-- 000
Cycle 4:	0x00e:	je dest	# Not taken

- state set according to `addl` instruction
- combinational logic starting to react to state changes

SEQ Operation





Cycle 1:	`0x000: irmovl $0x100,%ebx # %ebx <-- 0x100`
Cycle 2:	`0x006: irmovl $0x200,%edx # %edx <-- 0x200`
Cycle 3:	`0x00c: addl %edx,%ebx # %ebx <-- 0x300 CC <-- 000`
Cycle 4:	`0x00e: je dest # Not taken`

- state set according to `addl` instruction
- combinational logic generates results for `je` instruction

SEQ Semantics

- Achieve the same effect as a sequential execution of the assignment shown in the tables of Figures 4.16 to 4.19
  - Though all of the state updates occur simultaneously at the clock rises to the next cycle.
  - A problem: popl %esp need to sequentially write two registers. So the register file control logic must process it.
- Principle: never needs to read back the state updated by an instruction in order to complete the processing of this instruction (P295)
  - If so, the update must happen in the instruction cycle
  - E.g. pushl semantics
  - E.g. no one instruction need to both set and then read the condition codes.

SEQ CPU Implementation

jc@fudan.edu.cn

Main Topics

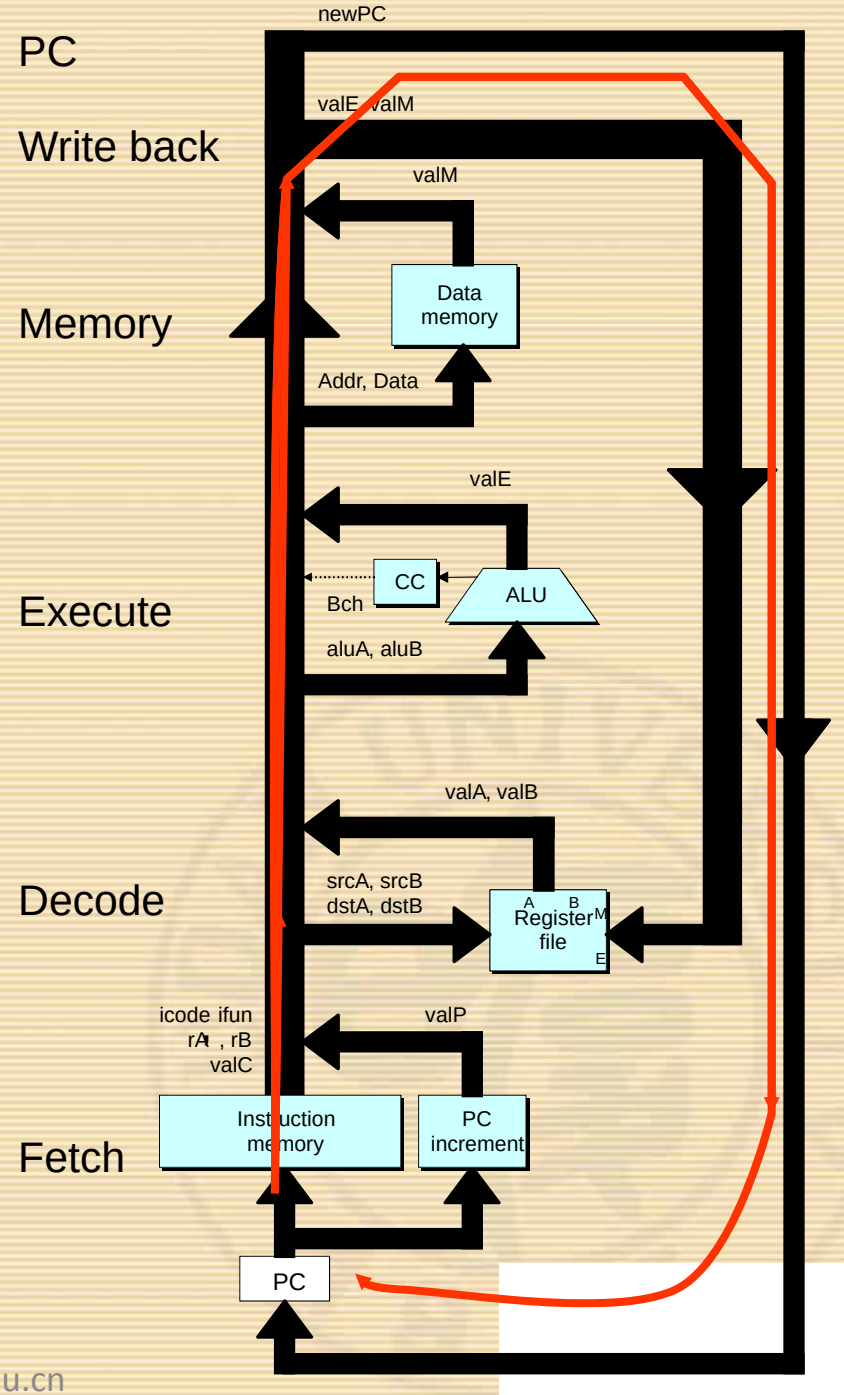
- The implementation of a sequential CPU ---- SEQ
  - Every Instruction finished in one cycle.
  - Instruction executes in sequential
  - No two instructions execute in parallel or overlap
- An revised version of SEQ ---- SEQ+
  - Modify the PC Update stage of SEQ
  - to show the difference between ISA and implementation

Some Macros P299

Name	Value	Meaning
INOP	0	Code for **`nop`** instruction
IHALT	1	Code for **`halt`** instruction
IRRMOVL	2	Code for **`rrmovl`** instruction
IIRMOVL	3	Code for **`irmovl`** instruction
IRMMOVL	4	Code for **`rmmovl`** instruction
IMRMOVL	5	Code for **`mrmovl`** instruction
IOPL	6	Code for integer op instructions
IJXX	7	Code for jump instructions
.....	.....	.....
IPOPL	B	Code for **`popl`** instruction
RESP	6	Register ID for %esp
RENONE	8	Indicates no register file access
ALUADD	0	Function for addition operation

SEQ Hardware Structure P292

- Stages
  - Fetch
    - Read instruction from memory
  - Decode
    - Read program registers
  - Execute
    - Compute value or address
  - Memory
    - Read or write data
  - Write Back
    - Write program registers
  - PC
    - Update program counter
- Instruction Flow
  - Read instruction at address specified by PC
  - Process through stages
  - Update program counter

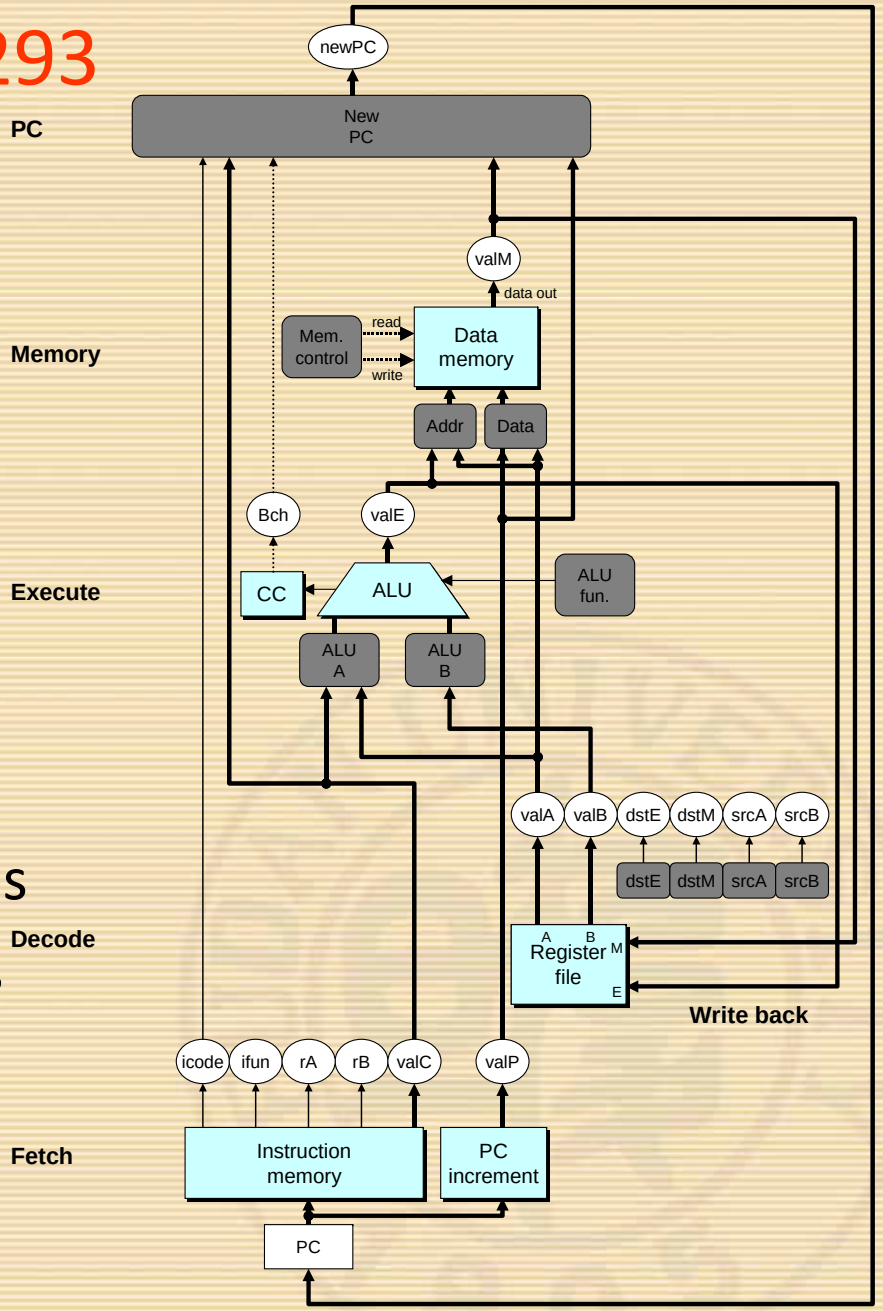


Difference between semantics and implementation

- ISA
  - Every stage may update some states, these updates occur sequentially
- SEQ
  - All the state update operations occur simultaneously at clock rising (except memory and CC)

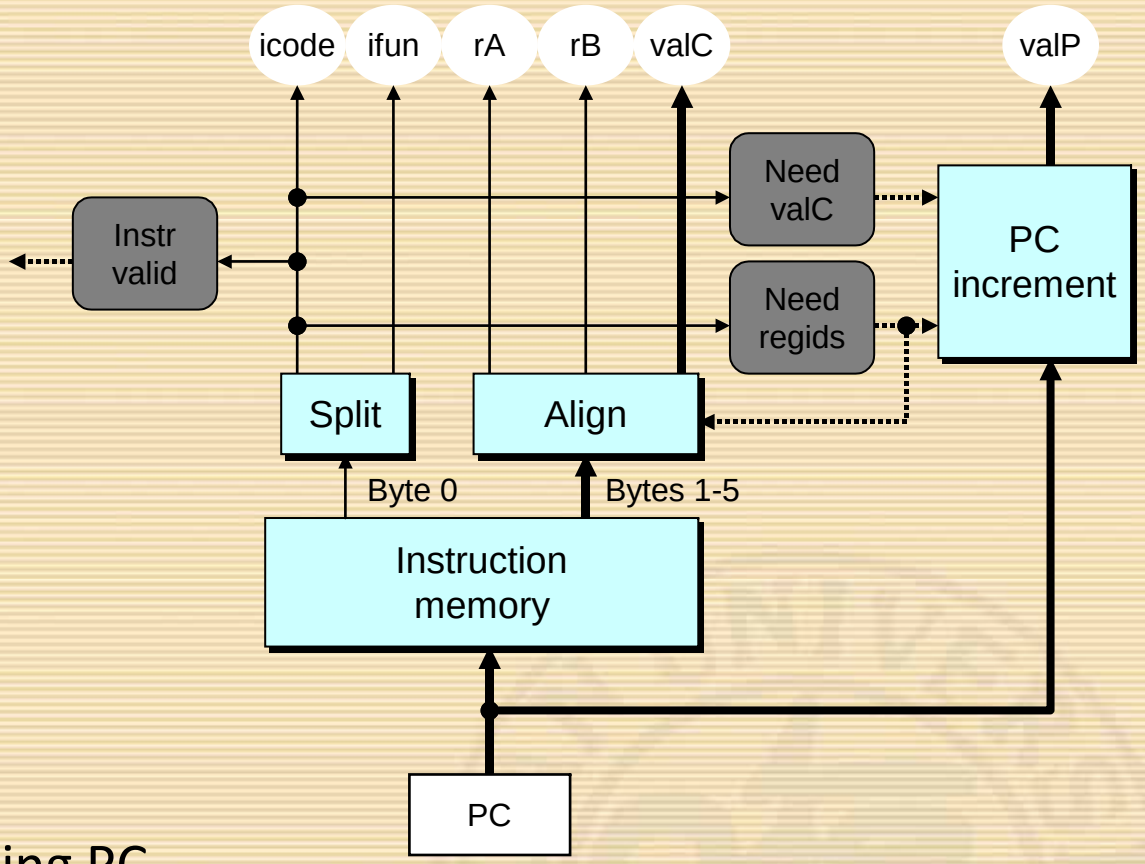
SEQ Hardware P293

- Key
  - Blue boxes: predesigned hardware blocks
    - E.g., memories, ALU
  - Gray boxes: control logic
    - Describe in HCL
  - White ovals : labels for signals
  - Thick lines: 32-bit word values
  - Thin lines: 4-8 bit values
  - Dotted lines: 1-bit values



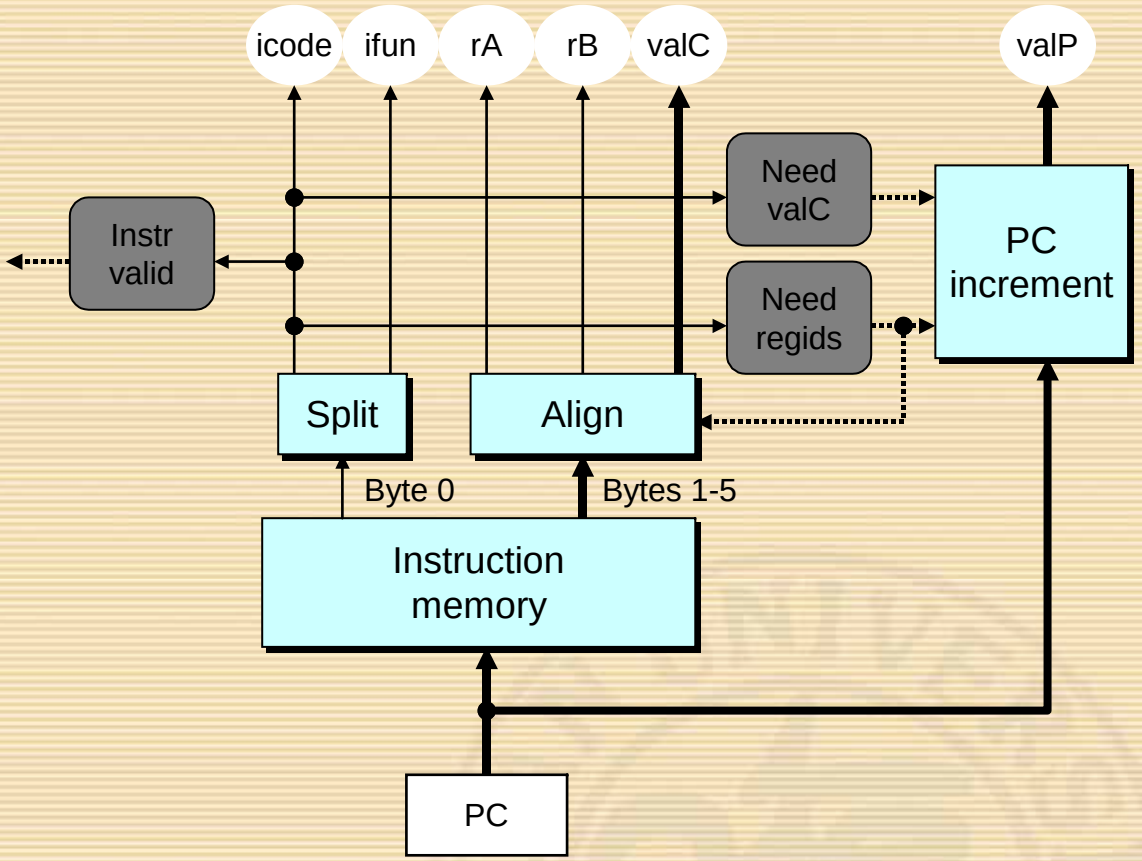
Fetch Logic

Figure 4.25 P299



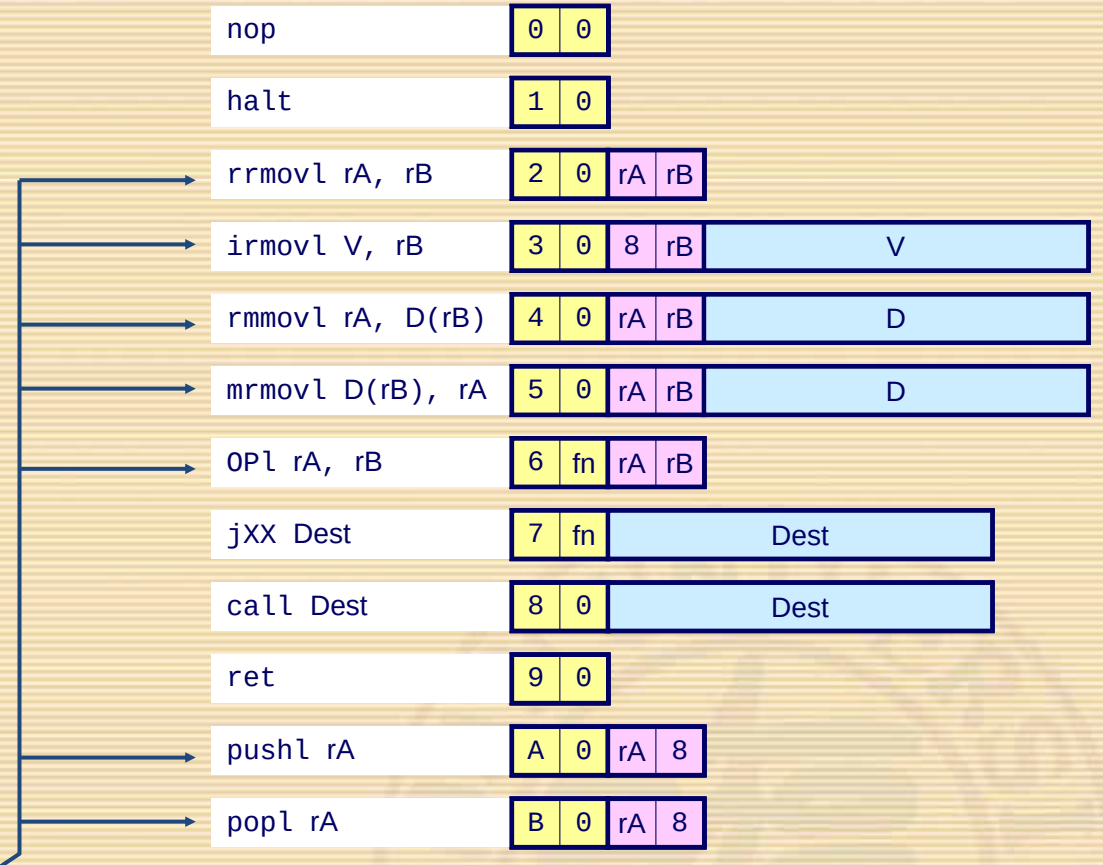
- Predefined Blocks
  - PC: Register containing PC
  - Instruction memory: Read 6 bytes (PC to PC+5)
  - Split: Divide instruction byte into icode and ifun
  - Align: Get fields for rA, rB, and valC

Fetch Logic



- Control Logic
  - Instr. Valid: Is this instruction valid?
  - Need regids: Does this instruction have a register bytes?
  - Need valC: Does this instruction have a constant word?

Fetch Control Logic P300

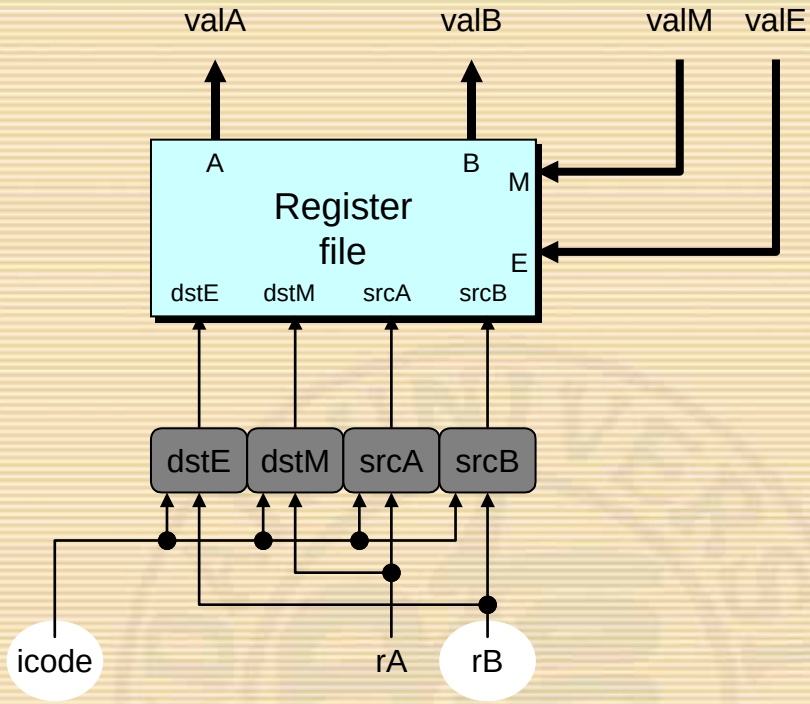



```
bool need_regids =
        icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
                   IIRMOVL, IRMMOVL, IMRMOVL };
bool instr_valid = icode in
        { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
          IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };
```

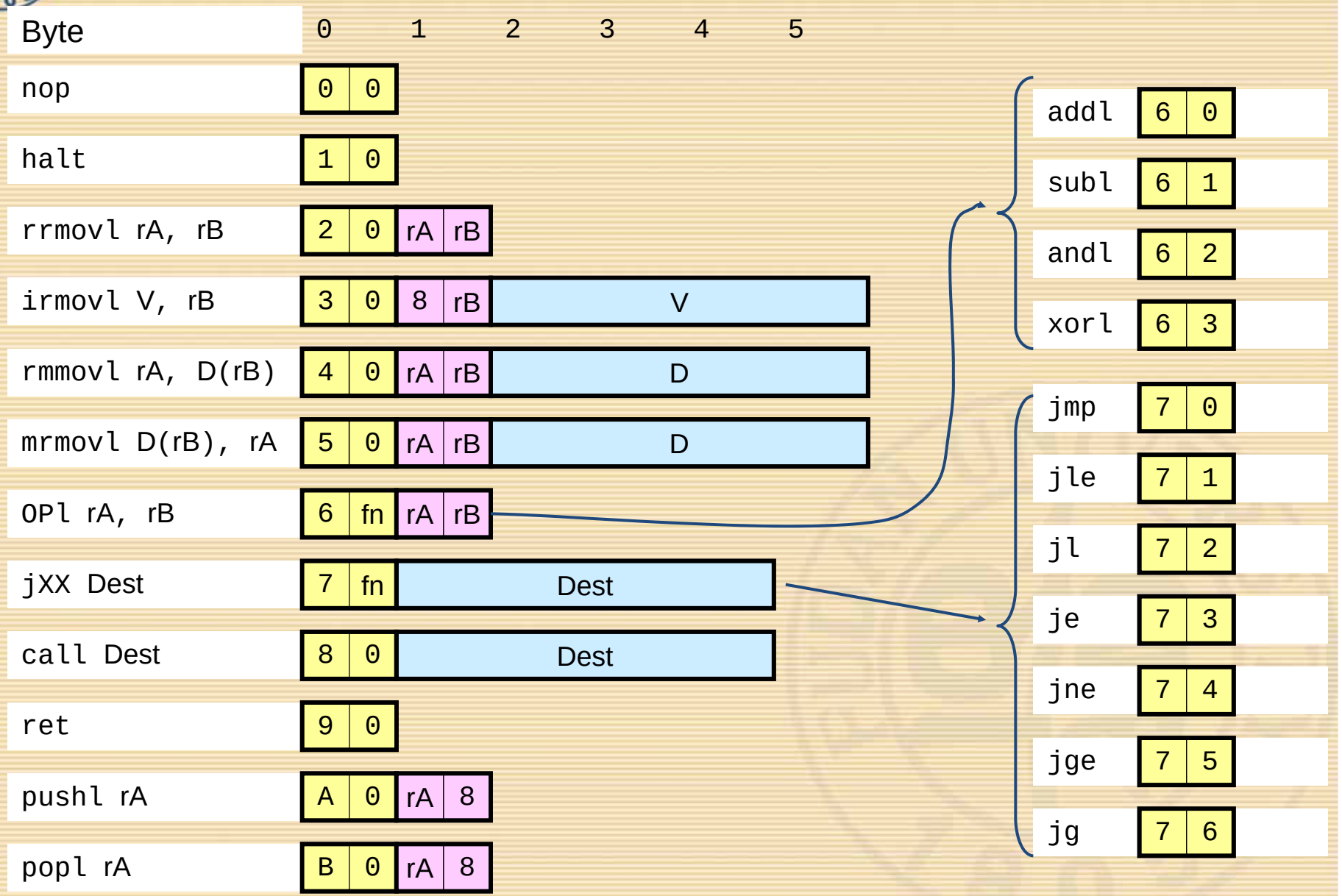
Decode & Write-Back Logic

- Register File
  - Read ports A, B
  - Write ports E, M
  - Addresses are register IDs or 8 (no access)
- Control Logic
  - srcA, srcB: read port addresses
  - dstE, dstM: write port addresses

Figure 4.26 P300



Y86 Instruction Set P259



A Source

	OPl rA, rB	
Decode	valA ← R[rA]	Read operand A
	`rmmovl` rA, D(rB)	
Decode	valA ← R[rA]	Read operand A
	`popl` rA	
Decode	valA ← R[%esp]	Read stack pointer
	jXX Dest	
Decode		No operand
	`call` Dest	
Decode		No operand
	`ret`	
Decode	valA ← R[%esp]	Read stack pointer

```
int srcA = [
        icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL  } : rA;
        icode in { IPOPL, IRET } : RESP;
        1 : RNONE; # Don't need register
];
```

E Destination

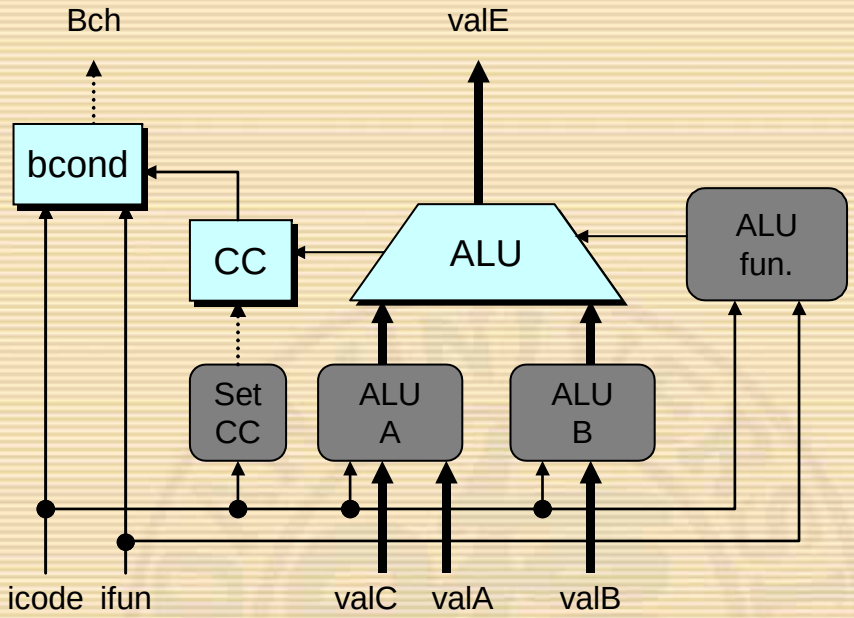
	OPl rA, rB	
Write-back	R[rB] ← valE	Write back result
	`rmmovl` rA, D(rB)	
Write-back		None
	`popl` rA	
Write-back	R[%esp] ← valE	Update stack pointer
	jXX Dest	
Write-back		None
	`call` Dest	
Write-back	R[%esp] ← valE	Update stack pointer
	`ret`	
Write-back	R[%esp] ← valE	Update stack pointer

```
int dstE = [
        icode in { IRRMOVL, IIRMOVL, IOPL} : rB;
        icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
        1 : RNONE;  # Don't need register
];
```

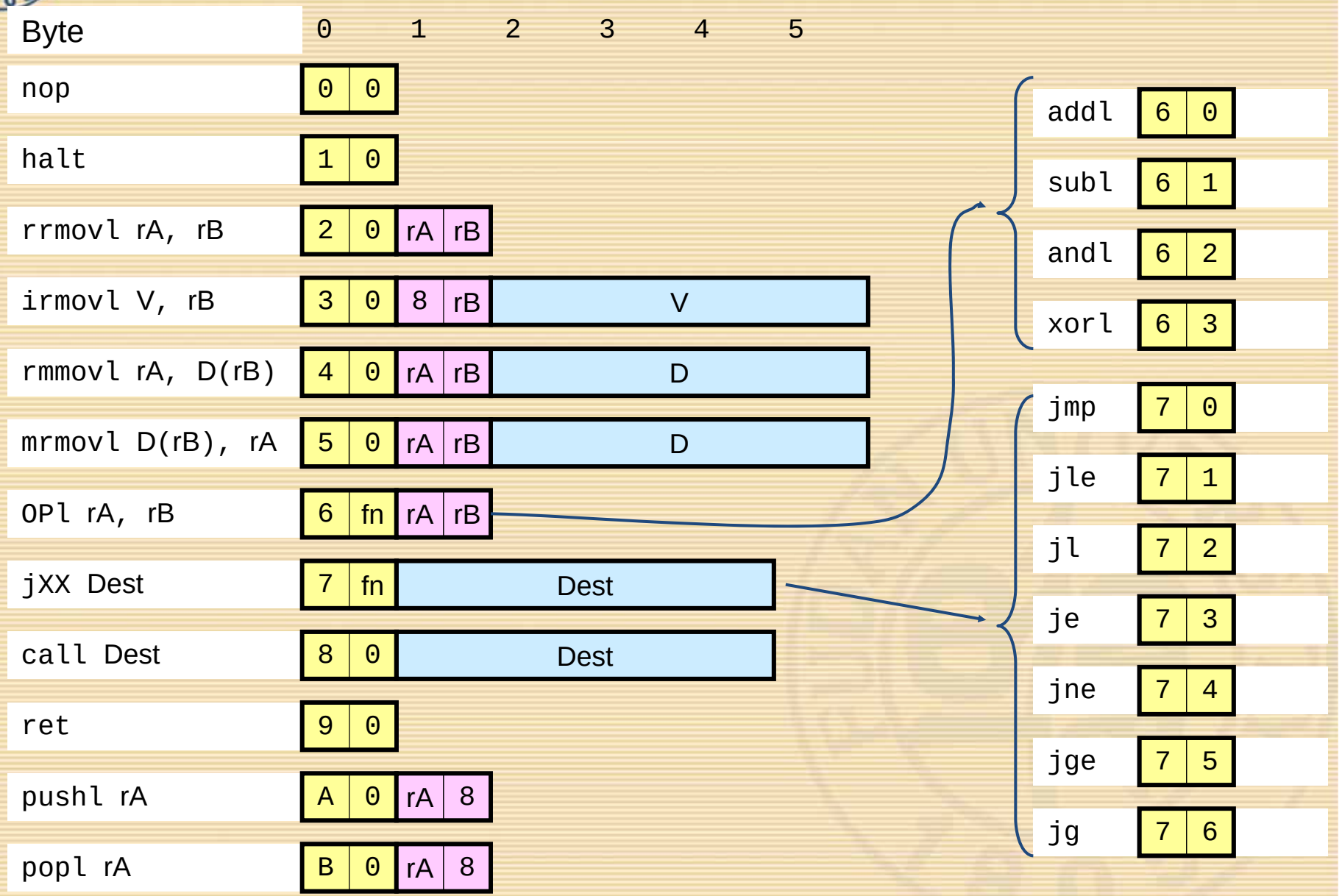
Execute Logic

Figure 4.27 P302

- Units
  - ALU
    - Implements 4 required functions
    - Generates condition code values
  - CC
    - Register with 3 condition code bits
  - bcond
    - Computes branch flag
- Control Logic
  - Set CC: Should condition code register be loaded?
  - ALU A: Input A to ALU
  - ALU B: Input B to ALU
  - ALU fun: What function should ALU compute?



Y86 Instruction Set P259



ALU A Input

	OPl rA, rB	
Execute	valE ← valB OP valA	Perform ALU operation
	`rmmovl` rA, D(rB)	
Execute	valE ← valB + valC	Compute effective address
	`popl` rA	
Execute	valE ← valB + 4	Increment stack pointer
	jXX Dest	
Execute		No operation
	`call` Dest	
Execute	valE ← valB + –4	Decrement stack pointer
	`ret`	
Execute	valE ← valB + 4	Increment stack pointer

```
int aluA = [
        icode in { IRRMOVL, IOPL } : valA;
        icode in { IIRMOVL, IRMMOVL, IMRMOVL } : valC;
        icode in { ICALL, IPUSHL } : -4;
        icode in { IRET, IPOPL } : 4;
        # Other instructions don't need ALU
];
```

ALU Operation

	OPl rA, rB	
Execute	valE ← valB OP valA	Perform ALU operation
	`rmmovl` rA, D(rB)	
Execute	valE ← valB + valC	Compute effective address
	`popl` rA	
Execute	valE ← valB + 4	Increment stack pointer
	jXX Dest	
Execute		No operation
	`call` Dest	
Execute	valE ← valB + –4	Decrement stack pointer
	`ret`	
Execute	valE ← valB + 4	Increment stack pointer

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

Condition Set

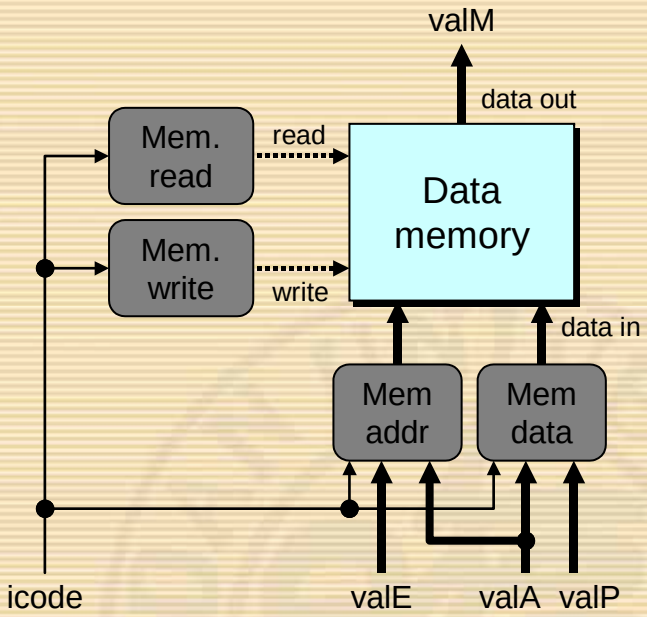
- Bool set_cc = icode in { IOPL };

- We will not discuss the detail of Bcond
  - Though it is also a control unit

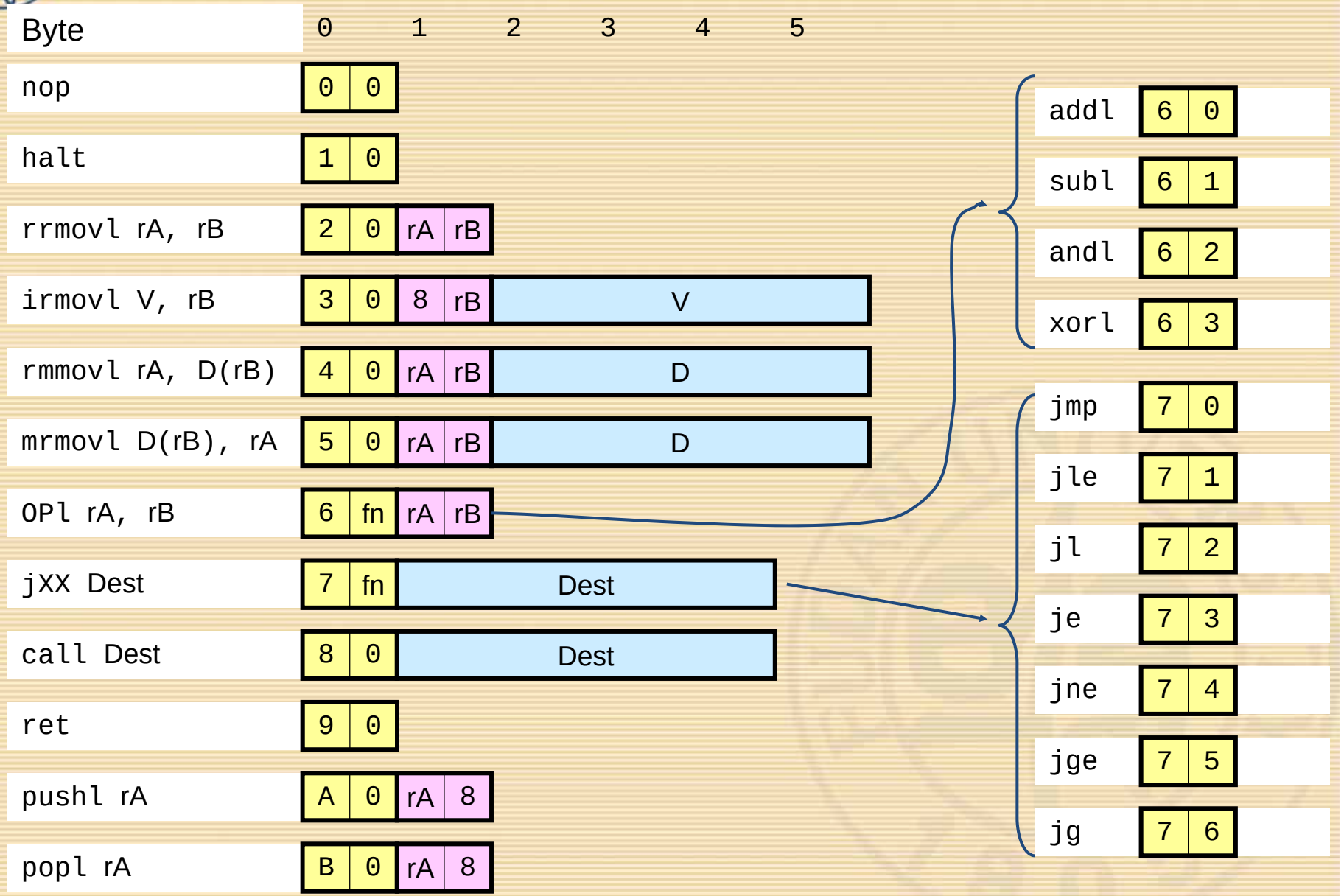
Memory Logic

Figure 4.28 P303

- Memory
  - Reads or writes memory word
- Control Logic
  - Mem. read: should word be read?
  - Mem. write: should word be written?
  - Mem. addr.: Select address
  - Mem. data.: Select data



Y86 Instruction Set P259



Memory Address

	OPl rA, rB	
Memory		No operation

	`rmmovl` rA, D(rB)	
Memory	M_4[valE] ← valA	Write value to memory

	`popl` rA	
Memory	valM ← M_4[valA]	Read from stack

	jXX Dest	
Memory		No operation

	`call` Dest	
Memory	M_4[valE] ← valP	Write return value on stack

	`ret`	
Memory	valM ← M_4[valA]	Read return address

```
int mem_addr = [
        icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : valE;
        icode in { IPOPL, IRET } : valA;
        # Other instructions don't need address
];
```

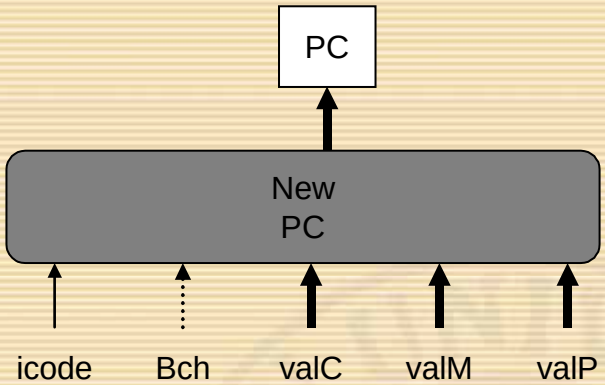
Memory Read

	OPl rA, rB	
Memory		No operation
	`rmmovl` rA, D(rB)	
Memory	M_4[valE] ← valA	Write value to memory
	`popl` rA	
Memory	valM ← M_4[valA]	Read from stack
	jXX Dest	
Memory		No operation
	`call` Dest	
Memory	M_4[valE] ← valP	Write return value on stack
	`ret`	
Memory	valM ← M_4[valA]	Read return address

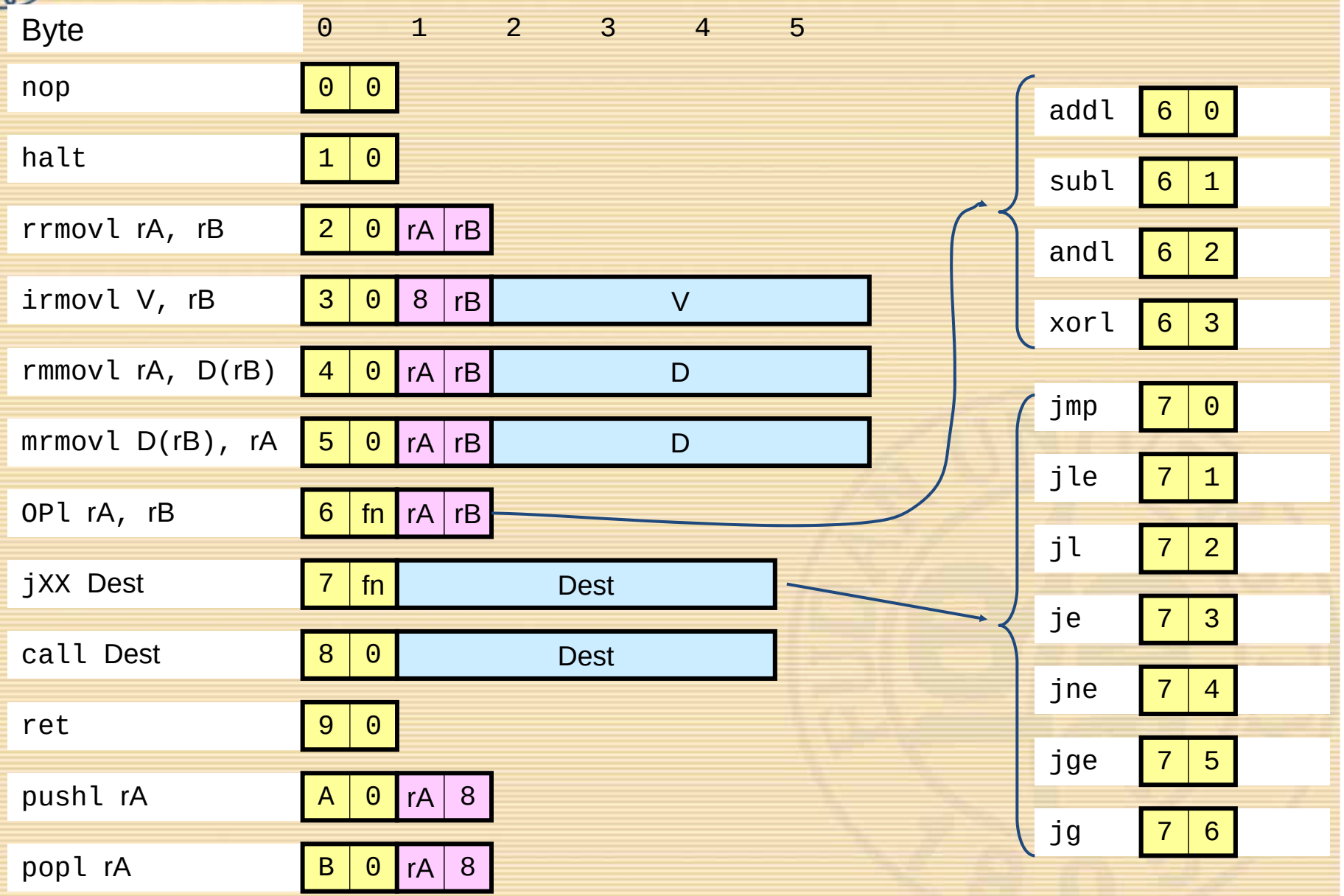
PC Update Logic

Figure 4.29 P304

- New PC
  - Select next value of PC



Y86 Instruction Set P259



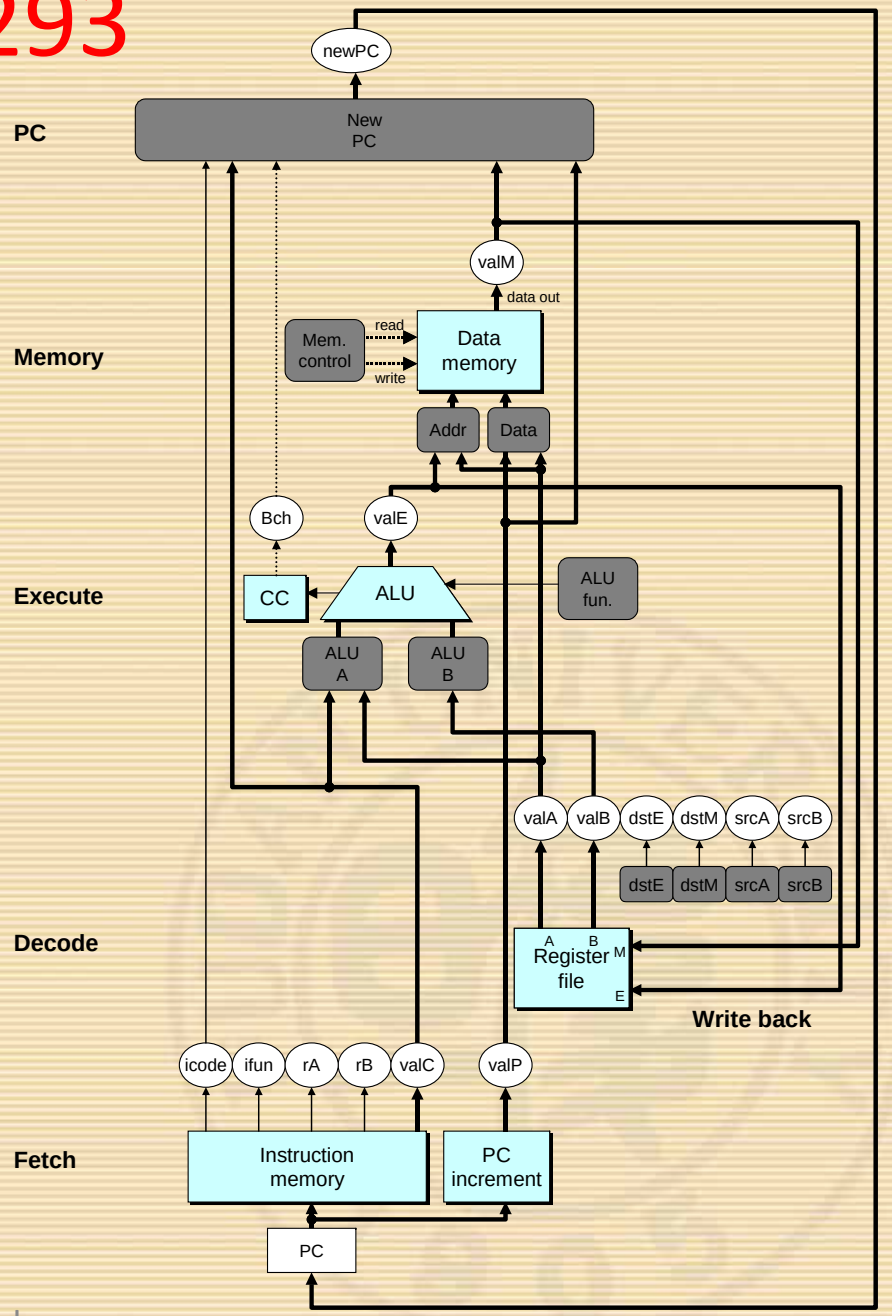
PC Update

	OPl rA, rB	
PC update	PC ← valP	Update PC
	`rmmovl` rA, D(rB)	
PC update	PC ← valP	Update PC
	`popl` rA	
PC update	PC ← valP	Update PC
	jXX Dest	
PC update	PC ← Bch ? valC : valP	Update PC
	`call` Dest	
PC update	PC ← valC	Set PC to destination
	`ret`	
PC update	PC ← valM	Set PC to return address

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

SEQ Hardware P293

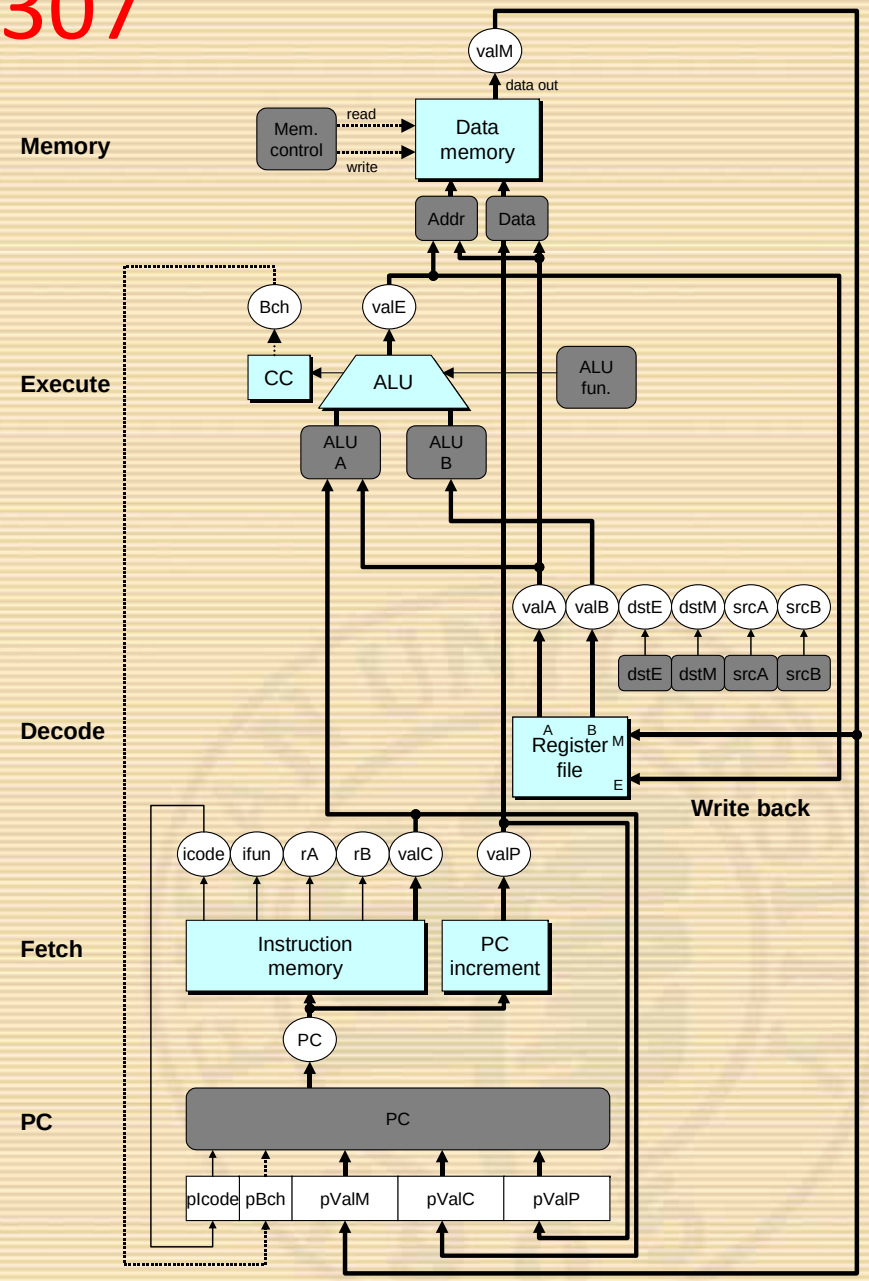
- Stages occur in sequence
- One operation in process at a time



SEQ+ Hardware P307

- Still sequential implementation
- Reorder PC stage to put at beginning

- PC Stage
  - Task is to select PC for current instruction
  - Based on results computed by previous instruction
- Processor State
  - PC is no longer stored in register
  - But, can determine PC based on other stored information



PC Computation

```
Int pc= [
    pIcode == ICALL : pValC;
    pIcode == IJXX && bBch : pValC;
    PIcode == IRET : pValM;
    1 : pValP;
    ];
```

SEQ Summary

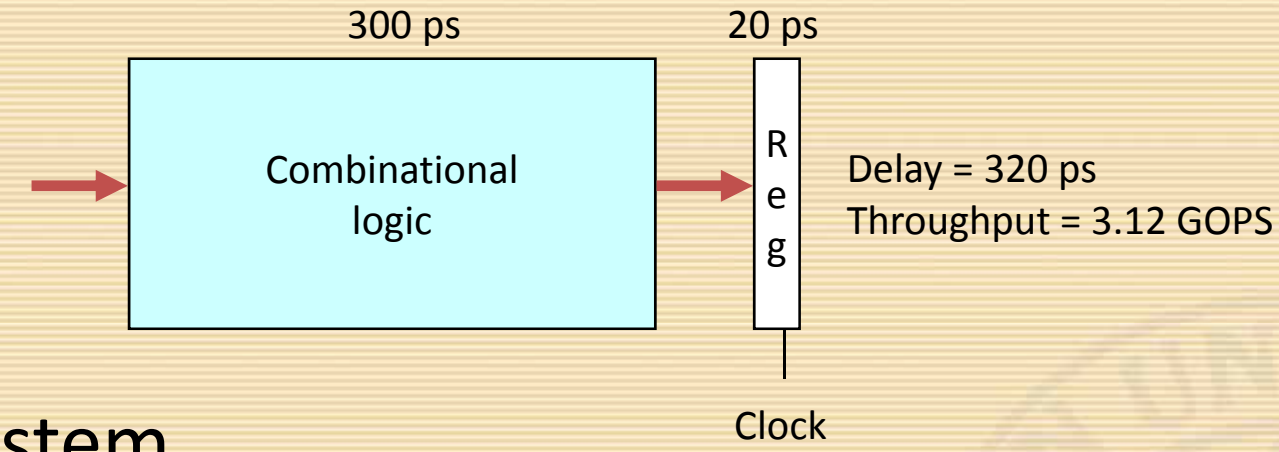
- Implementation
  - Express every instruction as series of simple steps
  - Follow same general flow for each instruction type
  - Assemble registers, memories, predesigned combinational blocks
  - Connect with control logic
- Limitations
  - Too slow to be practical
  - In one cycle, must propagate 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

Lecutre

jc@fudan.edu.cn

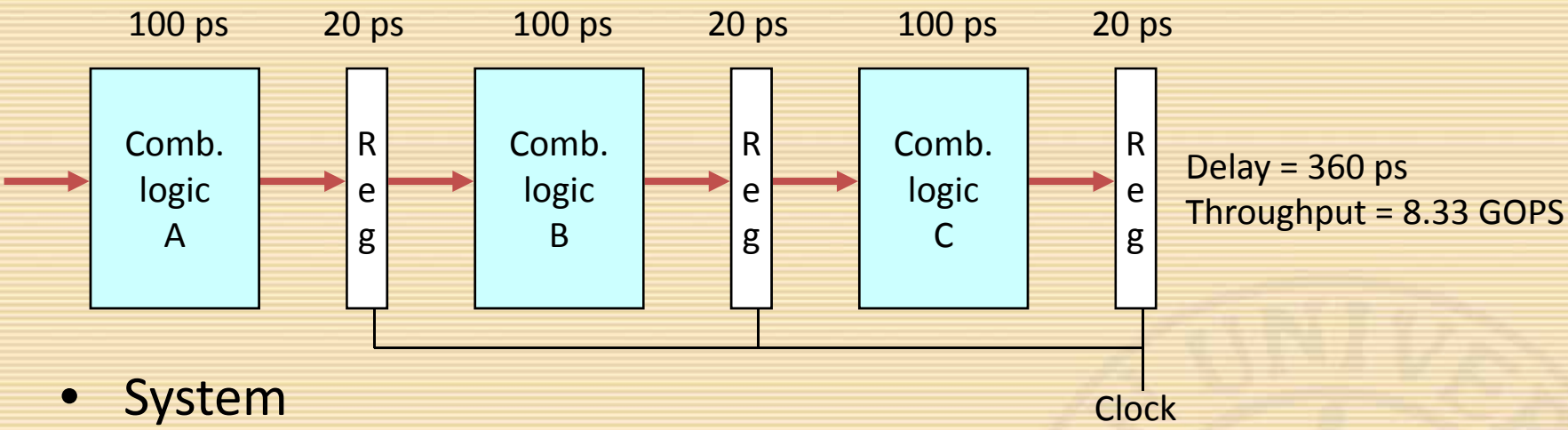
Pipelined Implementation

Computational Example



- System
  - Computation requires total of 300 picoseconds
  - Additional 20 picoseconds to save result in register
  - Can must have clock cycle of at least 320 ps

3-Way Pipelined Version



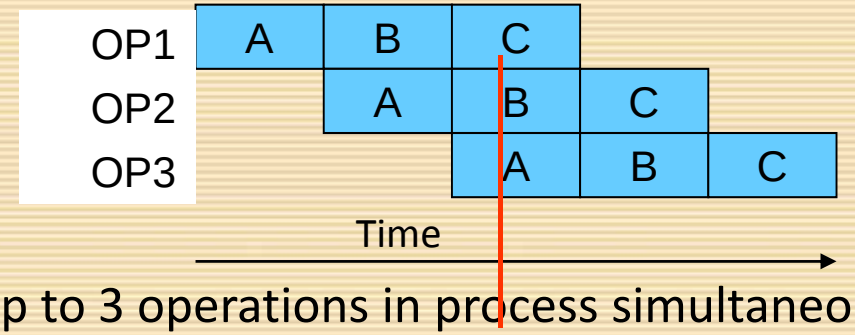
- System
  - Divide combinational logic into 3 blocks of 100 ps each
  - Can begin new operation as soon as previous one passes through stage A.
    - Begin new operation every 120 ps
  - Overall latency increases
    - 360 ps from start to finish

Pipeline Diagrams

- Unpipelined

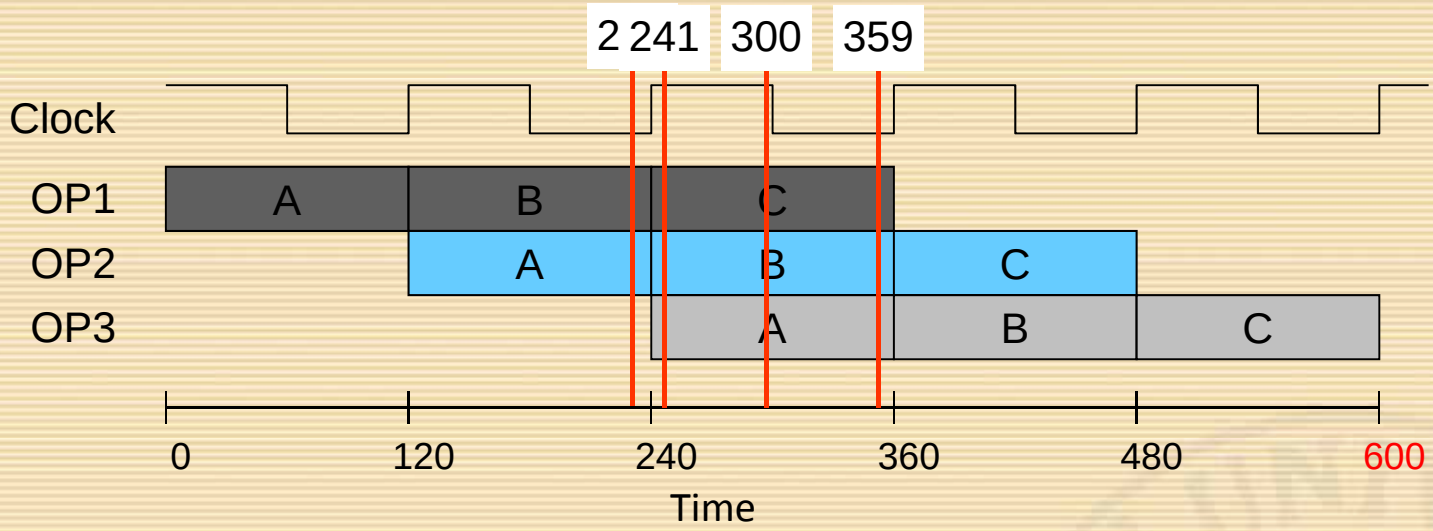
  - Cannot start new operation until previous one completes

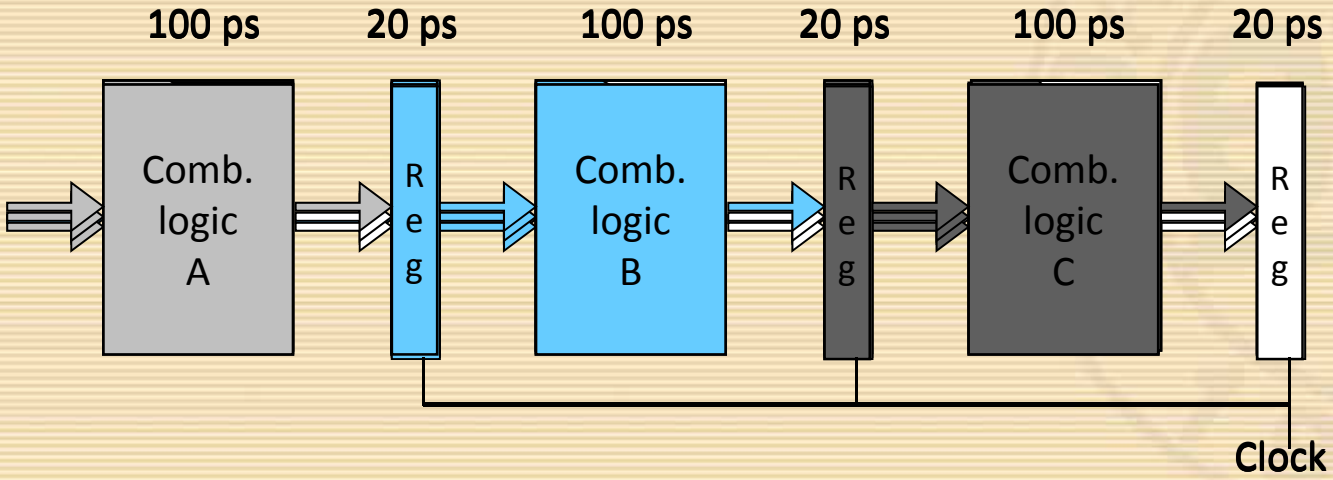
- 3-Way Pipelined



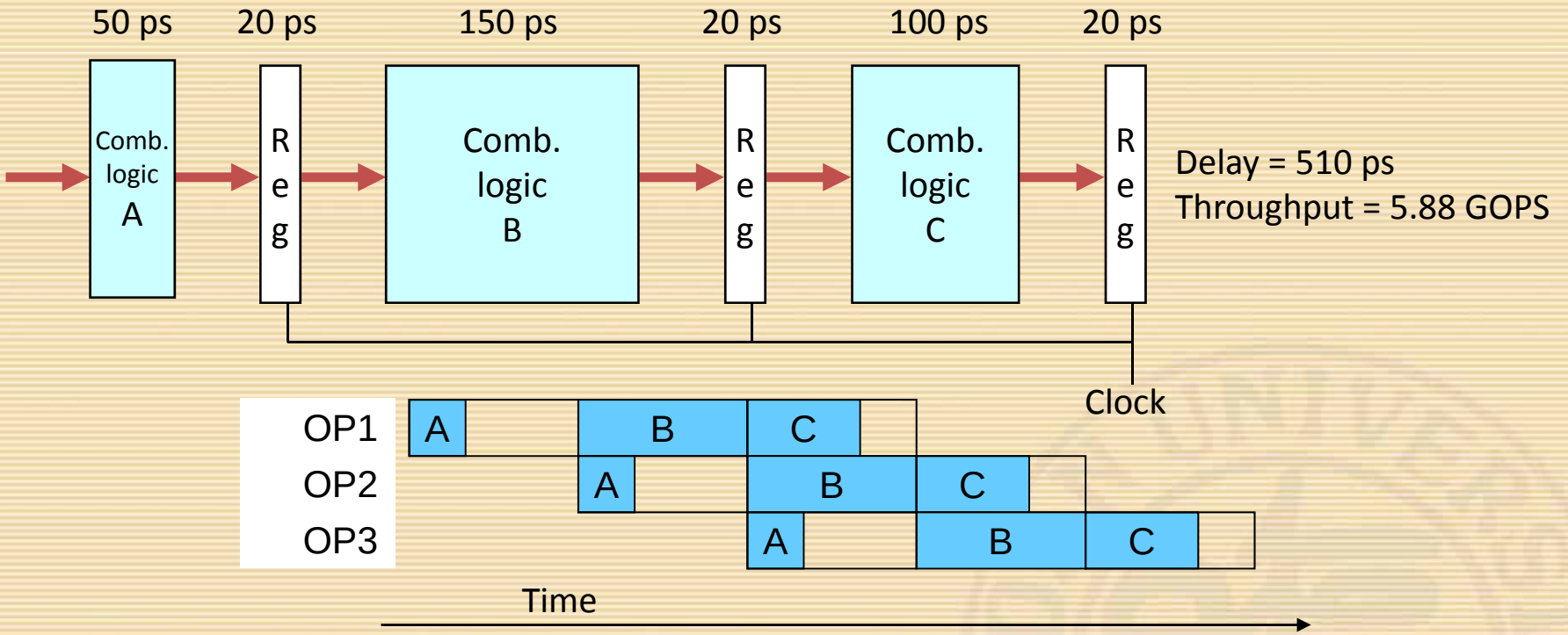
  - Up to 3 operations in process simultaneously

Operating a Pipeline



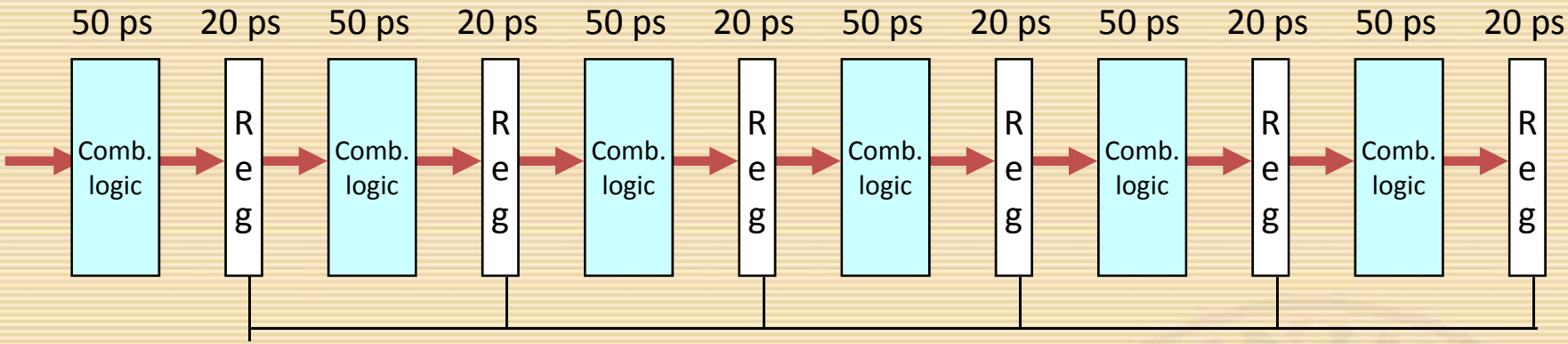


Limitations: Nonuniform Delays



- Throughput limited by slowest stage
- Other stages sit idle for much of the time
- Challenging to partition system into balanced stages

Limitations: Register Overhead



Delay = 420 ps, Throughput = 14.29 GOPS

- As try to deepen pipeline, overhead of loading registers becomes more significant
- Percentage of clock cycle spent loading register:
  - 1-stage pipeline: 6.25%
  - 3-stage pipeline: 16.67%
  - 6-stage pipeline: 28.57%
- High speeds of modern processor designs obtained through very deep pipelining

Overhead:开销

Adding Pipeline Registers

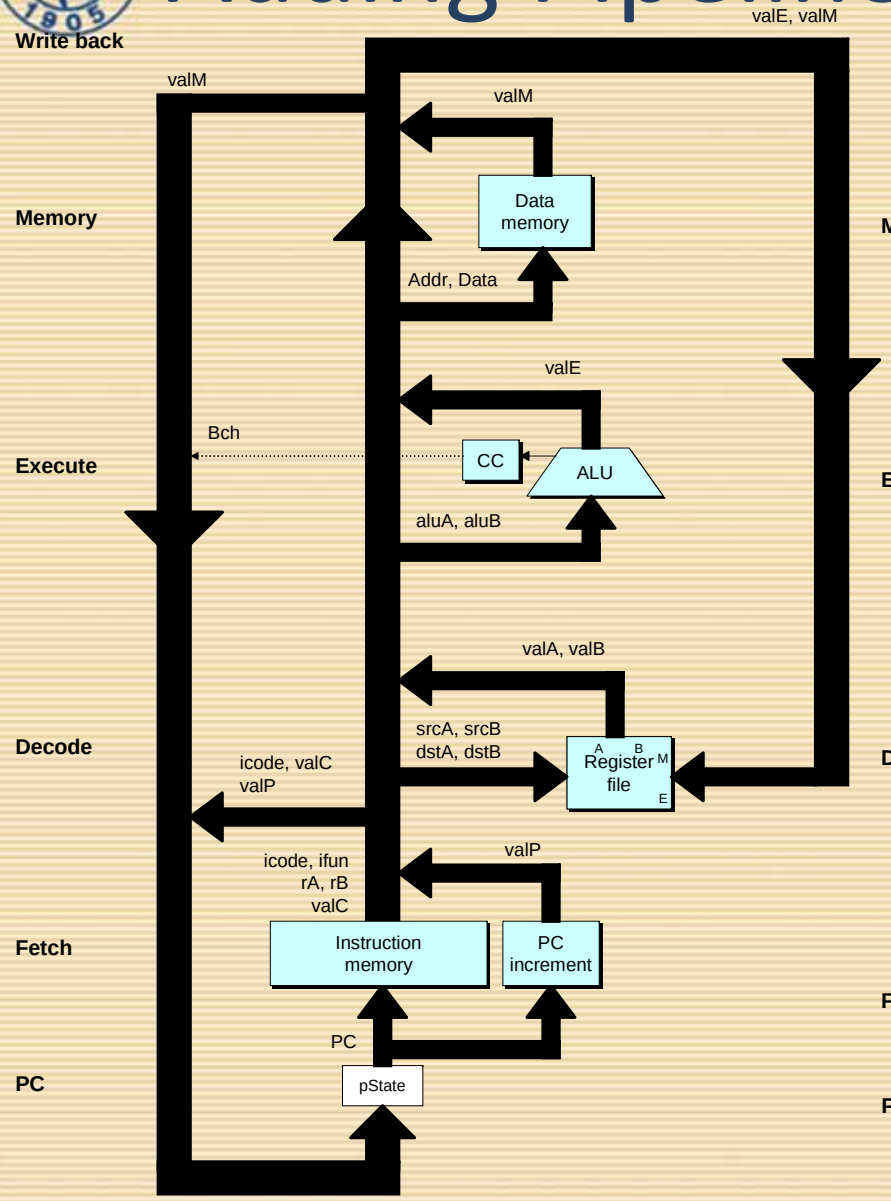


Figure 4.30 P306

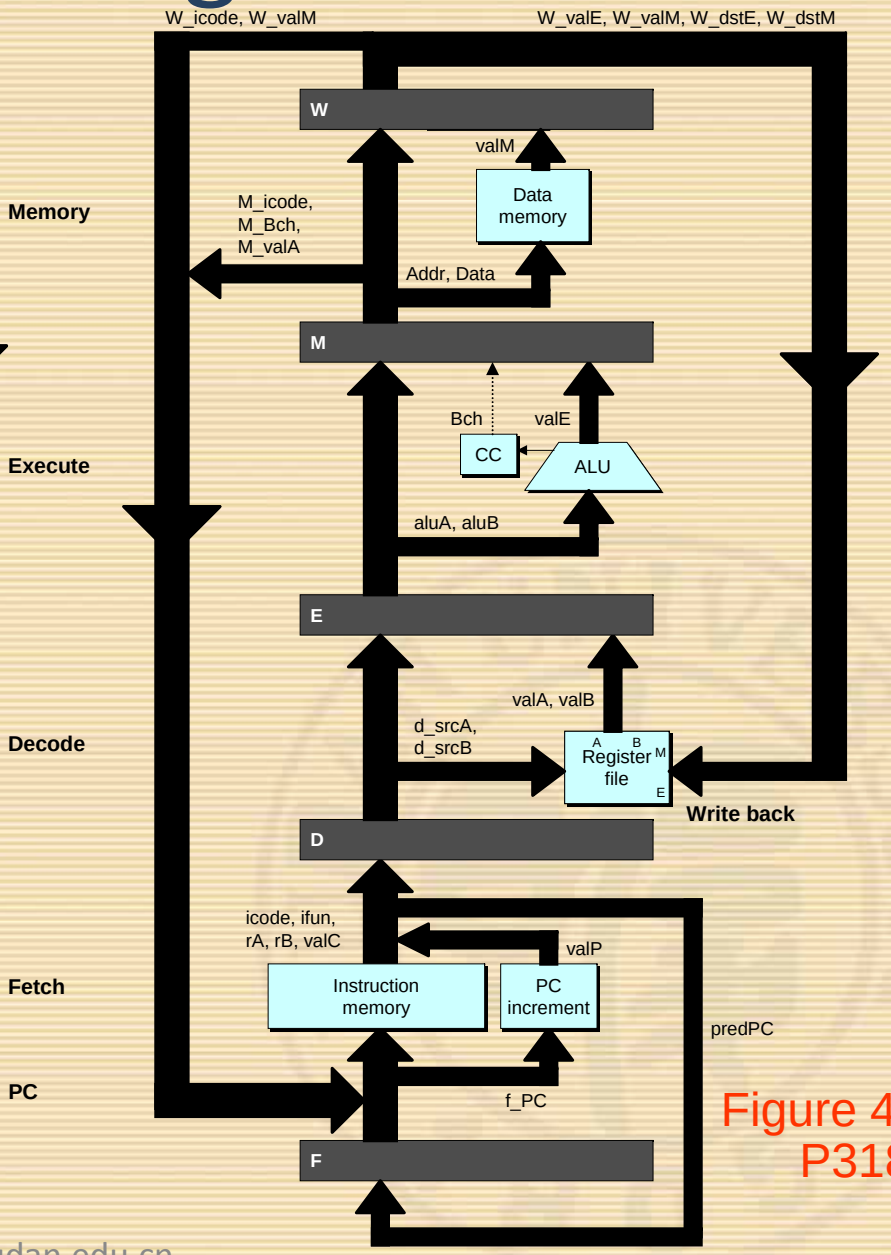
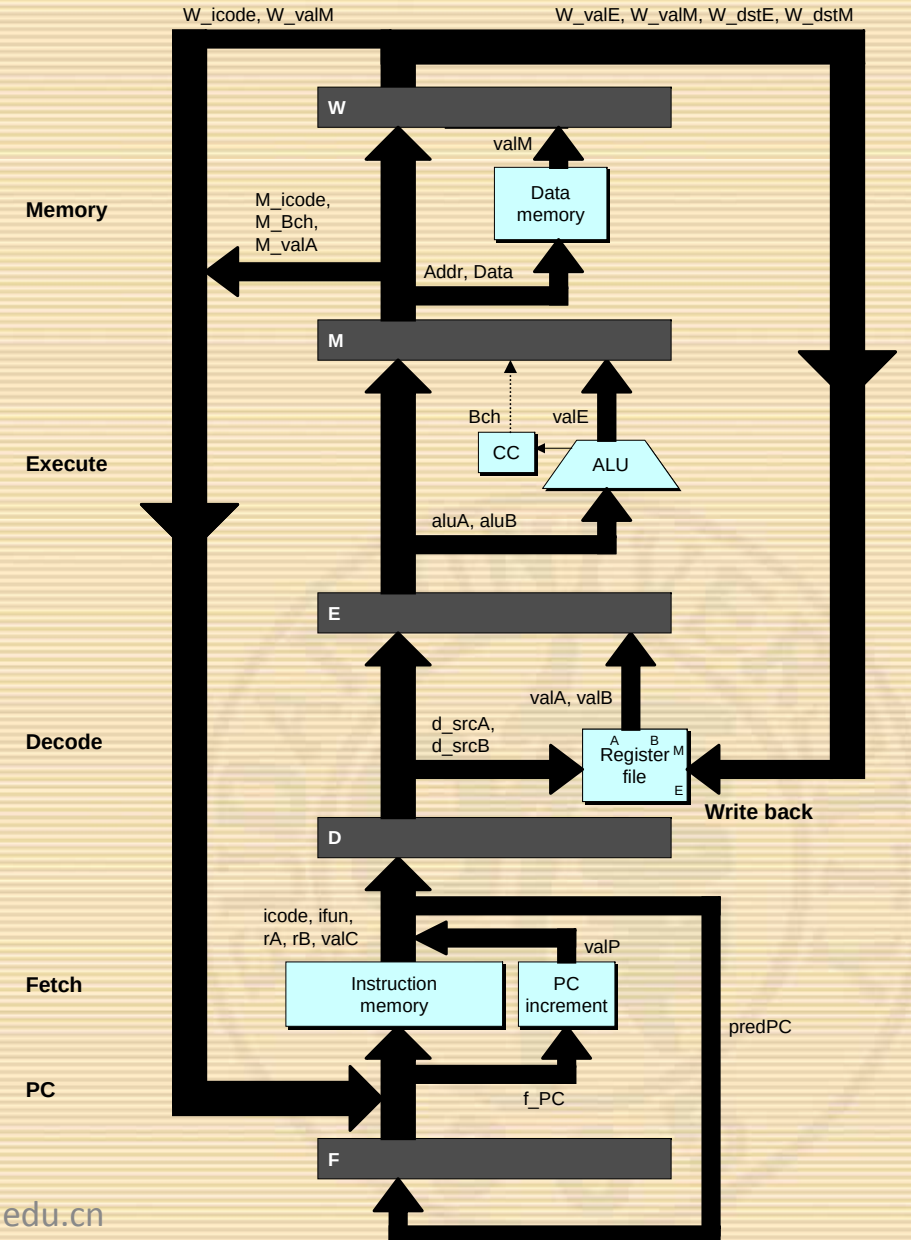


Figure 4.39 P318

Pipeline Stages

- Fetch
  - Select current PC
  - Read instruction
  - Compute incremented PC
- Decode
  - Read program registers
- Execute
  - Operate ALU
- Memory
  - Read or write data memory
- Write Back
  - Update register file

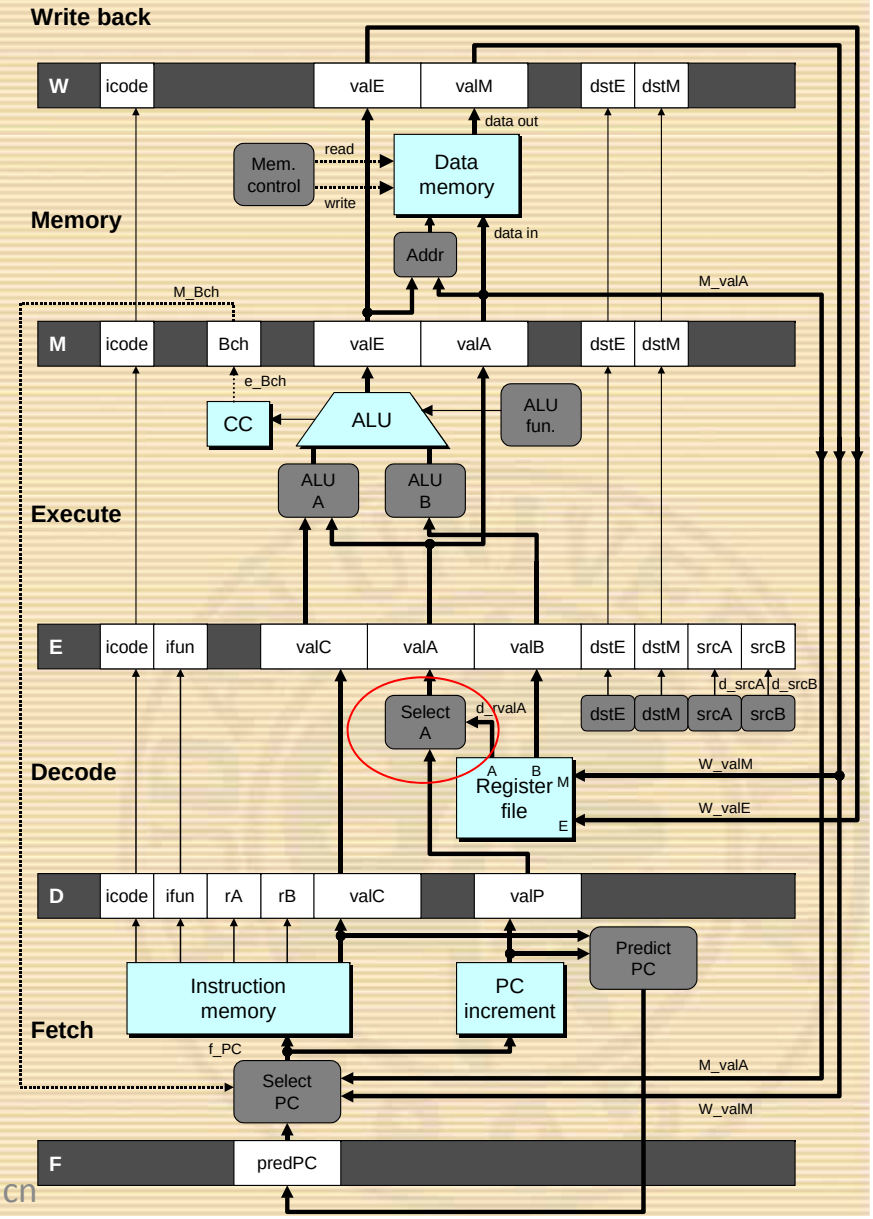


PIPE- Hardware

Figure 4.41 P320

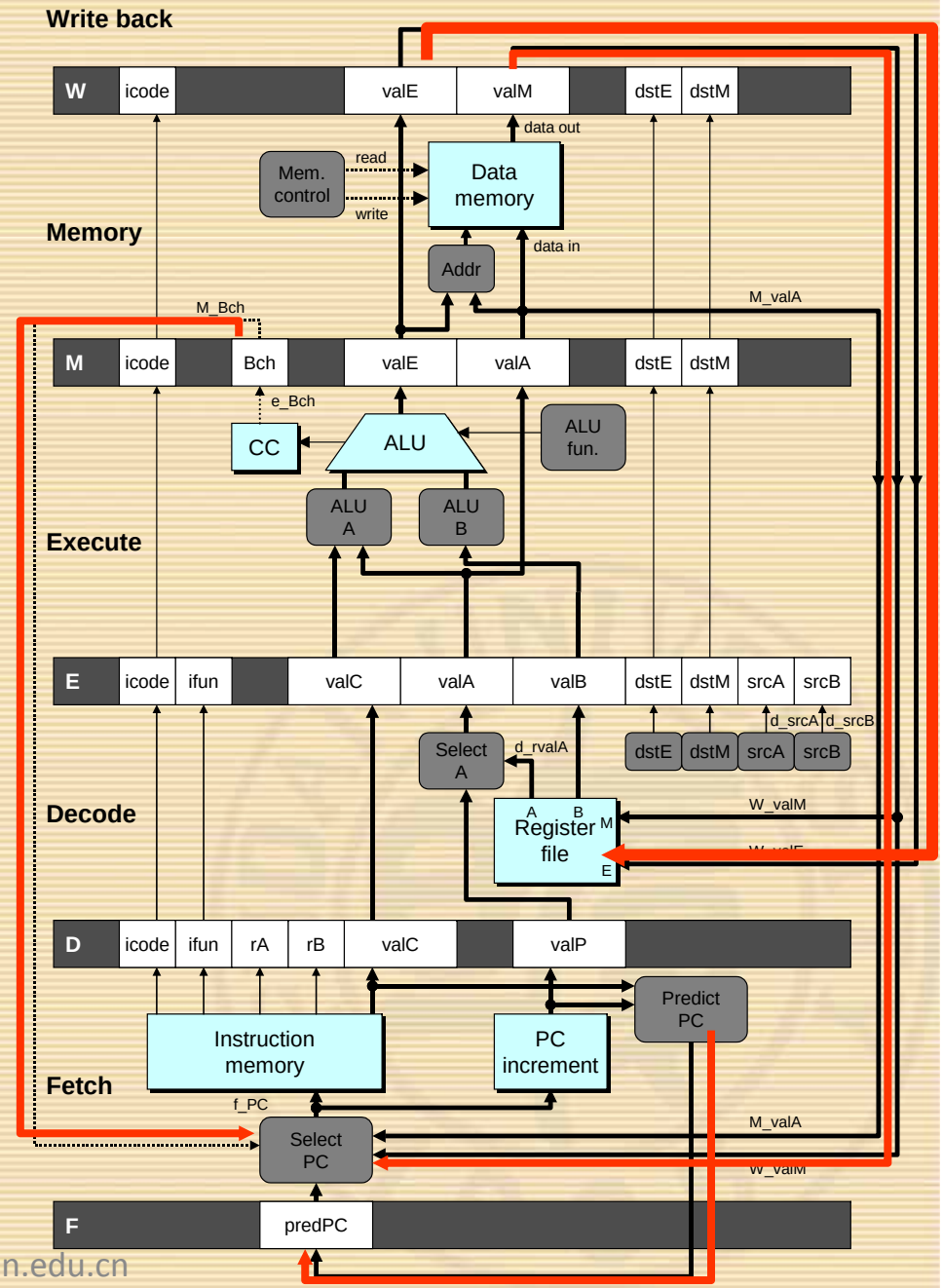
- Pipeline registers hold intermediate values from instruction execution

- Forward (Upward) Paths
  - Values passed from one stage to next
  - Cannot jump past stages
    - e.g., valC passes through decode



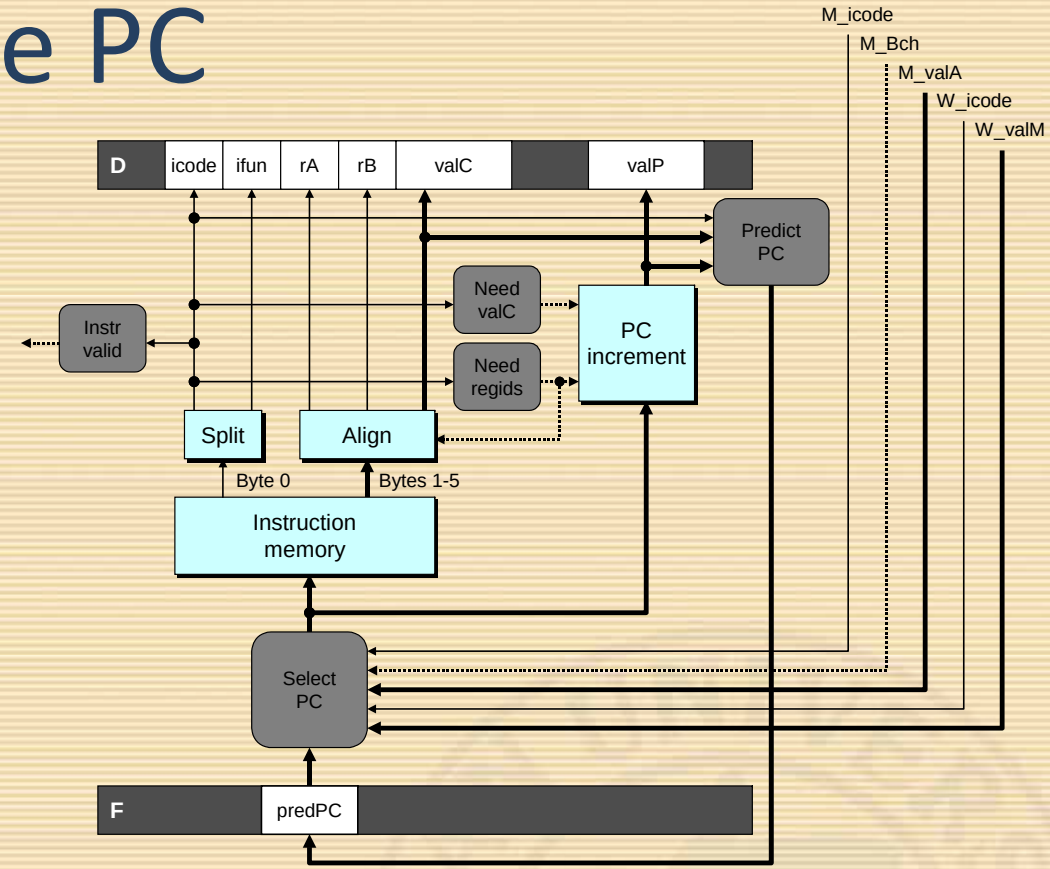
Feedback Paths

- Predicted PC
  - Guess value of next PC
- Branch information
  - Jump taken/not-taken
  - Fall-through or target address
- Return point
  - Read from memory
- Register updates
  - To register file write ports



Predicting the PC

Figure 4.56 P338



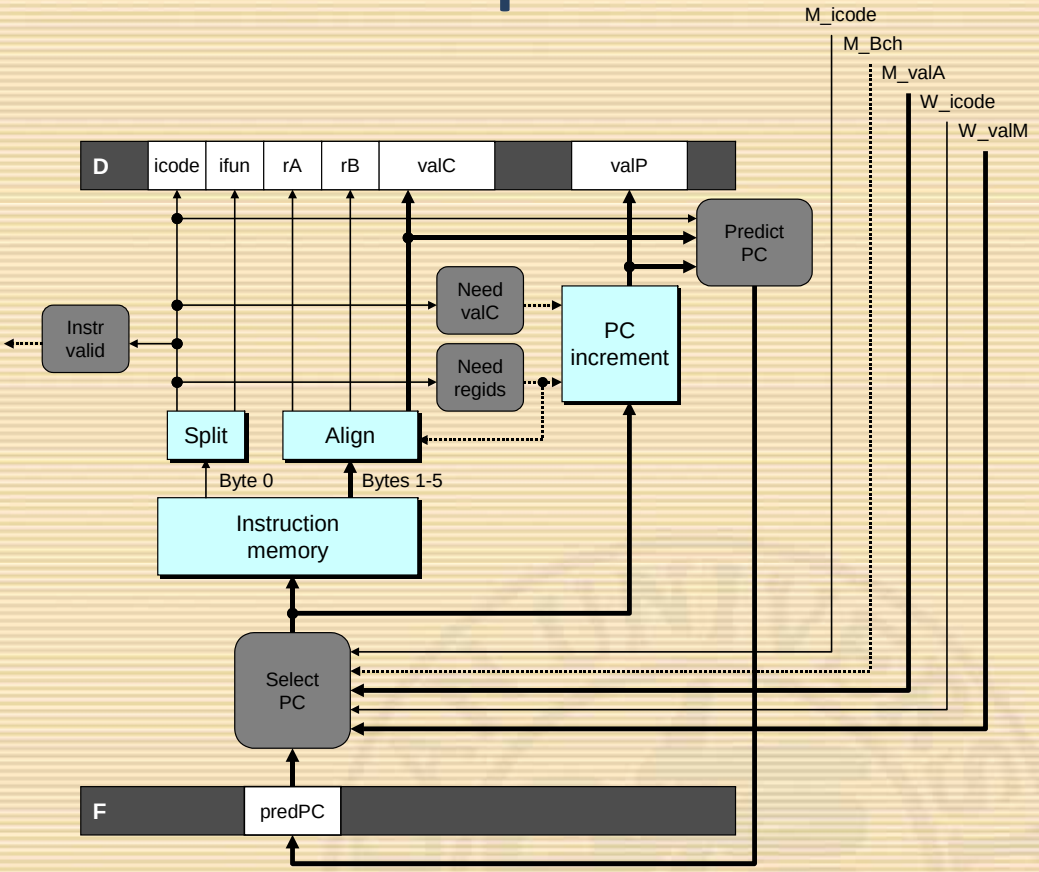
- Start fetch of new instruction after current one has completed fetch stage
  - Not enough time to reliably determine next instruction
- Guess which instruction will follow
  - Recover if prediction was incorrect

Our Prediction Strategy

- Instructions that Don't Transfer Control
  - Predict next PC to be valP
  - Always reliable
- Call and Unconditional Jumps
  - Predict next PC to be valC (destination)
  - Always reliable
- Conditional Jumps
  - Predict next PC to be valC (destination)
  - Only correct if branch is taken
    - Typically right 60% of time
- Return Instruction
  - Don't try to predict

Recovering from PC Misprediction

Figure 4.56 P338



- Mispredicted Jump
  - Will see branch flag once instruction reaches memory stage
  - Can get fall-through PC from valA
- Return Instruction
  - Will get return PC when `ret` reaches write-back stage

Select PC

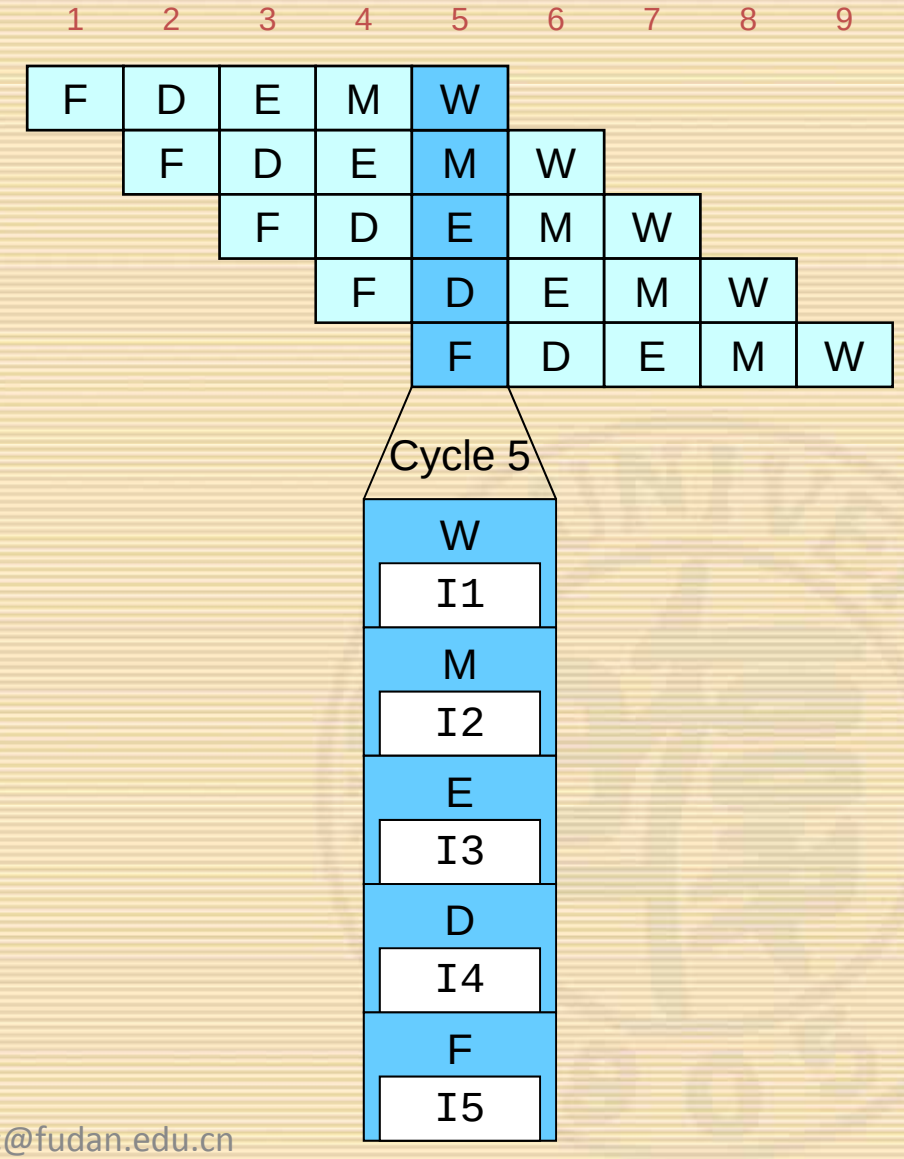
P338 and Figure 4.56

```
int f_PC = [
        #mispredicted branch. Fetch at incremented PC
                M_icode == IJXX && !M_Bch : M_valA;
        #completion of RET instruciton
                W_icode == IRET : W_valM;
        #default: Use predicted value of PC
                1: F_predPC
];
Int new_F_predPC = [
        f_icode in {IJXX, ICALL} : f_valC;
        1: f_valP;
];
```

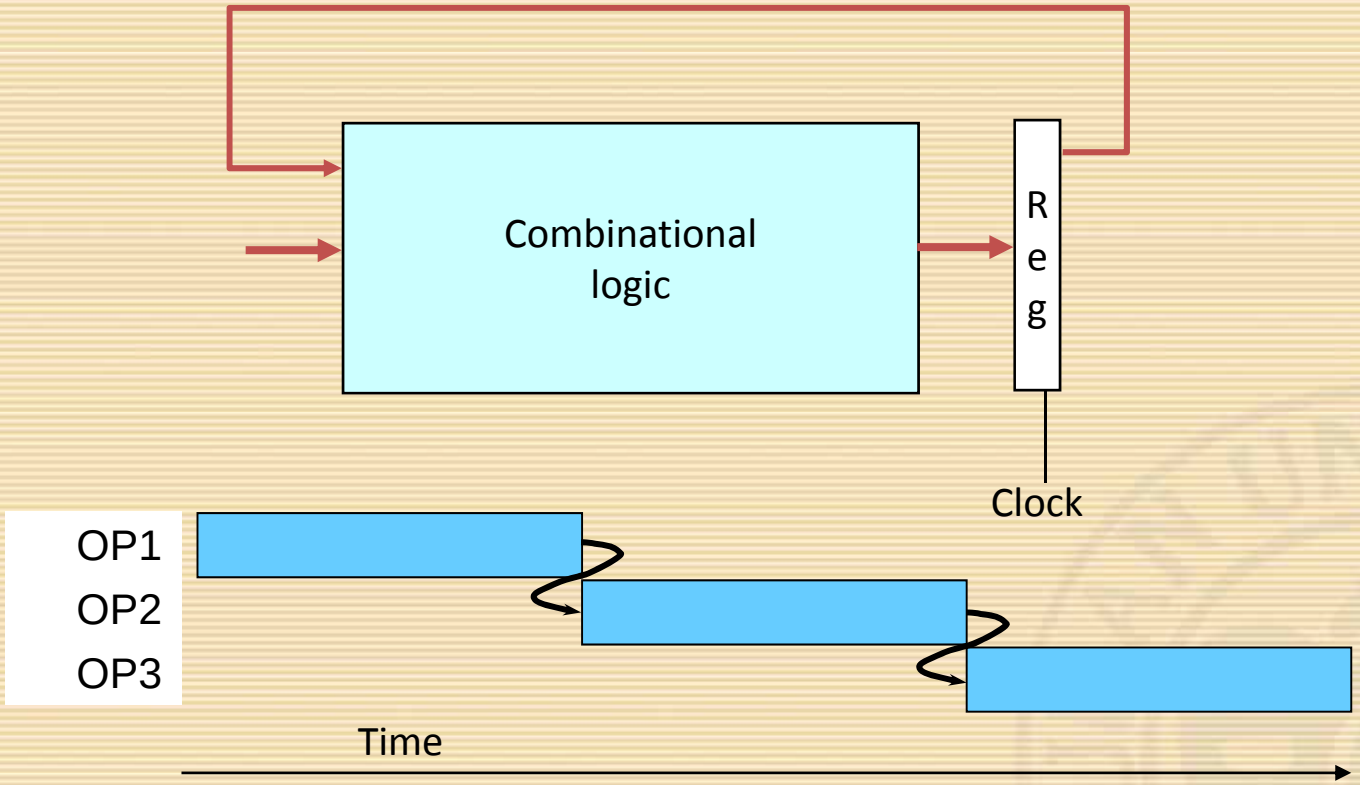
Pipeline Demonstration

Figure 4.40 P319

```
irmovl   $1,%eax   #I1
irmovl   $2,%ecx   #I2
irmovl   $3,%edx   #I3
irmovl   $4,%ebx   #I4
halt               #I5
```

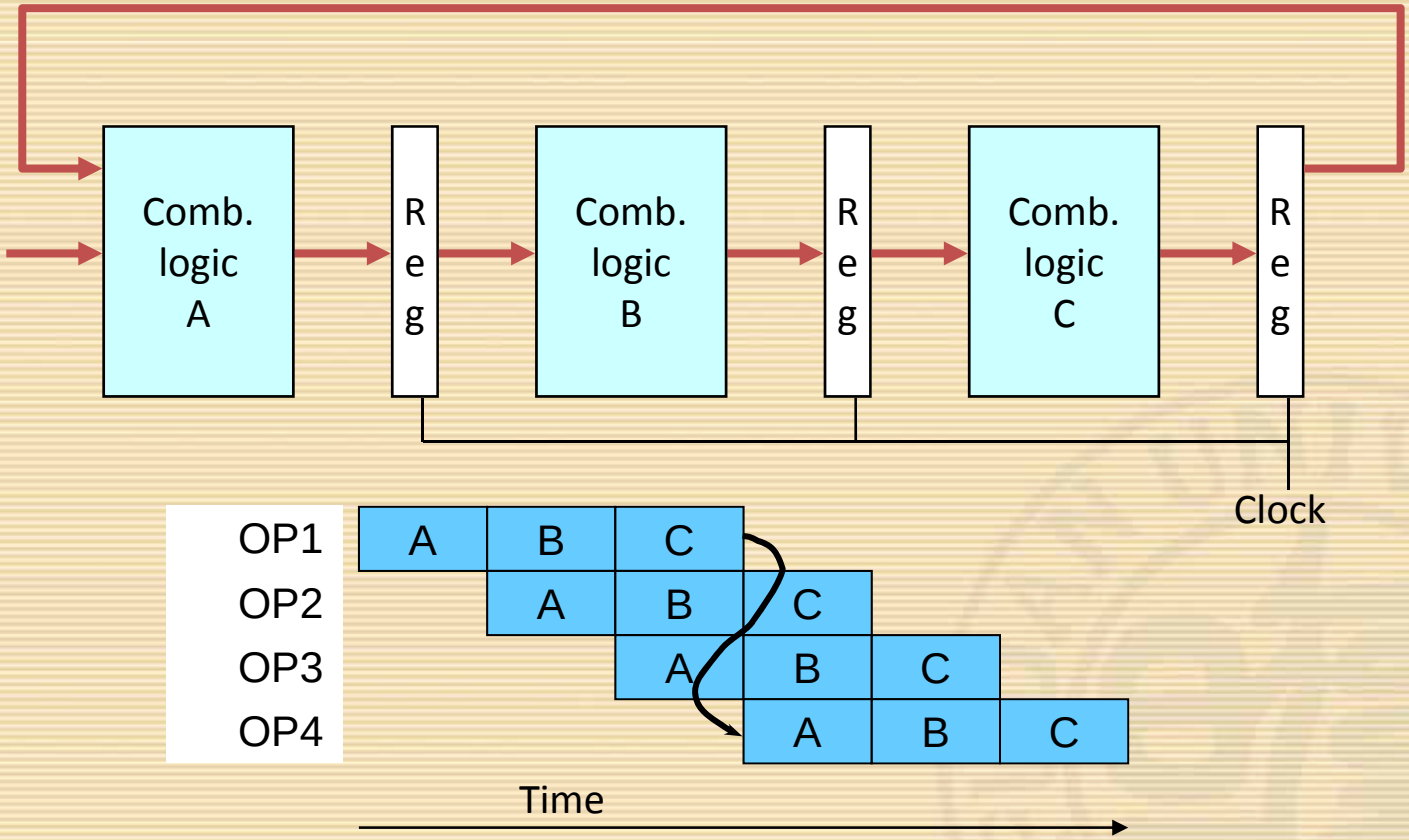


Data Dependencies



- System
  - Each operation depends on result from preceding one

Data Hazards



- Result does not feed back around in time for next operation
- Pipelining has changed behavior of system

Data Dependencies in Processors

```
1    irmovl $50, %eax
2    addl %eax , %ebx
3    mrmovl 100( %ebx ), %edx
```

- Result from one instruction used as operand for another
  - Read-after-write (RAW) dependency
- Very common in actual programs
- Must make sure our pipeline handles these properly
  - Get correct results
  - Minimize performance impact

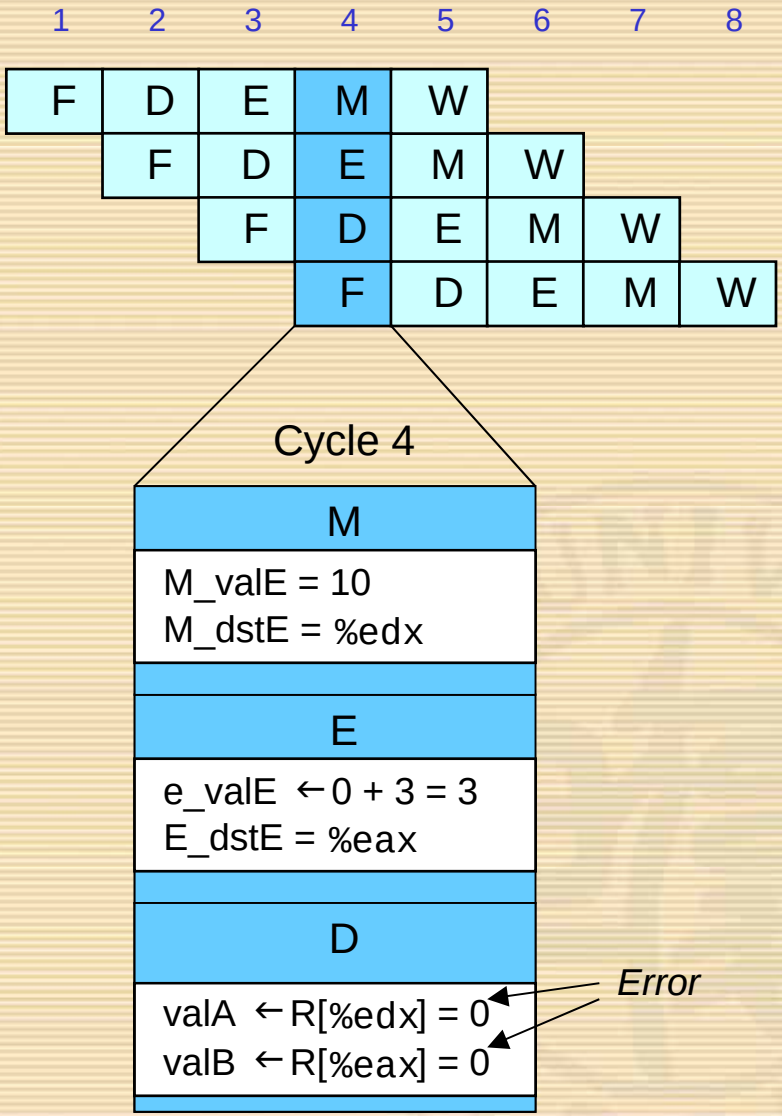
Control Dependence

- Example:
  - Loop:
    - Subl %edx, %ebx
    - Jne targ
    - Irmovl $10, %edx
    - Jmp loop
  - Targ:
    - Halt
  - The jne instruction create a control dependency.
    - Which instruction will be executed?

Data Dependencies

```
# demo-h0.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: addl %edx,%eax
0x00e: halt
```

Figure 4.45 P327



Data Dependencies: 1 nop

```
# demo-h1.ys

0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: addl %edx,%eax
0x00f: halt
```

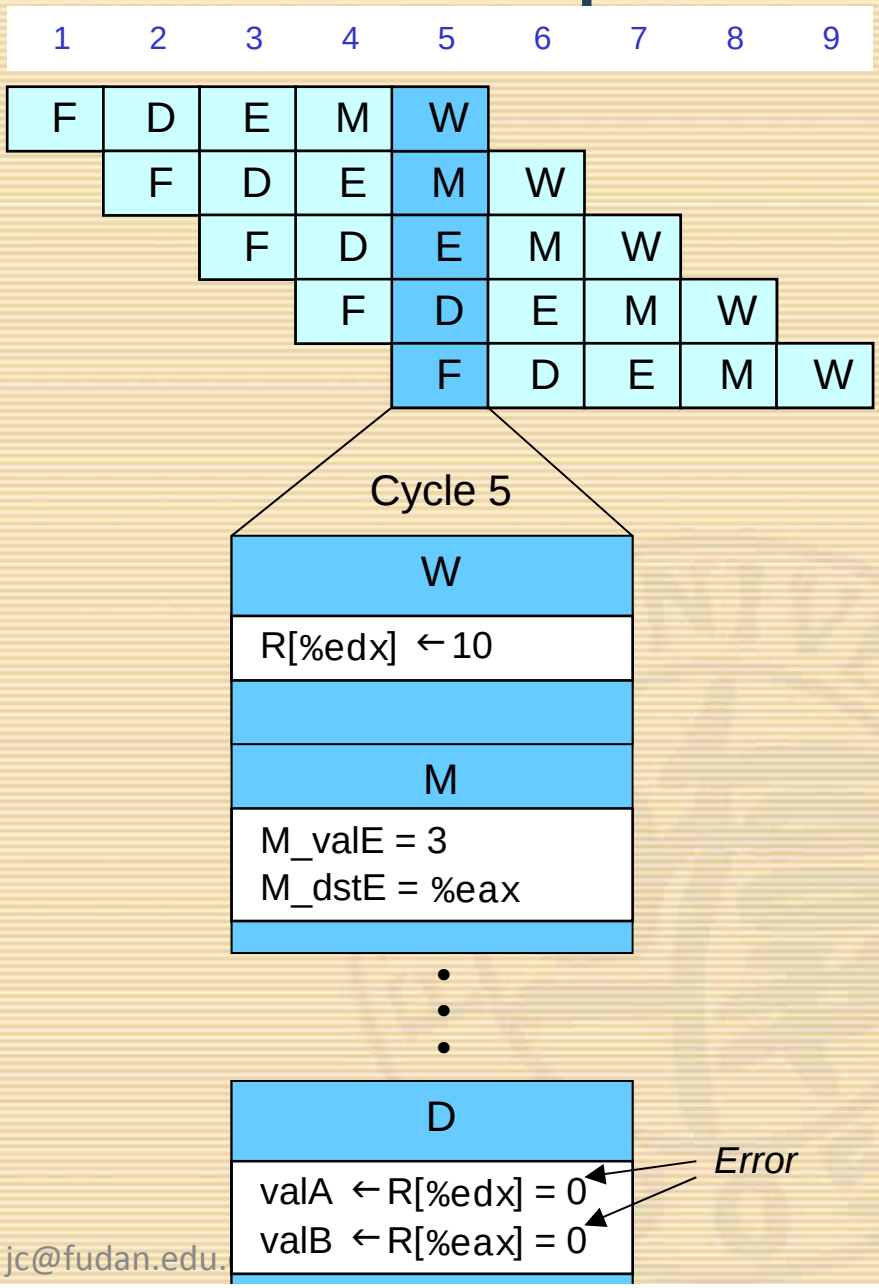
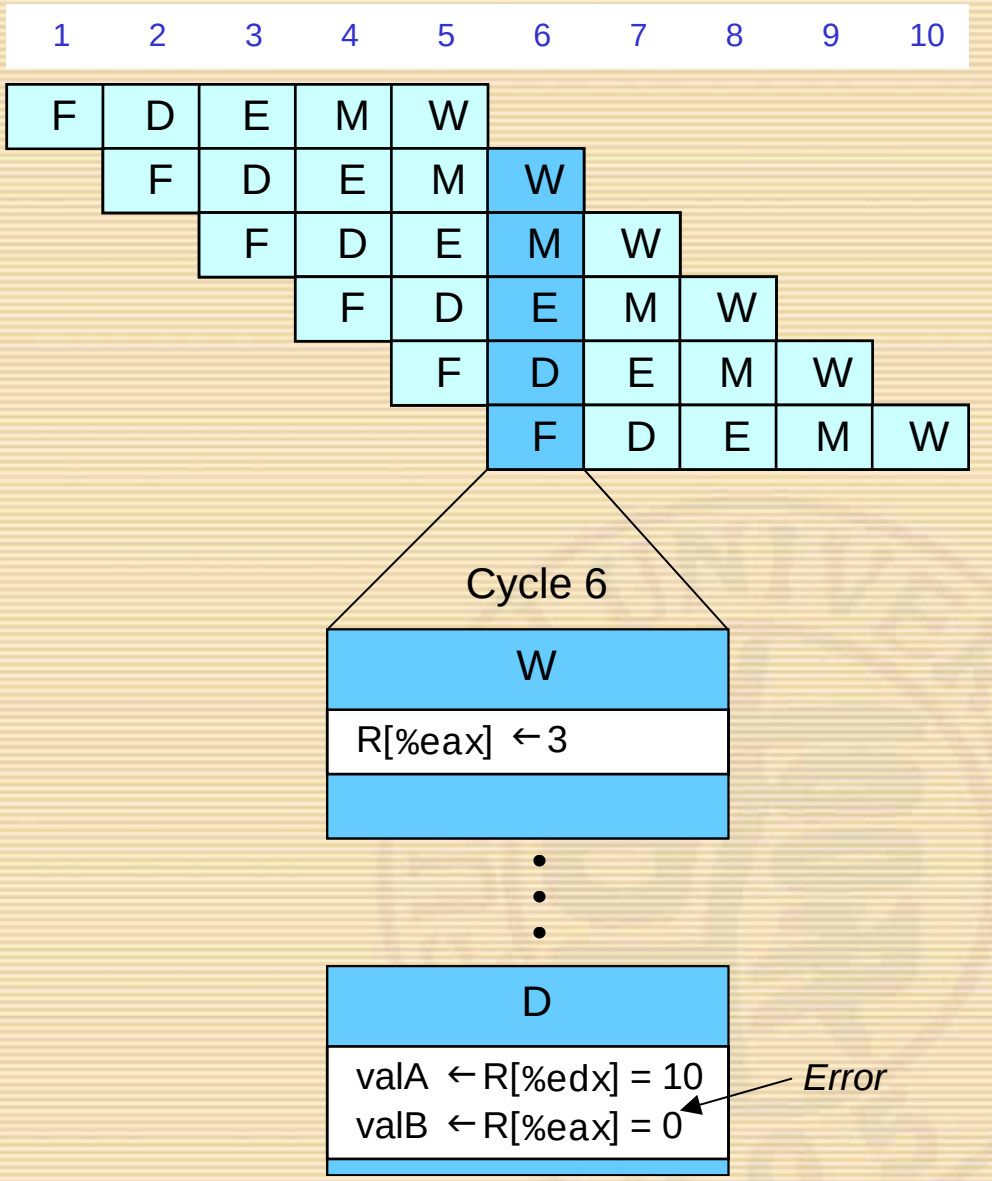


Figure 4.44 P326

Data Dependencies: 2 nops

```
# demo-h2.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: nop
0x00e: addl %edx,%eax
0x010: halt
```

Figure 4.43 P325



Data Dependencies: 3 nops

```
# demo-h3.ys
 0x000: irmovl $10,%edx
 0x006: irmovl  $3,%eax
 0x00c: nop
 0x00d: nop
 0x00e: nop
 0x00f: addl %edx,%eax
 0x011: halt
```

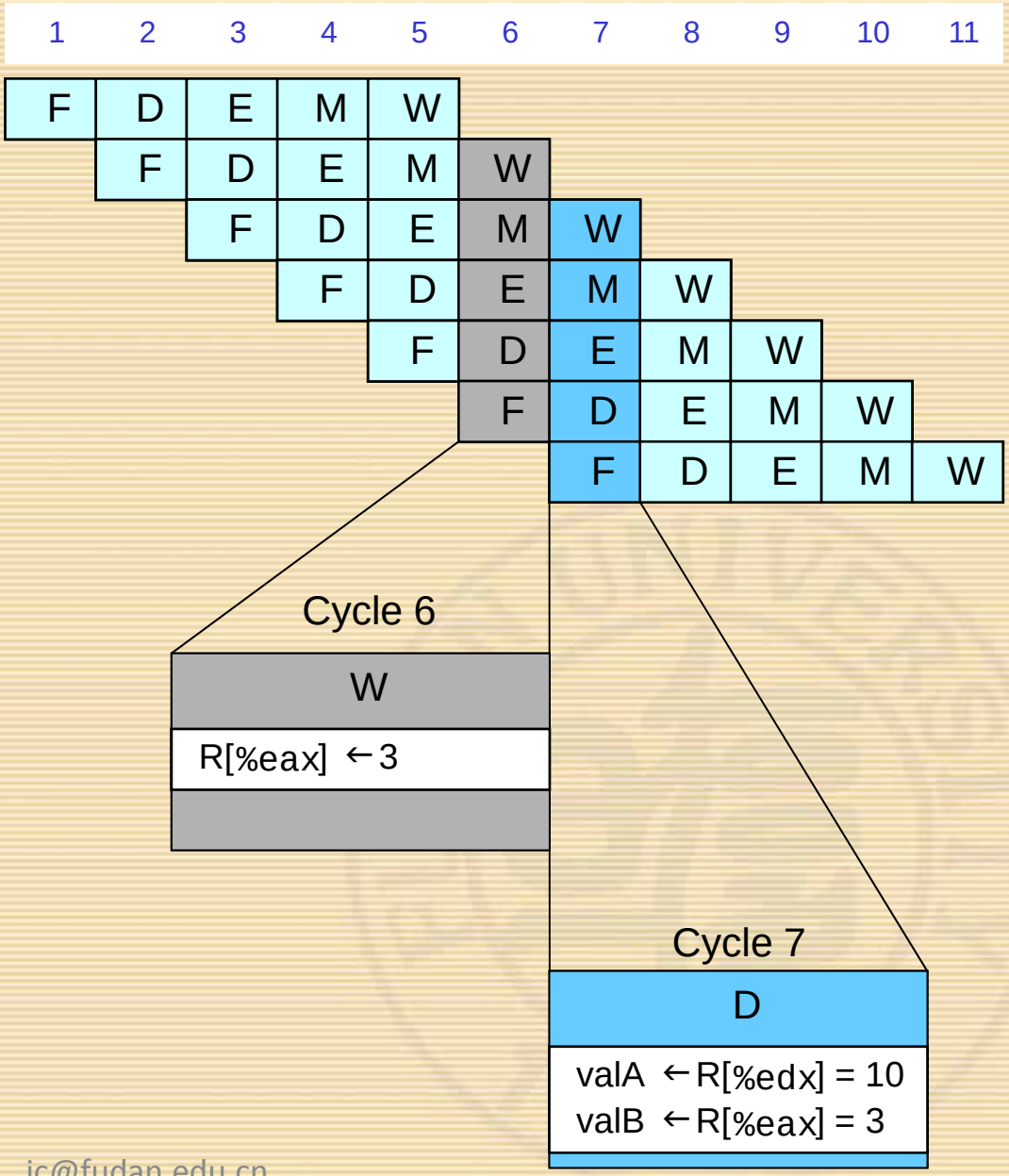


Figure 4.42 P324

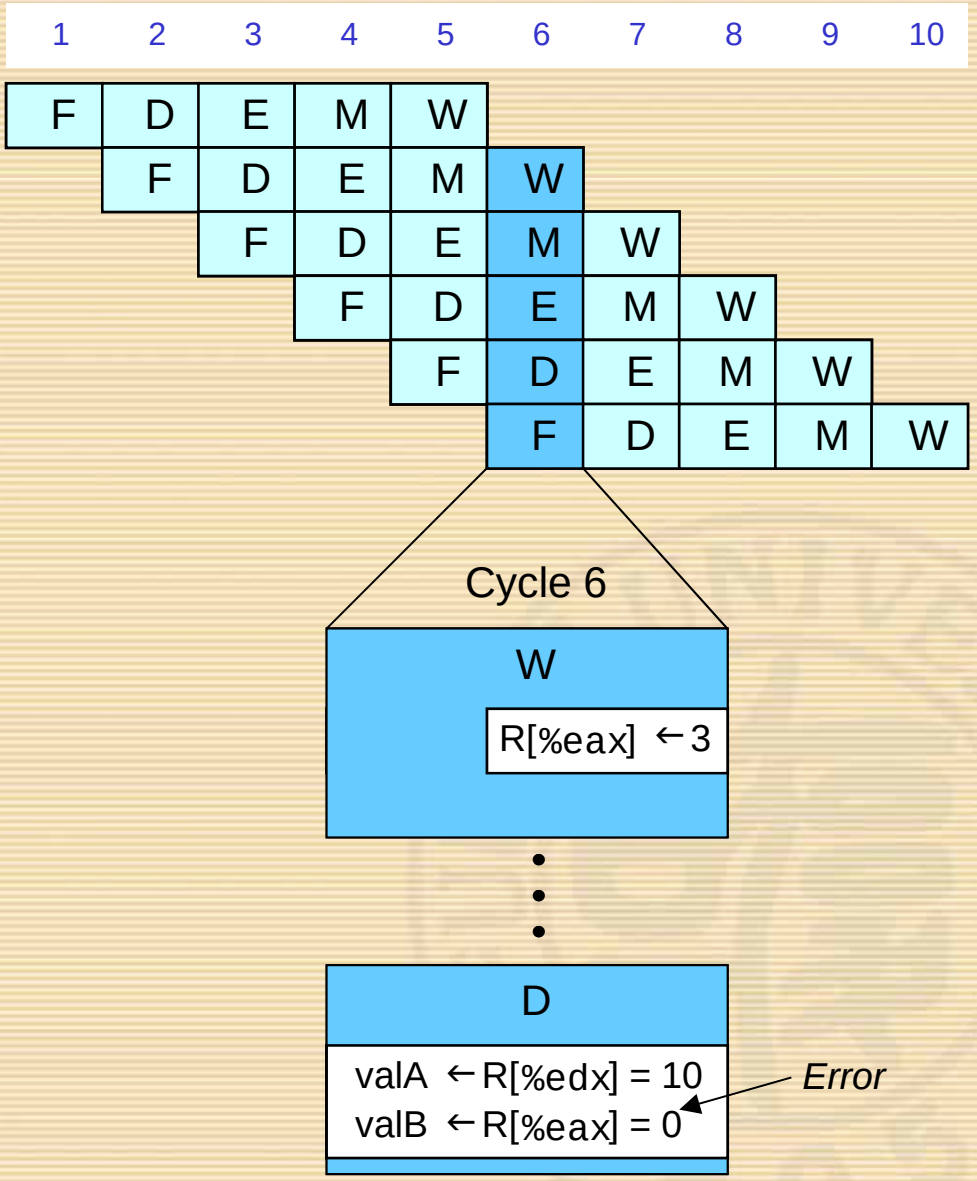
Classes of Data Hazards

- Hazards can potentially occur when one instruction updates part of the program state that read by a later instruction
- Program states:
  - Program registers
    - The hazards already identified.
  - Condition codes
    - Both written and read in the execute stage.
    - No hazards can arise
  - Program counter
    - Conflicts between updating and reading PC cause control hazards
  - Memory
    - Both written and read in the memory stage.
    - Without self-modified code, no hazards.

Data Dependencies: 2 nops

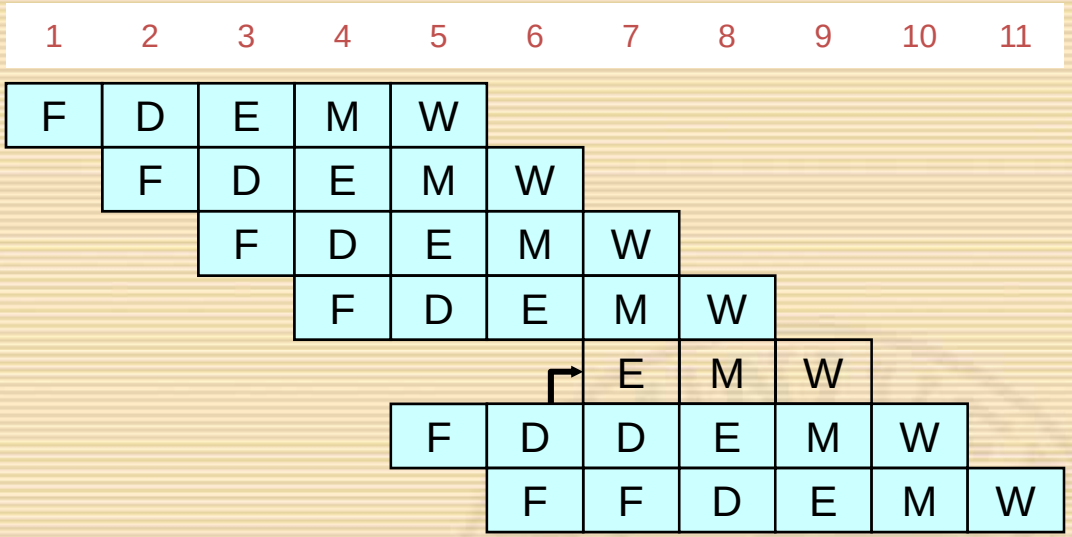
```
# demo-h2.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: nop
0x00e: addl %edx,%eax
0x010: halt
```

Figure 4.43 P325



Stalling for Data Dependencies

```
# demo-h2.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: nop
       bubble
0x00e: addl %edx,%eax
0x010: halt
```

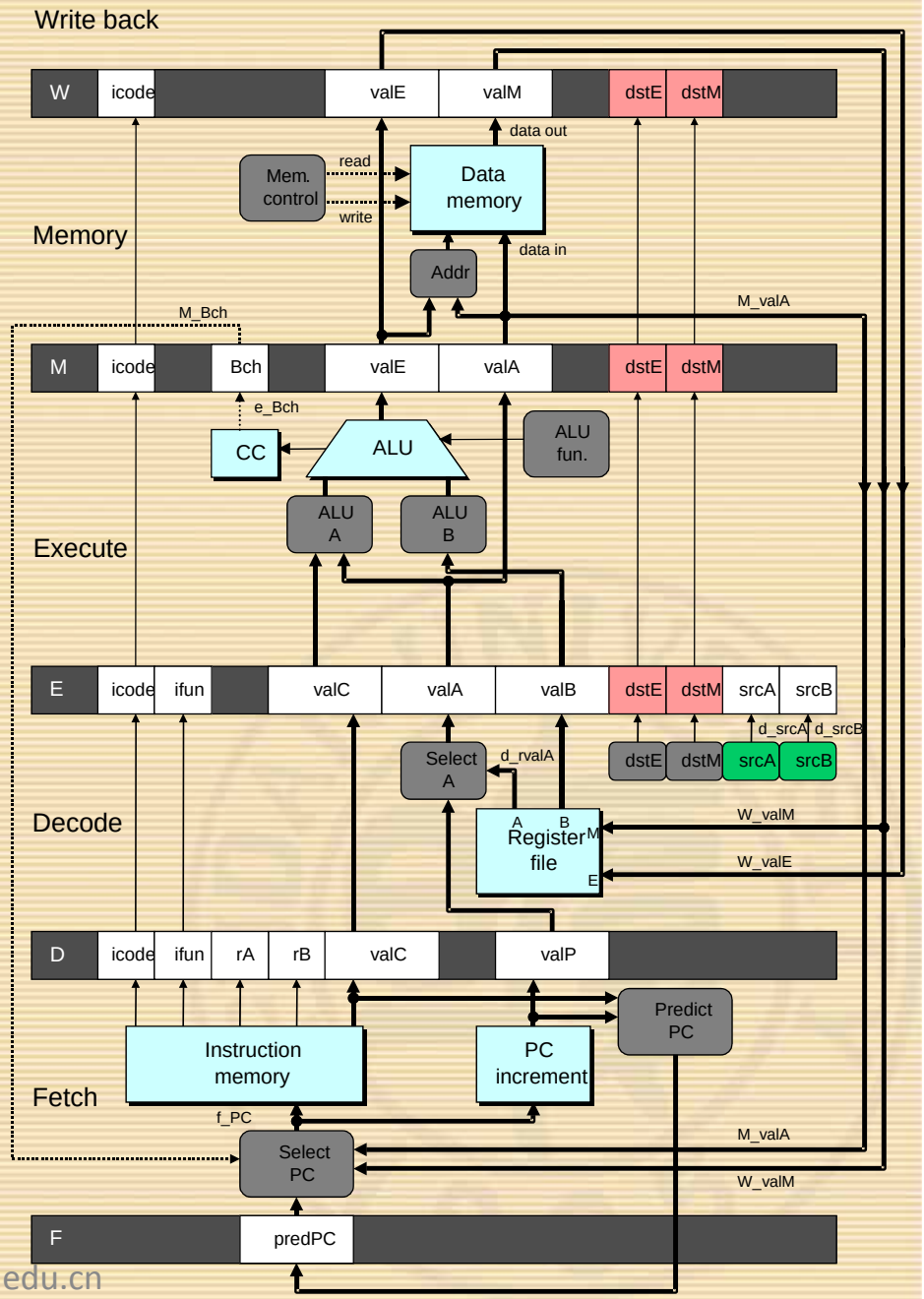


- If instruction follows too closely after one that writes register, slow it down
- Hold instruction in decode
- Dynamically inject nop into execute stage

Stall: 停止，迟延

Stall Condition

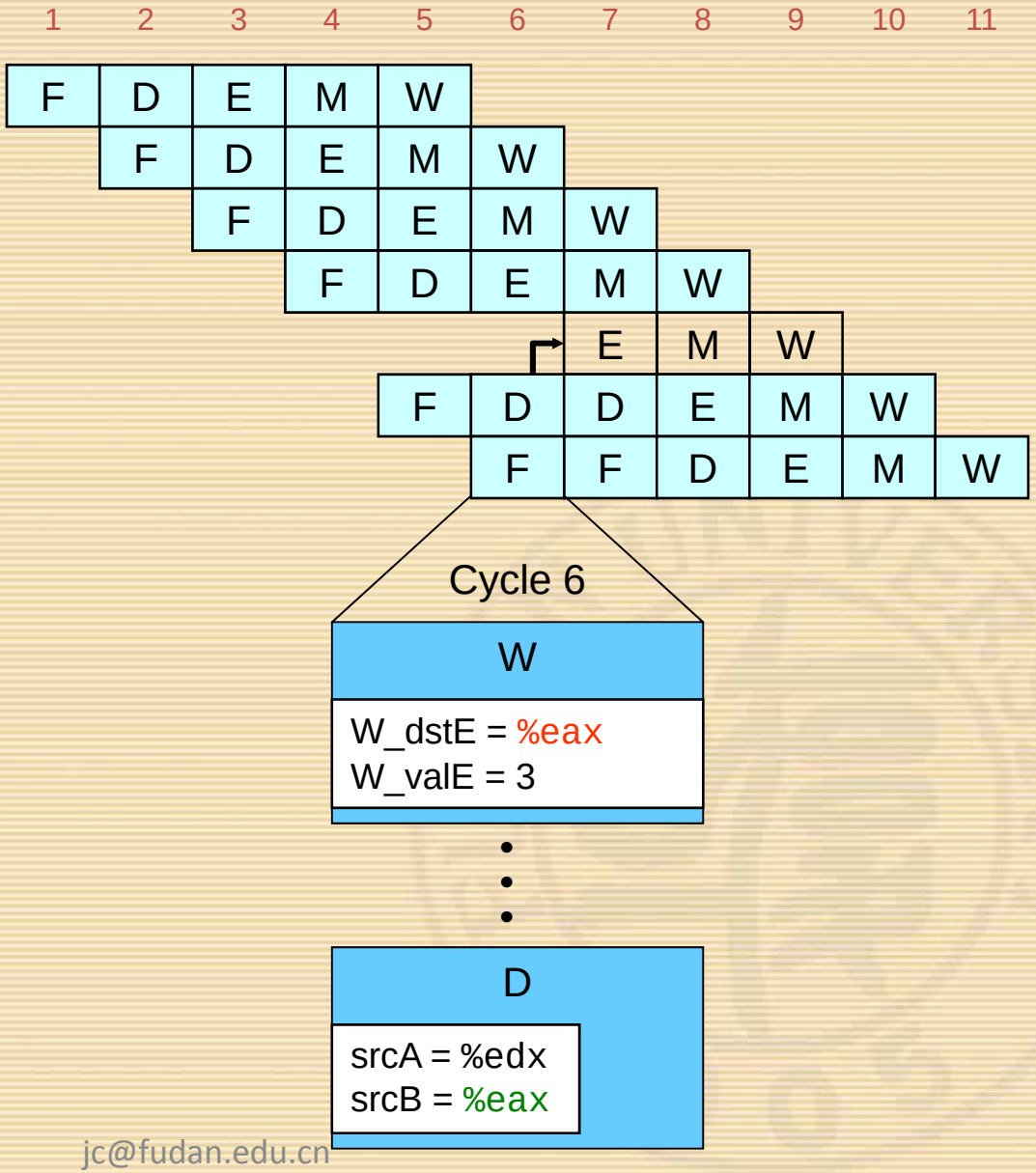
- Source Registers
  - srcA and srcB of current instruction in decode stage
- Destination Registers
  - dstE and dstM fields
  - Instructions in execute, memory, and write-back stages
- Condition
  - srcA==dstE or srcA==dstM
  - srcB==dstE or srcB==dstM
- Special Case
  - Don't stall for register ID 8
    - Indicates absence of register operand



Detecting Stall Condition

```
# demo-h2.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: nop
       bubble
0x00e: addl %edx,%eax
0x010: halt
```

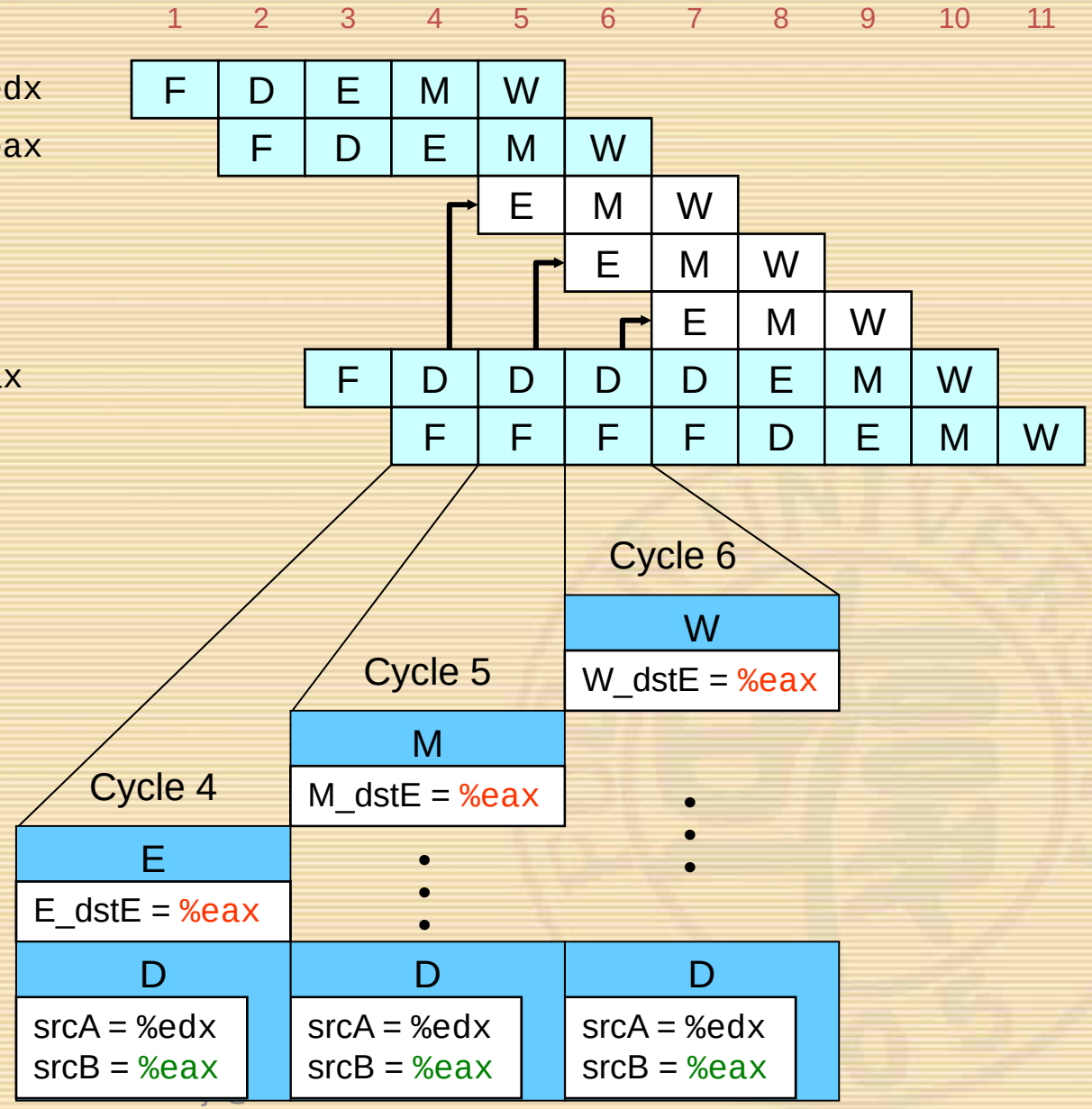
Figure 4.46 P328



Stalling X 3

```
# demo-h0.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
       bubble
       bubble
       bubble
0x00c: addl %edx,%eax
0x00e: halt
```

Figure 4.48 P329



What Happens When Stalling?

```
# demo-h0.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: addl %edx,%eax
0x00e: halt
```

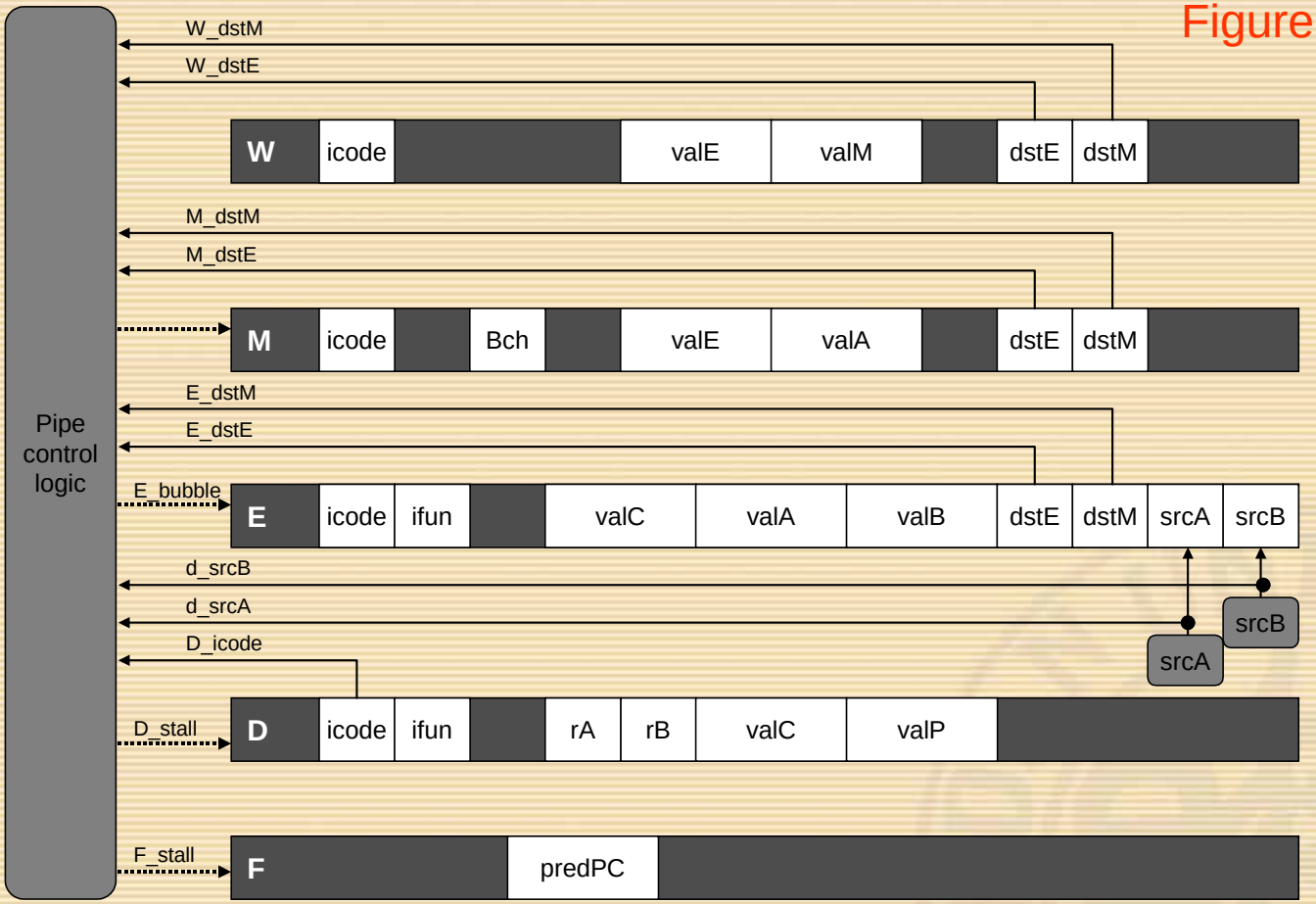
Figure 4.48 P329

	Cycle 8
Write Back	*bubble*
Memory	*bubble*
Execute	`0x00c: addl %edx,%eax`
Decode	`0x00e: halt`
Fetch	

- Stalling instruction held back in decode stage
- Following instruction stays in fetch stage
- Bubbles injected into execute stage
  - Like dynamically generated nop's
  - Move through later stages

Implementing Stalling

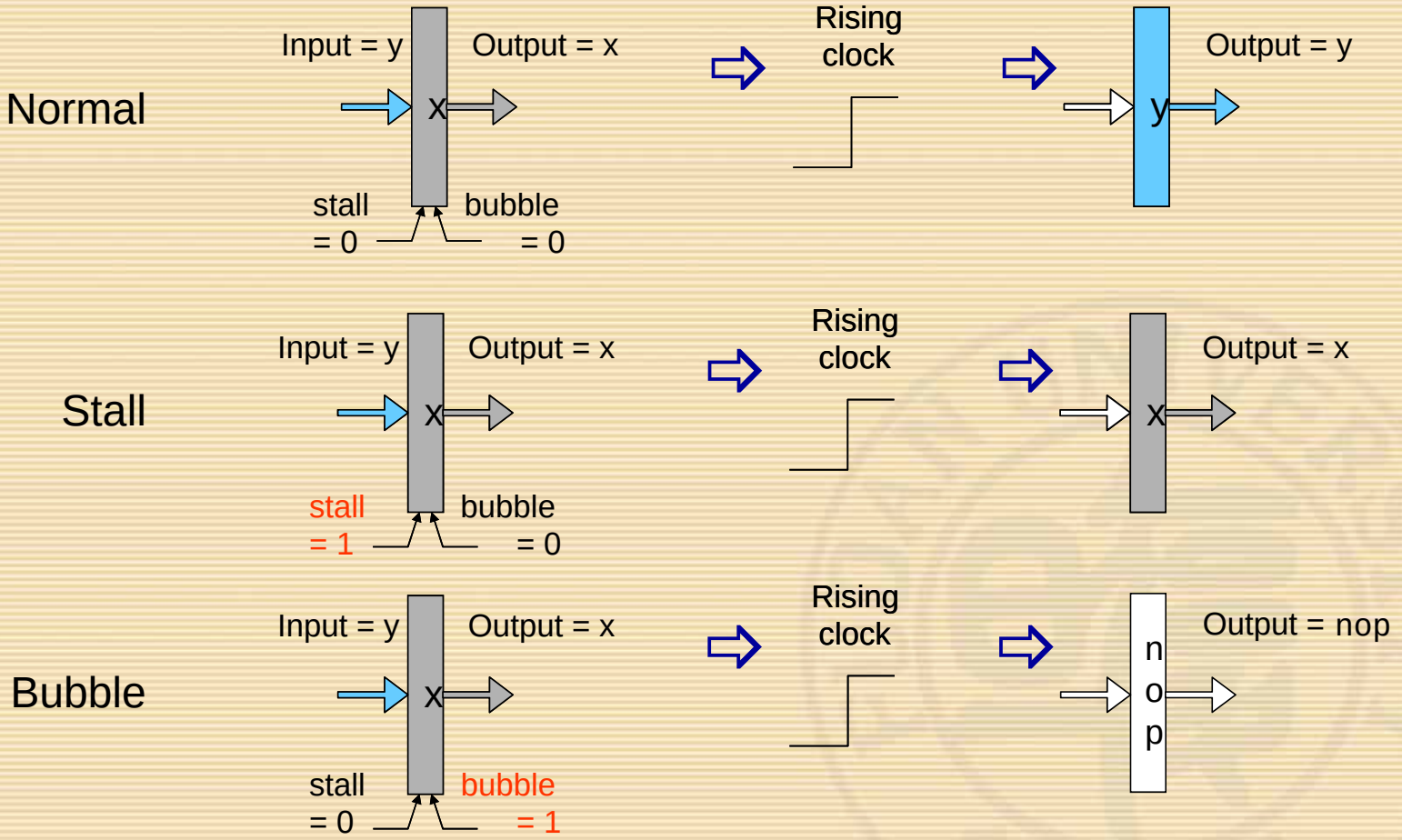
Figure 4.68 P351



- Pipeline Control
  - Combinational logic detects stall condition
  - Sets mode signals for how pipeline registers should update

Pipeline Register Modes

Figure 4.65 P348



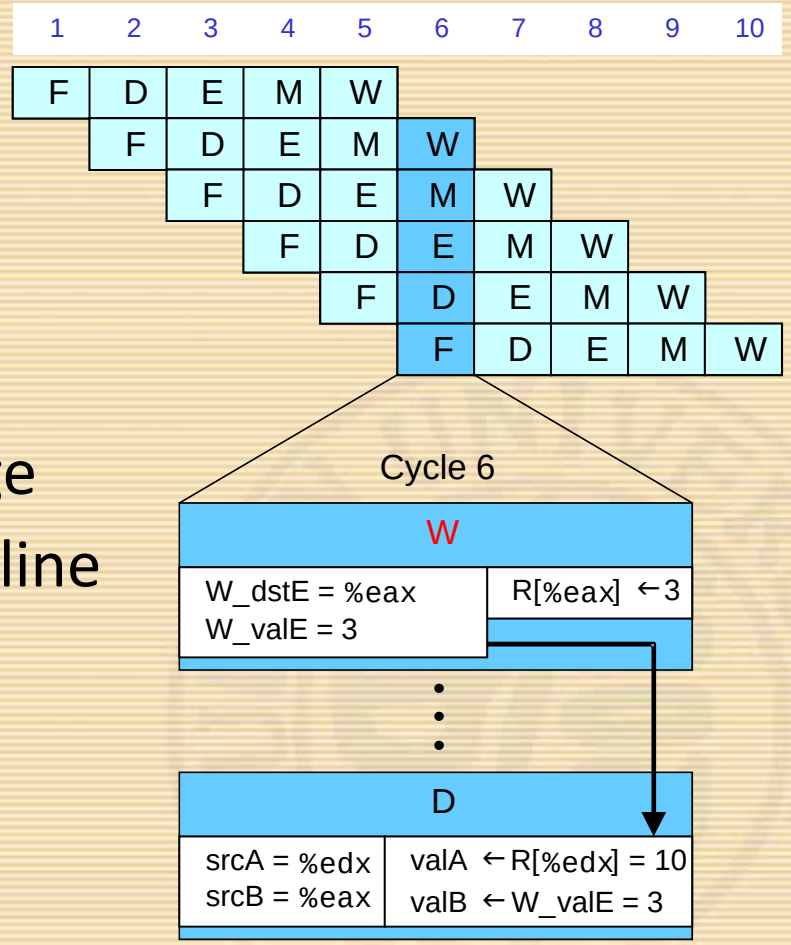
Data Forwarding

- Naive Pipeline
  - Register isn't written until completion of write-back stage
  - Source operands read from register file in decode stage
    - Needs to be in register file at start of stage
    - Performance is not good
- Observation
  - Value generated in execute or memory stage
- Trick
  - Pass value directly from generating instruction to decode stage
  - Needs to be available at end of decode stage

Data Forwarding Example

Figure 4.49 P331

```
# demo-h2.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: nop
0x00e: addl %edx,%eax
0x010: halt
```

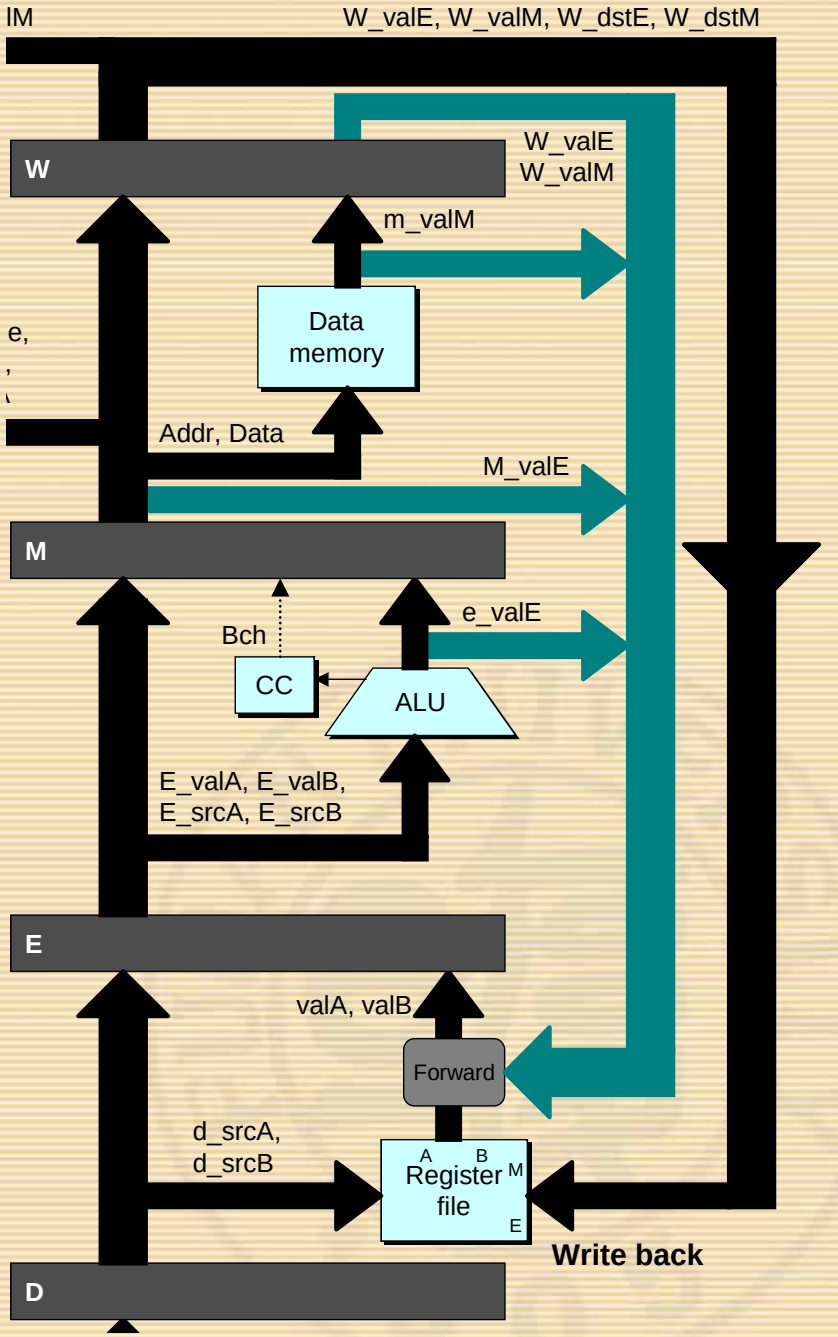


- `irmovl` in write-back stage
- Destination value in W pipeline register
- Forward as valB for decode stage

Bypass Paths

P333

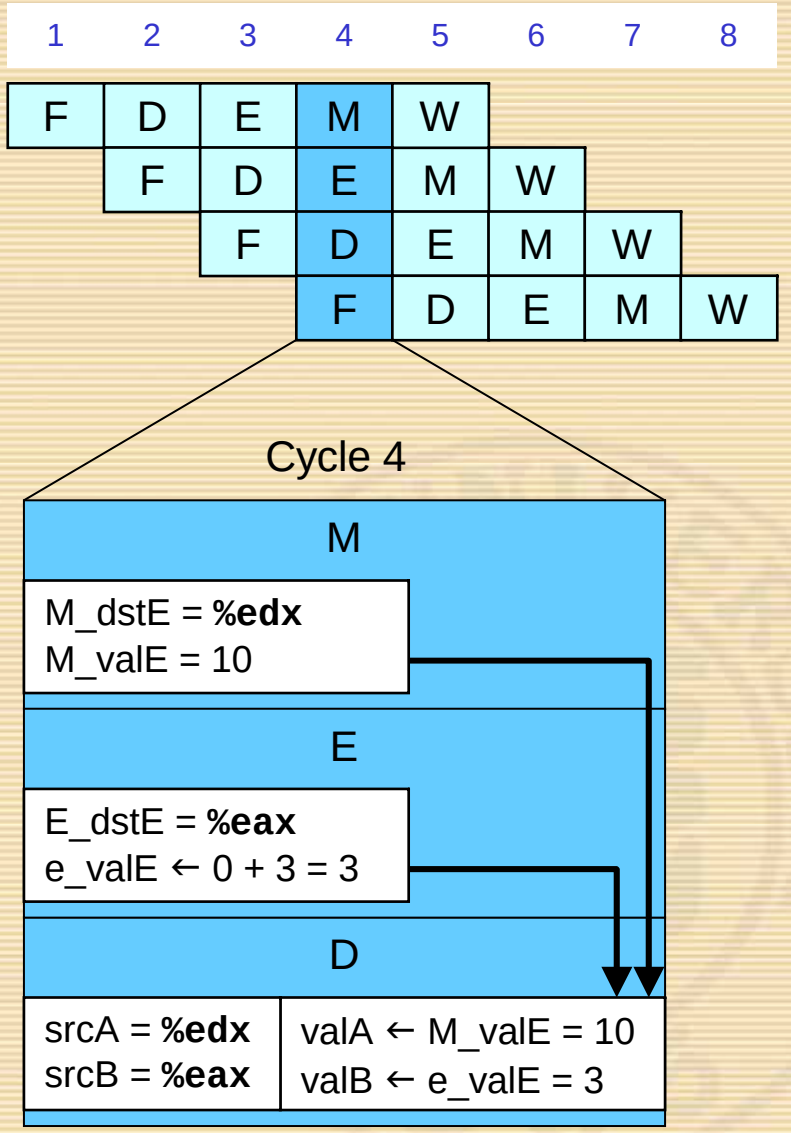
- Decode Stage
  - Forwarding logic selects valA and valB
  - Normally from register file
  - Forwarding: get valA or valB from later pipeline stage
- Forwarding Sources
  - Execute: valE
  - Memory: valE, valM
  - Write back: valE, valM



Data Forwarding Example #2

```
# demo-h0.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: addl %edx,%eax
0x00e: halt
```

Figure 4.51 P332



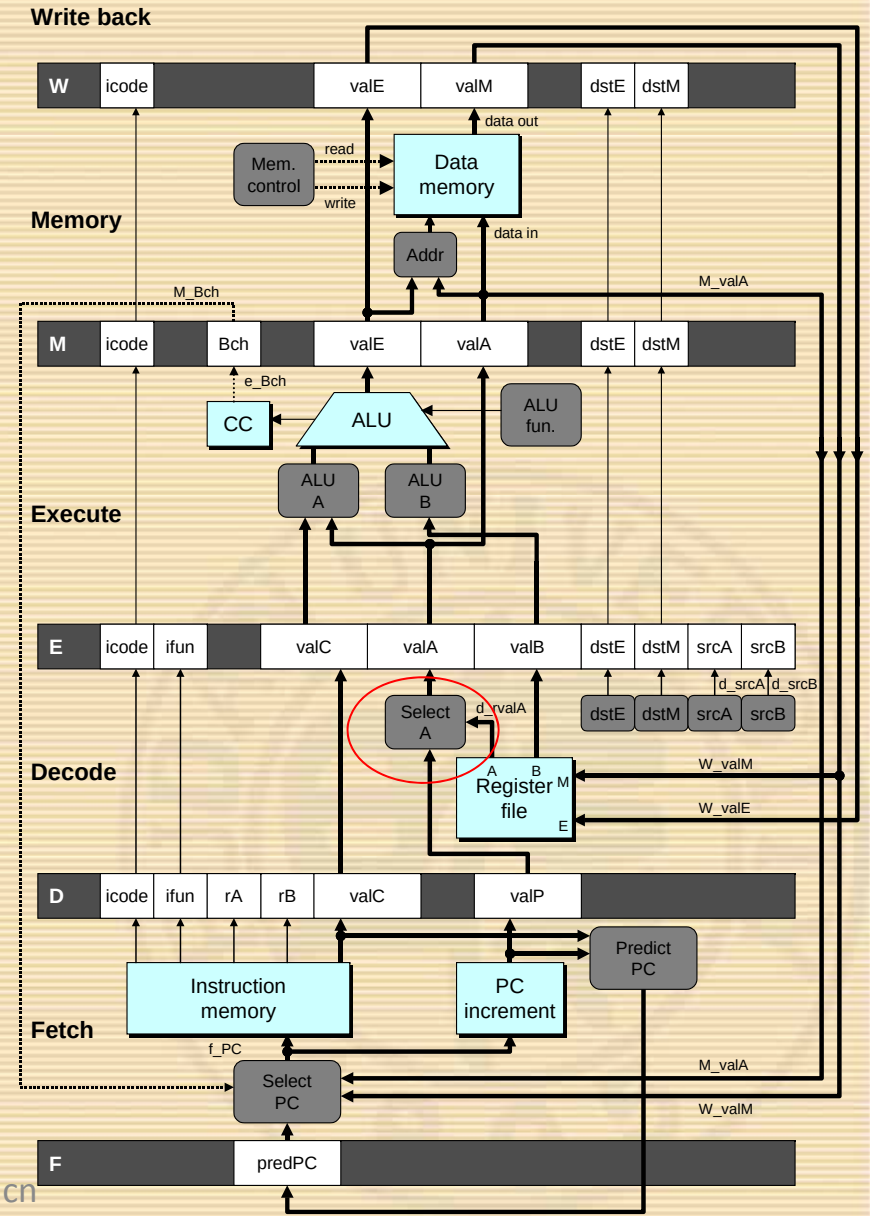
- Register %edx
  - Generated by ALU during previous cycle
  - Forward from memory stage as valA
- Register %eax
  - Value just generated by ALU
  - Forward from execute stage as valB

PIPE- Hardware

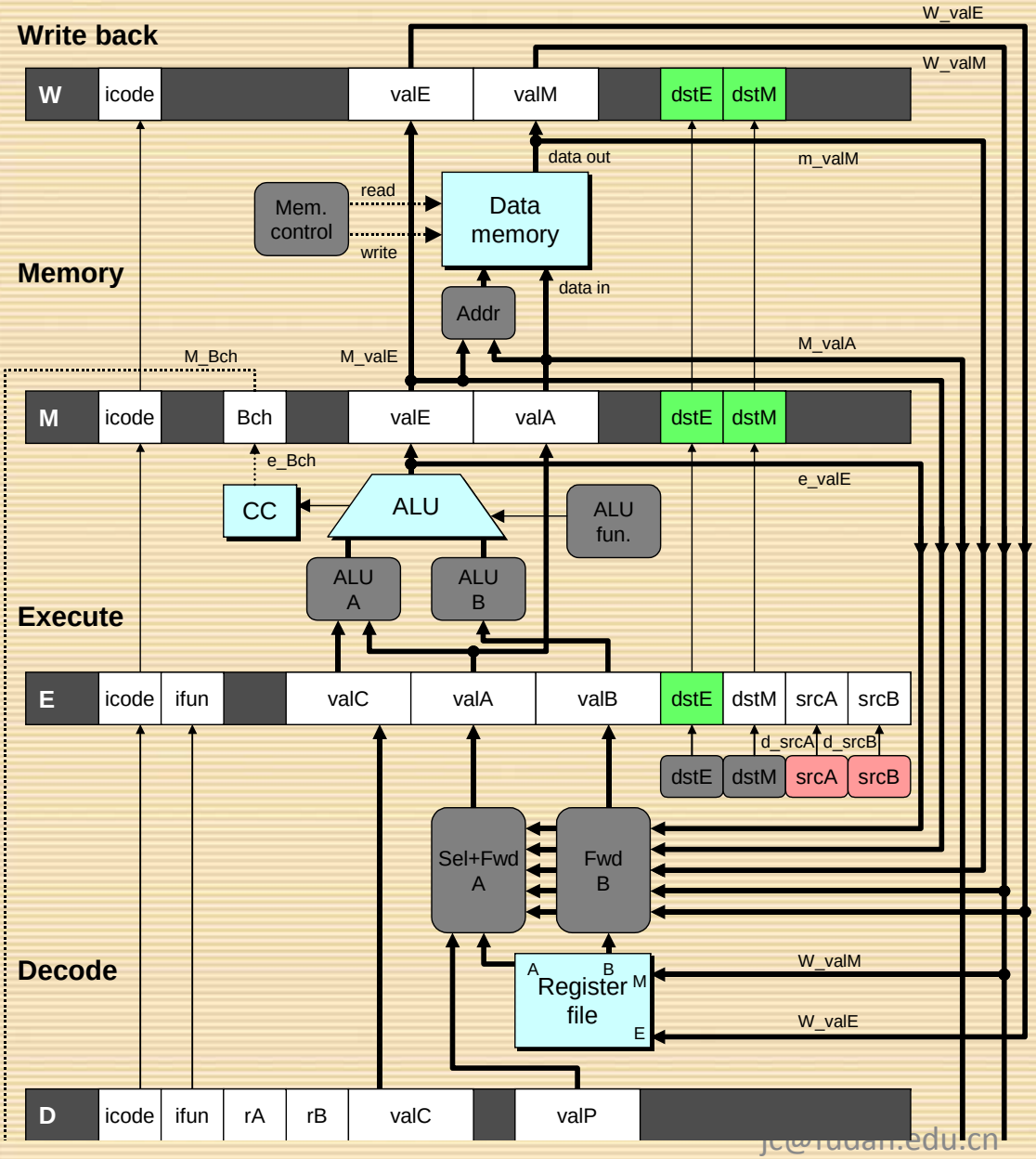
Figure 4.41 P320

- Pipeline registers hold intermediate values from instruction execution

- Forward (Upward) Paths
  - Values passed from one stage to next
  - Cannot jump past stages
    - e.g., valC passes through decode

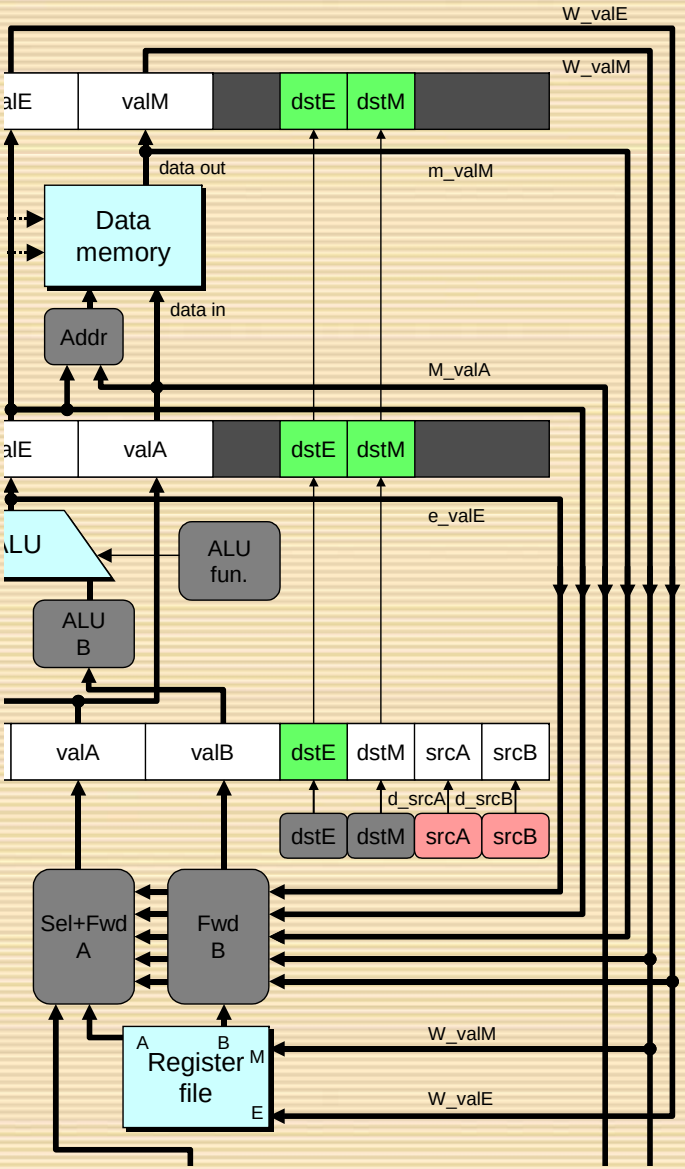


Implementing Forwarding Figure 4.53 P334



- Add additional feedback paths from E, M, and W pipeline registers into decode stage
- Create logic blocks to select from multiple sources for valA and valB in decode stage

Implementing Forwarding



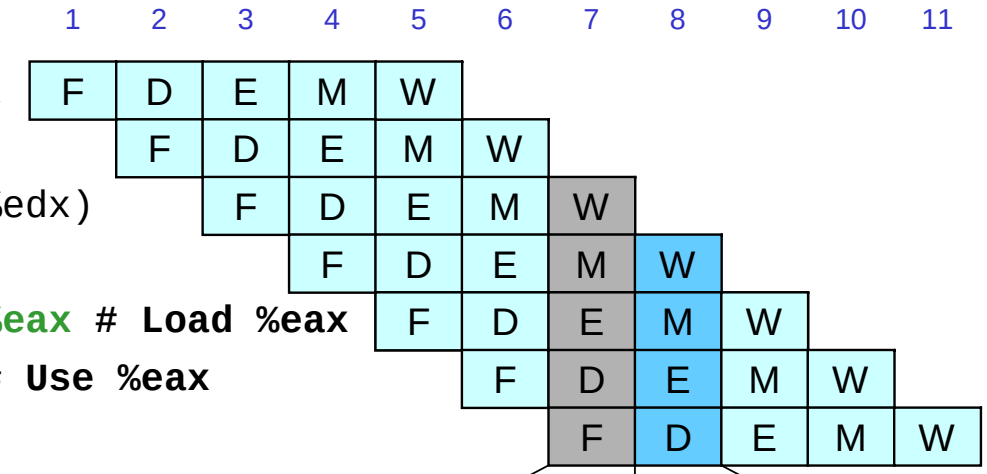
P340

```
## What should be the A value?
int new_E_valA = [
  # Use incremented PC
     D_icode in { ICALL, IJXX } : D_valP;
  # Forward valE from execute
     d_srcA == E_dstE : e_valE;
  # Forward valM from memory
     d_srcA == M_dstM : m_valM;
  # Forward valE from memory
     d_srcA == M_dstE : M_valE;
  # Forward valM from write back
     d_srcA == W_dstM : W_valM;
  # Forward valE from write back
     d_srcA == W_dstE : W_valE;
  # Use value read from register file
     1 : d_rvalA;
];
```

Figure 4.53 P334

Limitation of Forwarding

```
# demo-luh.ys
0x000: irmovl $128,%edx
0x006: irmovl  $3,%ecx
0x00c: rmmovl %ecx, 0(%edx)
0x012: irmovl $10,%ebx
0x018: mrmovl 0(%edx),%eax # Load %eax
0x01e: addl %ebx,%eax # Use %eax
0x020: halt
```



- Load-use dependency
  - Value needed by end of decode stage in cycle 7
  - Value read from memory in memory stage of cycle 8

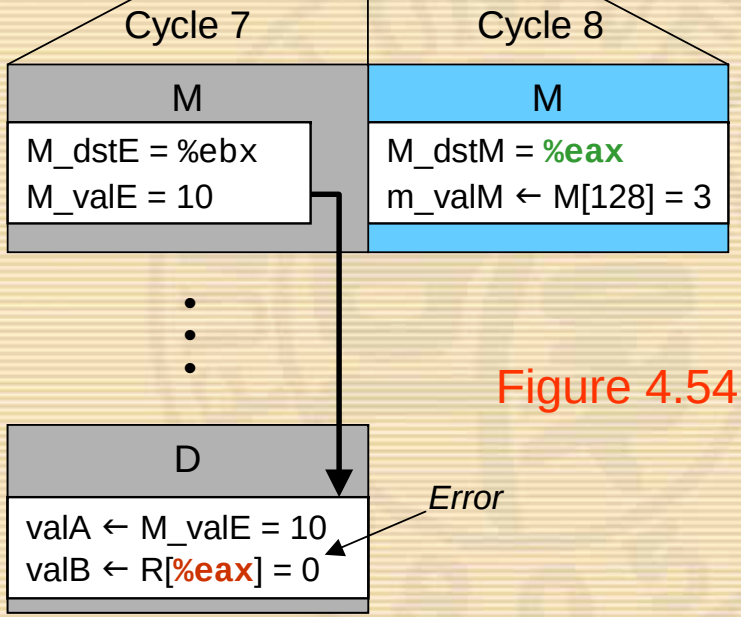
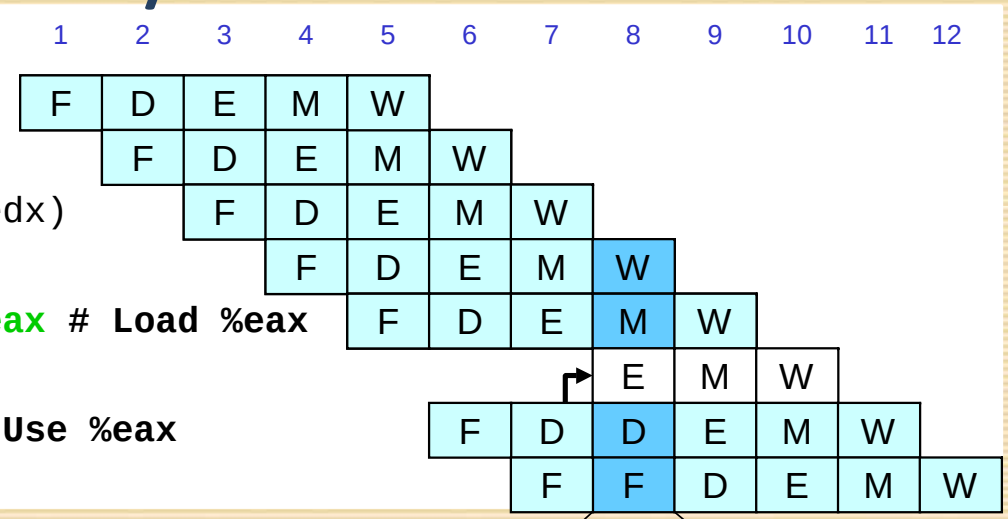


Figure 4.54 P335

Avoiding Load/Use Hazard

```
# demo-luh.ys
0x000: irmovl $128,%edx
0x006: irmovl  $3,%ecx
0x00c: rmmovl %ecx, 0(%edx)
0x012: irmovl $10,%ebx
0x018: mrmovl 0(%edx),%eax # Load %eax
       bubble
0x01e: addl %ebx,%eax # Use %eax
0x020: halt
```



- Stall using instruction for one cycle
- Can then pick up loaded value by forwarding from memory stage

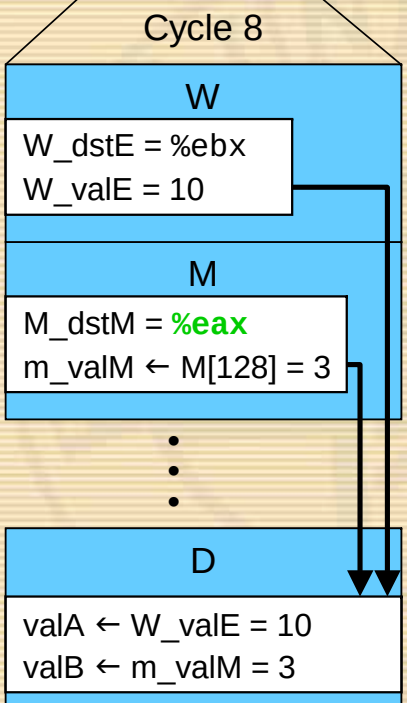
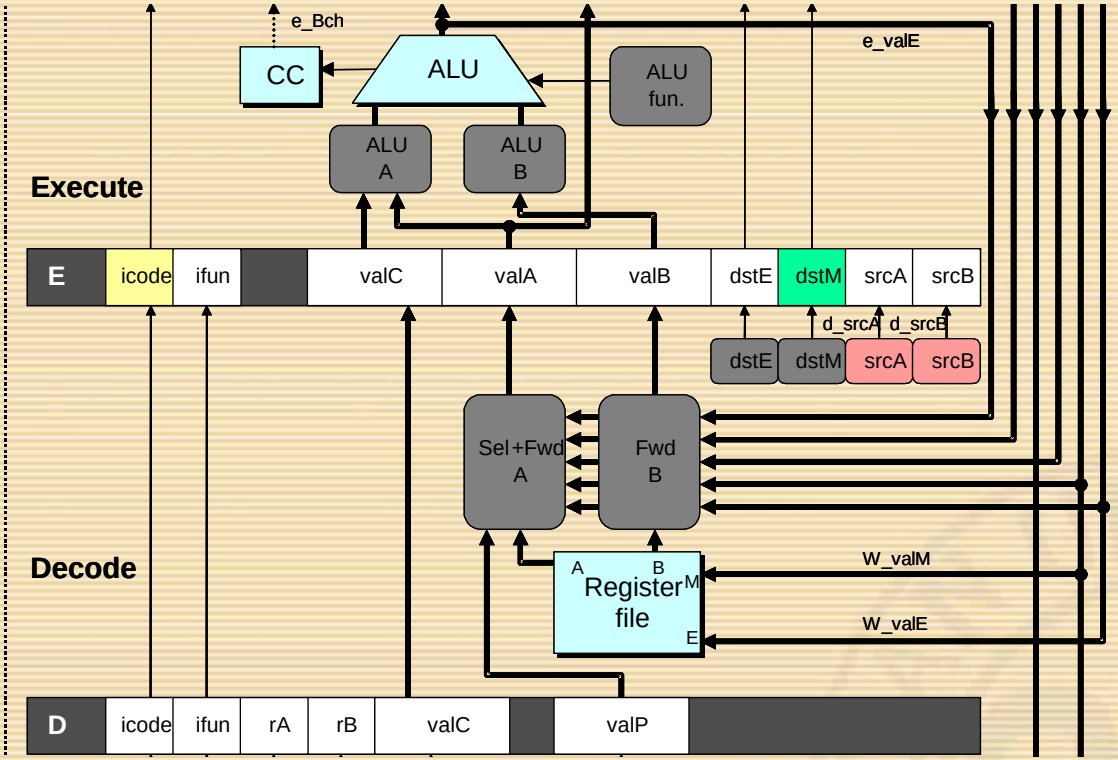


Figure 4.55 P336

Detecting Load/Use Hazard



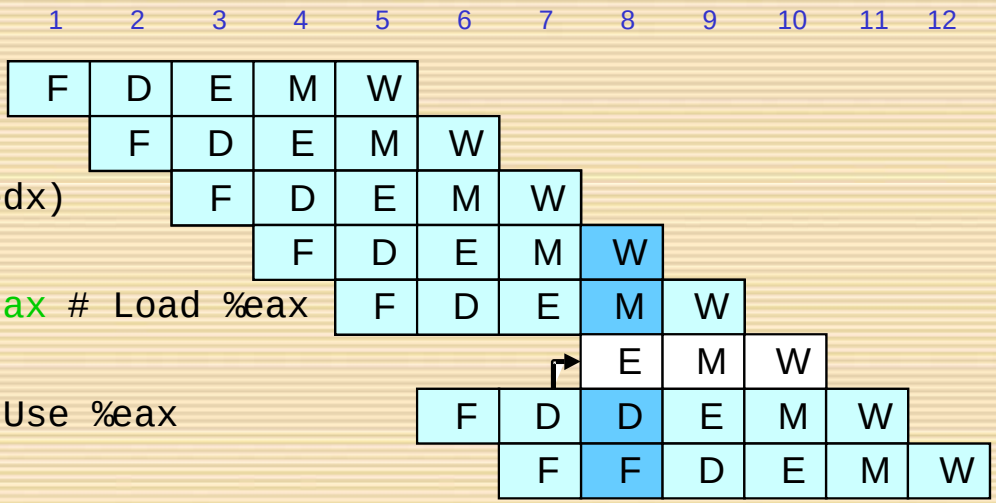
Condition	Trigger
Load/Use Hazard	E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB }

Figure 4.64 P347

Control for Load/Use Hazard

P345

```
# demo-luh.ys
0x000: irmovl $128,%edx
0x006: irmovl  $3,%ecx
0x00c: rmmovl %ecx, 0(%edx)
0x012: irmovl $10,%ebx
0x018: mrmovl 0(%edx),%eax # Load %eax
       bubble
0x01e: addl %ebx,%eax # Use %eax
0x020: halt
```



- Stall instructions in fetch and decode stages
- Inject bubble into

Condition	F	D	E	M	W
Load/Use Hazard	stall	stall	bubble	normal	normal

Figure 4.66 P348

Branch Misprediction Example

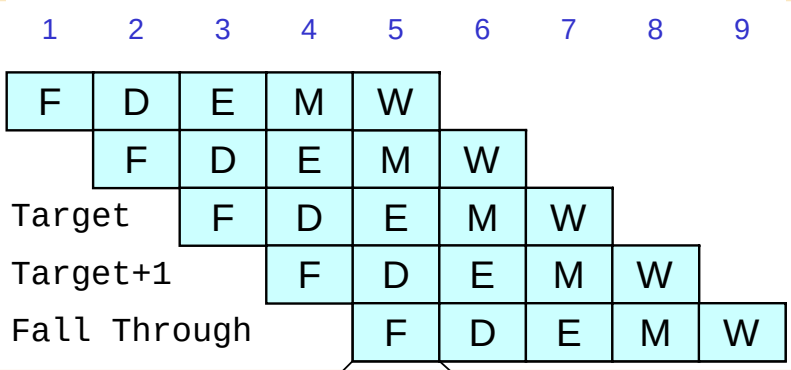
demo-j.ys

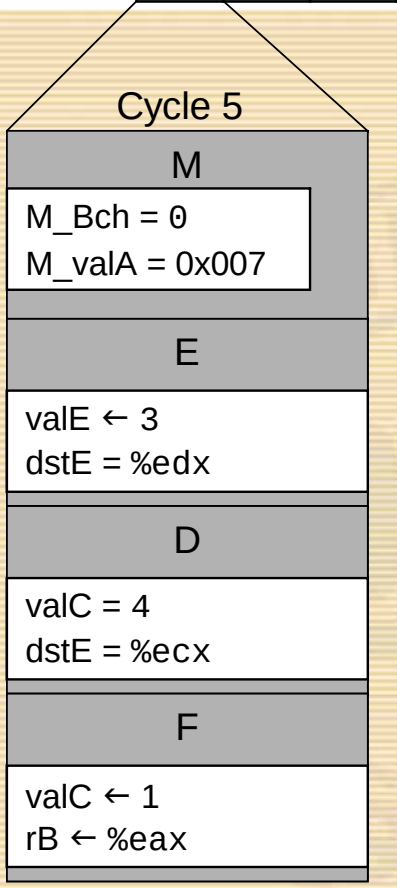
```
0x000:     xorl %eax,%eax
0x002:     jne  t                  # Not taken
0x007:     irmovl $1, %eax         # Fall through
0x00d:     nop
0x00e:     nop
0x00f:     nop
0x010:     halt
0x011: t:  irmovl $3, %edx         # Target (Should not execute)
0x017:     irmovl $4, %ecx         # Should not execute
0x01d:     irmovl $5, %edx         # Should not execute
```

– Should only execute first 7 instructions

Branch Misprediction Trace

```
# demo-j
0x000:     xorl %eax,%eax
0x002:     jne t # Not taken
0x011: t:  irmovl $3, %edx # Target
0x017:     irmovl $4, %ecx # Target+1
0x007:     irmovl $1, %eax # Fall Through
```





– Incorrectly execute two instructions at branch target

Return Example

demo-ret.ys

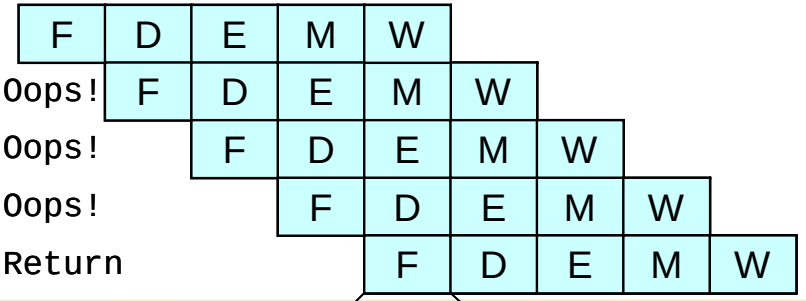
```
0x000:    irmovl Stack,%esp  # Initialize stack pointer
0x006:    nop                # Avoid hazard on %esp
0x007:    nop
0x008:    nop
0x009:    call p             # Procedure call
0x00e:    irmovl $5,%esi     # Return point
0x014:    halt
0x020: .pos 0x20
0x020: p: nop                   # procedure
0x021:    nop
0x022:    nop
0x023:    ret
0x024:    irmovl $1,%eax        # Should not be executed
0x02a:    irmovl $2,%ecx        # Should not be executed
0x030:    irmovl $3,%edx        # Should not be executed
0x036:    irmovl $4,%ebx        # Should not be executed
0x100: .pos 0x100
0x100: Stack:                   # Stack: Stack pointer
```

- Require lots of nops to avoid data hazards

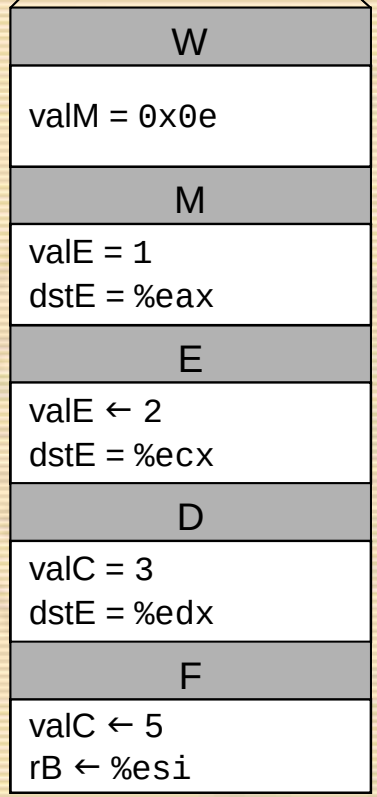
Incorrect Return Example

```
# demo-ret

0x023:    ret
0x024:    irmovl $1,%eax # Oops!
0x02a:    irmovl $2,%ecx # Oops!
0x030:    irmovl $3,%edx # Oops!
0x00e:    irmovl $5,%esi # Return
```



- Incorrectly execute 3 instructions following `ret`



Handling Misprediction

```
# demo-j.ys
0x000:     xorl %eax,%eax
0x002:     jne target # Not taken
0x011: t:  irmovl $2,%edx # Target
           bubble
0x017:     irmovl $3,%ebx # Target+1
           bubble
0x007:     irmovl $1,%eax # Fall through
0x00d:     nop
```

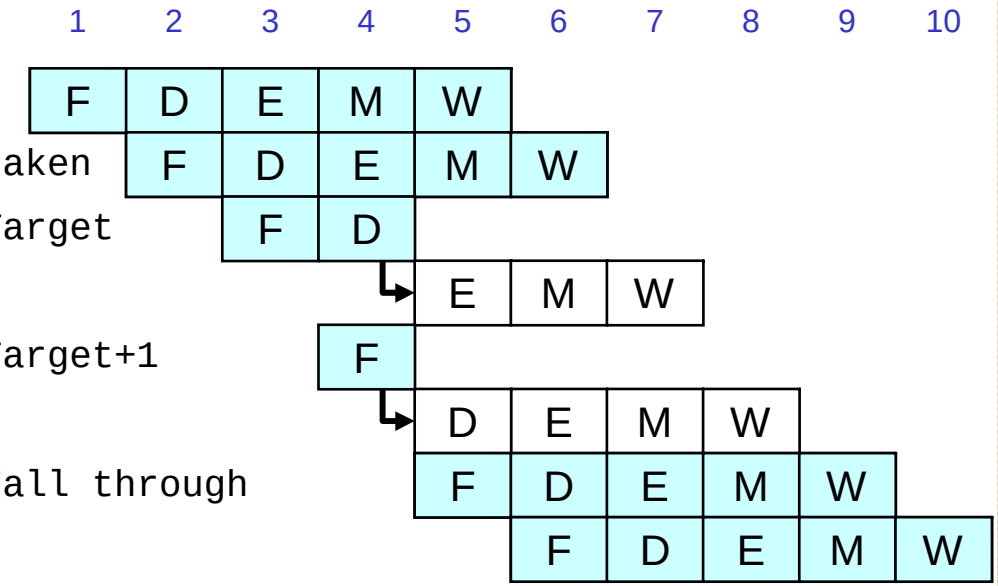


Figure 4.63 P346

- Predict branch as taken
  - Fetch 2 instructions at target
- Cancel when mispredicted
  - Detect branch not-taken in execute stage
  - On following cycle, replace instructions in execute and decode by bubbles
  - No side effects have occurred yet

Detecting Mispredicted Branch

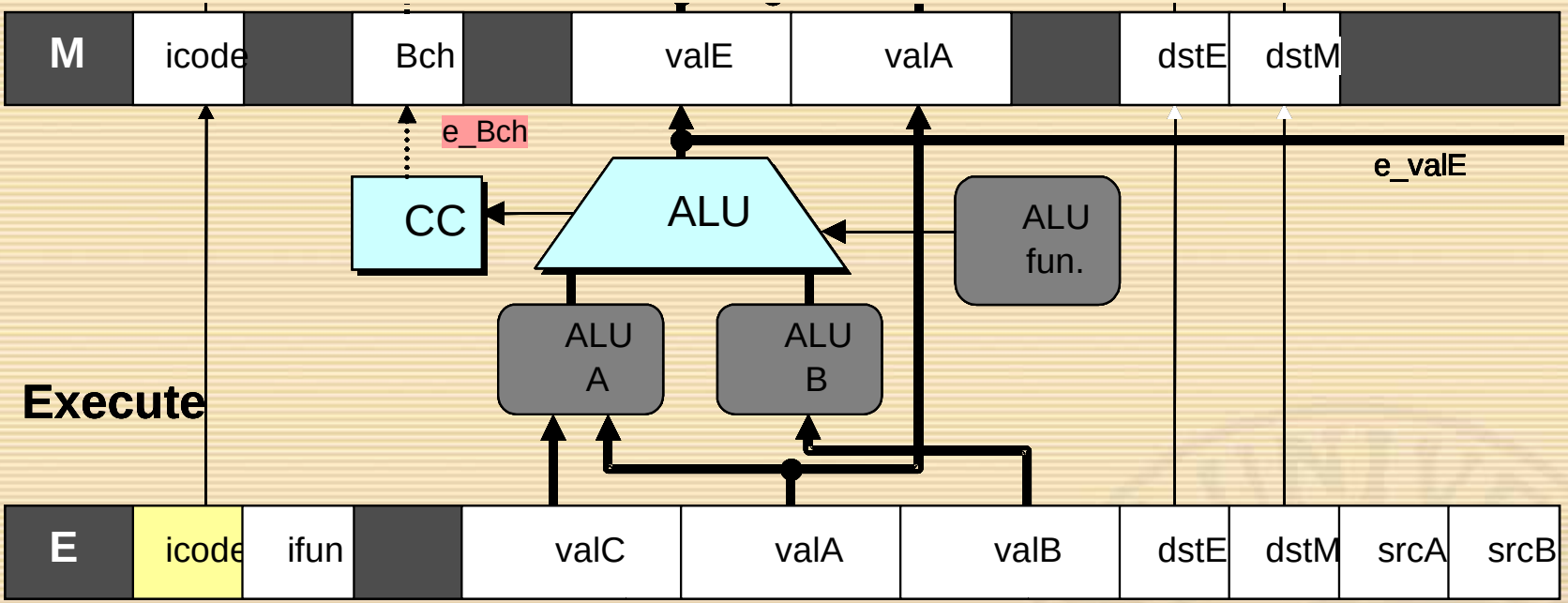


Figure 4.64 P347

Condition	Trigger
Mispredicted Branch	E_icode = IJXX & !e_Bch

Control for Misprediction

```
# demo-j.ys
0x000:     xorl %eax,%eax
0x002:     jne  t # Not taken
0x011: t:  irmovl $2,%edx # Target
           bubble
0x017:     irmovl $3,%ebx # Target+1
           bubble
0x007:     irmovl $1,%eax # Fall through
0x00d:     nop
```

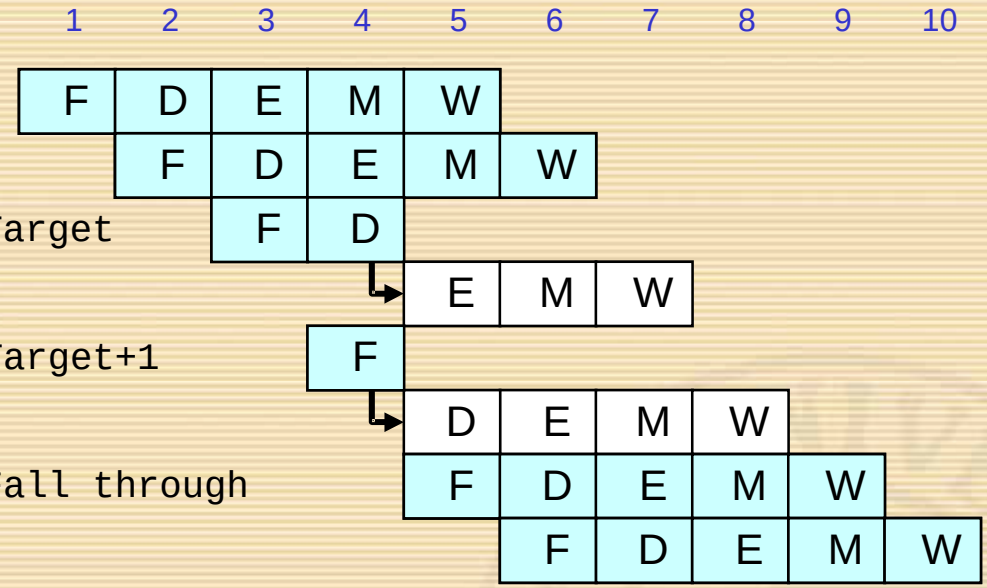


Figure 4.63 P346

Condition	F	D	E	M	W
Mispredicted Branch	normal	bubble	bubble	normal	normal

Figure 4.66 P348

Return Example

demo-retb.ys

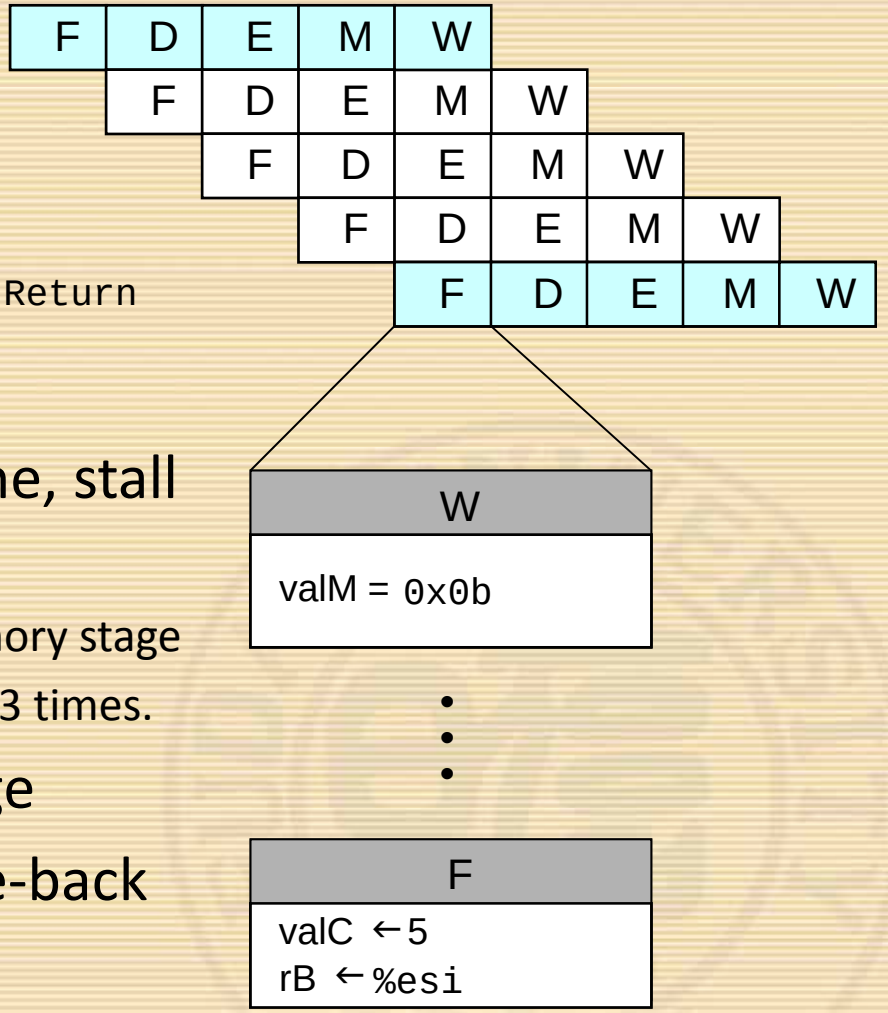
```
0x000:    irmovl Stack,%esp  # Initialize stack pointer
0x006:    call p             # Procedure call
0x00b:    irmovl $5,%esi     # Return point
0x011:    halt
0x020: .pos 0x20
0x020: p: irmovl $-1,%edi    # procedure
0x026:    ret
0x027:    irmovl $1,%eax     # Should not be executed
0x02d:    irmovl $2,%ecx     # Should not be executed
0x033:    irmovl $3,%edx     # Should not be executed
0x039:    irmovl $4,%ebx     # Should not be executed
0x100: .pos 0x100
0x100: Stack:                # Stack: Stack pointer
```

– Previously executed three additional instructions

Correct Return Example

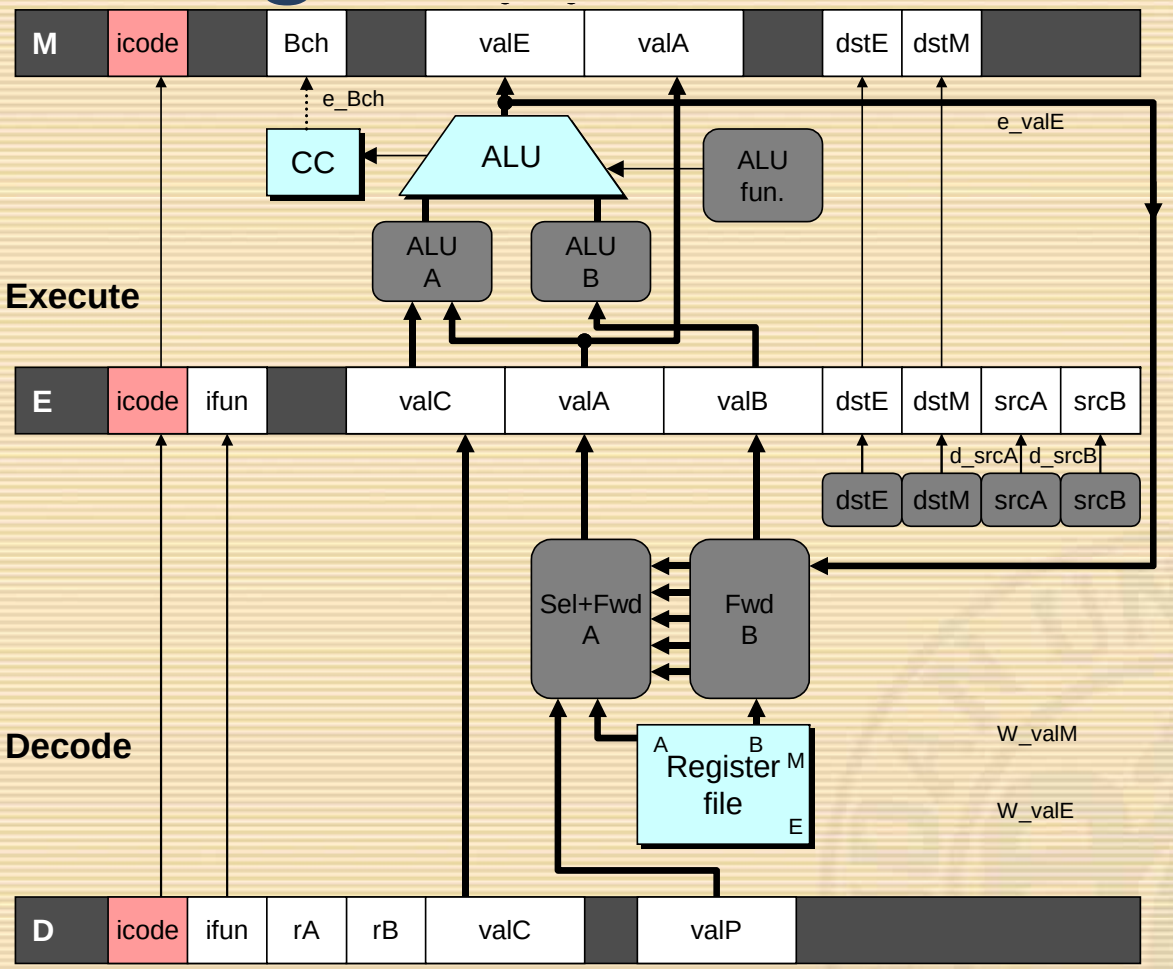
```
# demo-retb
0x026:    ret
          bubble
          bubble
          bubble
0x00b:    irmovl $5,%esi # Return
```

Figure 4.61 P344



- As `ret` passes through pipeline, stall at fetch stage
  - While in decode, execute, and memory stage
  - fetch the same instruction after ret 3 times.
- Inject bubble into decode stage
- Release stall when reach write-back stage

Detecting Return

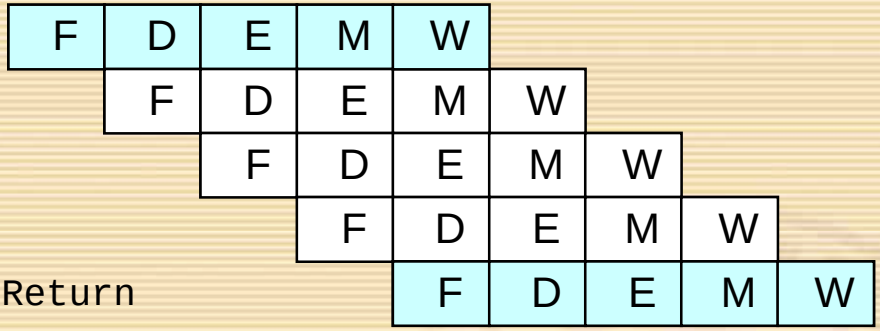


Condition	Trigger
Processing ret	IRET in { D_icode, E_icode, M_icode }

Figure 4.64 P347

Control for Return

```
# demo-retb
0x026:    ret
          bubble
          bubble
          bubble
0x00b:    irmovl $5,%esi # Return
```



Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal

Figure 4.66 P348

Special Control Cases

- Detection Figure 4.64 P347

Condition	Trigger
Processing ret	IRET in { D_icode, E_icode, M_icode }
Load/Use Hazard	E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB }
Mispredicted Branch	E_icode = IJXX & !e_Bch

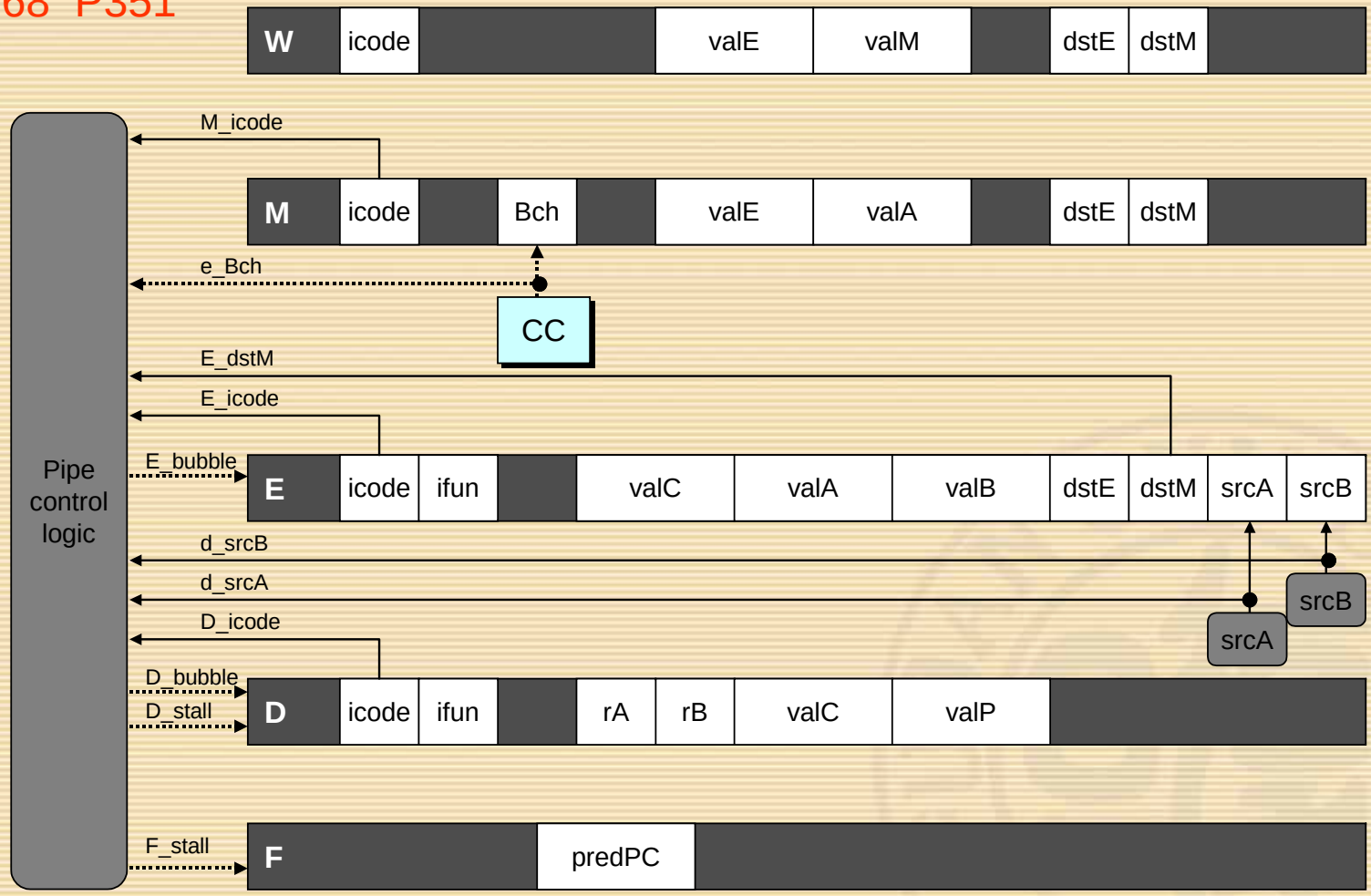
Figure 4.66 P348

-

Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Mispredicted Branch	normal	bubble	bubble	normal	normal

Implementing Pipeline Control

Figure 4.68 P351



- Combinational logic generates pipeline control signals
- Action occurs at start of following cycle

Initial Version of Pipeline Control

```
bool F_stall =
    # Conditions for a load/use hazard
    E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB } ||
    # Stalling at fetch while ret passes through pipeline
    IRET in { D_icode, E_icode, M_icode };

bool D_stall =
    # Conditions for a load/use hazard
    E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB };

bool D_bubble =
    # Mispredicted branch
    (E_icode == IJXX && !e_Bch) ||
    # Stalling at fetch while ret passes through pipeline
     IRET in { D_icode, E_icode, M_icode };

bool E_bubble =
    # Mispredicted branch
    (E_icode == IJXX && !e_Bch) ||
    # Load/use hazard
    E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB};
```

Control Combinations

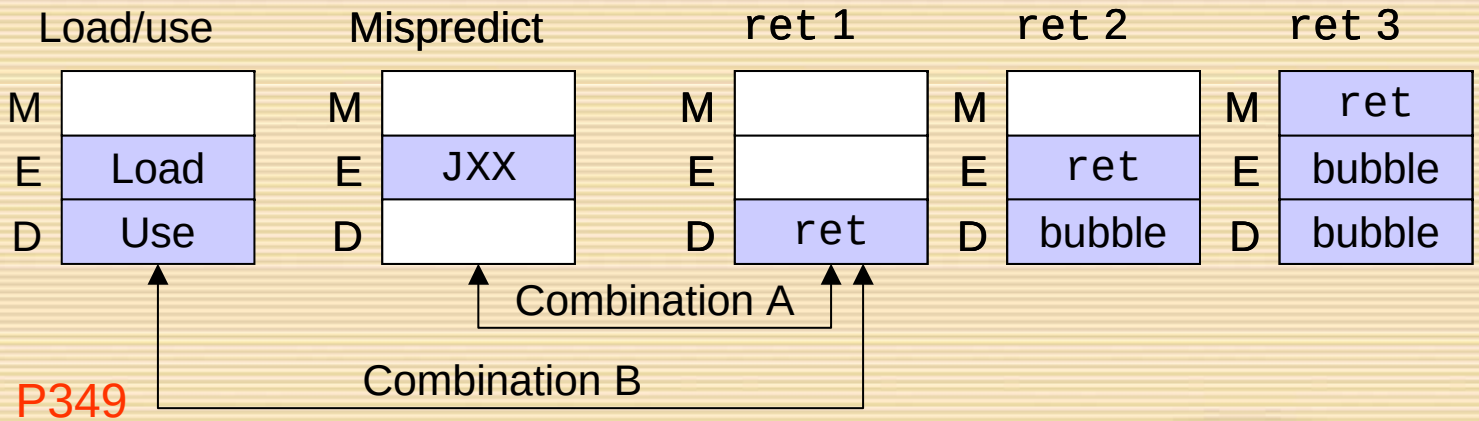
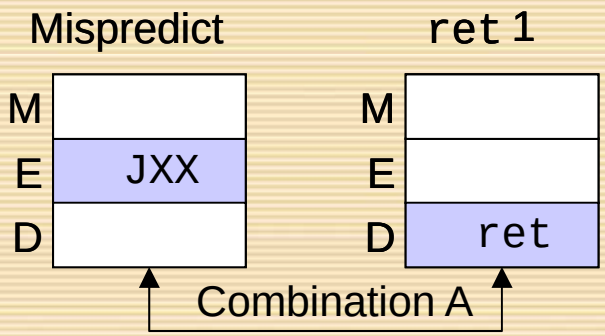
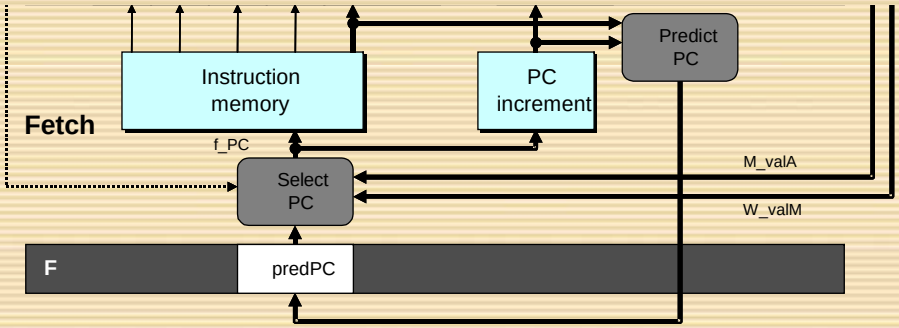


Figure 4.67 P349

- – Special cases that can arise on same clock cycle
- Combination A
  - – Not-taken branch
  - – `ret` instruction at branch target
- Combination B
  - – Instruction that reads from memory to %esp
  - – Followed by `ret` instruction

Control Combination A

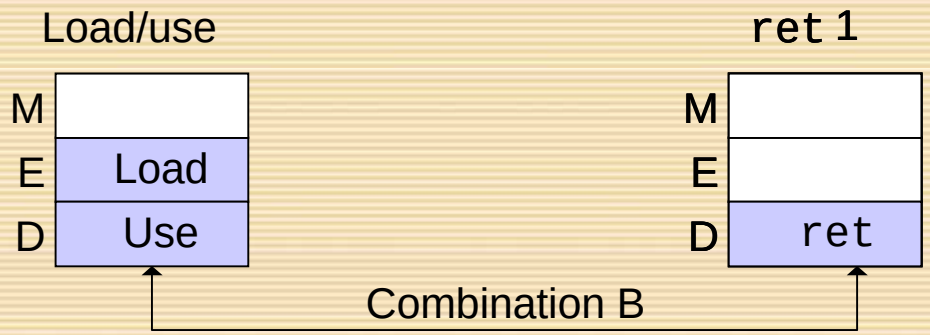




Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Mispredicted Branch	normal	bubble	bubble	normal	normal
Combination	*stall*	*bubble*	*bubble*	*normal*	*normal*

- Should be handled as mispredicted branch
- Stalls F pipeline register
- But PC selection logic will be using M_valA anyhow

Control Combination B

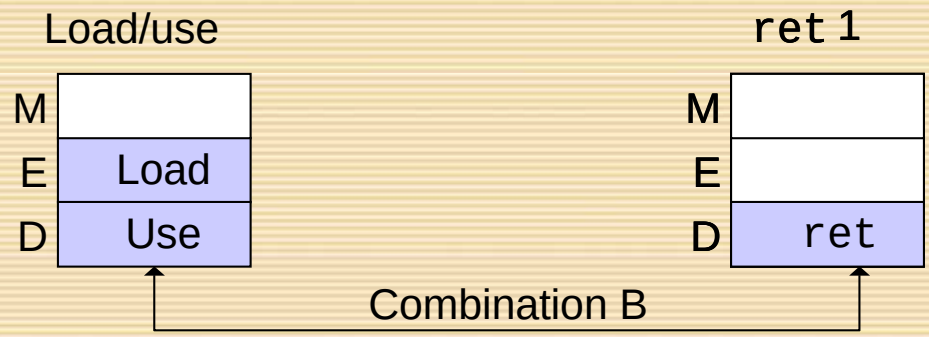


Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Combination	*stall*	*bubble + stall*	*bubble*	*normal*	*normal*

– Would attempt to bubble *and* stall pipeline register D

– Signaled by processor as pipeline error

Handling Control Combination B



Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Combination	*stall*	*stall*	*bubble*	*normal*	*normal*

- Load/use hazard should get priority
- `ret` instruction should be held in decode stage for additional cycle

Corrected Pipeline Control Logic

```
bool D_bubble =
     # Mispredicted branch
     (E_icode == IJXX && !e_Bch) ||
     # Stalling at fetch while ret passes through pipeline
      IRET in { D_icode, E_icode, M_icode }
       # but not condition for a load/use hazard
       && !(E_icode in { IMRMOVL, IPOPL }
            && E_dstM in { d_srcA, d_srcB });
```

Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Combination	*stall*	*stall*	*bubble*	*normal*	*normal*

- Load/use hazard should get priority
- `ret` instruction should be held in decode stage for additional cycle

Pipeline Summary

- Data Hazards
  - Most handled by forwarding
    - No performance penalty
  - Load/use hazard requires one cycle stall
- Control Hazards
  - Cancel instructions when detect mispredicted branch
    - Two clock cycles wasted
  - Stall fetch stage while `ret` passes through pipeline
    - Three clock cycles wasted
- Control Combinations
  - Must analyze carefully
  - First version had subtle bug
    - Only arises with unusual instruction combination

Performance Metrics

- Clock rate
  - Measured in Megahertz or Gigahertz
  - Function of stage partitioning and circuit design
    - Keep amount of work per stage small
- Rate at which instructions executed
  - CPI: cycles per instruction
  - On average, how many clock cycles does each instruction require?
  - Function of pipeline design and benchmark programs
    - E.g., how frequently are branches mispredicted?

CPI for PIPE

- CPI ≈ 1.0
  - Fetch instruction each clock cycle
  - Effectively process new instruction almost every cycle
    - Although each individual instruction has latency of 5 cycles
- CPI > 1.0
  - Sometimes must stall or cancel branches
- Computing CPI
  - C clock cycles
  - I instructions executed to completion
  - B bubbles injected (C = I + B)

$$CPI = C/I = (I+B)/I = 1.0 + B/I$$

  - Factor B/I represents average penalty due to bubbles

CPI for PIPE (Cont.)

B/I = LP + MP + RP

Typical Values

- LP: Penalty due to load/use hazard stalling
  - Fraction of instructions that are loads — 0.25
  - Fraction of load instructions requiring stall — 0.20
  - Number of bubbles injected each time — 1

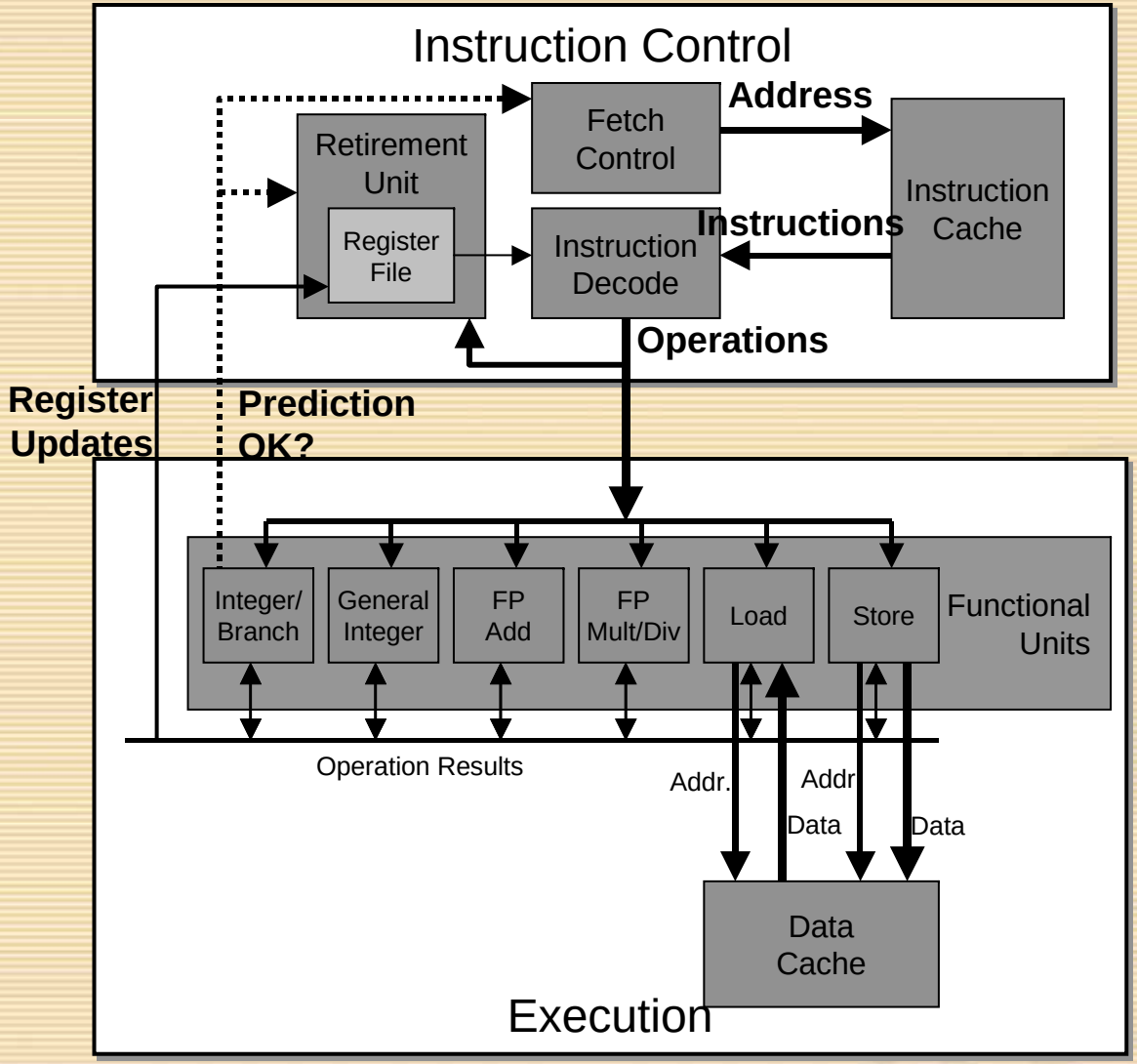
  ⇒ LP = 0.25 * 0.20 * 1 = 0.05
- MP: Penalty due to mispredicted branches
  - Fraction of instructions that are cond. jumps — 0.20
  - Fraction of cond. jumps mispredicted — 0.40
  - Number of bubbles injected each time — 2

  ⇒ MP = 0.20 * 0.40 * 2 = 0.16
- RP: Penalty due to `ret` instructions
  - Fraction of instructions that are returns — 0.02
  - Number of bubbles injected each time — 3

  ⇒ RP = 0.02 * 3 = 0.06
- Net effect of penalties 0.05 + 0.16 + 0.06 = 0.27

  ⇒ CPI = 1.27 (Not bad!)S

Modern CPU Design



Processor Summary

- Design Technique
  - Create uniform framework for all instructions
    - Want to share hardware among instructions
  - Connect standard logic blocks with bits of control logic
- Operation
  - State held in memories and clocked registers
  - Computation done by combinational logic
  - Clocking of registers/memories sufficient to control overall behavior
- Enhancing Performance
  - Pipelining increases throughput and improves resource utilization
  - Must make sure maintains ISA behavior

Code Optimization I: Machine Independent Optimizations

jc@fudan.edu.cn

Basic Requirements of Codes

- Robustness
- Efficiency
- Extensibility

Why

- Constant factors
  - Easily see 10:1 performance range depending on how code is written
  - Must optimize at multiple levels:
    - algorithm, data representations, procedures, and loops
- Must understand system to optimize performance
  - How programs are compiled and executed
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality

Optimizing Compilers

- Provide efficient mapping of program to machine
  - register allocation
  - code selection and ordering
  - eliminating minor inefficiencies
- Don't (usually) improve asymptotic efficiency
  - up to programmer to select best overall algorithm
  - big-O savings are (often) more important than constant factors
    - but constant factors also matter
- Have difficulty overcoming "optimization blockers"
  - potential memory aliasing
  - potential procedure side-effects

Limitations of Optimizing Compilers

- Operate Under Fundamental Constraint
  - Must not cause any change in program behavior under any possible condition
  - Often prevents it from making optimizations when would only affect behavior under pathological conditions.
- Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles
  - e.g., data ranges may be more limited than variable types suggest
- Most analysis is performed only within procedures
  - whole-program analysis is too expensive in most cases
- Most analysis is based only on *static* information
  - compiler has difficulty anticipating run-time inputs
- When in doubt, the compiler must be conservative

Machine-Independent Optimizations

- Optimizations that you should do regardless of processor / compiler

- Code Motion
  - Reduce frequency with which computation performed
    - If it will always produce same result
    - Especially moving code out of loop

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

?

Machine-Independent Optimizations

- Optimizations that you should do regardless of processor / compiler

- Code Motion
  - Reduce frequency with which computation performed
    - If it will always produce same result
    - Especially moving code out of loop

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```



```
for (i = 0; i < n; i++) {
  int ni = n*i;
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
}
```

Compiler-Generated Code Motion

- Most compilers do a good job with array code + simple loop structures

- Code Generated by GCC

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

```
for (i = 0; i < n; i++) {
  int ni = n*i;
  int *p = a+ni;
  for (j = 0; j < n; j++)
    *p++ = b[j];
}
```

```
  imull %ebx,%eax         # i*n
  movl 8(%ebp),%edi       # a
  leal (%edi,%eax,4),%edx # p = a+i*n (scaled by 4)
# Inner Loop
.L40:
  movl 12(%ebp),%edi      # b
  movl (%edi,%ecx,4),%eax # b+j  (scaled by 4)
  movl %eax,(%edx)        # *p = b[j]
  addl $4,%edx            # p++  (scaled by 4)
  incl %ecx               # j++
  jl .L40                 # loop if j<n
```

Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide

  `16*x --> x << 4`

  - Utility machine dependent
  - Depends on cost of multiply or divide instruction
  - On Pentium II or III, integer multiply only requires 4 CPU cycles
- Recognize sequence of products

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```


?

Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide

  `16*x --> x << 4`

  - Utility machine dependent
  - Depends on cost of multiply or divide instruction
  - On Pentium II or III, integer multiply only requires 4 CPU cycles
- Recognize sequence of products

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

```
int ni = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
  ni += n;
}
```

Make Use of Registers

- Reading and writing registers is much faster than reading/writing memory

- Limitation
  - Compiler not always able to determine whether variable can be held in register
  - Possibility of *Aliasing*
  - See example later

Machine-Independent Opts. (Cont.)

- Share Common Subexpressions
  - Reuse portions of expressions
  - Compilers often not very sophisticated in exploiting arithmetic properties

```
/* Sum neighbors of i,j */
up =    val[(i-1)*n + j];
down =  val[(i+1)*n + j];
left =  val[i*n   + j-1];
right = val[i*n   + j+1];
sum = up + down + left + right;
```

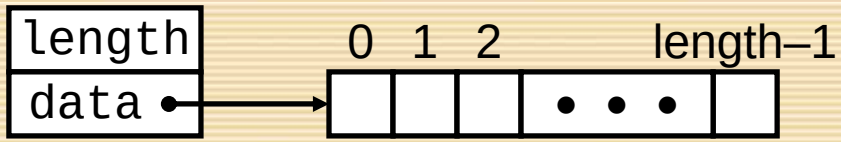
3 multiplications: i*n, (i–1)*n, (i+1)*n

```
int inj = i*n + j;
up =    val[inj - n];
down =  val[inj + n];
left =  val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

1 multiplication: i*n

```
  leal -1(%edx),%ecx  # i-1
  imull %ebx,%ecx     # (i-1)*n
  leal 1(%edx),%eax   # i+1
  imull %ebx,%eax     # (i+1)*n
  imull %ebx,%edx     # i*n
```

Vector ADT



- Procedures

  `vec_ptr new_vec(int len)`

  - Create vector of specified length

  `int get_vec_element(vec_ptr v, int index, int *dest)`

  - Retrieve vector element, store at *dest
  - Return 0 if out of bounds, 1 if successful

  `int *get_vec_start(vec_ptr v)`

  - Return pointer to start of vector data

  – Similar to array implementations in Pascal, ML, Java

  - E.g., always do bounds checking

Optimization Example

```
void combine1(vec_ptr v, int *dest)
{
  int i;
  *dest = 0;
  for (i = 0; i < vec_length(v); i++) {
    int val;
    get_vec_element(v, i, &val);
    *dest += val;
  }
}
```

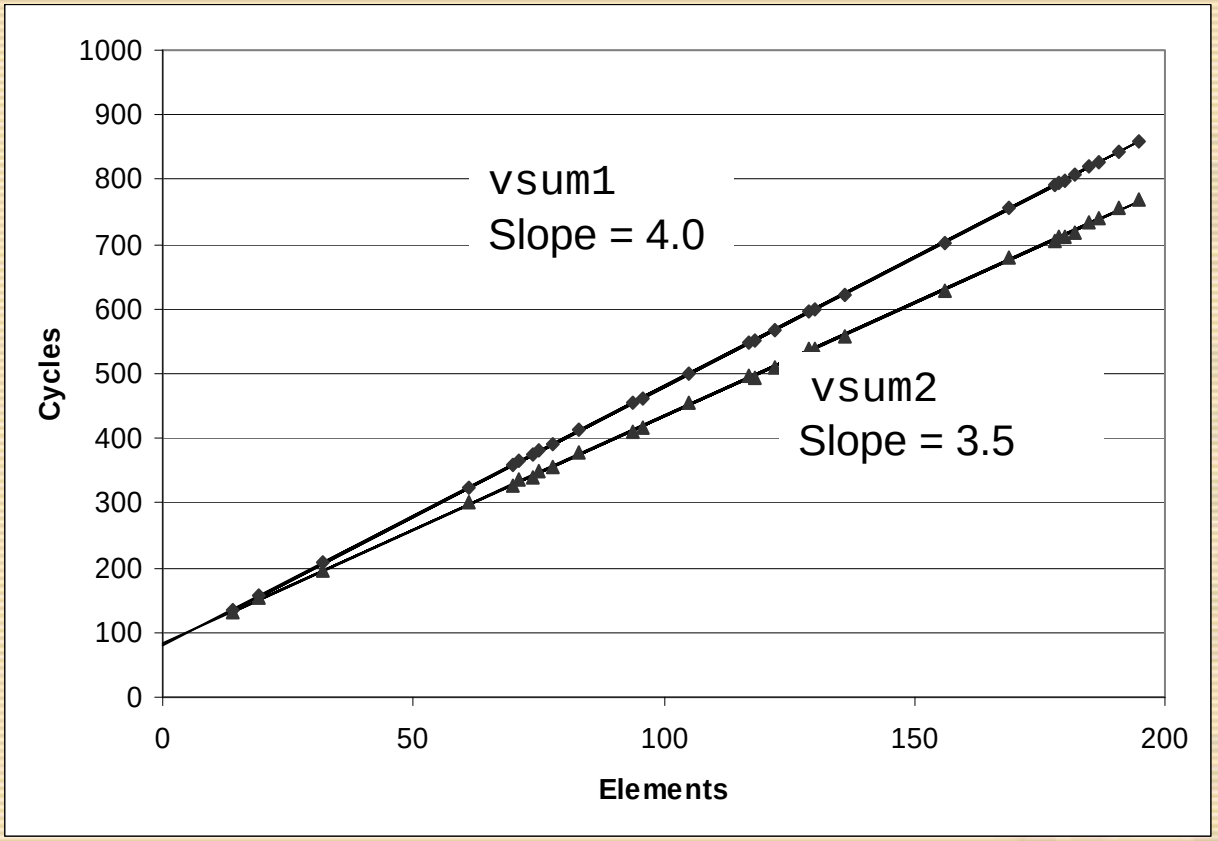
- Procedure
  - Compute sum of all elements of vector
  - Store result at destination location

Time Scales

- Absolute Time
  - Typically use nanoseconds
    - 10^{-9} seconds
  - Time scale of computer instructions
- Clock Cycles
  - Most computers controlled by high frequency clock signal
  - Typical Range
    - 100 MHz
      - 10^{8} cycles per second
      - Clock period = 10ns
    - 3.2 GHz
      - 3.2 X 10^{9} cycles per second
      - Clock period = 0.3125ns

Cycles Per Element

- Convenient way to express performance of program that operates on vectors or lists
- Length = n
- T = CPE*n + Overhead



Optimization Example

```
void combine1(vec_ptr v, int *dest)
{
  int i;
  *dest = 0;
  for (i = 0; i < vec_length(v); i++) {
    int val;
    get_vec_element(v, i, &val);
    *dest += val;
  }
}
```

- Procedure
  - Compute sum of all elements of integer vector
  - Store result at destination location
  - Vector data structure and operations defined via abstract data type
- Pentium II/III Performance: Clock Cycles / Element
  - 42.06 (Compiled -g) 31.25 (Compiled -O2)

Understanding Loop

```
void combine1-goto(vec_ptr v, int *dest)
{
    int i = 0;
    int val;
    *dest = 0;
    if (i >= vec_length(v))
      goto done;
  loop:
    get_vec_element(v, i, &val);
    *dest += val;
    i++;
    if (i < vec_length(v))
      goto loop
  done:
}
```

1 iteration

- Inefficiency
  - Procedure vec_length called every iteration
  - Even though result always the same

Move `vec_length` Call Out of Loop

```
void combine2(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  *dest = 0;
  for (i = 0; i < length; i++) {
    int val;
    get_vec_element(v, i, &val);
    *dest += val;
  }
}
```

- Optimization
  - Move call to `vec_length` out of inner loop
    - Value does not change from one iteration to next
    - Code motion
  - CPE: 20.66 (Compiled -O2)
    - `vec_length` requires only constant time, but significant overhead

Optimization Blocker: Procedure Calls

- *Why couldn't the compiler move* `vec_len` *out of the inner loop?*
  - Procedure may have side effects
    - Alters global state each time called
  - Function may not return same value for given arguments
    - Depends on other parts of global state
    - Procedure `lower` could interact with `strlen`
- *Why doesn't compiler look at code for* `vec_len`*?*
  - Linker may overload with different version
    - Unless declared static
  - Inter-procedural optimization is not used extensively due to cost
- Warning:
  - Compiler treats procedure call as a black box
  - Weak optimizations in and around them

Reduction in Strength

```
void combine3(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  int *data = get_vec_start(v);
  *dest = 0;
  for (i = 0; i < length; i++) {
    *dest += data[i];
}
```

- Optimization
  - Avoid procedure call to retrieve each vector element
    - Get pointer to start of array before loop
    - Within loop just do pointer reference
    - Not as clean in terms of data abstraction
  - CPE: 6.00 (Compiled -O2)
    - Procedure calls are expensive!
    - Bounds checking is expensive

Eliminate Unneeded Memory Refs

```
void combine4(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  int *data = get_vec_start(v);
  int sum = 0;
  for (i = 0; i < length; i++)
    sum += data[i];
  *dest = sum;
}
```

- Optimization
  - Don't need to store in destination until end
  - Local variable `sum` held in register
  - Avoids 1 memory read, 1 memory write per cycle
  - CPE: 2.00 (Compiled -O2)
    - Memory references are expensive!

Detecting Unneeded Memory Refs.

Combine3

```
.L18:
        movl (%ecx,%edx,4),%eax
        addl %eax,(%edi)
        incl %edx
        cmpl %esi,%edx
        jl .L18
```

Combine4

```
.L24:
        addl (%eax,%edx,4),%ecx

        incl %edx
        cmpl %esi,%edx
        jl .L24
```

- Performance
  - Combine3
    - 5 instructions in 6 clock cycles
    - `addl` must read and write memory
  - Combine4
    - 4 instructions in 2 clock cycles

Optimization Blocker: Memory Aliasing

- Aliasing
  - Two different memory references specify single location
- Example
  - v: [3, 2, 17]
  - combine3(v, get_vec_start(v)+2)-->?
  - combine4(v, get_vec_start(v)+2)-->?

```
void combine3(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  int *data = get_vec_start(v);
  *dest = 0;
  for (i = 0; i < length; i++) {
    *dest += data[i];
}
```

```
void combine4(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  int *data = get_vec_start(v);
  int sum = 0;
  for (i = 0; i < length; i++)
    sum += data[i];
  *dest = sum;
}
```

Optimization Blocker: Memory Aliasing

- Aliasing
  - Two different memory references specify single location
- Example
  - `v: [3, 2, 17]`
  - `combine3(v, get_vec_start(v)+2)-->?`
  - `combine4(v, get_vec_start(v)+2)-->?`
- Observations
  - Easy to happen in C
    - Since address arithmetic is allowed
    - Direct access to storage structures
  - Get in habit of introducing local variables
    - Accumulating within loops
    - Your way of telling compiler not to check for aliasing

Machine-Independent Opt. Summary

- Code Motion
  - *Compilers are good at this for simple loop/array structures*
  - *Don't do well in presence of procedure calls and memory aliasing*
- Reduction in Strength
  - Shift, add instead of multiply or divide
    - *compilers are (generally) good at this*
    - *Exact trade-offs machine-dependent*
  - Keep data in registers rather than memory
    - *compilers are not good at this, since concerned with aliasing*
- Share Common Subexpressions
  - *compilers have limited algebraic reasoning capabilities*

Important Tools

- Measurement
  - Accurately compute time taken by code
    - Most modern machines have built in cycle counters
    - Using them to get reliable measurements is tricky
  - Profile procedure calling frequencies
    - Unix tool gprof
- Observation
  - Generating assembly code
    - Lets you see what optimizations compiler can make
    - Understand capabilities/limitations of particular compiler

Code Profiling Example

- Task
  - Count word frequencies in document
  - Produce sorted list of words from most frequent to least
- Steps
  - Convert strings to lowercase
  - Apply hash function
  - Read words and insert into hash table
    - Mostly list operations
    - Maintain counter for each unique word
  - Sort results
- Data Set
  - Collected works of Shakespeare
  - 946,596 total words, 26,596 unique
  - Initial implementation: 9.2 seconds

Shakespeare's most frequent words

29,801	the
27,529	and
21,029	I
20,957	to
18,514	of
15,370	a
14010	you
12,936	my
11,722	in
11,519	that

Code Profiling

- Augment Executable Program with Timing Functions
  - Computes (approximate) amount of time spent in each function
  - Time computation method
    - Periodically (~ every 10ms) interrupt program
    - Determine what function is currently executing
    - Increment its timer by interval (e.g., 10ms)
  - Also maintains counter for each function indicating number of times called
- Using

```
gcc –O2 –pg prog. –o prog
./prog
```

    - Executes in normal fashion, but also generates file `gmon.out`

```
gprof prog
```

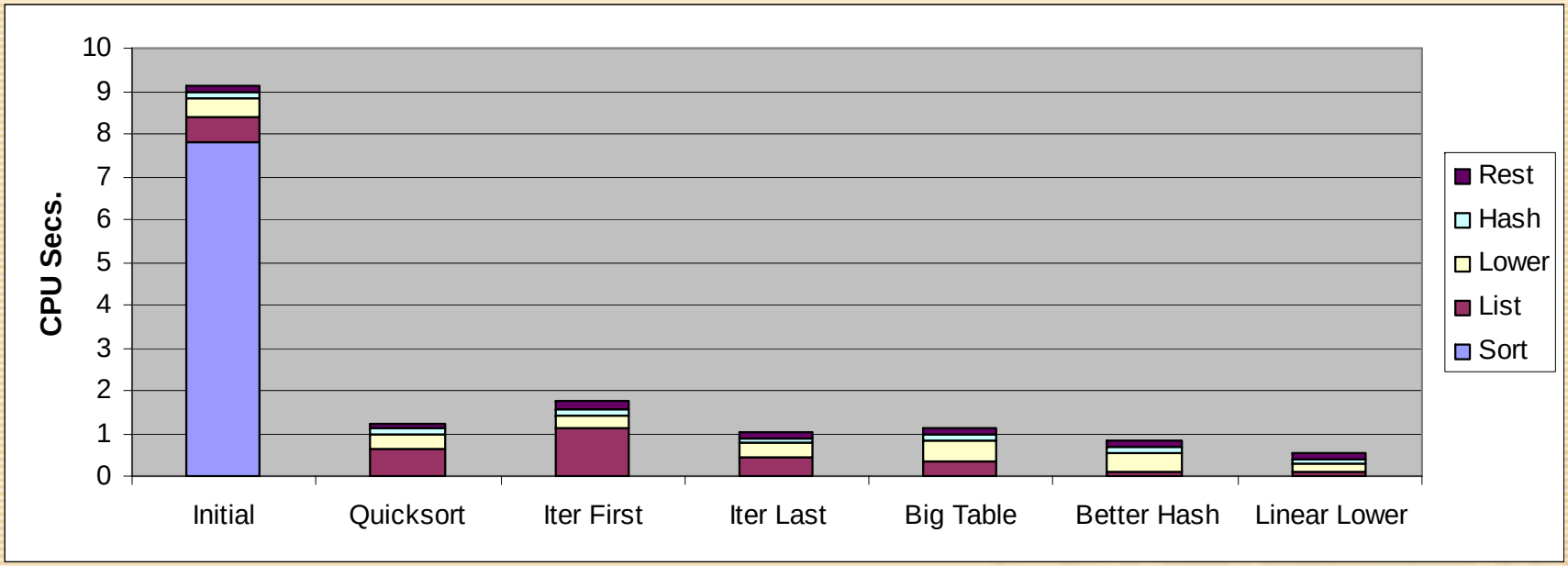
    - Generates profile information based on `gmon.out`

Profiling Results

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
86.60	8.21	8.21	1	8210.00	8210.00	sort_words
5.80	8.76	0.55	946596	0.00	0.00	lower1
4.75	9.21	0.45	946596	0.00	0.00	find_ele_rec
1.27	9.33	0.12	946596	0.00	0.00	h_add

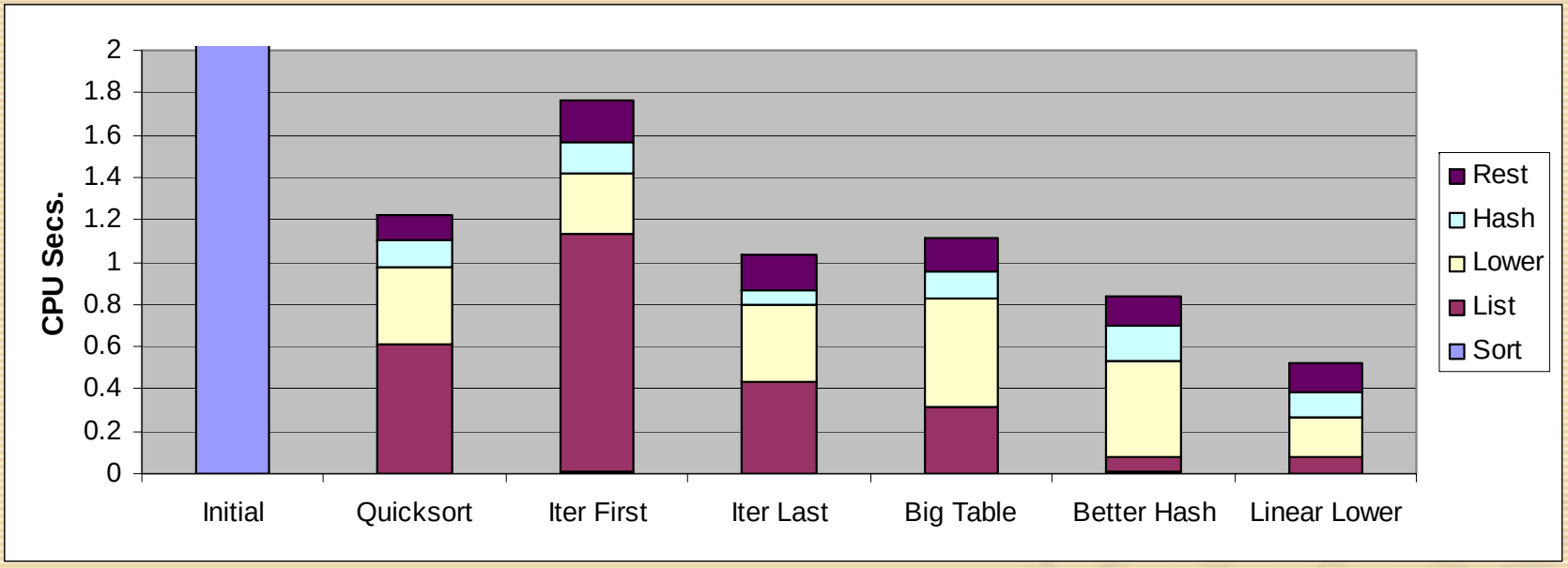
- Call Statistics
  - Number of calls and cumulative time for each function
- Performance Limiter
  - Using inefficient sorting algorithm
  - Single call uses 87% of CPU time

Code Optimizations



- First step: Use more efficient sorting function
- Library function `qsort`

Further Optimizations



- Iter first: Use iterative function to insert elements into linked list
  - Causes code to slow down
- Iter last: Iterative function, places new entry at end of list
  - Tend to place most common words at front of list
- Big table: Increase number of hash buckets
- Better hash: Use more sophisticated hash function
- Linear lower: Move `strlen` out of loop

Profiling Observations

- Benefits
  - Helps identify performance bottlenecks
  - Especially useful when have complex system with many components
- Limitations
  - Only shows performance for data tested
  - E.g., linear lower did not show big gain, since words are short
    - Quadratic inefficiency could remain lurking in code
  - Timing mechanism fairly coarse
    - Only works for programs that run for > 3 seconds

Code Optimization II: Machine Dependent Optimizations

jc@fudan.edu.cn

Previous Best Combining Code

```
void combine4(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  int *data = get_vec_start(v);
  int sum = 0;
  for (i = 0; i < length; i++)
    sum += data[i];
  *dest = sum;
}
```

- Task
  - Compute sum of all elements in vector
  - Vector represented by C-style abstract data type
  - Achieved CPE of 2.00
    - Cycles per element

General Forms of Combining

```
void abstract_combine4(vec_ptr v, data_t *dest)
{
  int i;
  int length = vec_length(v);
  data_t *data = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP data[i];
  *dest = t;
}
```

- Data Types
  - Use different declarations for `data_t`
  - `int`
  - `float`
  - `double`
- Operations
  - Use different definitions of `OP` and `IDENT`
  - `+` / `0`
  - `*` / `1`

Machine Independent Opt. Results

- Optimizations
  - Reduce function calls and memory references within loop

Method	Integer		Floating Point	
	+	*	+	*
Abstract -g	42.06	41.86	41.44	160.00
Abstract -O2	31.25	33.25	31.25	143.00
Move vec_length	20.66	21.25	21.15	135.00
data access	6.00	9.00	8.00	117.00
Accum. in temp	2.00	4.00	3.00	5.00

- Performance Anomaly
  - Computing FP product of all elements exceptionally slow.
  - Very large speedup when accumulate in temporary
  - Caused by quirk of IA32 floating point
    - Memory uses 64-bit format, register use 80
    - Benchmark data caused overflow of 64 bits, but not 80

Pointer Code

```
void combine4p(vec_ptr v, int *dest)
{
  int length = vec_length(v);
  int *data = get_vec_start(v);
  int *dend = data+length;
  int sum = 0;
  while (data < dend) {
    sum += *data;
    data++;
  }
  *dest = sum;
}
```

- Optimization
  - Use pointers rather than array references
  - CPE: 3.00 (Compiled -O2)
    - We're not making progress here!

Warning: Some compilers do better job optimizing array code

Pointer vs. Array Code Inner Loops

- Array Code

```
.L24:                           # Loop:
  addl (%eax,%edx,4),%ecx       # sum += data[i]
  incl %edx                     # i++
  cmpl %esi,%edx                # i:length
  jl .L24                       # if < goto Loop
```

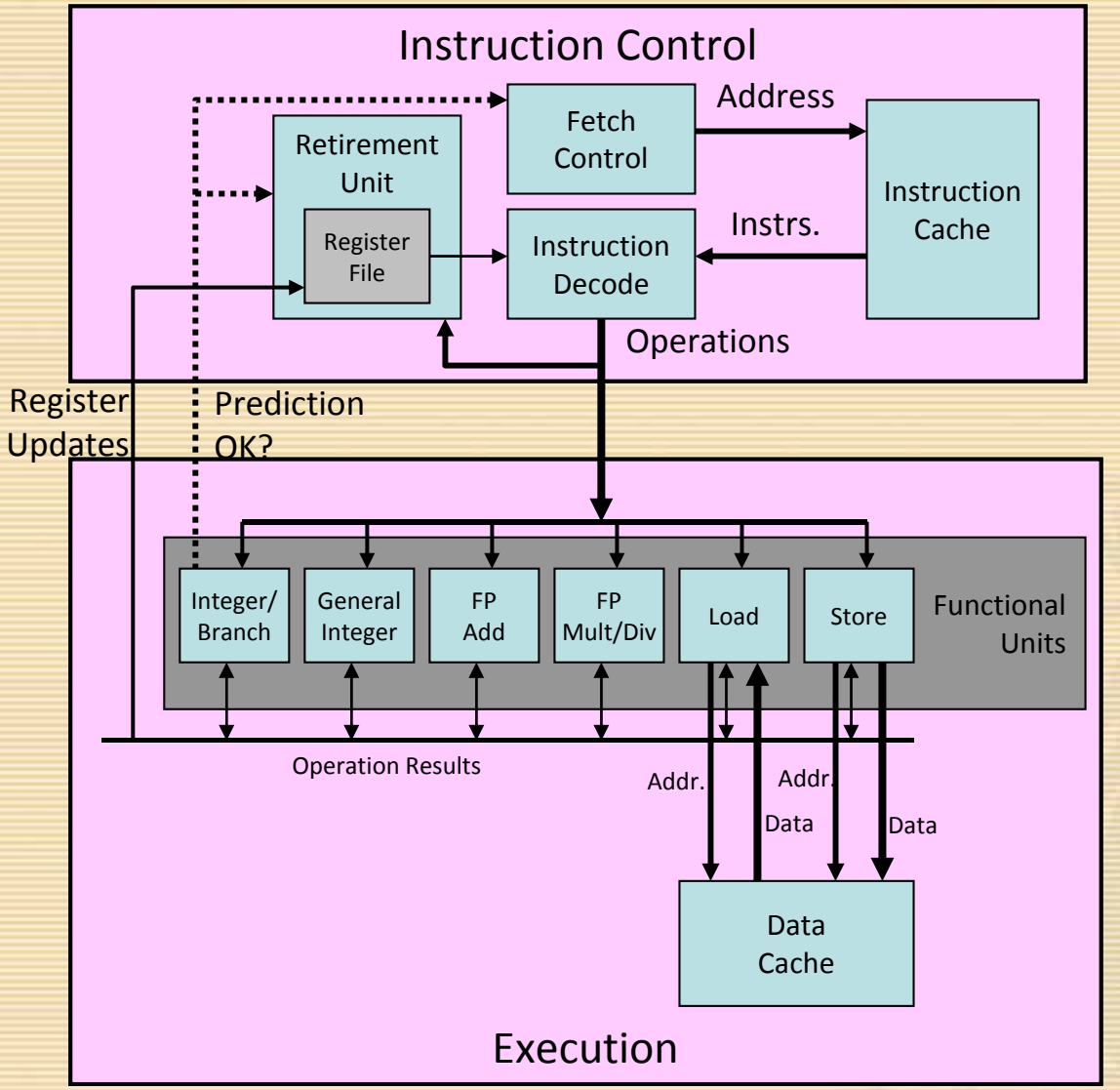
- Pointer Code

```
.L30:                   # Loop:
  addl (%eax),%ecx      #  sum += *data
  addl $4,%eax          #  data ++
  cmpl %edx,%eax        #  data:dend
  jb .L30               #  if < goto Loop
```

- Performance
  - Array Code: 4 instructions in 2 clock cycles
  - Pointer Code: Almost same 4 instructions in 3 clock cycles

Modern CPU Design

Page 396

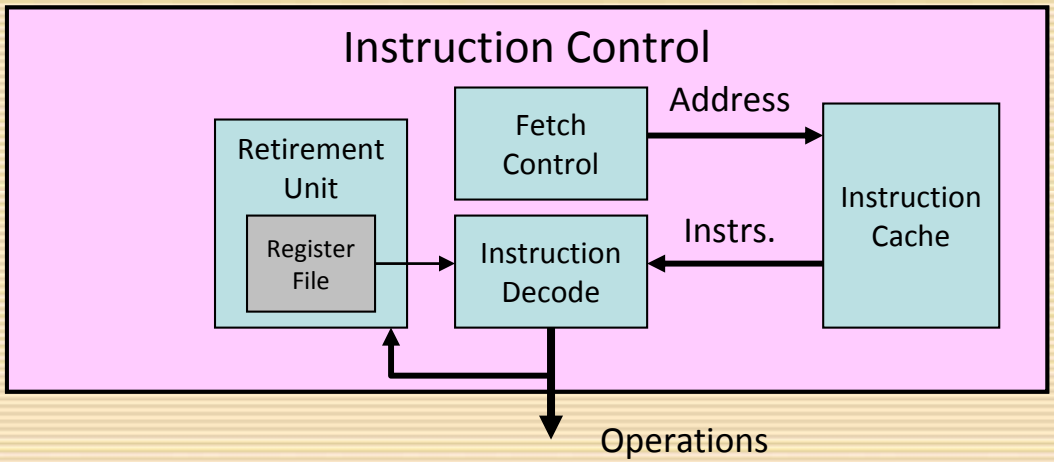


CPU Capabilities of Pentium III

- Multiple Instructions Can Execute in Parallel
  - 1 load
  - 1 store
  - 2 integer (one may be branch)
  - 1 FP Addition
  - 1 FP Multiplication or Division
- Some Instructions Take > 1 Cycle, but Can be Pipelined

Instruction	Latency	Cycles/Issue
Load / Store	3	1
Integer Multiply	4	1
Integer Divide	36	36
Double/Single FP Multiply	5	2
Double/Single FP Add	3	1
Double/Single FP Divide	38	38

Instruction Control



- Grabs Instruction Bytes From Memory
  - Based on current PC + predicted targets for predicted branches
  - Hardware dynamically guesses whether branches taken/not taken and (possibly) branch target
- Translates Instructions Into *Operations*
  - Primitive steps required to perform instruction
  - Typical instruction requires 1–3 operations
- Converts Register References Into *Tags*
  - Abstract identifier linking destination of one operation with sources of later operations

Translation Example

- Version of Combine4
  - Integer data, multiply operation

```
.L24:                              # Loop:
  imull (%eax,%edx,4),%ecx         # t *= data[i]
  incl %edx                        # i++
  cmpl %esi,%edx                   # i:length
  jl .L24                          # if < goto Loop
```

- Translation of First Iteration

```
.L24:
  imull (%eax,%edx,4),%ecx

  incl %edx
  cmpl %esi,%edx
  jl .L24
```

```
load (%eax,%edx.0,4) ➔ t.1
imull t.1, %ecx.0    ➔ %ecx.1
incl %edx.0          ➔ %edx.1
cmpl %esi, %edx.1    ➔ cc.1
jl-taken cc.1
```

Translation Example #1

```
imull (%eax,%edx,4),%ecx
```

```
load (%eax,%edx.0,4) ➔ t.1
imull t.1, %ecx.0    ➔ %ecx.1
```

- Split into two operations
  - `load` reads from memory to generate temporary result `t.1`
  - Multiply operation just operates on registers
- Operands
  - Registers `%eax` does not change in loop. Values will be retrieved from register file during decoding
  - Register `%ecx` changes on every iteration. Uniquely identify different versions as `%ecx.0`, `%ecx.1`, `%ecx.2`, …
    - Register *renaming*
    - Values passed directly from producer to consumers

Translation Example #2

```
incl %edx
```

```
incl %edx.0          ➔ %edx.1
```

– Register `%edx` changes on each iteration.
  Rename as `%edx.0`, `%edx.1`, `%edx.2`, …

Translation Example #3

```
cmpl %esi,%edx
```

```
cmpl %esi, %edx.1  ➔ cc.1
```

- Condition codes are treated similar to registers
- Assign tag to define connection between producer and consumer

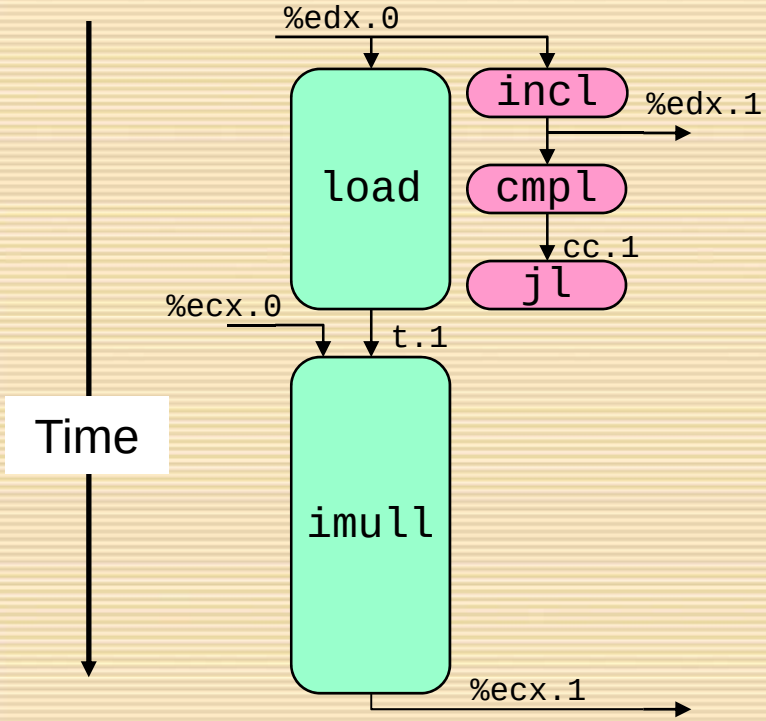
Translation Example #4

```
jl .L24
```

```
jl-taken cc.1
```

- Instruction control unit determines destination of jump
- Predicts whether will be taken and target
- Starts fetching instruction at predicted destination
- Execution unit simply checks whether or not prediction was OK
- If not, it signals instruction control
  - Instruction control then “invalidates” any operations generated from misfetched instructions
  - Begins fetching and decoding instructions at correct target

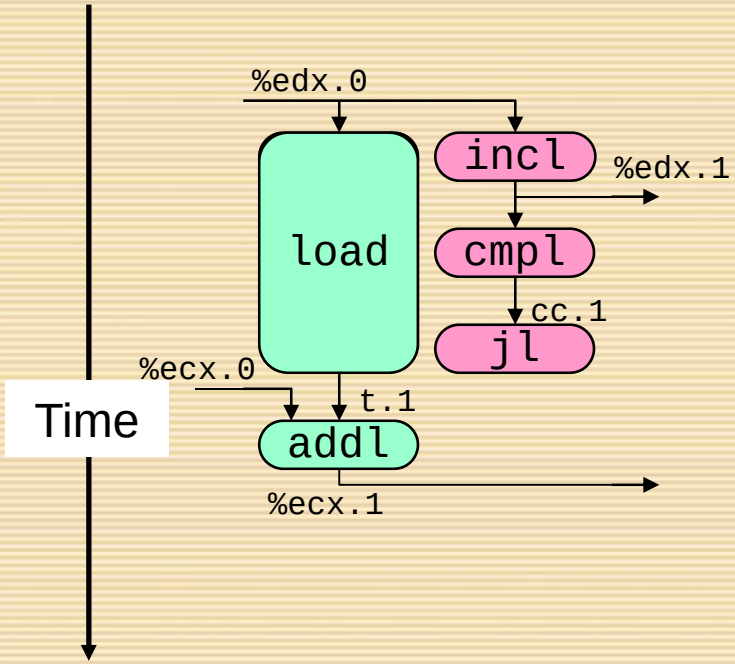
Visualizing Operations




```
load (%eax,%edx,4) ➔ t.1
imull t.1, %ecx.0  ➔ %ecx.1
incl %edx.0        ➔ %edx.1
cmpl %esi, %edx.1  ➔ cc.1
jl-taken cc.1
```

- Operations
  - Vertical position denotes time at which executed
    - Cannot begin operation until operands available
  - Height denotes latency
- Operands
  - Arcs shown only for operands that are passed within execution unit

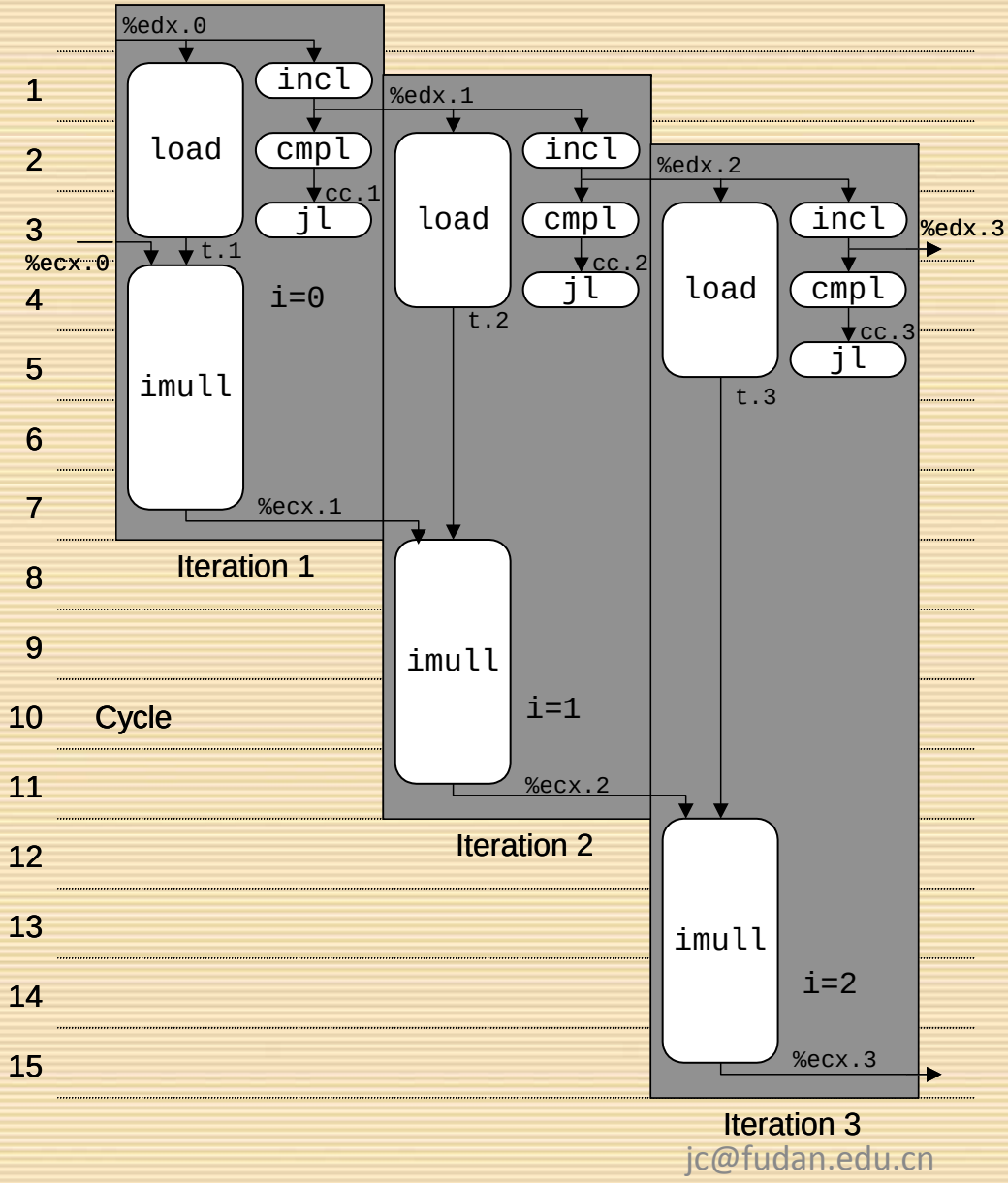
Visualizing Operations (cont.)




```
load (%eax,%edx,4) ➔ t.1
iaddl t.1, %ecx.0  ➔ %ecx.1
incl %edx.0        ➔ %edx.1
cmpl %esi, %edx.1  ➔ cc.1
jl-taken cc.1
```

- Operations
  - Same as before, except that add has latency of 1

3 Iterations of Combining Product



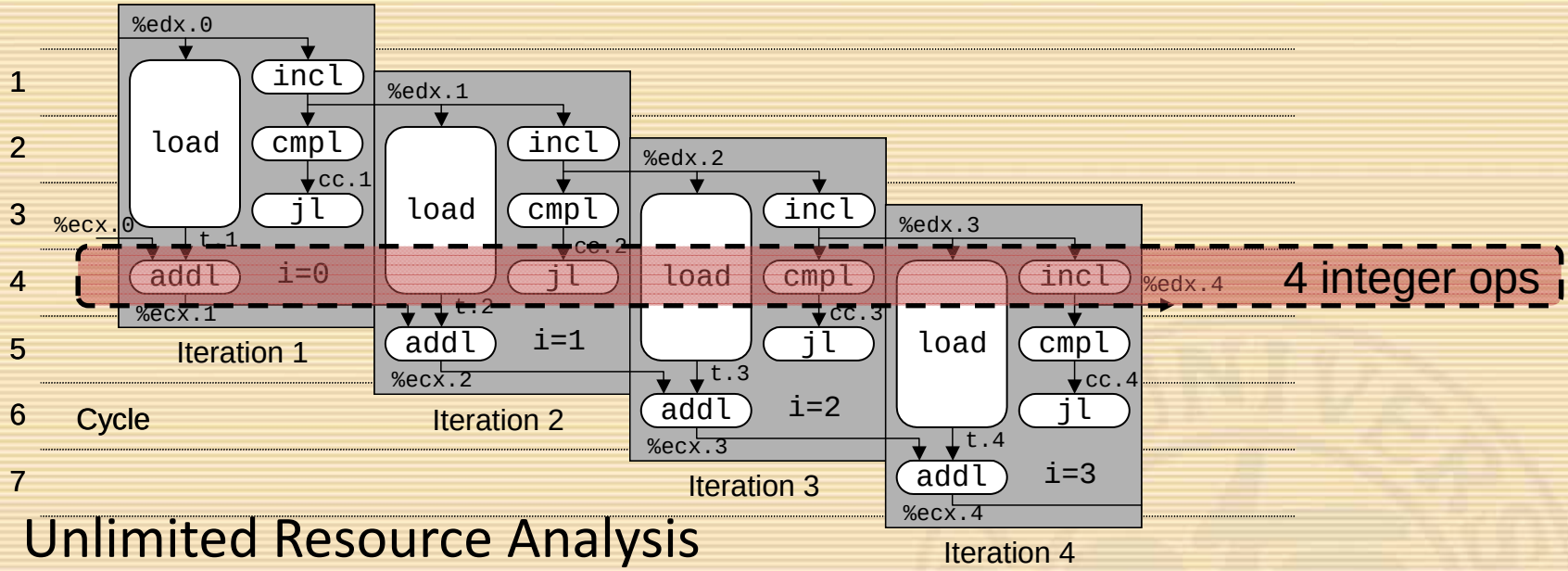
Page 404

- Unlimited Resource Analysis
  - Assume operation can start as soon as operands available
  - Operations for multiple iterations overlap in time
- Performance
  - Limiting factor becomes latency of integer multiplier
  - Gives CPE of 4.0

jc@fudan.edu.cn

4 Iterations of Combining Sum

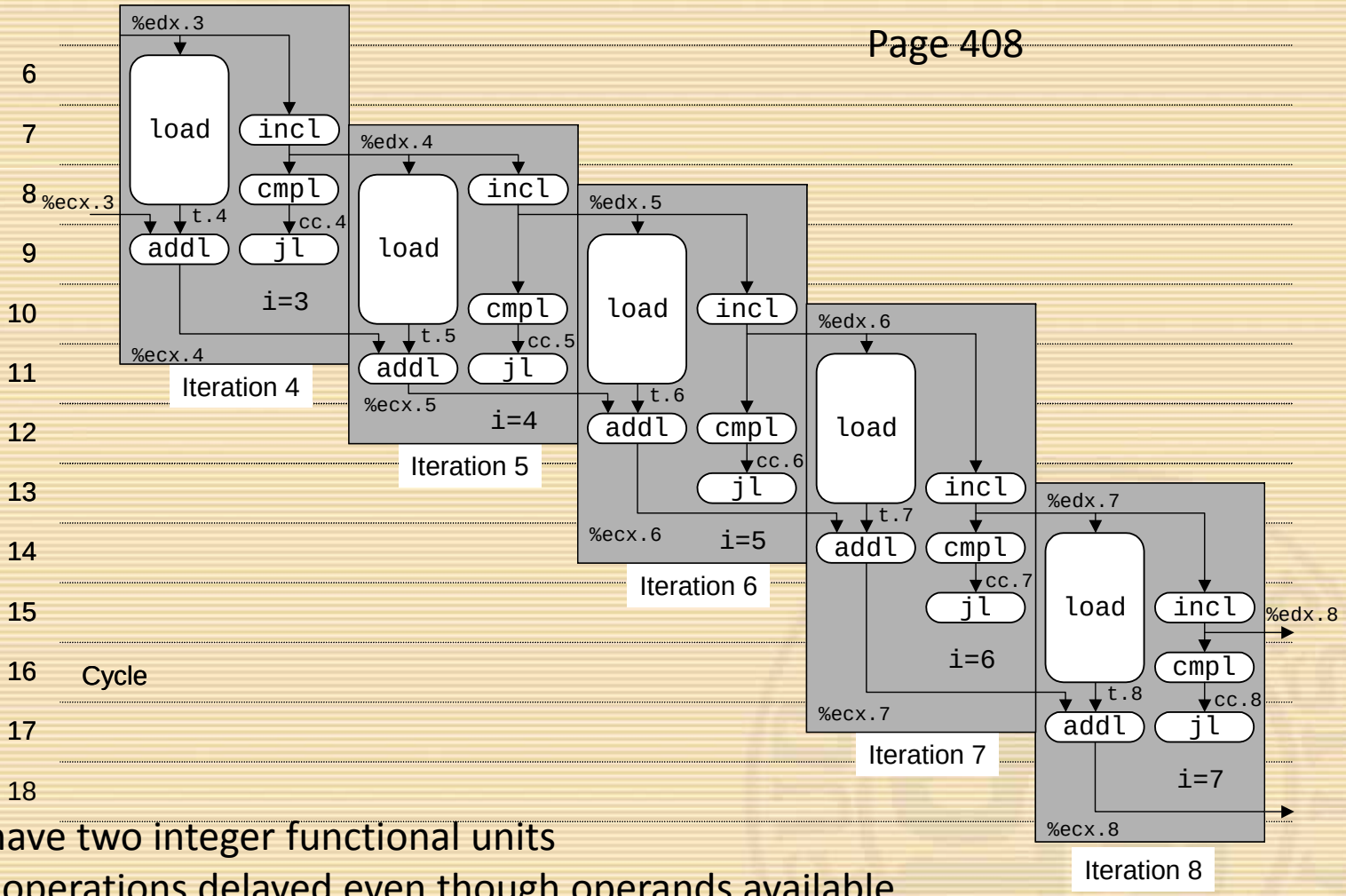
Page 405



- Unlimited Resource Analysis
- Performance
  - Can begin a new iteration on each clock cycle
  - Should give CPE of 1.0
  - Would require executing 4 integer operations in parallel

Combining Sum: Resource Constraints

Page 408



  - Only have two integer functional units
  - Some operations delayed even though operands available
  - Set priority based on program order
- **Performance**
  - Sustain CPE of 2.0

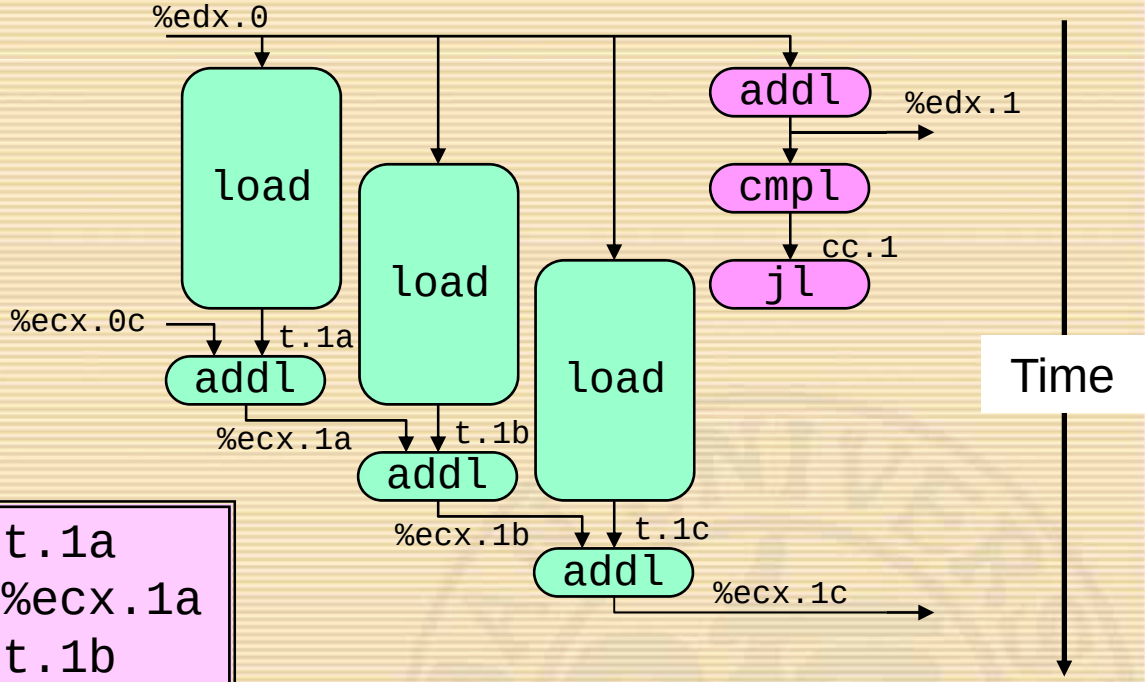
Loop Unrolling

```
void combine5(vec_ptr v, int *dest)
{
  int length = vec_length(v);
  int limit = length-2;
  int *data = get_vec_start(v);
  int sum = 0;
  int i;
  /* Combine 3 elements at a time */
  for (i = 0; i < limit; i+=3) {
    sum += data[i] + data[i+2]
           + data[i+1];
  }
  /* Finish any remaining elements */
  for (; i < length; i++) {
    sum += data[i];
  }
  *dest = sum;
}
```

- Optimization
  - Combine multiple iterations into single loop body
  - Amortizes loop overhead across multiple iterations
  - Finish extras at end
  - Measured CPE = 1.33

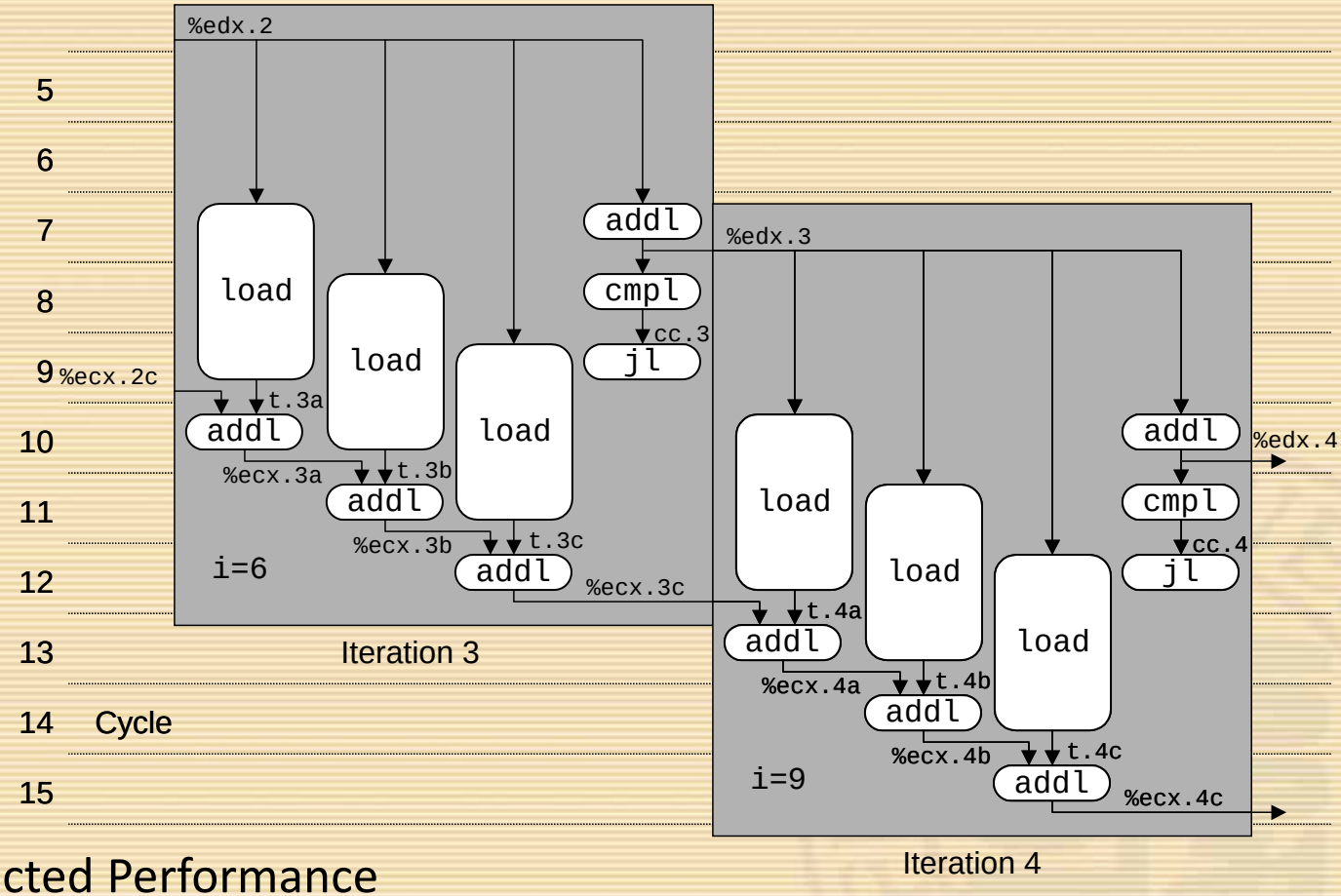
Visualizing Unrolled Loop

- Loads can pipeline, since don't have dependencies
- Only one set of loop control operations




```
load (%eax,%edx.0,4)   ➔ t.1a
iaddl t.1a, %ecx.0c    ➔ %ecx.1a
load 4(%eax,%edx.0,4)  ➔ t.1b
iaddl t.1b, %ecx.1a    ➔ %ecx.1b
load 8(%eax,%edx.0,4)  ➔ t.1c
iaddl t.1c, %ecx.1b    ➔ %ecx.1c
iaddl $3,%edx.0        ➔ %edx.1
cmpl %esi, %edx.1      ➔ cc.1
jl-taken cc.1
```

Executing with Loop Unrolling



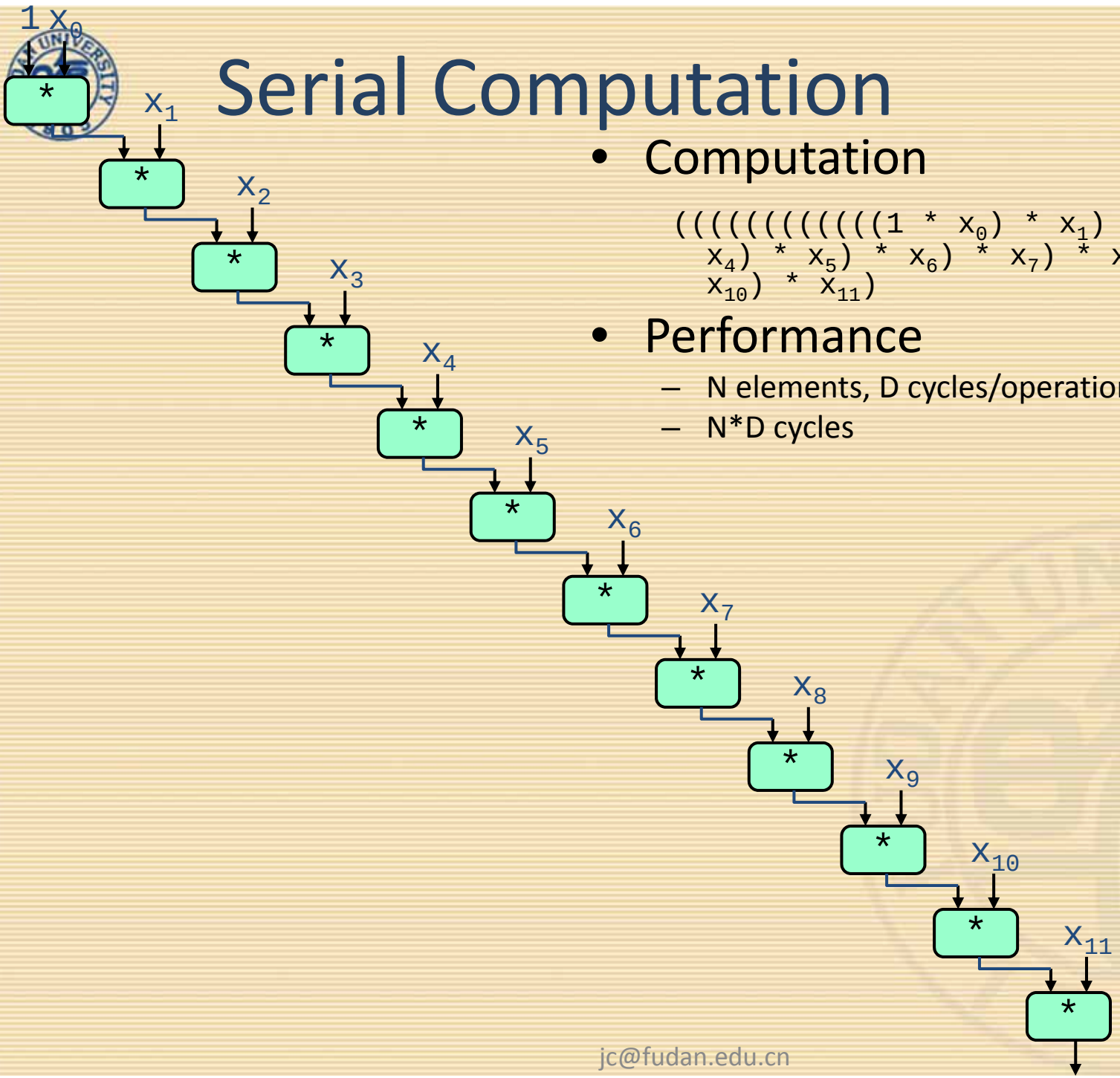
- Predicted Performance
  - Can complete iteration in 3 cycles
  - Should give CPE of 1.0
- Measured Performance
  - CPE of 1.33, One iteration every 4 cycles

Effect of Unrolling

Unrolling Degree		1	2	3	4	8	16
Integer	Sum	2.00	1.50	1.33	1.50	1.25	1.06
Integer	Product	4.00					
FP	Sum	3.00					
FP	Product	5.00					

- Only helps integer sum for our examples
  - Other cases constrained by functional unit latencies
- Effect is nonlinear with degree of unrolling
  - Many subtle effects determine exact scheduling of operations

Serial Computation



- Computation

$$((((((((((((1 * x_0) * x_1) * x_2) * x_3) * x_4) * x_5) * x_6) * x_7) * x_8) * x_9) * x_{10}) * x_{11})$$

- Performance
  - N elements, D cycles/operation
  - N*D cycles

Parallel Loop Unrolling

```
void combine6(vec_ptr v, int *dest)
{
  int length = vec_length(v);
  int limit = length-1;
  int *data = get_vec_start(v);
  int x0 = 1;
  int x1 = 1;
  int i;
  /* Combine 2 elements at a time */
  for (i = 0; i < limit; i+=2) {
    x0 *= data[i];
    x1 *= data[i+1];
  }
  /* Finish any remaining elements */
  for (; i < length; i++) {
    x0 *= data[i];
  }
  *dest = x0 * x1;
}
```

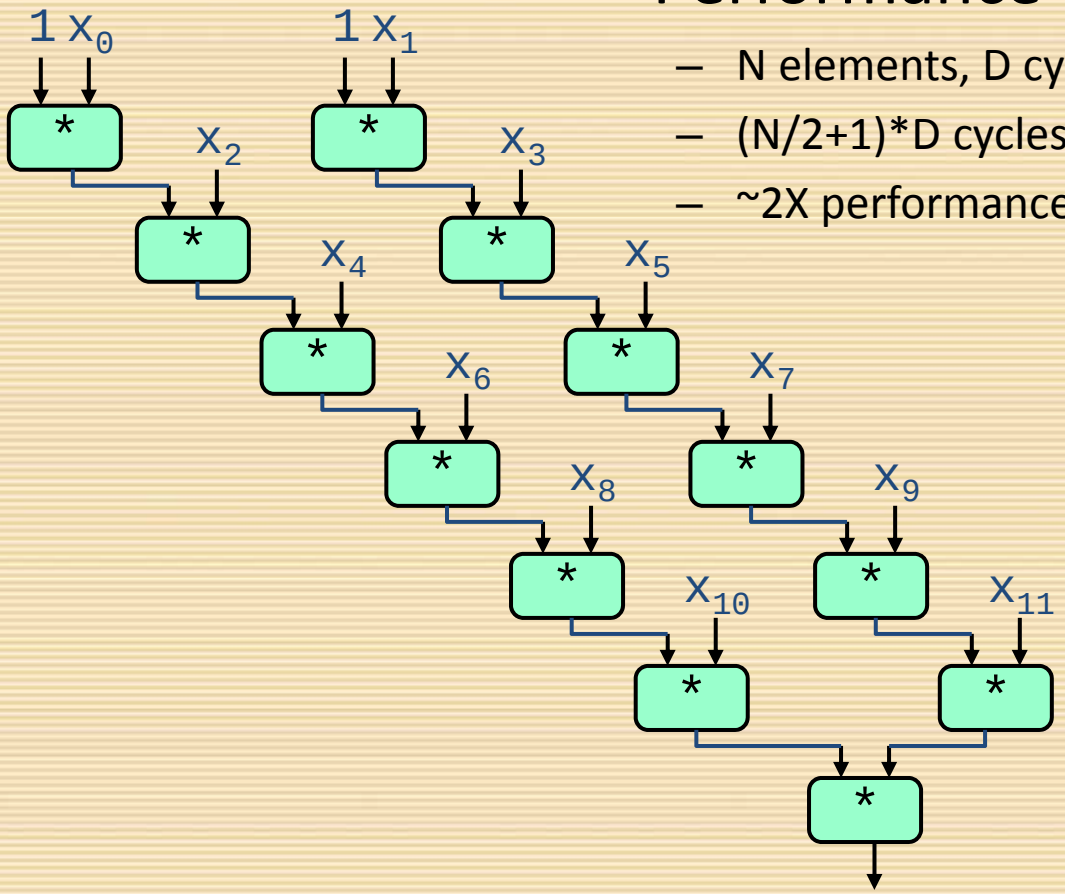
- Code Version
  - Integer product
- Optimization
  - Accumulate in two different products
    - Can be performed simultaneously
  - Combine at end
- Performance
  - CPE = 2.0
  - 2X performance

Dual Product Computation

- Computation

$$((((((1 * x_0) * x_2) * x_4) * x_6) * x_8) * x_{10}) *$$
$$((((((1 * x_1) * x_3) * x_5) * x_7) * x_9) * x_{11})$$

- Performance
  - N elements, D cycles/operation
  - (N/2+1)*D cycles
  - ~2X performance improvement

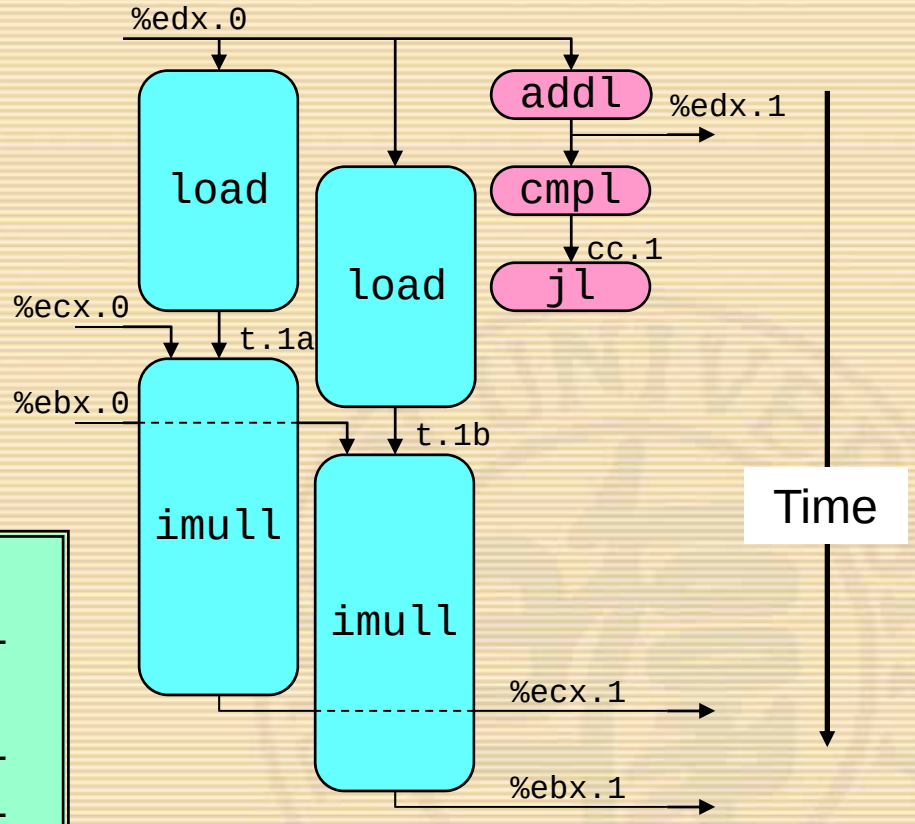


Requirements for Parallel Computation

- Mathematical
  - Combining operation must be associative & commutative
    - OK for integer multiplication
    - Not strictly true for floating point
      - OK for most applications
- Hardware
  - Pipelined functional units
  - Ability to dynamically extract parallelism from code

Visualizing Parallel Loop

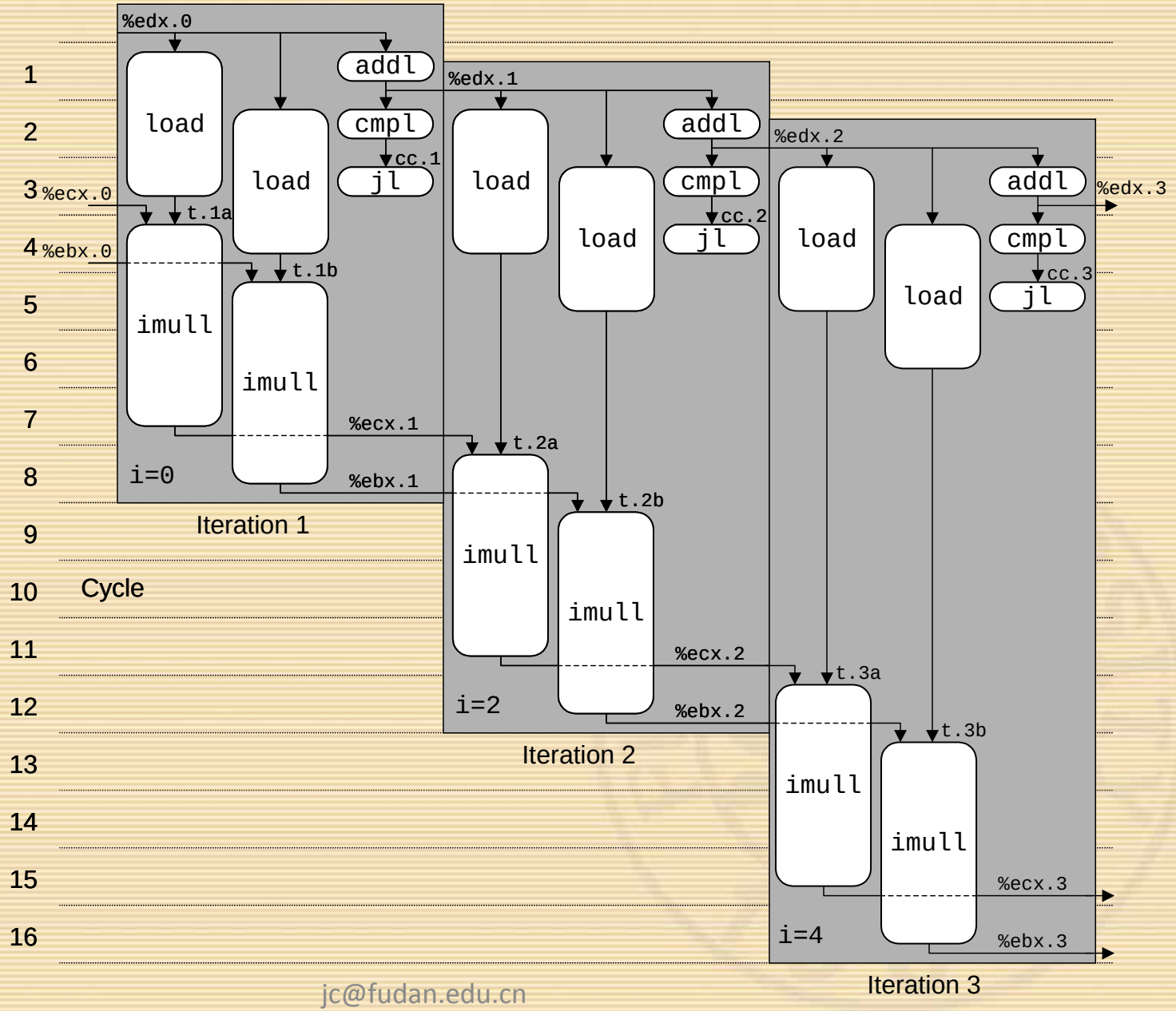
- Two multiplies within loop no longer have data dependency
- Allows them to pipeline




```
load (%eax,%edx.0,4)   ➔ t.1a
imull t.1a, %ecx.0     ➔ %ecx.1
load 4(%eax,%edx.0,4)  ➔ t.1b
imull t.1b, %ebx.0     ➔ %ebx.1
iaddl $2,%edx.0        ➔ %edx.1
cmpl %esi, %edx.1      ➔ cc.1
jl-taken cc.1
```

Executing with Parallel Loop

- Predicted Performance
  - Can keep 4-cycle multiplier busy performing two simultaneous multiplications
  - Gives CPE of 2.0



Optimization Results for Combining

Method	Integer		Floating Point	
	+	*	+	*
Abstract -g	42.06	41.86	41.44	160.00
Abstract -O2	31.25	33.25	31.25	143.00
Move vec_length	20.66	21.25	21.15	135.00
data access	6.00	9.00	8.00	117.00
Accum. in temp	2.00	4.00	3.00	5.00
Pointer	3.00	4.00	3.00	5.00
Unroll 4	1.50	4.00	3.00	5.00
Unroll 16	*1.06*	4.00	3.00	5.00
2 X 2	1.50	2.00	2.00	2.50
4 X 4	1.50	*2.00*	*1.50*	2.50
8 X 4	1.25	*1.25*	1.50	*2.00*
Theoretical Opt.	1.00	1.00	1.00	2.00
Worst : Best	*39.7*	*33.5*	*27.6*	*80.0*

Parallel Unrolling: Method #2

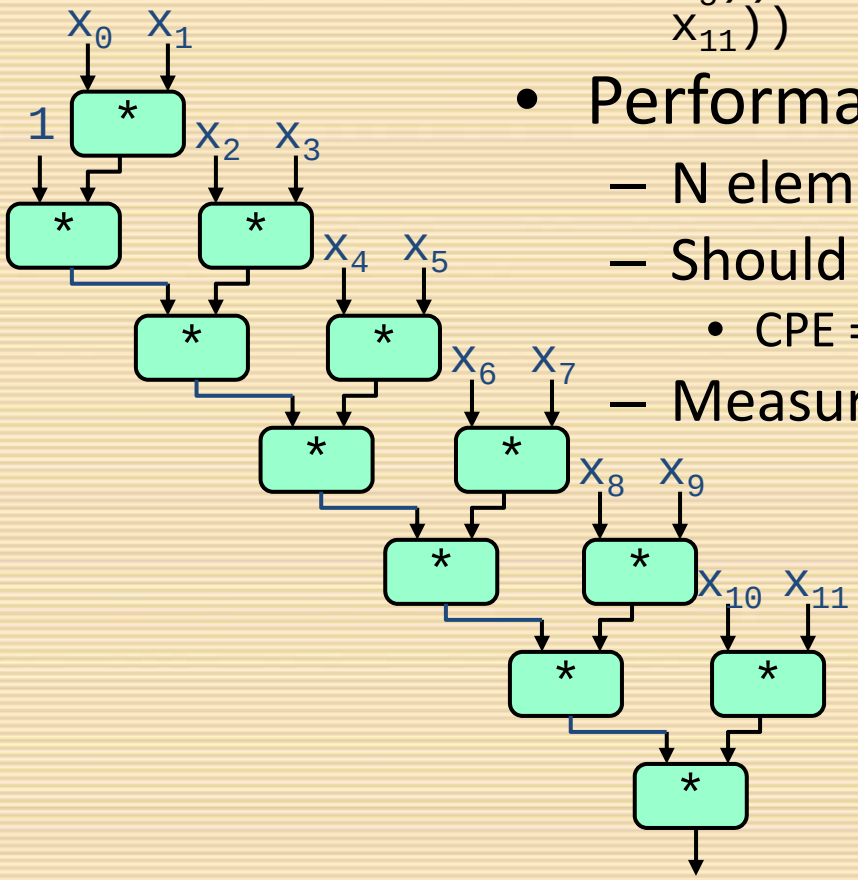
```
void combine6aa(vec_ptr v, int *dest)
{
  int length = vec_length(v);
  int limit = length-1;
  int *data = get_vec_start(v);
  int x = 1;
  int i;
  /* Combine 2 elements at a time */
  for (i = 0; i < limit; i+=2) {
    x *= (data[i] * data[i+1]);
  }
  /* Finish any remaining elements */
  for (; i < length; i++) {
    x *= data[i];
  }
  *dest = x;
}
```

- Code Version
  - Integer product
- Optimization
  - Multiply pairs of elements together
  - And then update product
  - “Tree height reduction”
- Performance
  - CPE = 2.5

Method #2 Computation

- Computation

```
((((((1 * (x0 * x1)) * (x2 * x3)) * (x4 *
  x5)) * (x6 * x7)) * (x8 * x9)) * (x10 *
  x11))
```



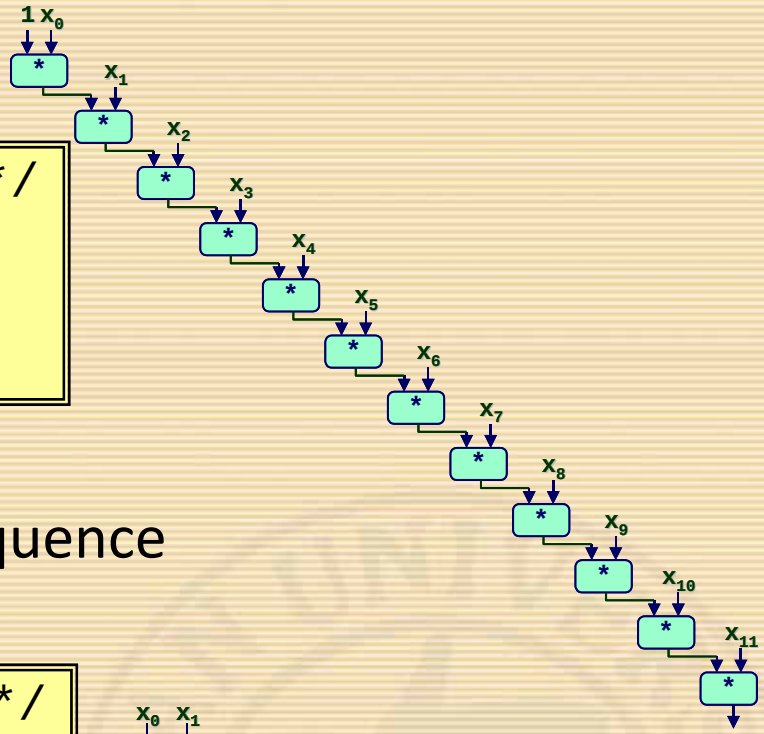
- Performance
  - N elements, D cycles/operation
  - Should be (N/2+1)*D cycles
    - CPE = 2.0
  - Measured CPE worse

Unrolling	CPE (measured)	CPE (theoretical)
2	2.50	2.00
3	1.67	1.33
4	1.50	1.00
6	1.78	1.00

Understanding Parallelism

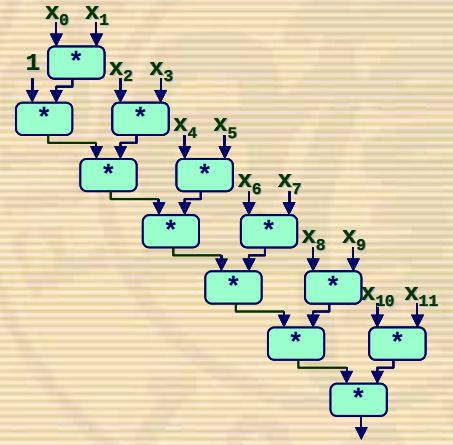
```
/* Combine 2 elements at a time */
for (i = 0; i < limit; i+=2) {
  x = (x * data[i]) * data[i+1];
}
```

- CPE = 4.00
- All multiplies performed in sequence




```
/* Combine 2 elements at a time */
for (i = 0; i < limit; i+=2) {
  x = x * (data[i] * data[i+1]);
}
```

- CPE = 2.50
- Multiplies overlap



Limitations of Parallel Execution

- Need Lots of Registers (Register Spilling)
  - To hold sums/products
  - Only 6 usable integer registers
    - Also needed for pointers, loop conditions
  - 8 FP registers
  - When not enough registers, must spill temporaries onto stack
    - Wipes out any performance gains
  - Not helped by renaming
    - Cannot reference more operands than instruction set allows
    - Major drawback of IA32 instruction set

Register Spilling Example

- Example
  - 8 X 8 integer product
  - 7 local variables share 1 register
  - See that are storing locals on stack
  - E.g., at -8(%ebp)

```
.L165:
      imull (%eax),%ecx
      movl -4(%ebp),%edi
      imull 4(%eax),%edi
      movl %edi,-4(%ebp)
      movl -8(%ebp),%edi
      imull 8(%eax),%edi
      movl %edi,-8(%ebp)
      movl -12(%ebp),%edi
      imull 12(%eax),%edi
      movl %edi,-12(%ebp)
      movl -16(%ebp),%edi
      imull 16(%eax),%edi
      movl %edi,-16(%ebp)
    …
      addl $32,%eax
      addl $8,%edx
      cmpl -32(%ebp),%edx
      jl .L165
```

Summary: Results for Pentium III

Method	Integer		Floating Point	
	+	*	+	*
Abstract -g	42.06	41.86	41.44	160.00
Abstract -O2	31.25	33.25	31.25	143.00
Move vec_length	20.66	21.25	21.15	135.00
data access	6.00	9.00	8.00	117.00
Accum. in temp	2.00	4.00	3.00	5.00
Unroll 4	1.50	4.00	3.00	5.00
Unroll 16	1.06	4.00	3.00	5.00
4 X 2	1.50	2.00	*1.50*	2.50
8 X 4	*1.25*	*1.25*	1.50	*2.00*
8 X 8	1.88	1.88	1.75	2.00
Worst : Best	*39.7*	*33.5*	*27.6*	*80.0*

– Biggest gain doing basic optimizations

Results for Alpha Processor

Method	Integer		Floating Point	
	+	*	+	*
Abstract -g	40.14	47.14	52.07	53.71
Abstract -O2	25.08	36.05	37.37	32.02
Move vec_length	19.19	32.18	28.73	32.73
data access	6.26	12.52	13.26	13.01
Accum. in temp	1.76	9.01	8.08	8.01
Unroll 4	1.51	9.01	6.32	6.32
Unroll 16	1.25	9.01	6.33	6.22
4 X 2	1.19	4.69	4.44	4.45
8 X 4	1.15	*4.12*	*2.34*	*2.01*
8 X 8	*1.11*	4.24	2.36	2.08
Worst : Best	*36.2*	*11.4*	*22.3*	*26.7*

- Overall trends very similar to those for Pentium III
- Even though very different architecture and compiler

Results for Pentium 4

Method	Integer		Floating Point	
	+	*	+	*
Abstract -g	35.25	35.34	35.85	38.00
Abstract -O2	26.52	30.26	31.55	32.00
Move vec_length	18.00	25.71	23.36	24.25
data access	3.39	31.56	27.50	28.35
Accum. in temp	2.00	14.00	5.00	7.00
Unroll 4	1.01	14.00	5.00	7.00
Unroll 16	1.00	14.00	5.00	7.00
4 X 2	1.02	7.00	2.63	3.50
8 X 4	1.01	*3.98*	*1.82*	*2.00*
8 X 8	*1.63*	4.50	2.42	2.31
Worst : Best	*35.2*	*8.9*	*19.7*	*19.0*

- Higher latencies (int * = 14, fp + = 5.0, fp * = 7.0)
  - Clock runs at 2.0 GHz
  - Not an improvement over 1.0 GHz P3 for integer *
- Avoids FP multiplication anomaly

What About Branches?

- Challenge
  - Instruction Control Unit must work well ahead of Exec. Unit
    - To generate enough operations to keep EU busy

```
80489f3:  movl   $0x1,%ecx
80489f8:  xorl   %edx,%edx
80489fa:  cmpl   %esi,%edx
80489fc:  jnl    8048a25
80489fe:  movl   %esi,%esi
8048a00:  imull  (%eax,%edx,4),%ecx
```

Executing

Fetching & Decoding

  - When encountered with conditional branch, cannot reliably determine where to continue fetching

Avoiding Branches

- On Modern Processor, Branches are Very Expensive
  - Unless prediction can be reliable
  - When possible, best to avoid altogether
- Example
  - Compute maximum of two values
    - 14 cycles when prediction correct, 29 cycles when incorrect
    - Overall: 20 cycles(?)

```
int max(int x, int y)
{
  return (x < y) ? y : x;
}
```

```
  movl 12(%ebp),%edx  # Get y
  movl 8(%ebp),%eax   # rval=x
  cmpl %edx,%eax      # rval:y
  jge L11             # skip when >=
  movl %edx,%eax      # rval=y
L11:
```

Avoiding Branches with Bit Tricks

- In style of Lab #1
- Use mask rather than conditionals

```
int bmax(int x, int y)
{
  int mask = -(x>y);
  return (mask & x) | (~mask & y);
}
```

- Compiler still uses conditional
  - 16 cycles when predict correctly, 32 otherwise

```
  xorl %edx,%edx      # mask = 0
  movl 8(%ebp),%eax
  movl 12(%ebp),%ecx
  cmpl %ecx,%eax
  jle L13             # skip if x<=y
  movl $-1,%edx       # mask = -1
L13:
```

Avoiding Branches with Bit Tricks

- Force compiler to generate desired code

```
int bvmax(int x, int y)
{
    volatile int t = (x>y);
    int mask = -t;
    return (mask & x) |
           (~mask & y);
}
```

```
  movl 8(%ebp),%ecx   # Get x
  movl 12(%ebp),%edx  # Get y
  cmpl %edx,%ecx      # x:y
  setg %al            # (x>y)
  movzbl %al,%eax     # Zero extend
  movl %eax,-4(%ebp)  # Save as t
  movl -4(%ebp),%eax  # Retrieve t
```

- `volatile` declaration forces value to be written to memory
  - Compiler must therefore generate code to compute `t`
  - Simplest way is `setg/movzbl` combination
- Not very elegant!
  - A hack to get control over compiler
- 22 clock cycles on all data
  - Better than misprediction

Conditional Move

- Added with P6 microarchitecture (PentiumPro onward)
- `cmov`*XX*`l %edx, %eax`
  - If condition *XX* holds, copy `%edx` to `%eax`
  - Doesn't involve any branching
  - Handled as operation within Execution Unit

```
movl 8(%ebp),%edx   # Get x
movl 12(%ebp),%eax  # rval=y
cmpl %edx, %eax     # rval:x
cmovll %edx,%eax    # If <, rval=x
```

- Current version of GCC won't use this instruction
  - Thinks it's compiling for a 386
- Performance
  - 14 cycles on all data

Machine-Dependent Opt. Summary

- Pointer Code
  - Look carefully at generated code to see whether helpful
- Loop Unrolling
  - Some compilers do this automatically
  - Generally not as clever as what can achieve by hand
- Exposing Instruction-Level Parallelism
  - Very machine dependent
- Warning:
  - Benefits depend heavily on particular machine
  - Best if performed by compiler
    - But GCC on IA32/Linux is not very good
  - Do only for performance-critical parts of code

Amdahl's Law P443

$$T_{new} = (1-\alpha)T_{old} + (\alpha T_{old})/k$$
$$= T_{old}[(1-\alpha) + \alpha/k]$$

$$S = T_{old} / T_{new} = 1/[(1-\alpha) + \alpha/k]$$

$$S_{\infty} = 1/(1-\alpha)$$

Role of Programmer

How should I write my programs, given that I have a good, optimizing compiler?

- Don't: Smash Code into Oblivion
  - Hard to read, maintain, & assure correctness
- Do:
  - Select best algorithm
  - Write code that's readable & maintainable
    - Procedures, recursion, without built-in constant limits
    - Even though these factors can slow down code
  - Eliminate optimization blockers
    - Allows compiler to do its job
- Focus on Inner Loops
  - Do detailed optimizations where code will be executed repeatedly
  - Will get most performance gain here

Chap. 5 Summary

- Machine Independent Optimization
  - Code Motion
    - Simpler Operation, Register / Memory, Sub-expression
  - Loop
  - Repeatedly Called Function
  - Repeatedly Referenced Memory Location (Aliasing)
- Machine Dependant Optimization
  - Instructions -> Operations, Latency, Issue Time
  - Pointer / Array, Loop Unrolling, Parallel
  - Load, Store

The Memory Hierarchy

jc@fudan.edu.cn

Random-Access Memory (RAM)

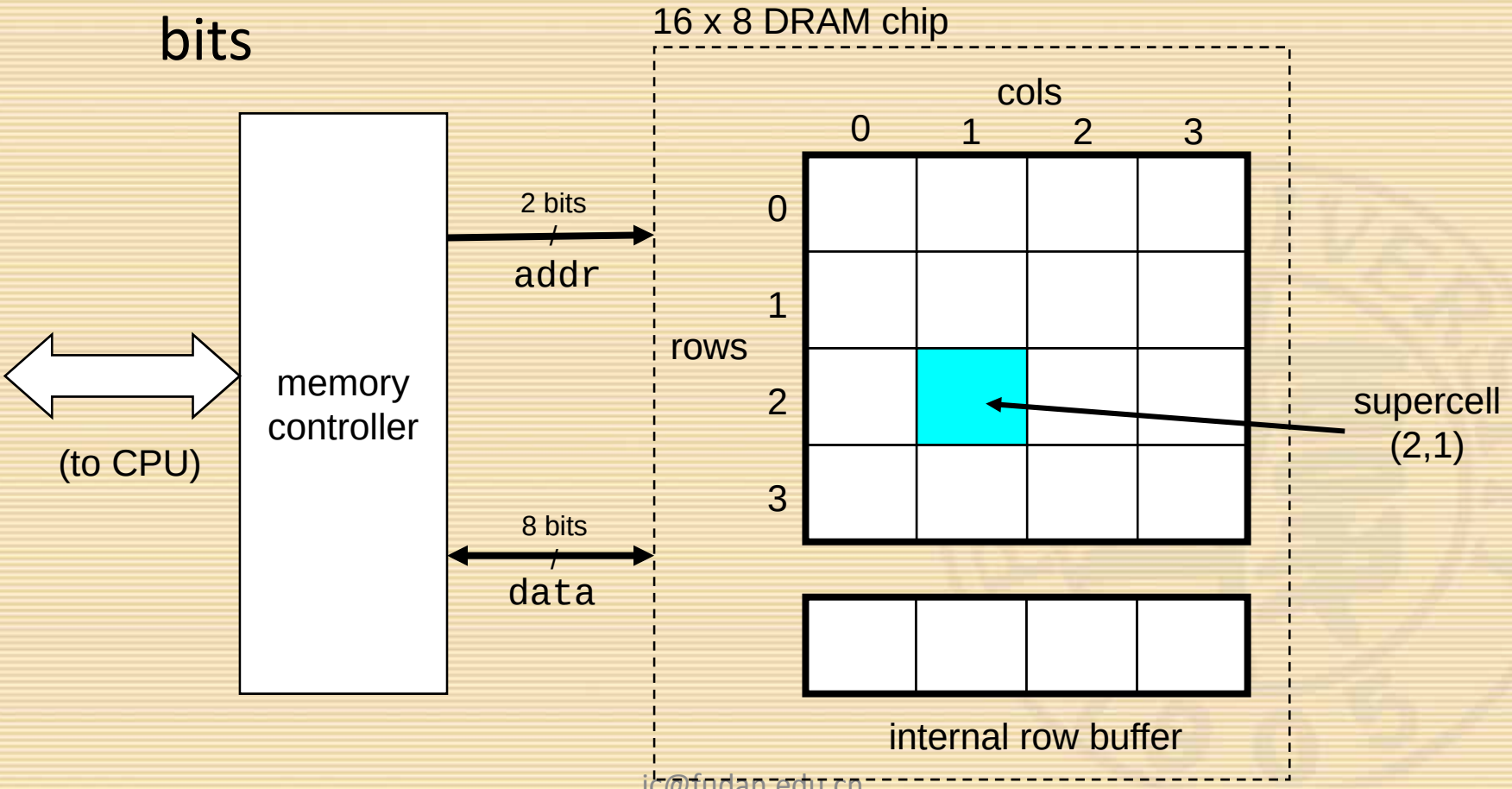
- Key features
  - RAM is packaged as a chip.
  - Basic storage unit is a cell (one bit per cell).
  - Multiple RAM chips form a memory.
- Static RAM (SRAM)
  - Each cell stores bit with a six-transistor circuit.
  - Retains value indefinitely, as long as it is kept powered.
  - Relatively insensitive to disturbances such as electrical noise.
  - Faster and more expensive than DRAM.
- Dynamic RAM (DRAM)
  - Each cell stores bit with a capacitor and transistor.
  - Value must be refreshed every 10-100 ms.
  - Sensitive to disturbances.
  - Slower and cheaper than SRAM.

SRAM vs DRAM Summary

	Tran. per bit	Access time	Persist?	Sensitive?	Cost	Applications
SRAM	6	1X	Yes	No	100x	cache memories
DRAM	1	10X	No	Yes	1X	Main memories, frame buffers

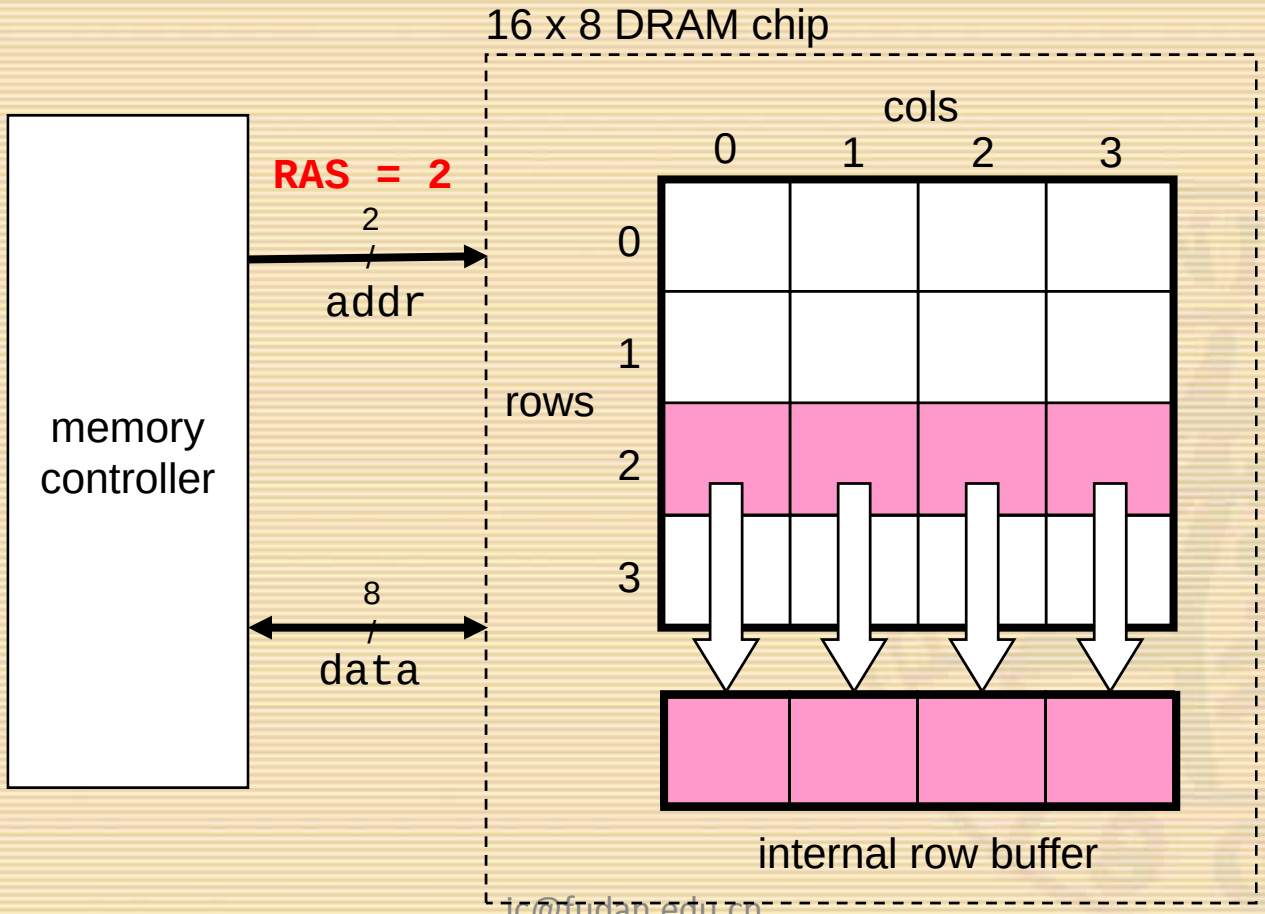
Conventional DRAM Organization

- d x w DRAM:
  - dw total bits organized as d supercells of size w bits



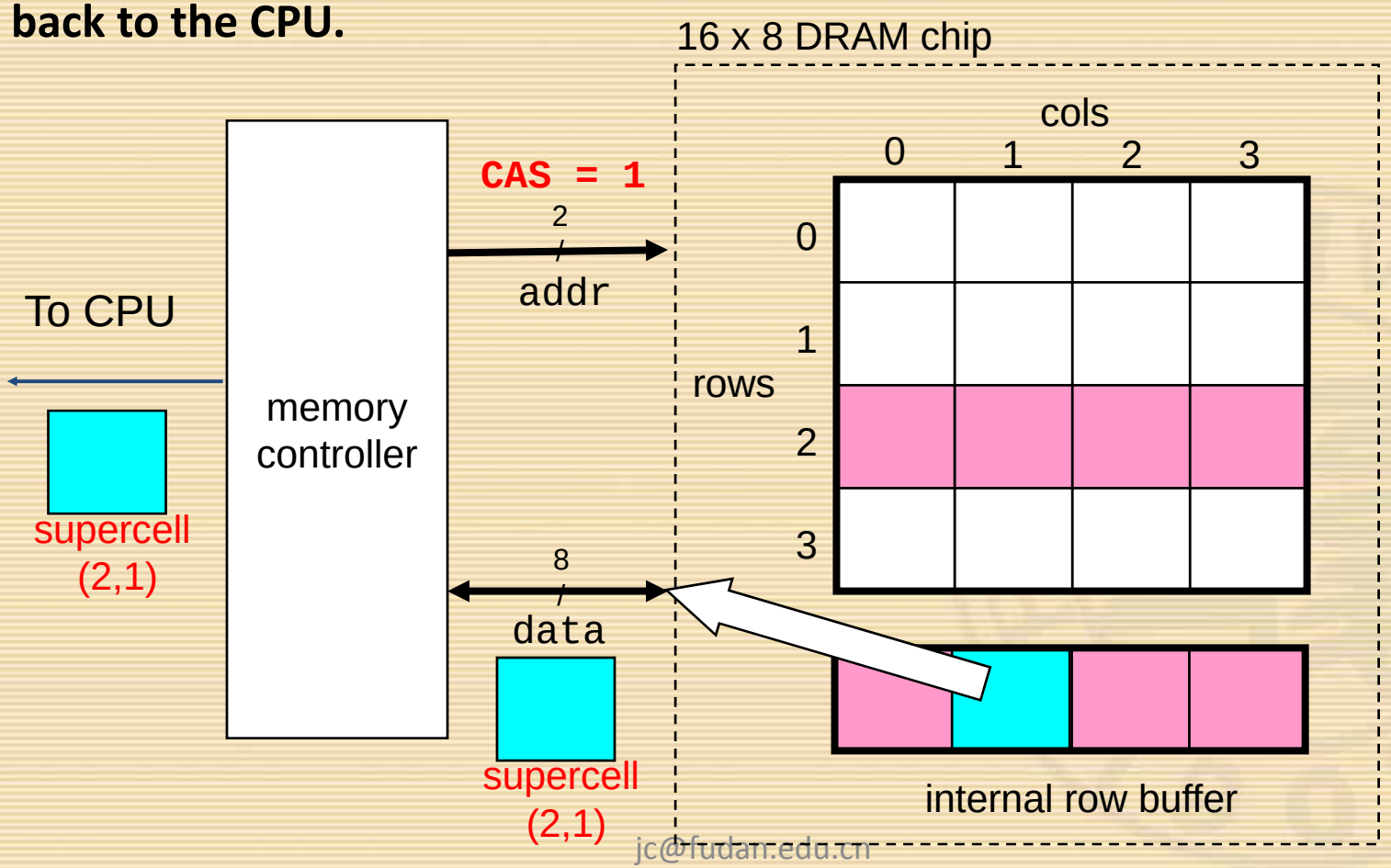
Reading DRAM Supercell (2,1)

- Step 1(a): Row access strobe (RAS) selects row 2.
- Step 1(b): Row 2 copied from DRAM array to row buffer.

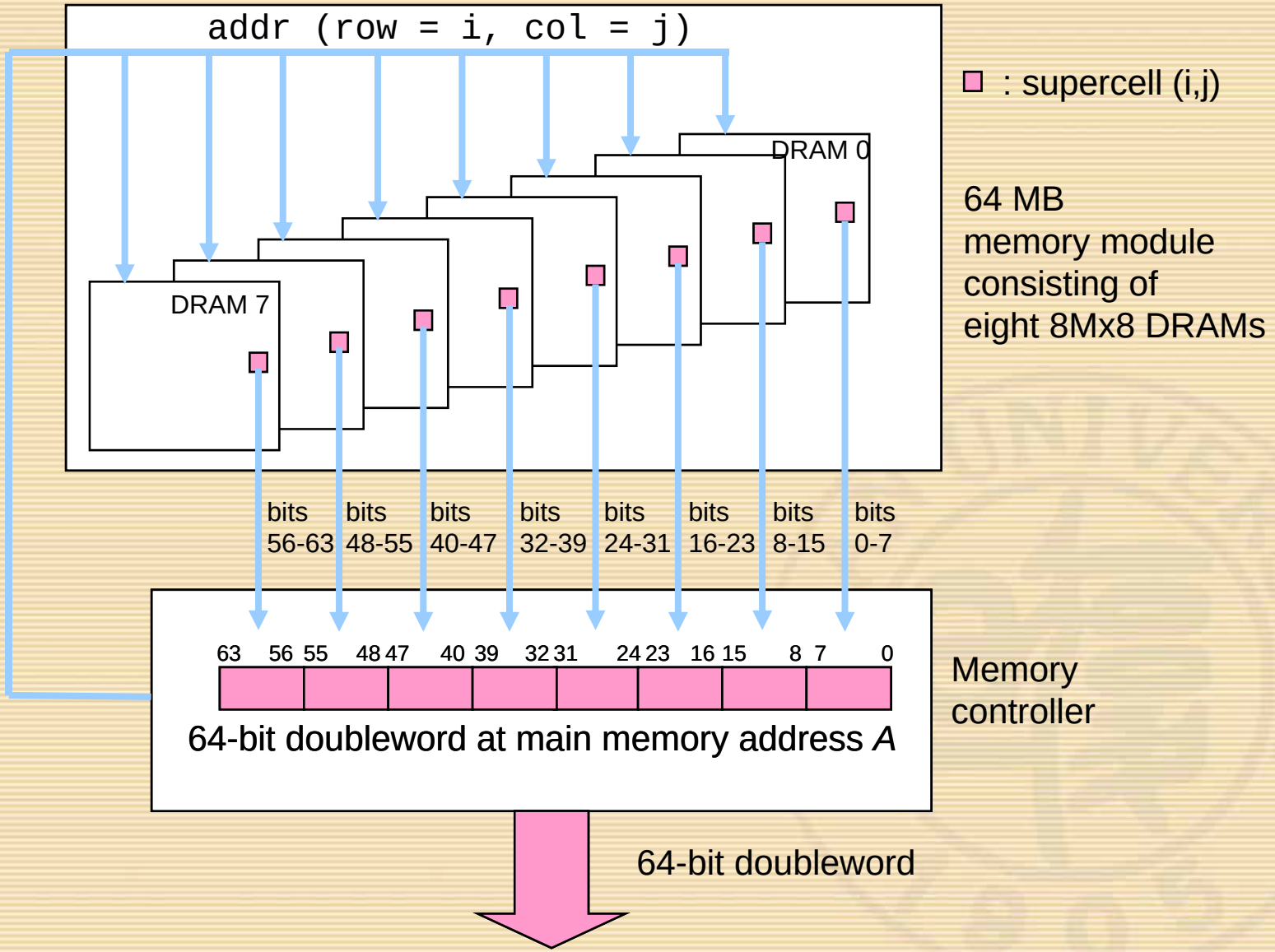


Reading DRAM Supercell (2,1)

- **Step 2(a): Column access strobe (CAS) selects column 1.**
- **Step 2(b): Supercell (2,1) copied from buffer to data lines, and eventually back to the CPU.**



Memory Modules



: supercell (i,j)

64 MB
memory module
consisting of
eight 8Mx8 DRAMs

Memory
controller

Enhanced DRAMs

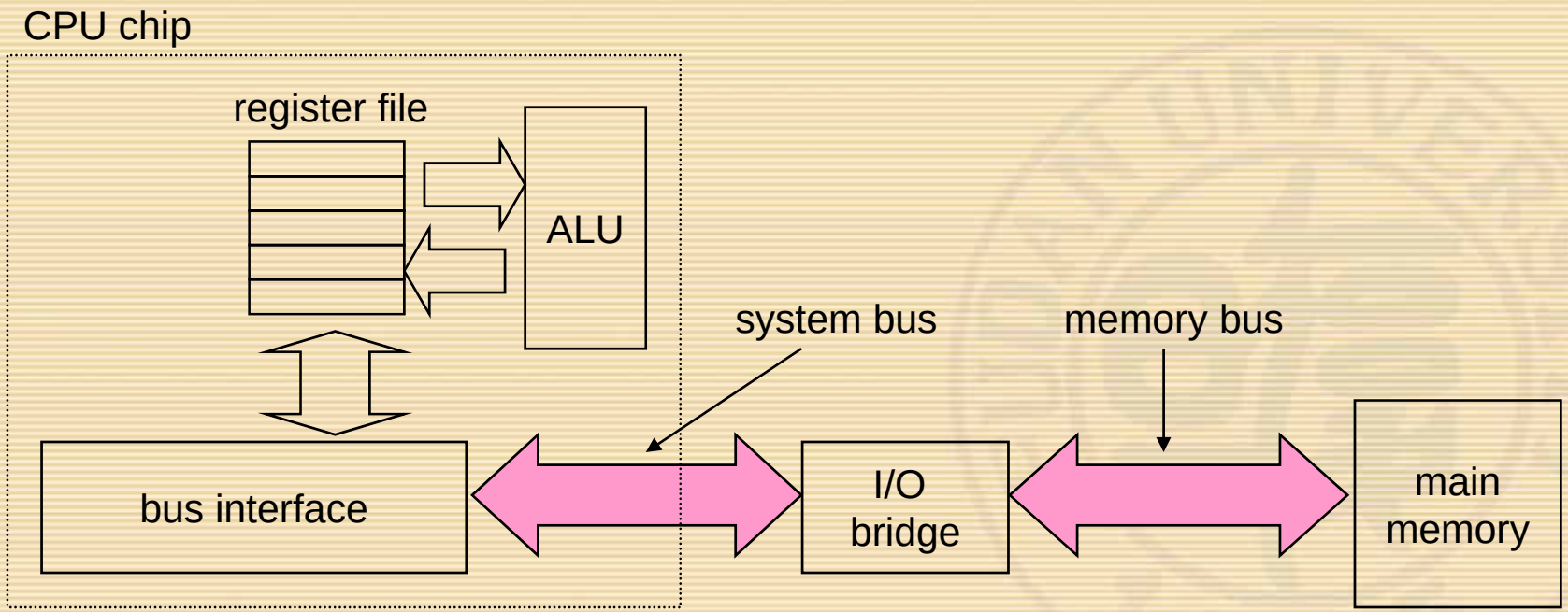
- All enhanced DRAMs are built around the conventional DRAM core.
  - Fast page mode DRAM (FPM DRAM)
    - Access contents of row with [RAS, CAS, CAS, CAS, CAS] instead of [(RAS,CAS), (RAS,CAS), (RAS,CAS), (RAS,CAS)].
  - Extended data out DRAM (EDO DRAM)
    - Enhanced FPM DRAM with more closely spaced CAS signals.
  - Synchronous DRAM (SDRAM)
    - Driven with rising clock edge instead of asynchronous control signals.
  - Double data-rate synchronous DRAM (DDR SDRAM)
    - Enhancement of SDRAM that uses both clock edges as control signals.
  - Video RAM (VRAM)
    - Like FPM DRAM, but output is produced by shifting row buffer
    - Dual ported (allows concurrent reads and writes)

Nonvolatile Memories

- DRAM and SRAM are volatile memories
  - Lose information if powered off.
- Nonvolatile memories retain value even if powered off.
  - Generic name is read-only memory (ROM).
  - Misleading because some ROMs can be read and modified.
- Types of ROMs
  - Programmable ROM (PROM)
  - Erasable programmable ROM (EPROM)
  - Electrically erasable PROM (EEPROM)
  - Flash memory
- Firmware
  - Program stored in a ROM
    - Boot time code, BIOS (basic input/output system)
    - graphics cards, disk controllers.

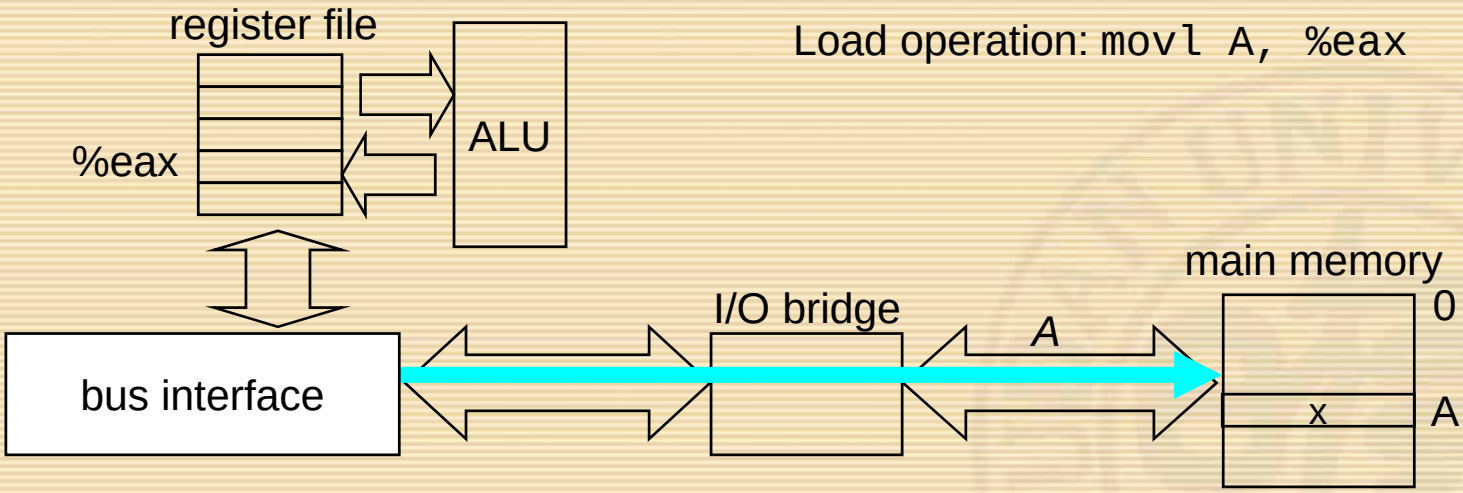
Typical Bus Structure Connecting CPU and Memory

- A bus is a collection of parallel wires that carry address, data, and control signals.
- Buses are typically shared by multiple devices.



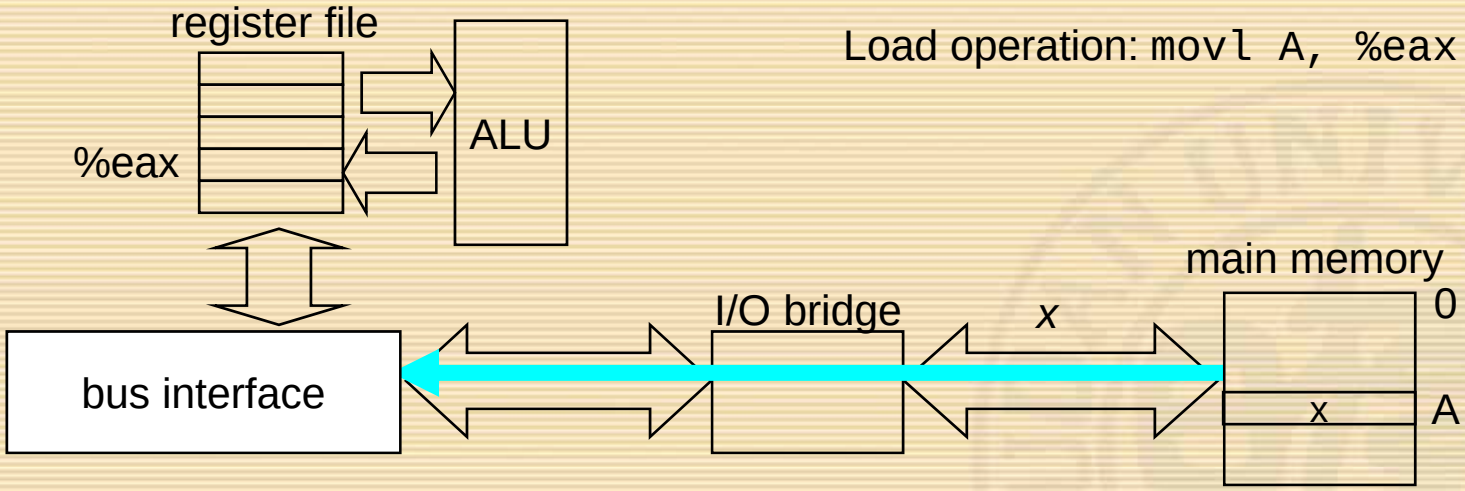
Memory Read Transaction (1)

- CPU places address A on the memory bus.



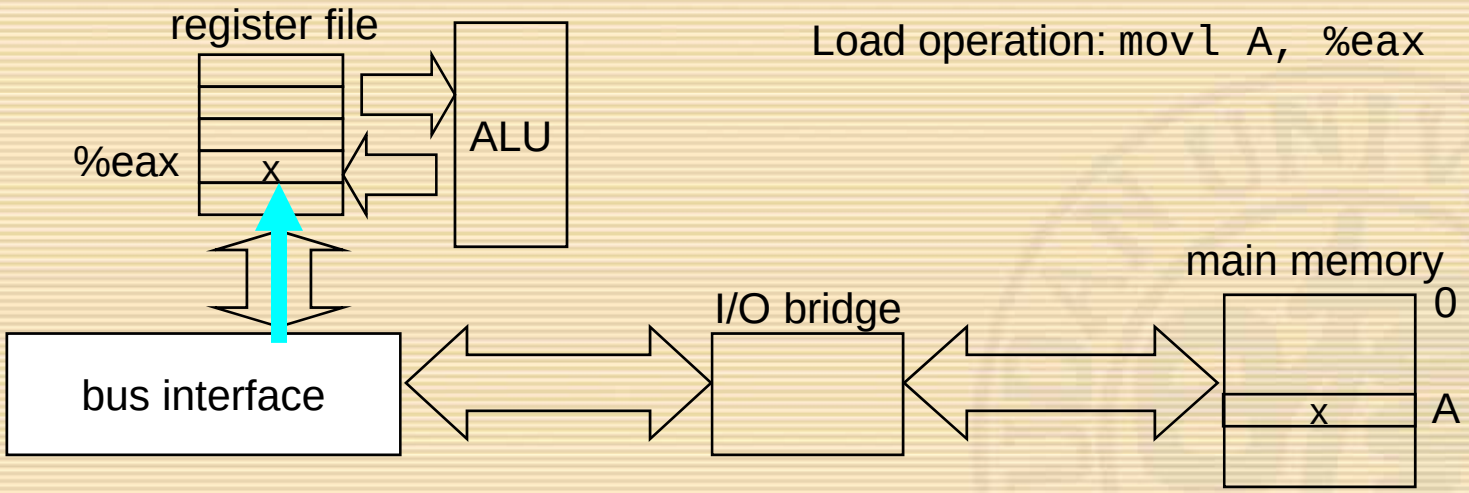
Memory Read Transaction (2)

- Main memory reads A from the memory bus, retrieves word x, and places it on the bus.



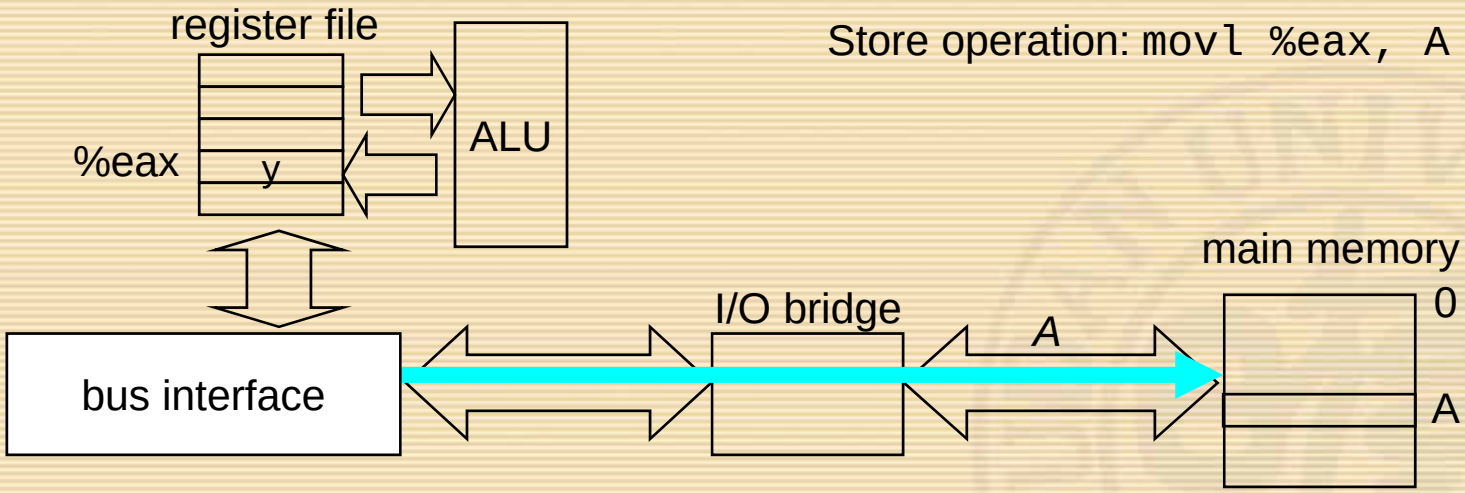
Memory Read Transaction (3)

- CPU read word x from the bus and copies it into register %eax.



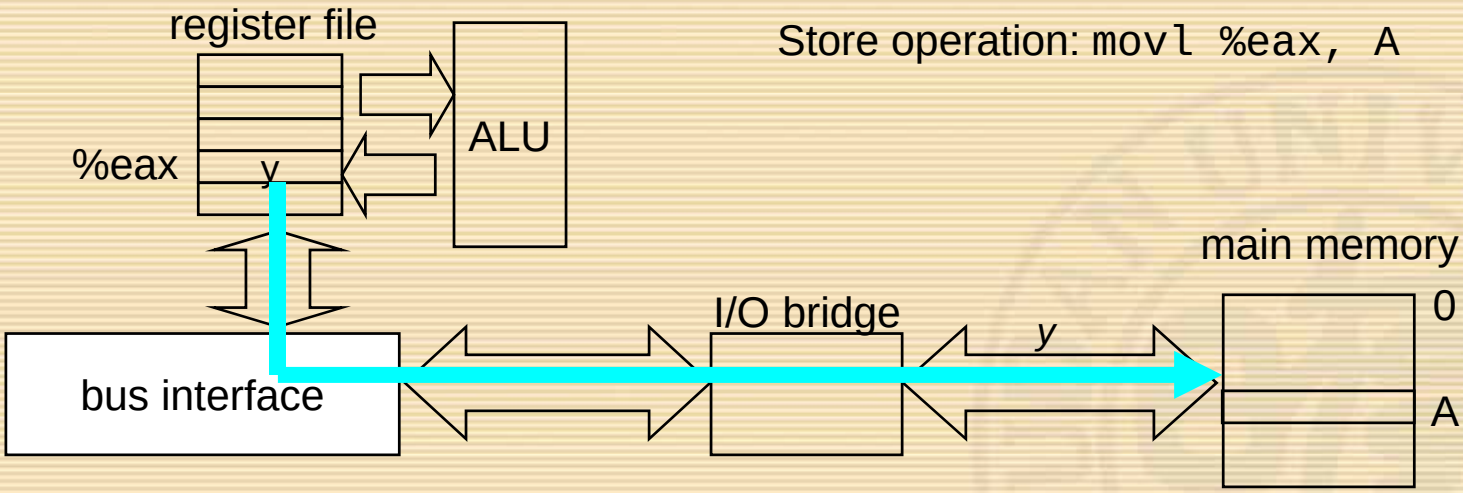
Memory Write Transaction (1)

- CPU places address A on bus. Main memory reads it and waits for the corresponding data word to arrive.



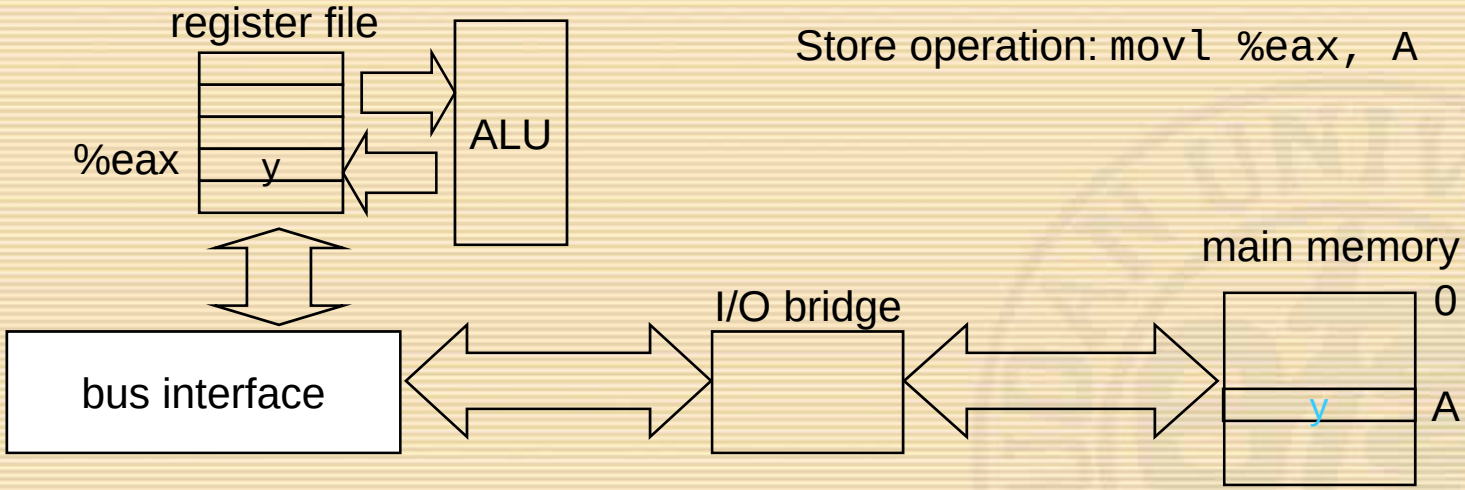
Memory Write Transaction (2)

- CPU places data word y on the bus.



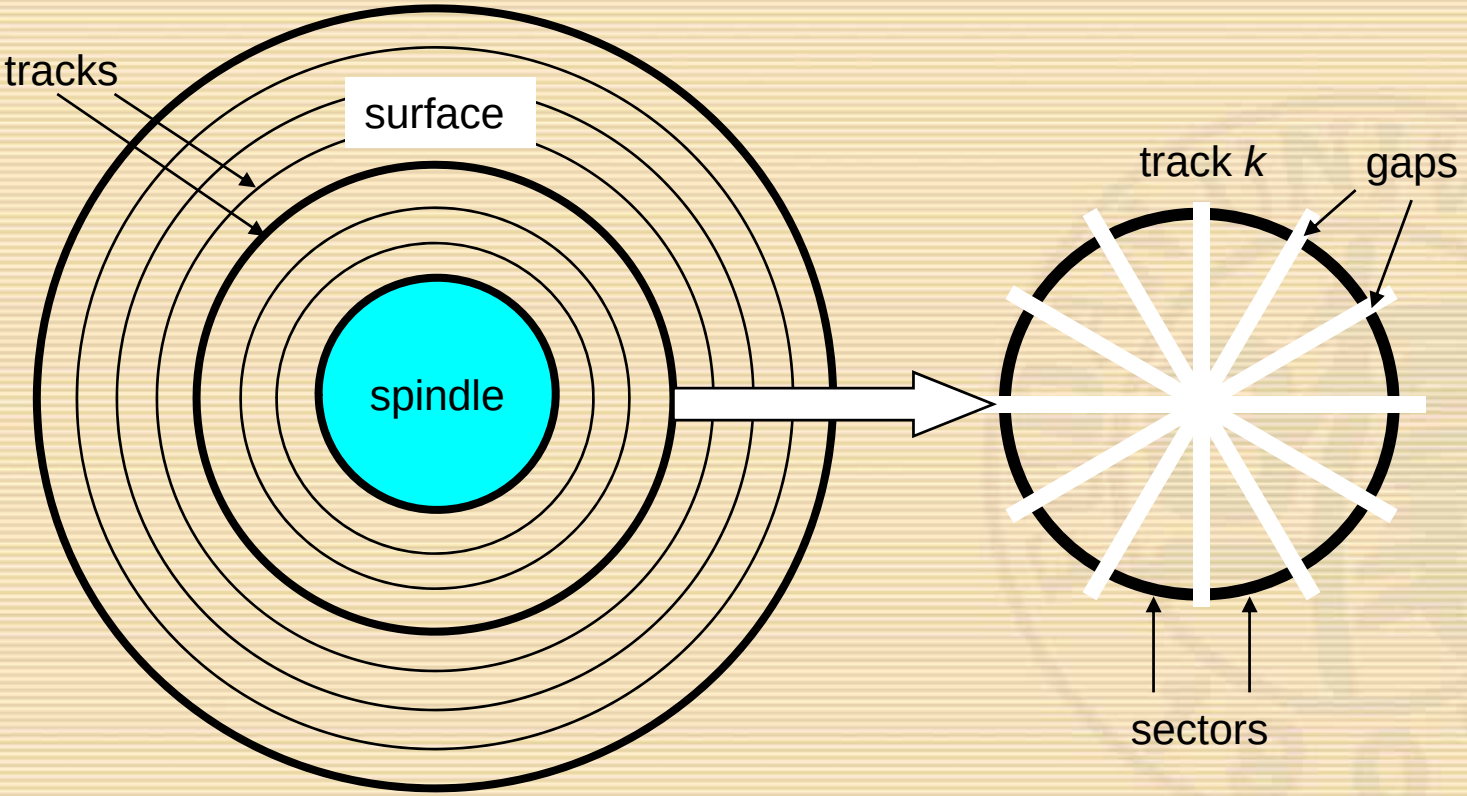
Memory Write Transaction (3)

- Main memory read data word y from the bus and stores it at address A.



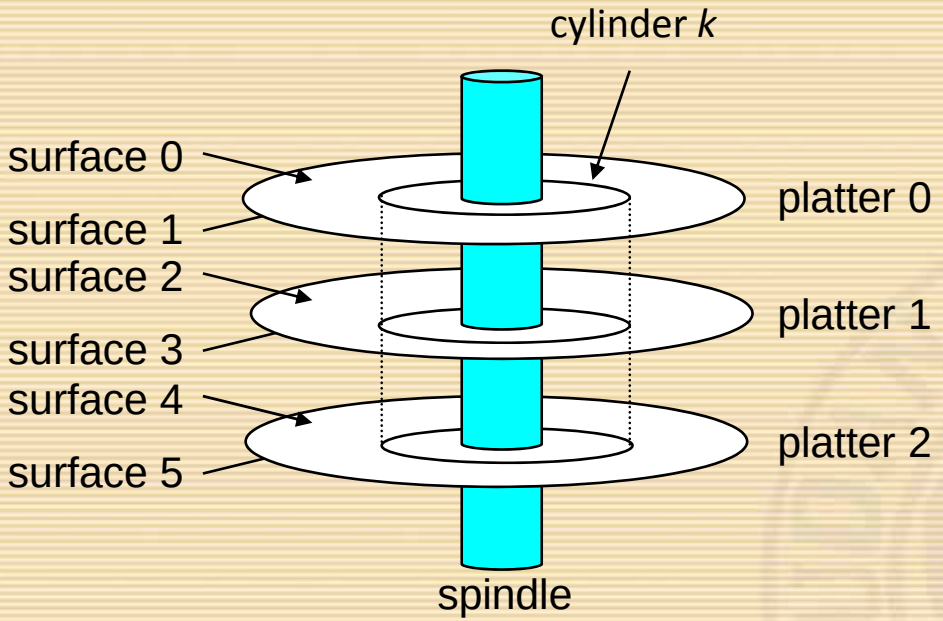
Disk Geometry

- Disks consist of platters, each with two surfaces.
- Each surface consists of concentric rings called tracks.
- Each track consists of sectors separated by gaps.



Disk Geometry (Muliple-Platter View)

- Aligned tracks form a cylinder.



Disk Capacity

- Capacity: maximum number of bits that can be stored.
  - Vendors express capacity in units of gigabytes (GB), where 1 GB = 10^9B.
- Capacity is determined by these technology factors:
  - Recording density (bits/in): number of bits that can be squeezed into a 1 inch segment of a track.
  - Track density (tracks/in): number of tracks that can be squeezed into a 1 inch radial segment.
  - Areal density (bits/in2): product of recording and track density.
- Modern disks partition tracks into disjoint subsets called recording zones
  - Each track in a zone has the same number of sectors, determined by the circumference of innermost track.
  - Each zone has a different number of sectors/track

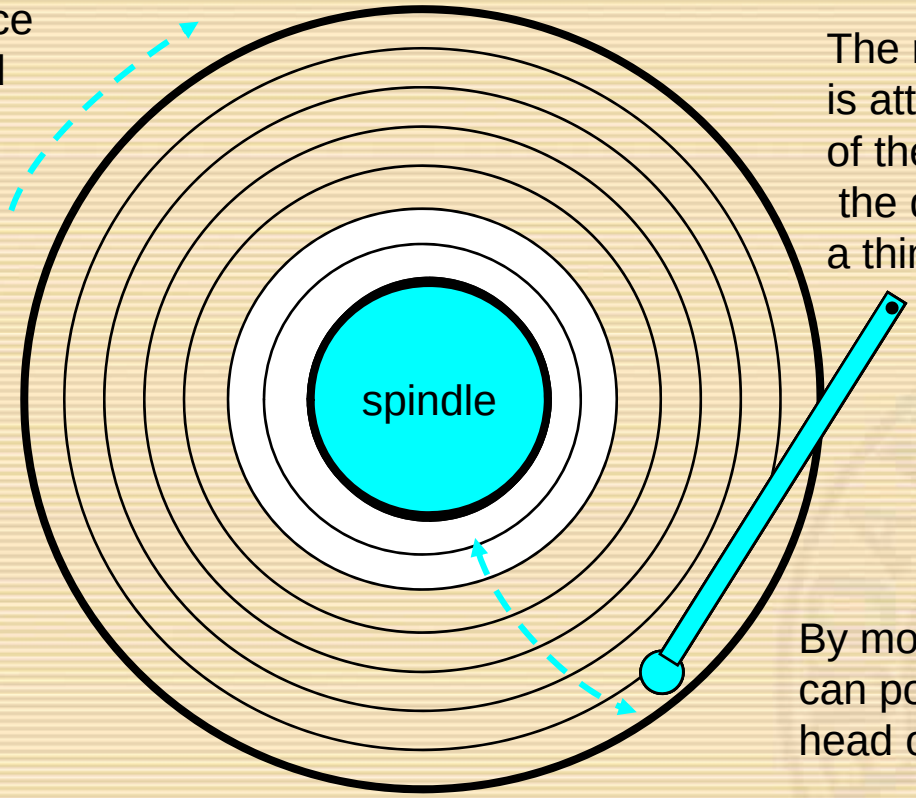
Computing Disk Capacity

- Capacity = (# bytes/sector) x (avg. # sectors/track) x
- (# tracks/surface) x (# surfaces/platter) x
- (# platters/disk)
- Example:
  - 512 bytes/sector
  - 300 sectors/track (on average)
  - 20,000 tracks/surface
  - 2 surfaces/platter
  - 5 platters/disk

- Capacity = 512 x 300 x 20000 x 2 x 5B
- = 30,720,000,000
- = 30.72 GB

Disk Operation (Single-Platter View)

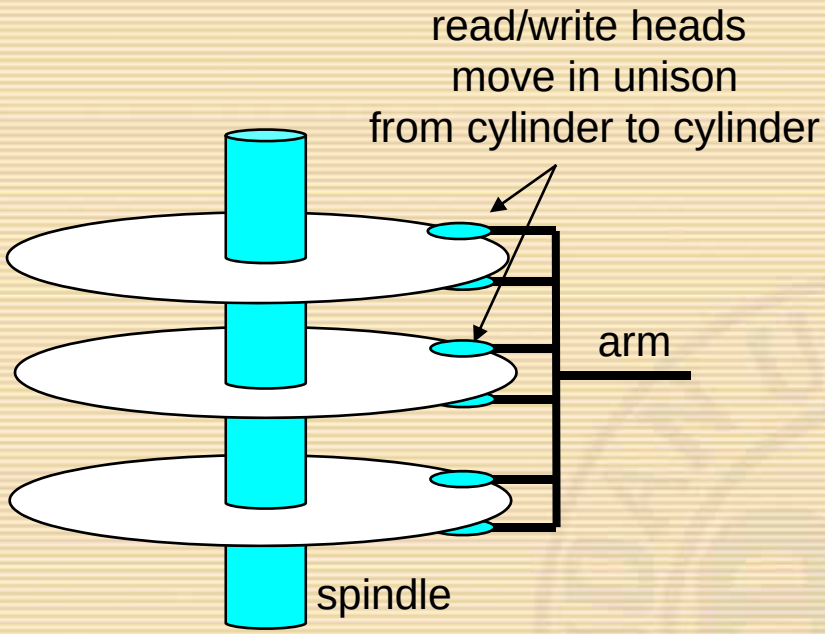
The disk surface spins at a fixed rotational rate



The read/write *head* is attached to the end of the *arm* and flies over the disk surface on a thin cushion of air.

By moving radially, the arm can position the read/write head over any track.

Disk Operation (Multi-Platter View)



Disk Access Time

- Average time to access some target sector approximated by :
  - Taccess = Tavg seek + Tavg rotation + Tavg transfer
- Seek time (Tavg seek)
  - Time to position heads over cylinder containing target sector.
  - Typical Tavg seek = 9 ms
- Rotational latency (Tavg rotation)
  - Time waiting for first bit of target sector to pass under r/w head.
  - Tavg rotation = 1/2 x 1/RPMs x 60 sec/1 min
- Transfer time (Tavg transfer)
  - Time to read the bits in the target sector.
  - Tavg transfer = 1/RPM x 1/(avg # sectors/track) x 60 secs/1 min.

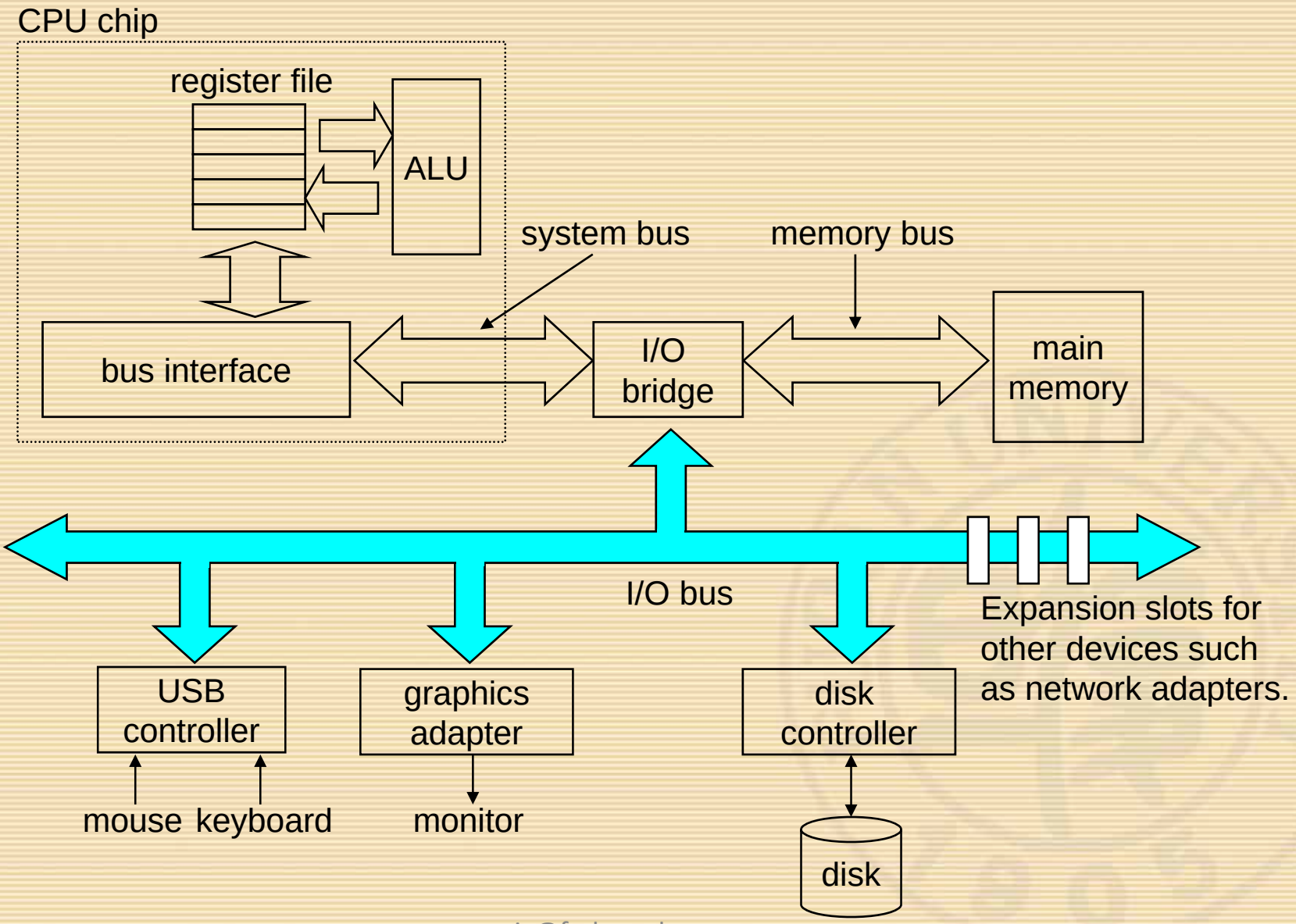
Disk Access Time Example

- Given:
  - Rotational rate = 7,200 RPM
  - Average seek time = 9 ms.
  - Avg # sectors/track = 400.
- Derived:
  - Tavg rotation = 1/2 x (60 secs/7200 RPM) x 1000 ms/sec = 4 ms.
  - Tavg transfer = 60/7200 RPM x 1/400 secs/track x 1000 ms/sec = 0.02 ms
  - Taccess = 9 ms + 4 ms + 0.02 ms
- Important points:
  - Access time dominated by seek time and rotational latency.
  - First bit in a sector is the most expensive, the rest are free.
  - SRAM access time is about 4 ns/doubleword, DRAM about 60 ns
    - Disk is about 40,000 times slower than SRAM,
    - 2,500 times slower then DRAM.

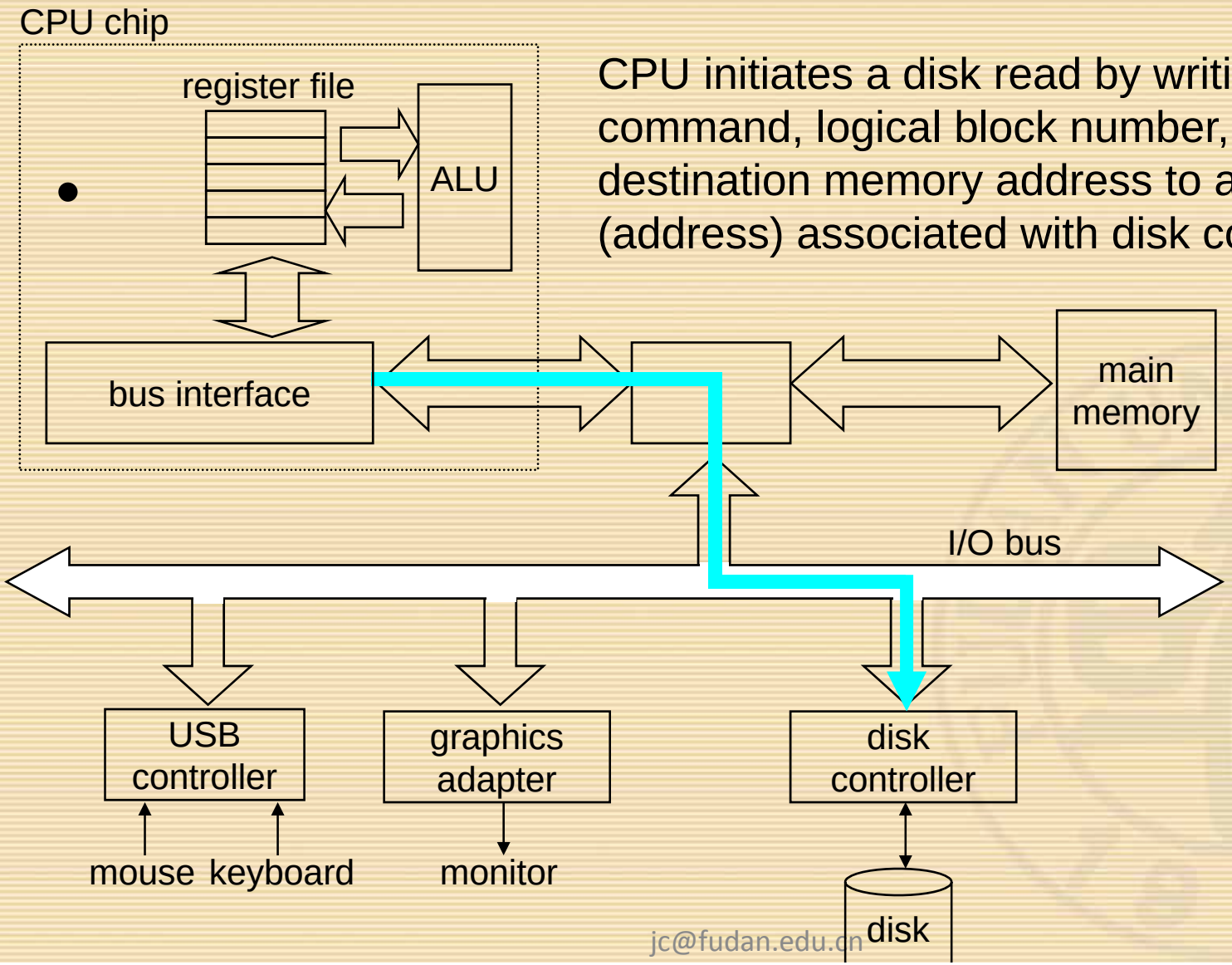
Logical Disk Blocks

- Modern disks present a simpler abstract view of the complex sector geometry:
  - The set of available sectors is modeled as a sequence of b-sized logical blocks (0, 1, 2, ...)
- Mapping between logical blocks and actual (physical) sectors
  - Maintained by hardware/firmware device called disk controller.
  - Converts requests for logical blocks into (surface,track,sector) triples.
- Allows controller to set aside spare cylinders for each zone. P471
  - Accounts for the difference in "formatted capacity" and "maximum capacity".

I/O Bus

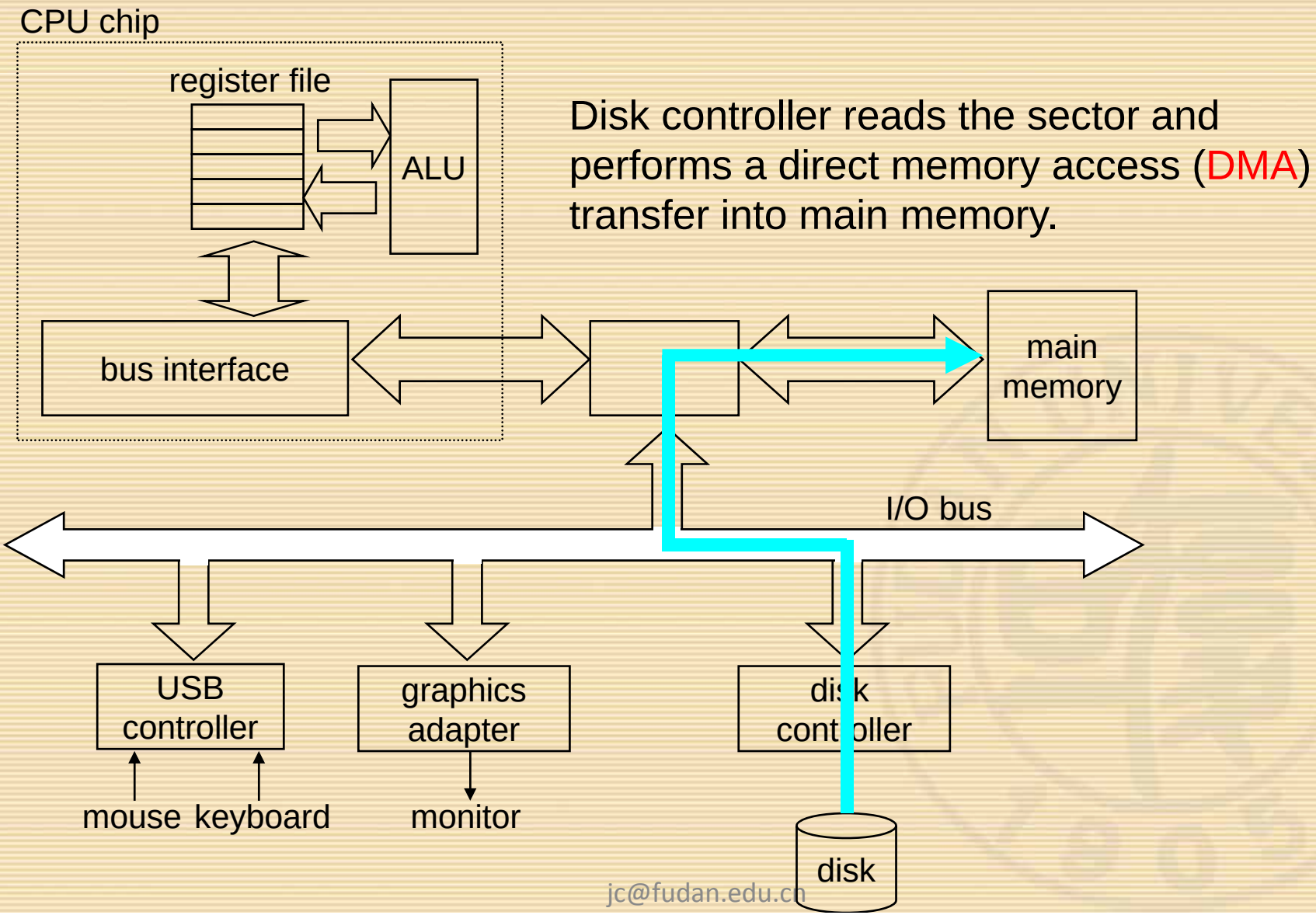

CPU chip
register file
ALU
system bus
memory bus
bus interface
I/O bridge
main memory
I/O bus
Expansion slots for other devices such as network adapters.
USB controller
graphics adapter
disk controller
mouse
keyboard
monitor
disk

Reading a Disk Sector (1)



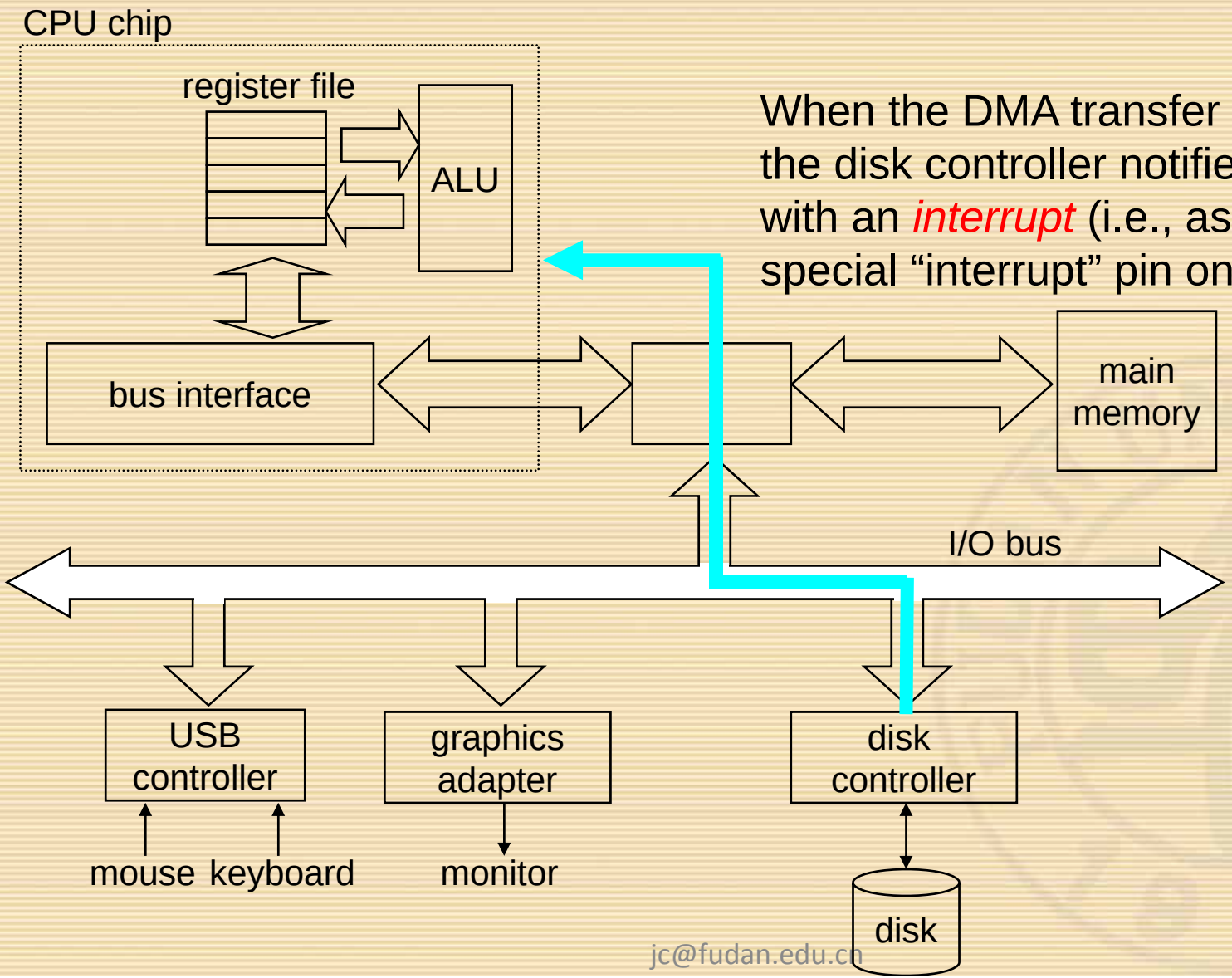
CPU initiates a disk read by writing a command, logical block number, and destination memory address to a port (address) associated with disk controller.

Reading a Disk Sector (2)



Disk controller reads the sector and performs a direct memory access (DMA) transfer into main memory.

Reading a Disk Sector (3)



When the DMA transfer completes, the disk controller notifies the CPU with an *interrupt* (i.e., asserts a special "interrupt" pin on the CPU)

Storage Trends

SRAM

metric	1980	1985	1990	1995	2000	*2000:1980*
$/MB	19,200	2,900	320	256	100	*190*
access (ns)	300	150	35	15	2	*100*

DRAM

metric	1980	1985	1990	1995	2000	*2000:1980*
$/MB	8,000	880	100	30	1	*8,000*
access (ns)	375	200	100	70	60	*6*
typical size(MB)	0.064	0.256	4	16	64	*1,000*

Disk

metric	1980	1985	1990	1995	2000	*2000:1980*
$/MB	500	100	8	0.30	0.05	*10,000*
access (ms)	87	75	28	10	8	*11*
typical size(MB)	1	10	160	1,000	9,000	*9,000*

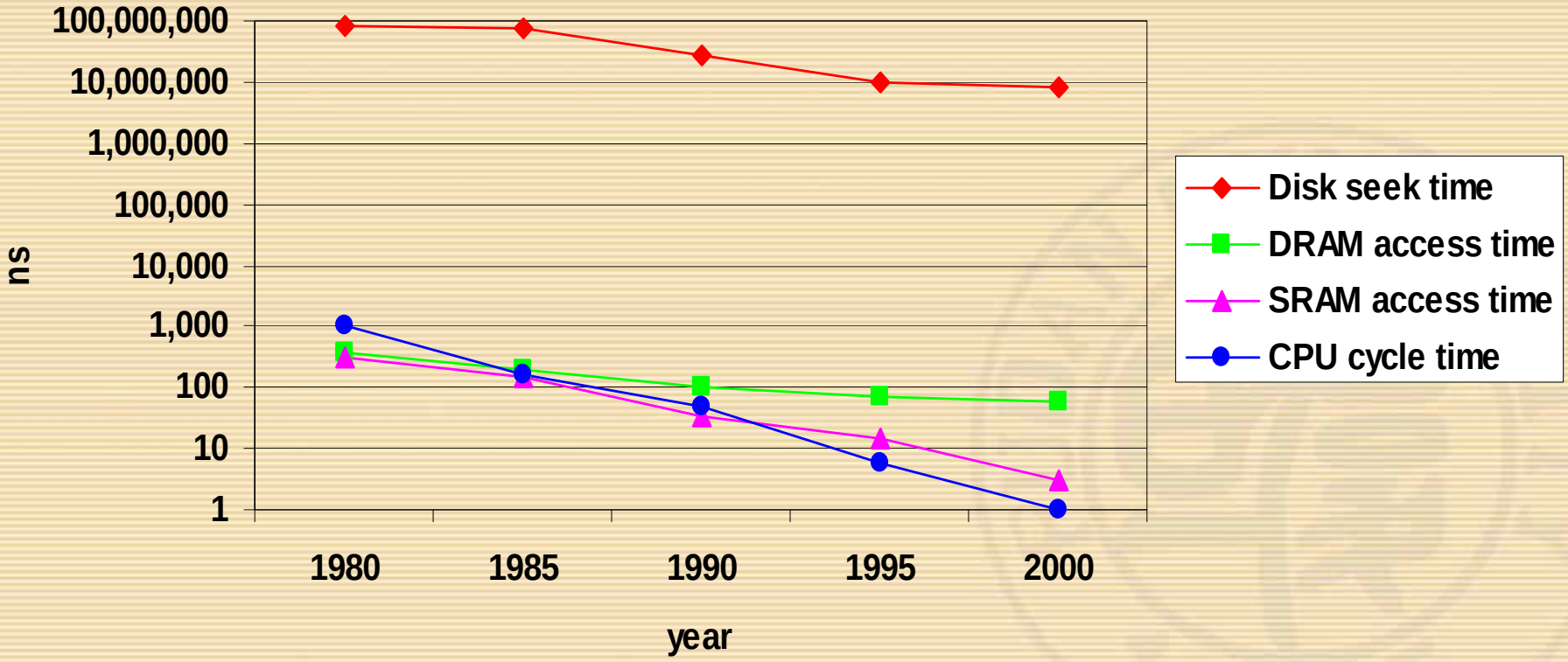
(Culled from back issues of Byte and PC Magazine)

CPU Clock Rates

	1980	1985	1990	1995	2000	*2000:1980*
processor 8080	286	386	Pent	P-III		
clock rate(MHz)	1	6	20	150	750	750
cycle time(ns)	1,000	166	50	6	1.6	750

The CPU-Memory Gap

- The increasing gap between DRAM, disk, and CPU speeds.



Locality

- Principle of Locality:
  - Programs tend to reuse data and instructions near those they have used recently, or that were recently referenced themselves.
  - Temporal locality: Recently referenced items are likely to be referenced in the near future.
  - Spatial locality: Items with nearby addresses tend to be referenced close together in time.

Locality Example:

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- Data
  - Reference array elements in succession (stride-1 reference pattern): Spatial locality
  - Reference sum each iteration: Temporal locality
- Instructions
  - Reference instructions in sequence: Spatial locality
  - Cycle through loop repeatedly: Temporal locality

Locality Example

- Claim: Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.
- Question: Does this function have good locality?

```
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Locality Example

- Question: Does this function have good locality?

```
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum
}
```

Locality Example

- Question: Can you permute the loops so that the function scans the 3-d array a[ ] with a stride-1 reference pattern (and thus has good spatial locality)?

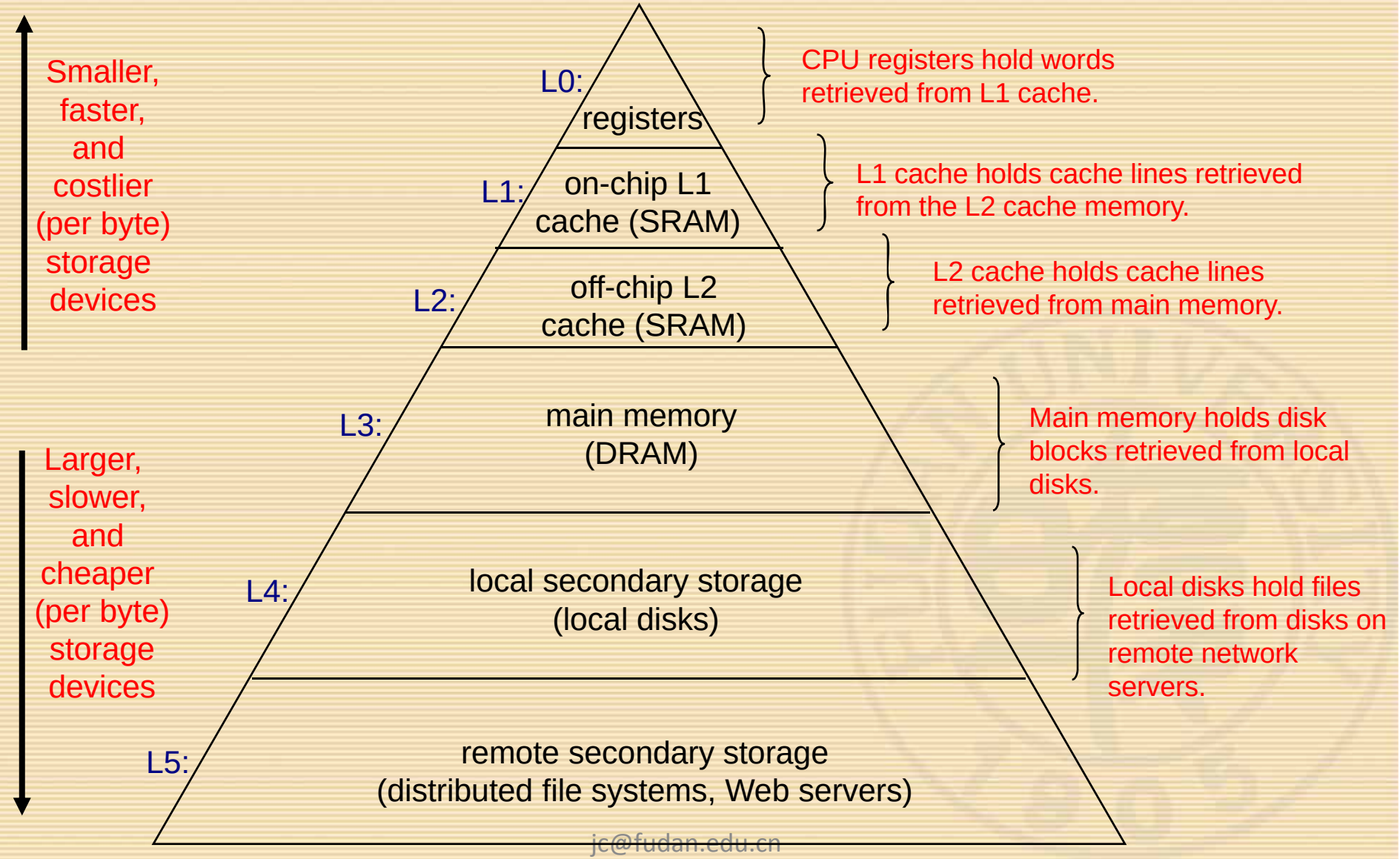
```
int sumarray3d(int a[N][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                sum += a[k][i][j];
    return sum;
}
```

Memory Hierarchies

- Some fundamental and enduring properties of hardware and software:
  - Fast storage technologies cost more per byte and have less capacity.
  - The gap between CPU and main memory speed is widening.
  - Well-written programs tend to exhibit good locality.
- These fundamental properties complement each other beautifully.
- They suggest an approach for organizing memory and storage systems known as a memory hierarchy.

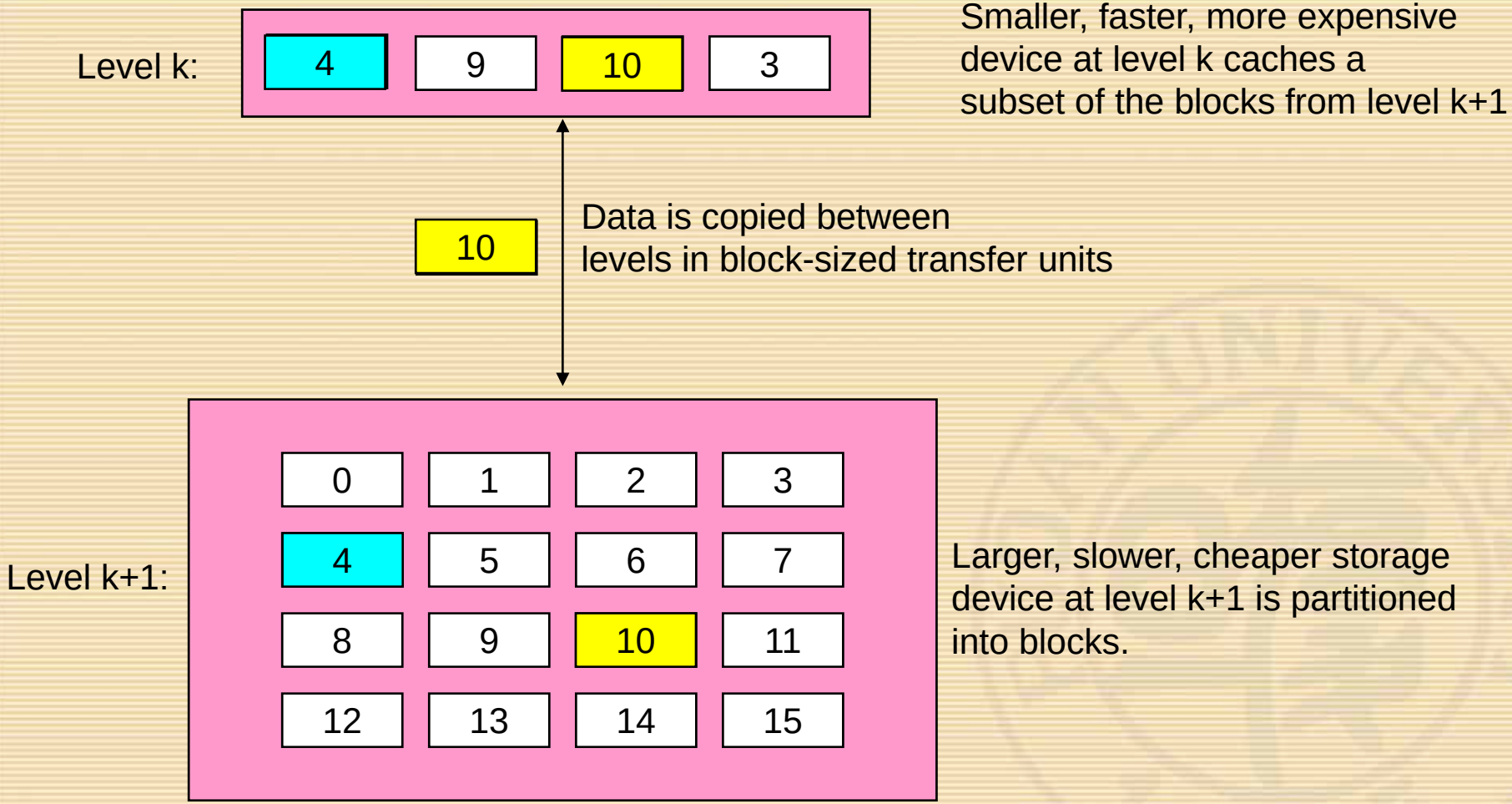
An Example Memory Hierarchy



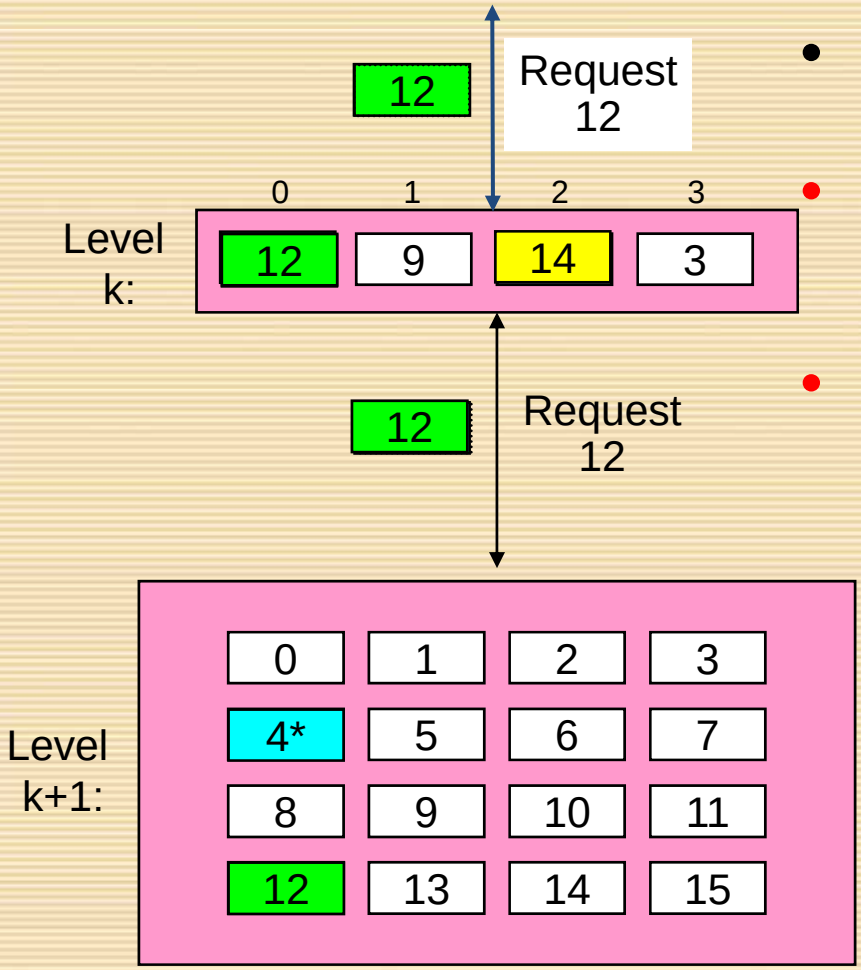
Caches

- Cache: A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.
- Fundamental idea of a memory hierarchy:
  - For each k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1.
- Why do memory hierarchies work?
  - Programs tend to access the data at level k more often than they access the data at level k+1.
  - Thus, the storage at level k+1 can be slower, and thus larger and cheaper per bit.
  - Net effect: A large pool of memory that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.

Caching in a Memory Hierarchy



General Caching Concepts



- Program needs object d, which is stored in some block b.
- Cache hit
  - Program finds b in the cache at level k. E.g., block 14.
- Cache miss
  - b is not at level k, so level k cache must fetch it from level k+1. E.g., block 12.
  - If level k cache is full, then some current block must be replaced (evicted). Which one is the "victim"?
    - Placement policy: where can the new block go? E.g., b mod 4
    - Replacement policy: which block should be evicted? E.g., LRU

General Caching Concepts

- Types of cache misses:
  - Cold (compulsary) miss
    - Cold misses occur because the cache is empty.
  - Conflict miss
    - Most caches limit blocks at level k+1 to a small subset (sometimes a singleton) of the block positions at level k.
    - E.g. Block i at level k+1 must be placed in block (i mod 4) at level k.
    - Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block.
    - E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time.
  - Capacity miss
    - Occurs when the set of active cache blocks (working set) is larger than the cache.

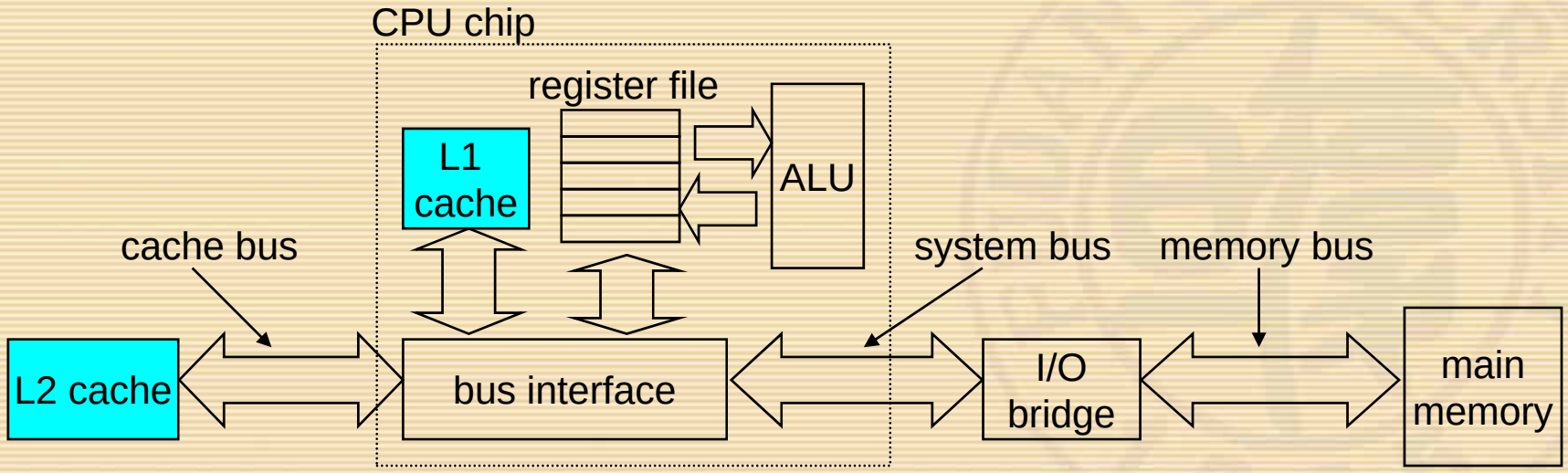
Examples of Caching in the Hierarchy

Cache Type	What Cached	Where Cached	Latency (cycles)	Managed By
Registers	4-byte word	CPU registers	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware
L1 cache	32-byte block	On-Chip L1	1	Hardware
L2 cache	32-byte block	Off-Chip L2	10	Hardware
Virtual Memory	4-KB page	Main memory	100	Hardware+ OS
Buffer cache	Parts of files	Main memory	100	OS
Network buffer cache	Parts of files	Local disk	10,000,000	AFS/NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

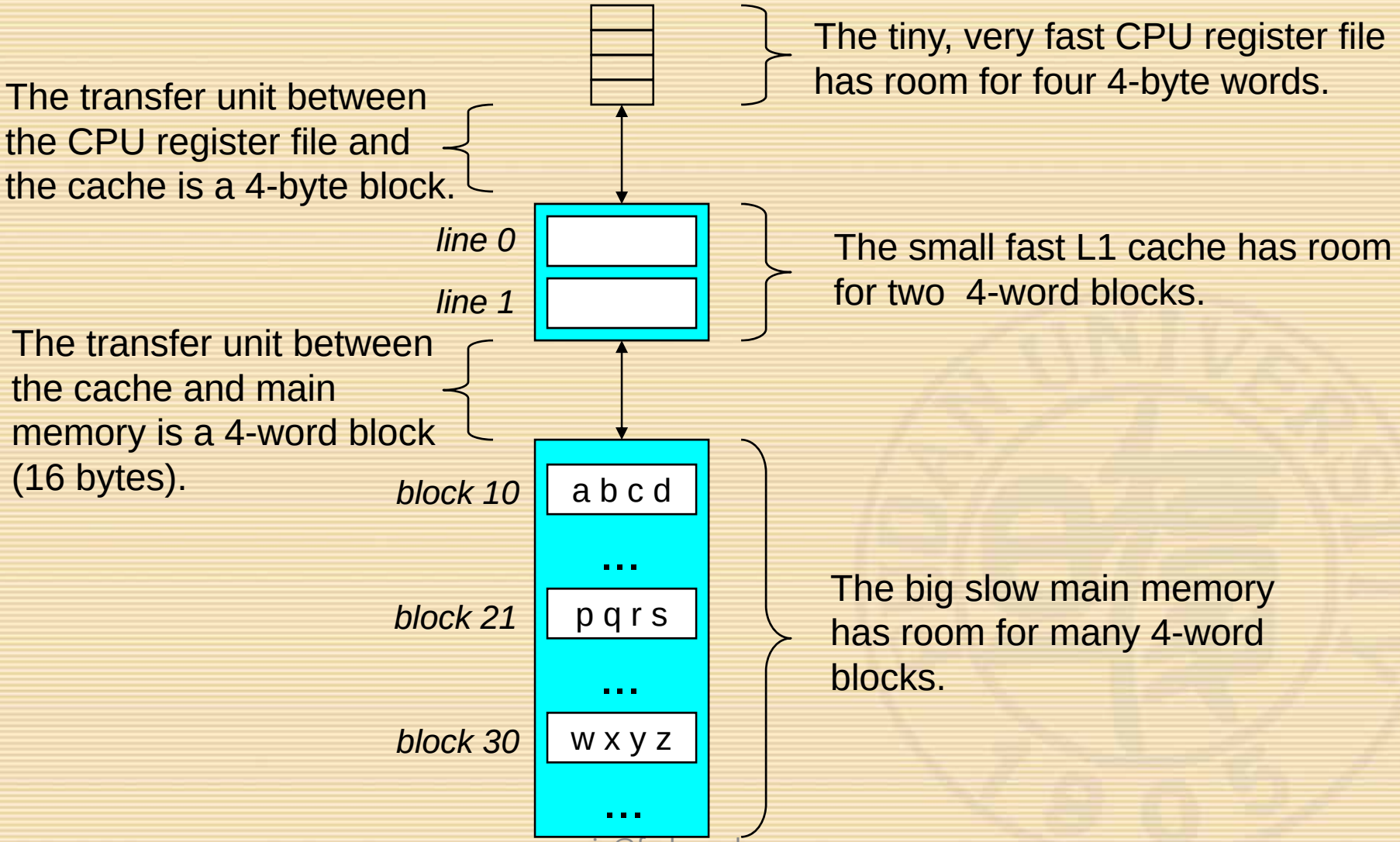
TLB: Translation Lookaside Buffer, maps virtual addresses to physical ones

Cache Memories

- Cache memories are small, fast SRAM-based memories managed automatically in hardware.
  - Hold frequently accessed blocks of main memory
- CPU looks first for data in L1, then in L2, then in main memory.
- Typical bus structure:



Inserting an L1 Cache Between the CPU and Main Memory

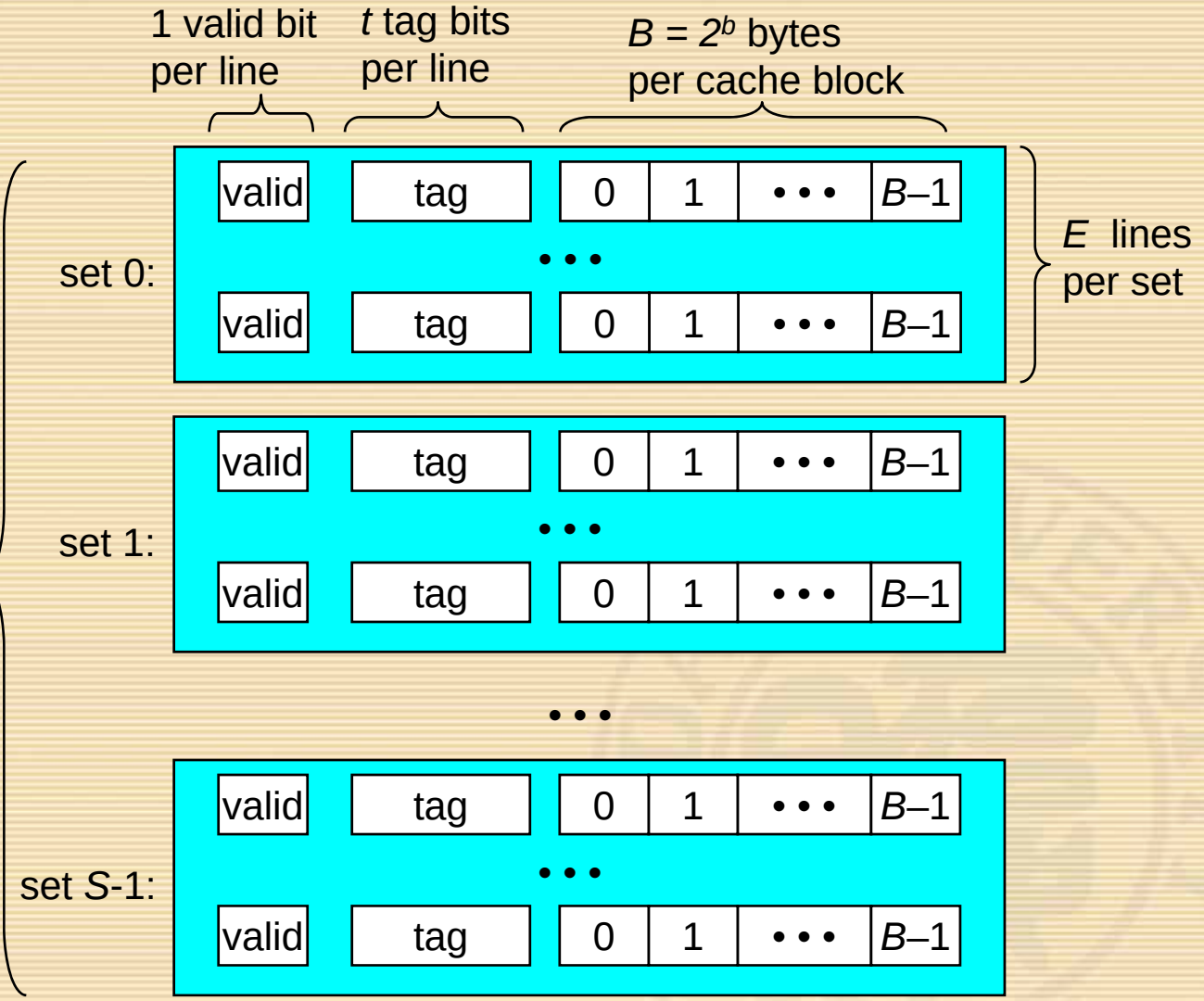


General Org of a Cache Memory

Cache is an array of sets.

Each set contains one or more lines.

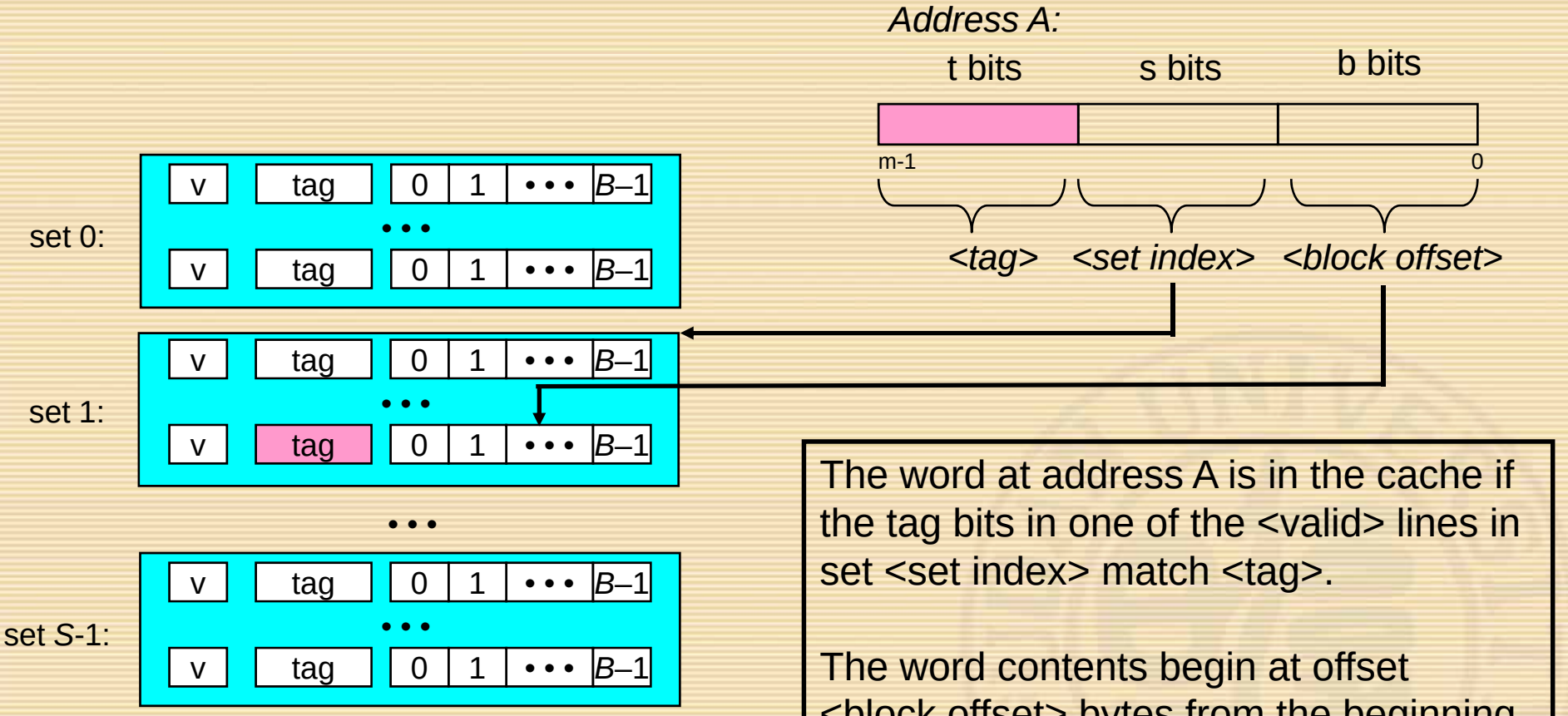
Each line holds a block of data.



Cache size: $C = B \times E \times S$ data bytes

jc@fudan.edu.cn

Addressing Caches

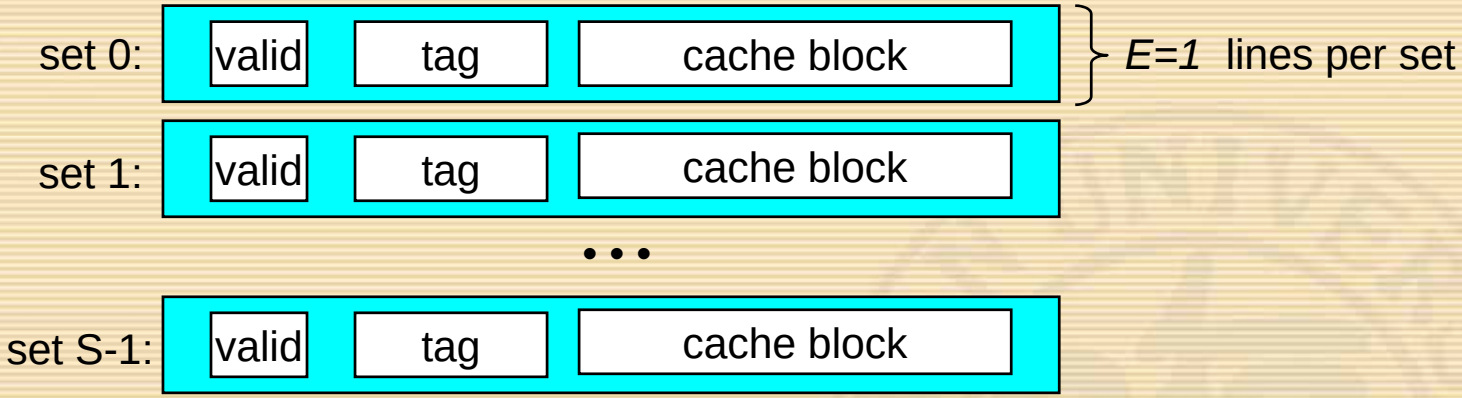


The word at address A is in the cache if the tag bits in one of the <valid> lines in set <set index> match <tag>.

The word contents begin at offset <block offset> bytes from the beginning of the block.

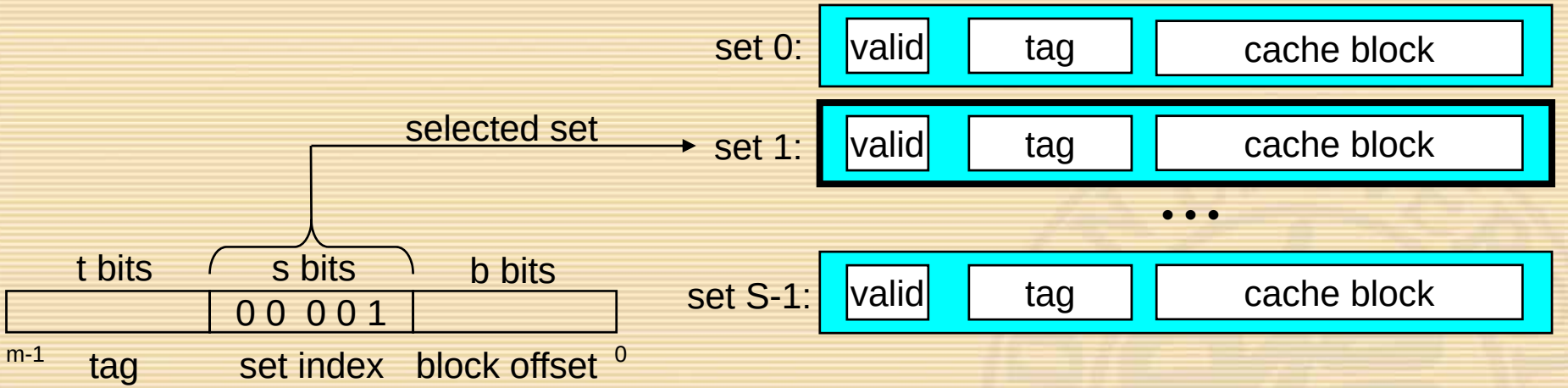
Direct-Mapped Cache

- Simplest kind of cache
- Characterized by exactly one line per set.



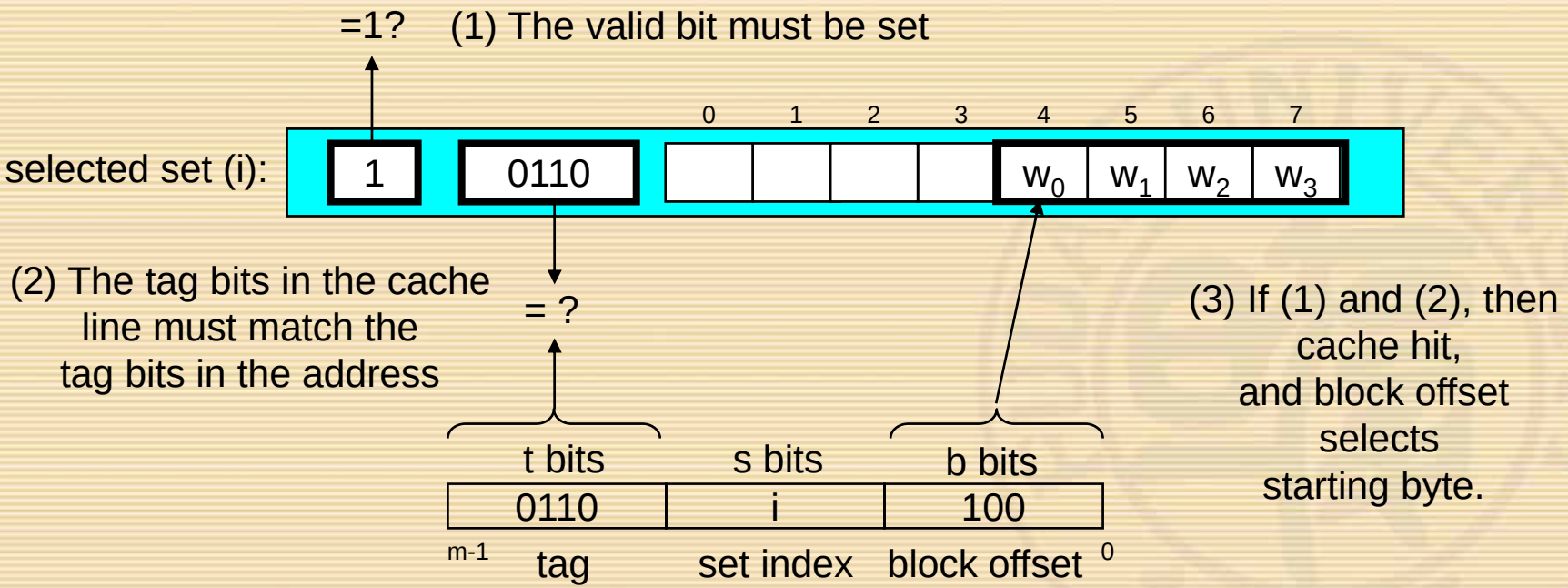
Accessing Direct-Mapped Caches

- Set selection
  - Use the set index bits to determine the set of interest.

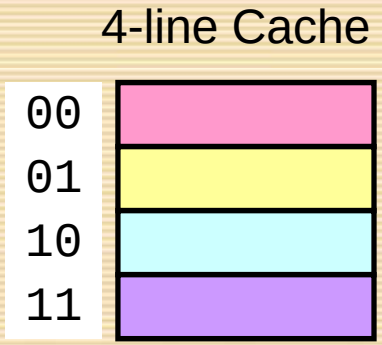


Accessing Direct-Mapped Caches

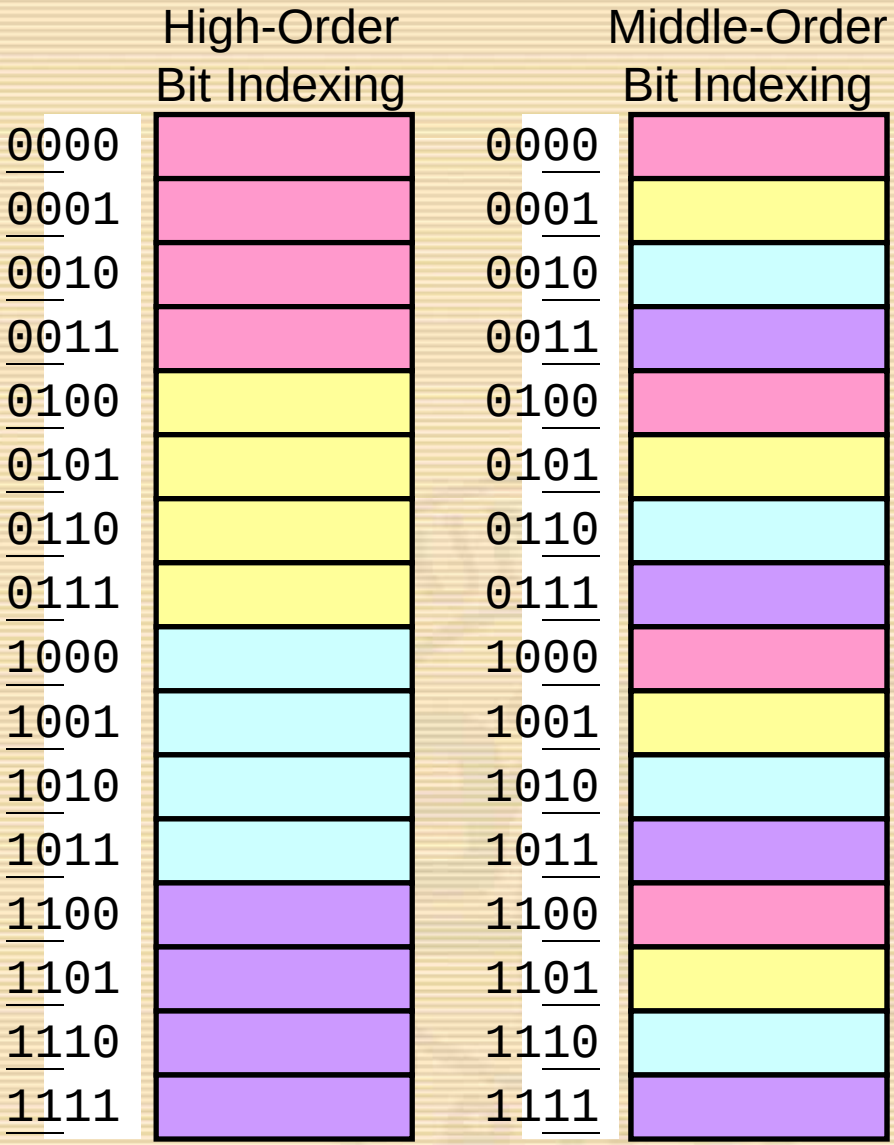
- Line matching and word selection
  - Line matching: Find a valid line in the selected set with a matching tag
  - Word selection: Then extract the word



Why Use Middle Bits as Set Index?

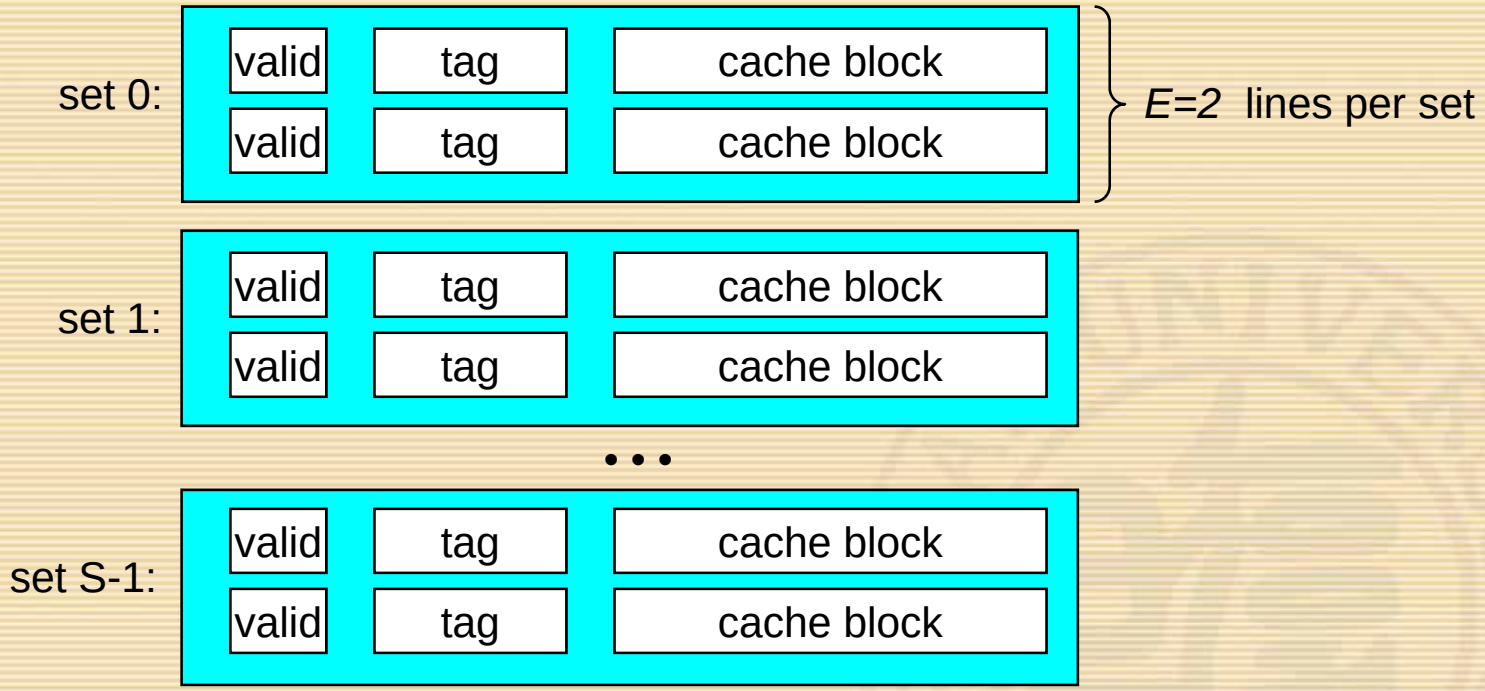


- High-Order Bit Indexing
  - Adjacent memory lines would map to same cache entry
  - Poor use of spatial locality
- Middle-Order Bit Indexing
  - Consecutive memory lines map to different cache lines
  - Can hold C-byte region of address space in cache at one time



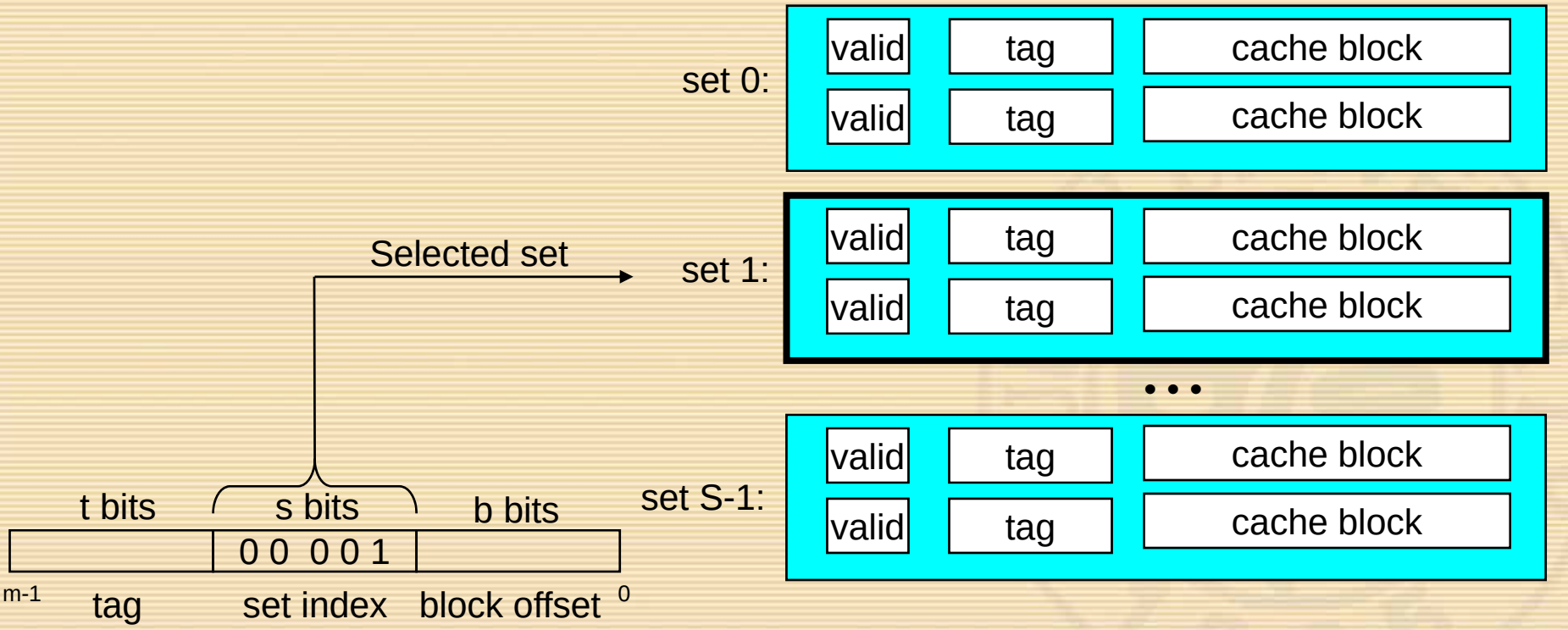
Set Associative Caches

- Characterized by more than one line per set



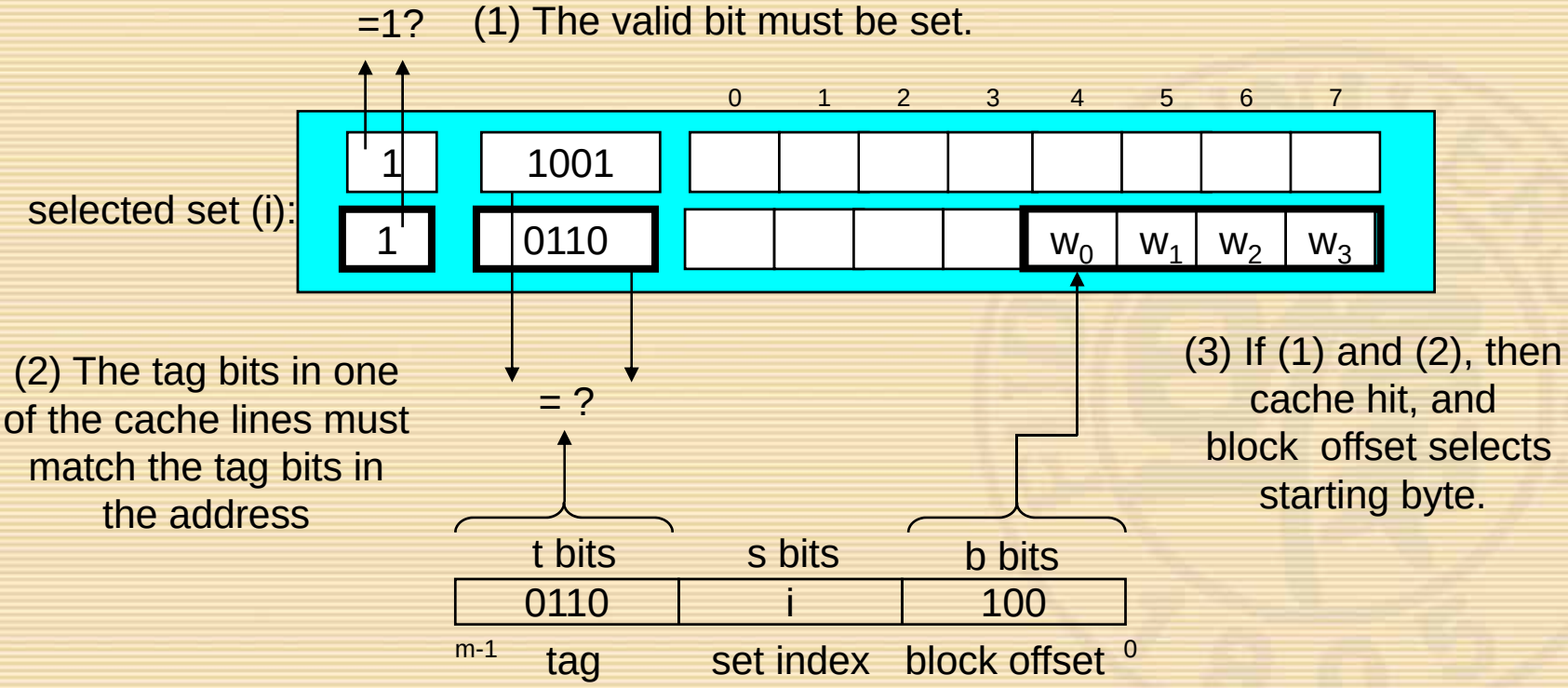
Accessing Set Associative Caches

- Set selection
  - identical to direct-mapped cache



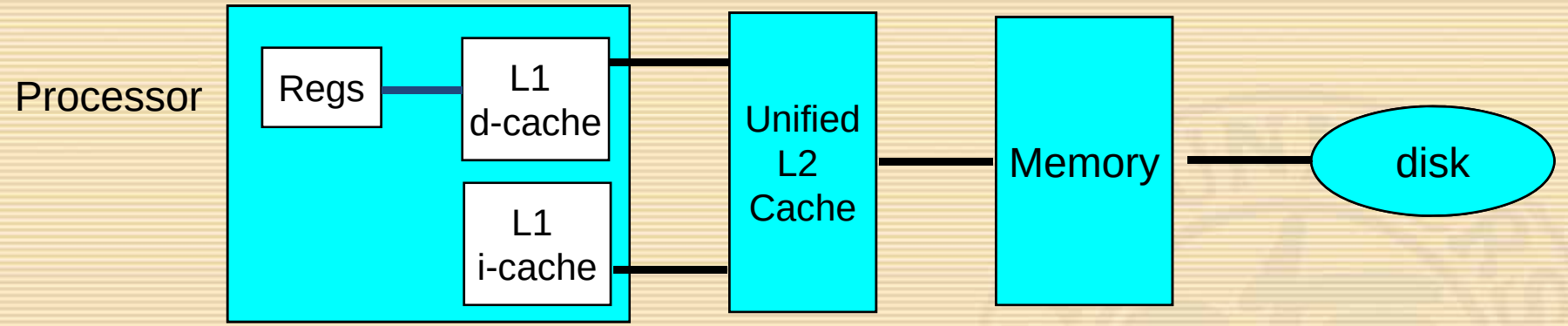
Accessing Set Associative Caches

- Line matching and word selection
  - must compare the tag in each valid line in the selected set.



Multi-Level Caches

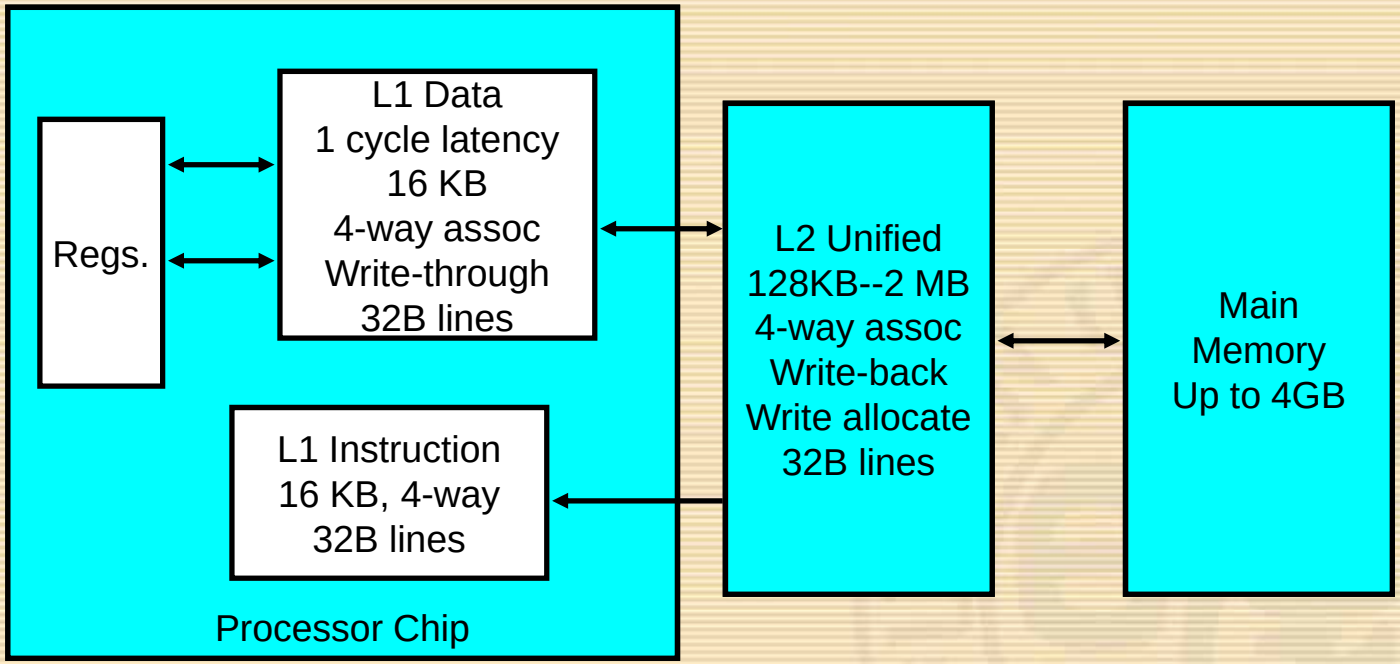
- Options: separate data and instruction caches, or a unified cache



size:	200 B	8-64 KB	1-4MB SRAM	128 MB DRAM	30 GB
speed:	3 ns	3 ns	6 ns	60 ns	8 ms
$/Mbyte:			$100/MB	$1.50/MB	$0.05/MB
line size:	8 B	32 B	32 B	8 KB	

larger, slower, cheaper

Intel Pentium Cache Hierarchy



Cache Performance Metrics

- Miss Rate
  - Fraction of memory references not found in cache (misses/references)
  - Typical numbers:
    - 3-10% for L1
    - can be quite small (e.g., < 1%) for L2, depending on size, etc.
- Hit Time
  - Time to deliver a line in the cache to the processor (includes time to determine whether the line is in the cache)
  - Typical numbers:
    - 1 clock cycle for L1
    - 3-8 clock cycles for L2
- Miss Penalty
  - Additional time required because of a miss
    - Typically 25-100 cycles for main memory

Writing Cache Friendly Code

- Repeated references to variables are good (temporal locality)
- Stride-1 reference patterns are good (spatial locality)
- Examples:
  - cold cache, 4-byte words, 4-word cache blocks

```
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Miss rate = 1/4 = 25%

```
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Miss rate = 100%

ScienceMark Membench				
	AMD Opteron 265	Intel Pentium D 830	Intel Petium M T2600	Dual Xeon 3.0GHz
Memory Bandwidth	4817.58 MB/s	4429.8 MB/s	3444.99 MB/s	4073.71 MB/s
L1 Cache Latency				
32 Bytes Stride	3 cycles/1.67ns	4 cycles/1.33ns	3 cycles/1.38ns	3 cycles/1.00ns
L2 Cache Latency				
4 Bytes Stride	3 cycles/1.67 ns	6 cycles/2.00 ns	3 cycles/1.38 ns	6 cycles/2.00 ns
16 Bytes Stride	5 cycles/2.79 ns	13 cycles/4.33 ns	5 cycles/2.31 ns	13 cycles/4.33 ns
64 Bytes Stride	17 cycles/9.47 ns	29 cycles/9.67 ns	14 cycles/6.46 ns	27 cycles/9.00 ns
256 Bytes Stride	12 cycles/6.69 ns	28 cycles/9.33 ns	14 cycles/6.46 ns	26 cycles/8.67 ns
512 Bytes Stride	13 cycles/7.24 ns	26 cycles/8.67 ns	14 cycles/6.46 ns	25 cycles/8.33 ns
Memory Latency				
4 Bytes Stride	3 cycles/1.67 ns	7 cycles/2.33 ns	4 cycles/1.85 ns	6 cycles/ 2.00 ns
16 Bytes Stride	12 cycles/6.69 ns	15 cycles/5.00 ns	13 cycles/6.00 ns	15 cycles/5.00 ns
64 Bytes Stride	48 cycles/26.75 ns	43 cycles/14.33 ns	53 cycles/24.46 ns	49 cycles/16.33 ns
256 Bytes Stride	103 cycles/57.40 ns	270 cycles/90.00 ns	202 cycles/93.22 ns	376 cycles/125.33 ns
512 Bytes Stride	106 cycles/59.07 ns	284 cycles/94.66 ns	205 cycles/94.61 ns	395 cycles/131.66 ns

Matrix Multiplication Example

- Major Cache Effects to Consider
  - Total cache size
    - Exploit temporal locality and keep the working set small (e.g., by using blocking)
  - Block size
    - Exploit spatial locality

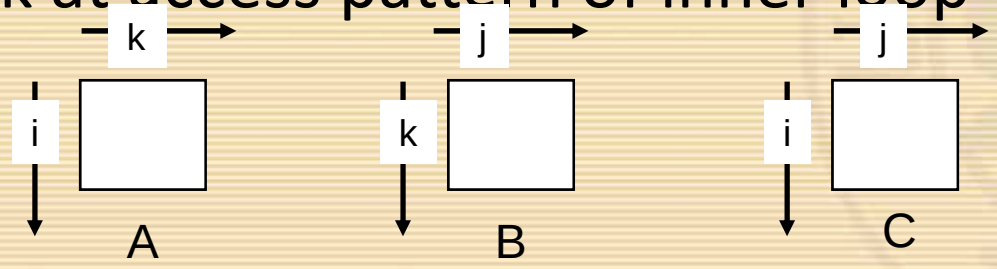
```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Variable sum held in register

- Description:
  - Multiply N x N matrices
  - O(N3) total operations
  - Accesses
    - N reads per source element
    - N values summed per destination
      - but may be able to hold in register

Miss Rate Analysis for Matrix Multiply

- Assume:
  - Line size = 32B (big enough for 4 64-bit words)
  - Matrix dimension (N) is very large
    - Approximate 1/N as 0.0
  - Cache is not even big enough to hold multiple rows
- Analysis Method:
  - Look at access pattern of inner loop

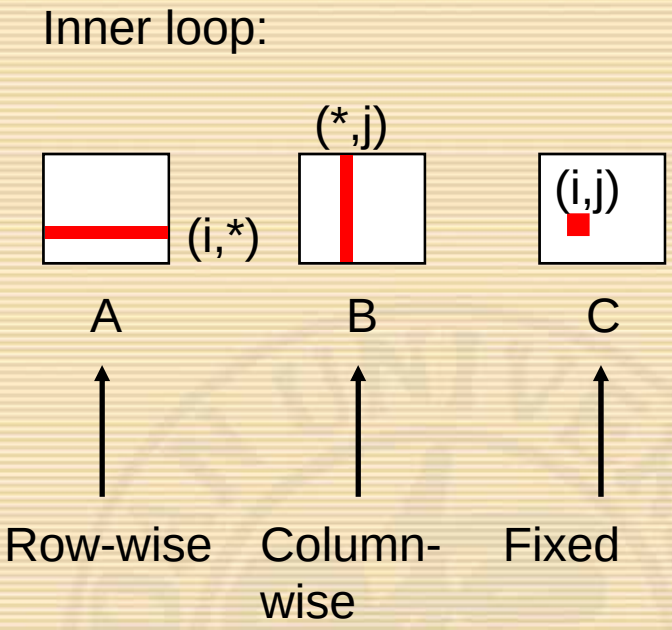


Layout of C Arrays in Memory (review)

- C arrays allocated in row-major order
  - each row in contiguous memory locations
- Stepping through columns in one row:
  - `for (i = 0; i < N; i++)`
    `sum += a[0][i];`
  - accesses successive elements
  - if block size (B) > 4 bytes, exploit spatial locality
    - compulsory miss rate = 4 bytes / B
- Stepping through rows in one column:
  - `for (i = 0; i < n; i++)`
    `sum += a[i][0];`
  - accesses distant elements
  - no spatial locality!
    - compulsory miss rate = 1 (i.e. 100%)

Matrix Multiplication (ijk)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

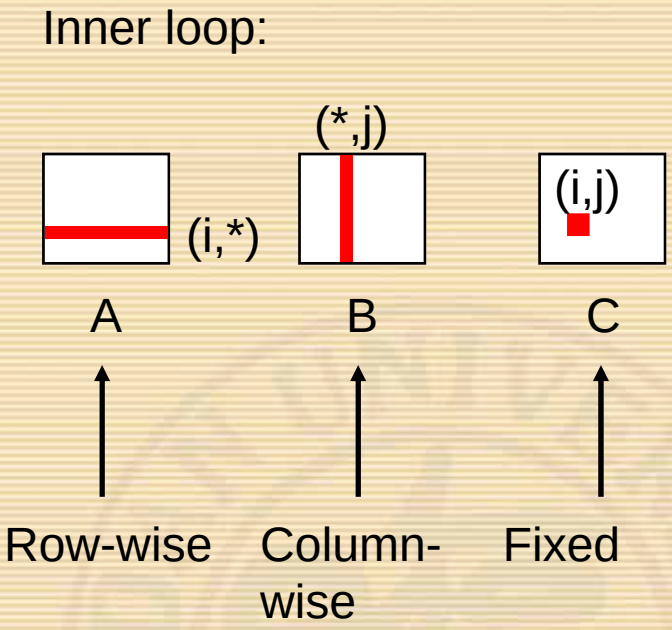


- Misses per Inner Loop Iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (jik)

```
/* jik */
for (j=0; j<n; j++)  {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

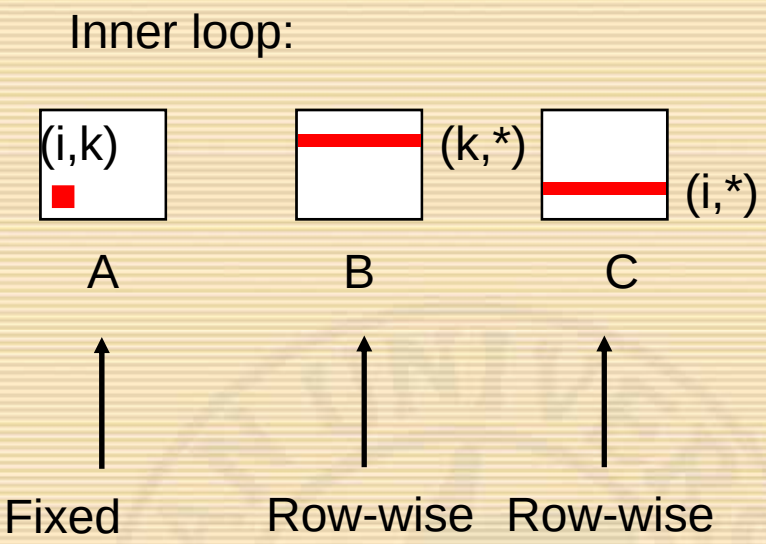


- Misses per Inner Loop Iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (kij)

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

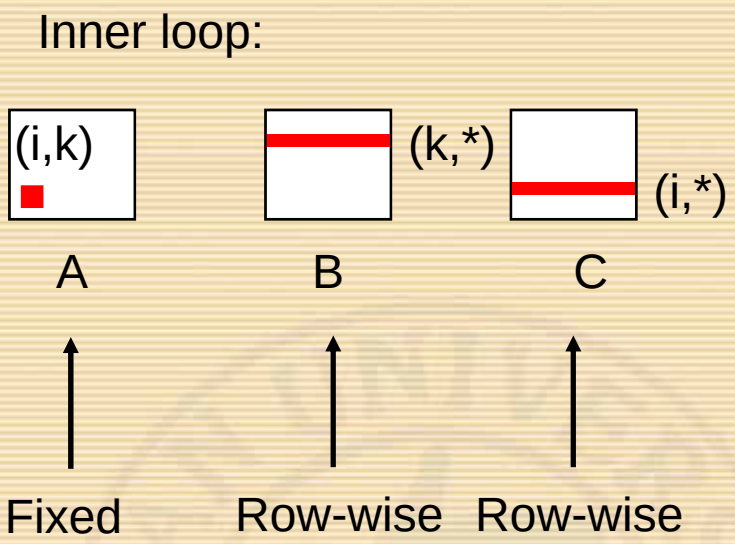


- Misses per Inner Loop Iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (ikj)

```
/* ikj */
for (i=0; i<n; i++) {
  for (k=0; k<n; k++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

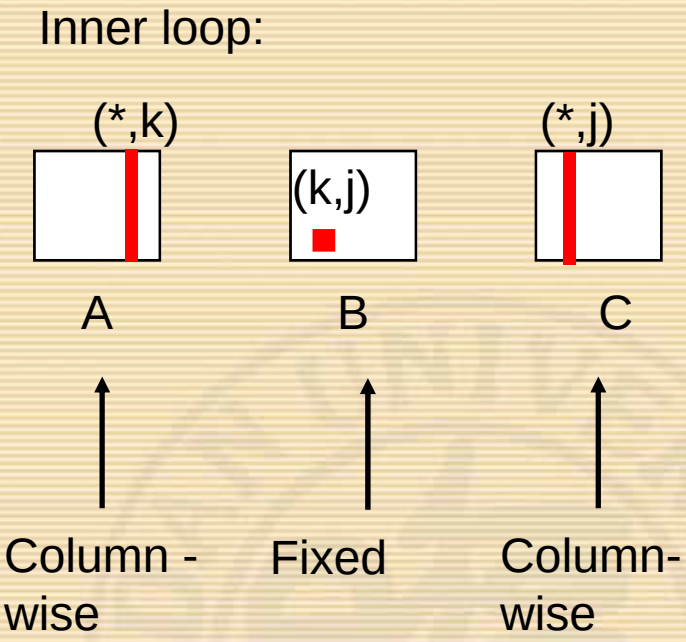


- Misses per Inner Loop Iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (jki)

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

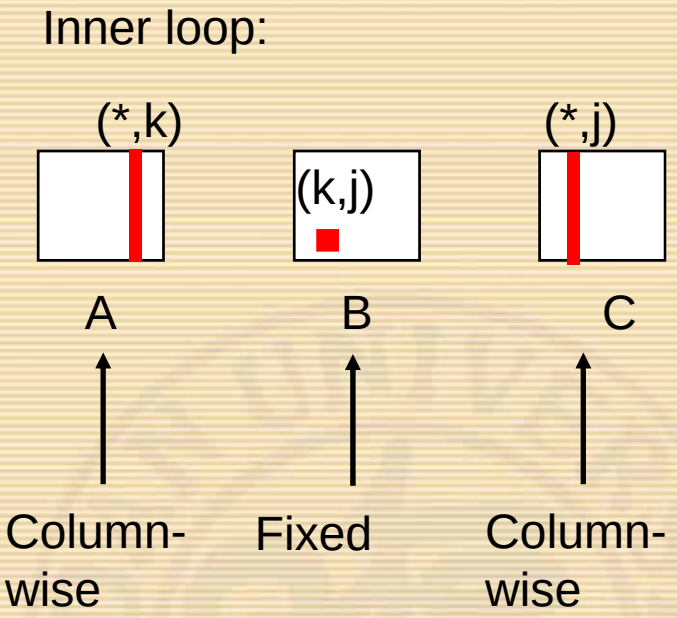


- Misses per Inner Loop Iteration:

A	B	C
1.0	0.0	1.0

Matrix Multiplication (kji)

```
/* kji */
for (k=0; k<n; k++) {
  for (j=0; j<n; j++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```



- Misses per Inner Loop Iteration:

A	B	C
1.0	0.0	1.0

Summary of Matrix Multiplication

ijk (& jik):
- 2 loads, 0 stores
- misses/iter = 1.25

```
for (i=0; i<n; i++)  {
   for (j=0; j<n; j++) {
      sum = 0.0;
      for (k=0; k<n; k++)
         sum += a[i][k] * b[k][j];
      c[i][j] = sum;
   }
}
```

kij (& ikj):
- 2 loads, 1 store
- misses/iter = 0.5

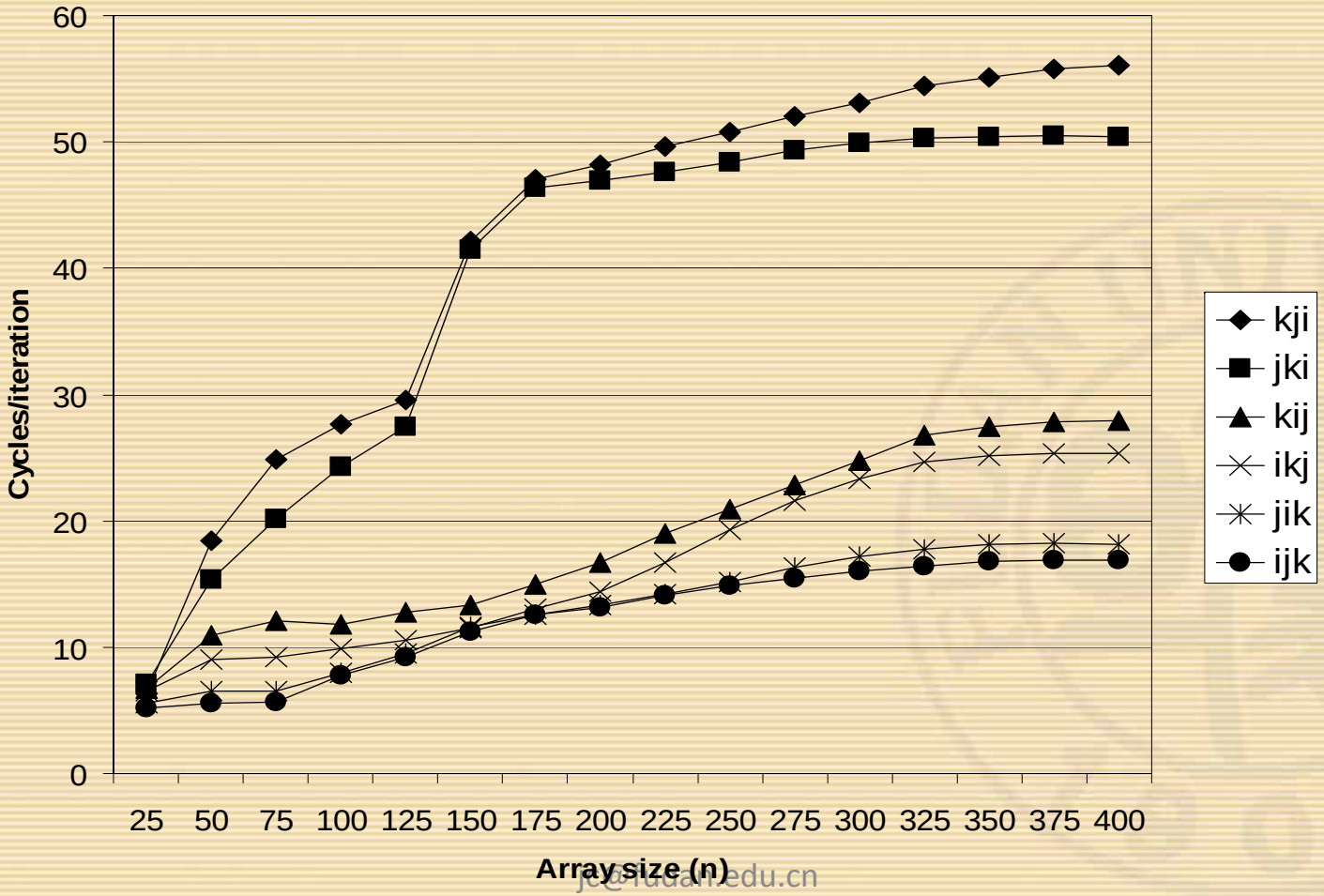
```
for (k=0; k<n; k++) {
   for (i=0; i<n; i++) {
      r = a[i][k];
      for (j=0; j<n; j++)
         c[i][j] += r * b[k][j];
   }
}
```

jki (& kji):
- 2 loads, 1 store
- misses/iter = 2.0

```
for (j=0; j<n; j++) {
   for (k=0; k<n; k++) {
      r = b[k][j];
      for (i=0; i<n; i++)
         c[i][j] += a[i][k] * r;
   }
}
```

Pentium Matrix Multiply Performance

- Miss rates are helpful but not perfect predictors.
  - Code scheduling matters, too.



Improving Temporal Locality by Blocking

- Example: Blocked matrix multiplication
  - "block" (in this context) does not mean "cache block".
  - Instead, it mean a sub-block within the matrix.
  - Example: N = 8; sub-block size = 4

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Key idea: Sub-blocks (i.e., A_{xy}) can be treated just like scalars.

$C_{11} = A_{11}B_{11} + A_{12}B_{21}$ $\quad$ $C_{12} = A_{11}B_{12} + A_{12}B_{22}$

$C_{21} = A_{21}B_{11} + A_{22}B_{21}$ $\quad$ $C_{22} = A_{21}B_{12} + A_{22}B_{22}$

Blocked Matrix Multiply (bijk)

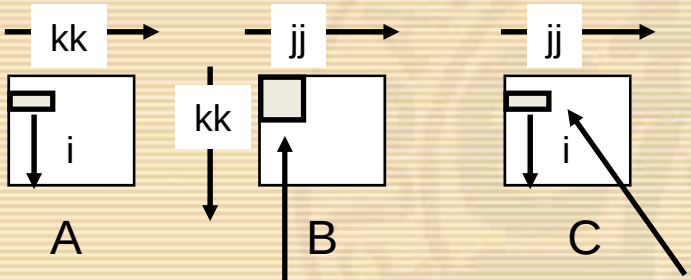
```
for (jj=0; jj<n; jj+=bsize) {
  for (i=0; i<n; i++)
    for (j=jj; j < min(jj+bsize,n); j++)
      c[i][j] = 0.0;
  for (kk=0; kk<n; kk+=bsize) {
    for (i=0; i<n; i++) {
      for (j=jj; j < min(jj+bsize,n); j++) {
        sum = 0.0
        for (k=kk; k < min(kk+bsize,n); k++) {
          sum += a[i][k] * b[k][j];
        }
        c[i][j] += sum;
      }
    }
  }
}
```

Blocked Matrix Multiply Analysis

- Innermost loop pair multiplies a *1 X bsize* sliver of *A* by a *bsize X bsize* block of *B* and accumulates into *1 X bsize* sliver of *C*
- Loop over *i* steps through *n* row slivers of *A* & *C*, using same *B*

```
for (i=0; i<n; i++) {
    for (j=jj; j < min(jj+bsize,n); j++) {
      sum = 0.0
      for (k=kk; k < min(kk+bsize,n); k++) {
        sum += a[i][k] * b[k][j];
      }
      c[i][j] += sum;
    }
```

Innermost Loop Pair



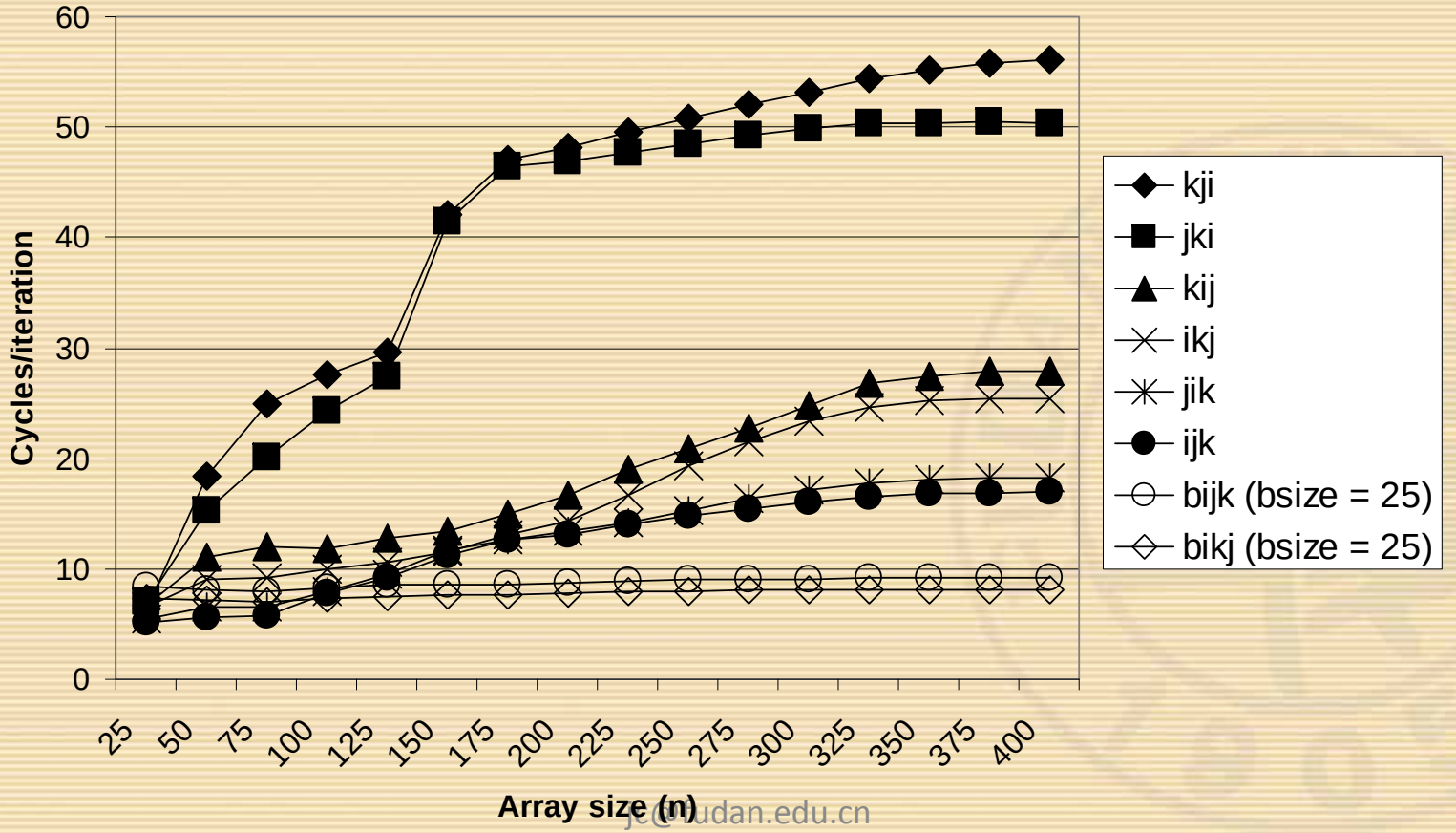
row sliver accessed *bsize* times

block reused *n* times in succession

Update successive elements of sliver

Blocked Matrix Multiply Performance

- Blocking (bijk and bikj) improves performance by a factor of two over unblocked versions (ijk and jik)
  - relatively insensitive to array size.



Concluding Observations

- Programmer can optimize for cache performance
  - How data structures are organized
  - How data are accessed
    - Nested loop structure
    - Blocking is a general technique
- All systems favor “cache friendly code”
  - Getting absolute optimum performance is very platform specific
    - Cache sizes, line sizes, associativities, etc.
  - Can get most of the advantage with generic code
    - Keep working set reasonably small (temporal locality)
    - Use small strides (spatial locality)

Virtual Memory

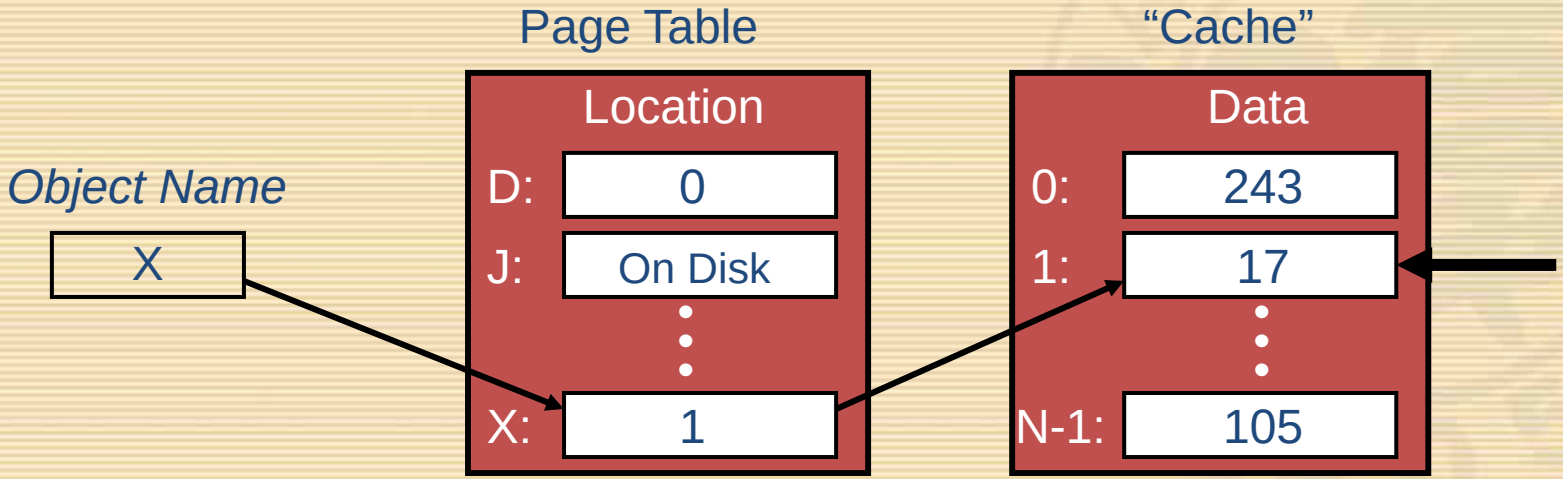
jc@fudan.edu.cn

Motivations for Virtual Memory

- Use Physical DRAM as a Cache for the Disk
  - Address space of a process can exceed physical memory size
  - Sum of address spaces of multiple processes can exceed physical memory
- Simplify Memory Management
  - Multiple processes resident in main memory.
    - Each process with its own address space
  - Only “active” code and data is actually in memory
    - Allocate more memory to process as needed.
- Provide Protection
  - One process can’t interfere with another.
    - because they operate in different address spaces.
  - User process cannot access privileged information
    - different sections of address spaces have different permissions.

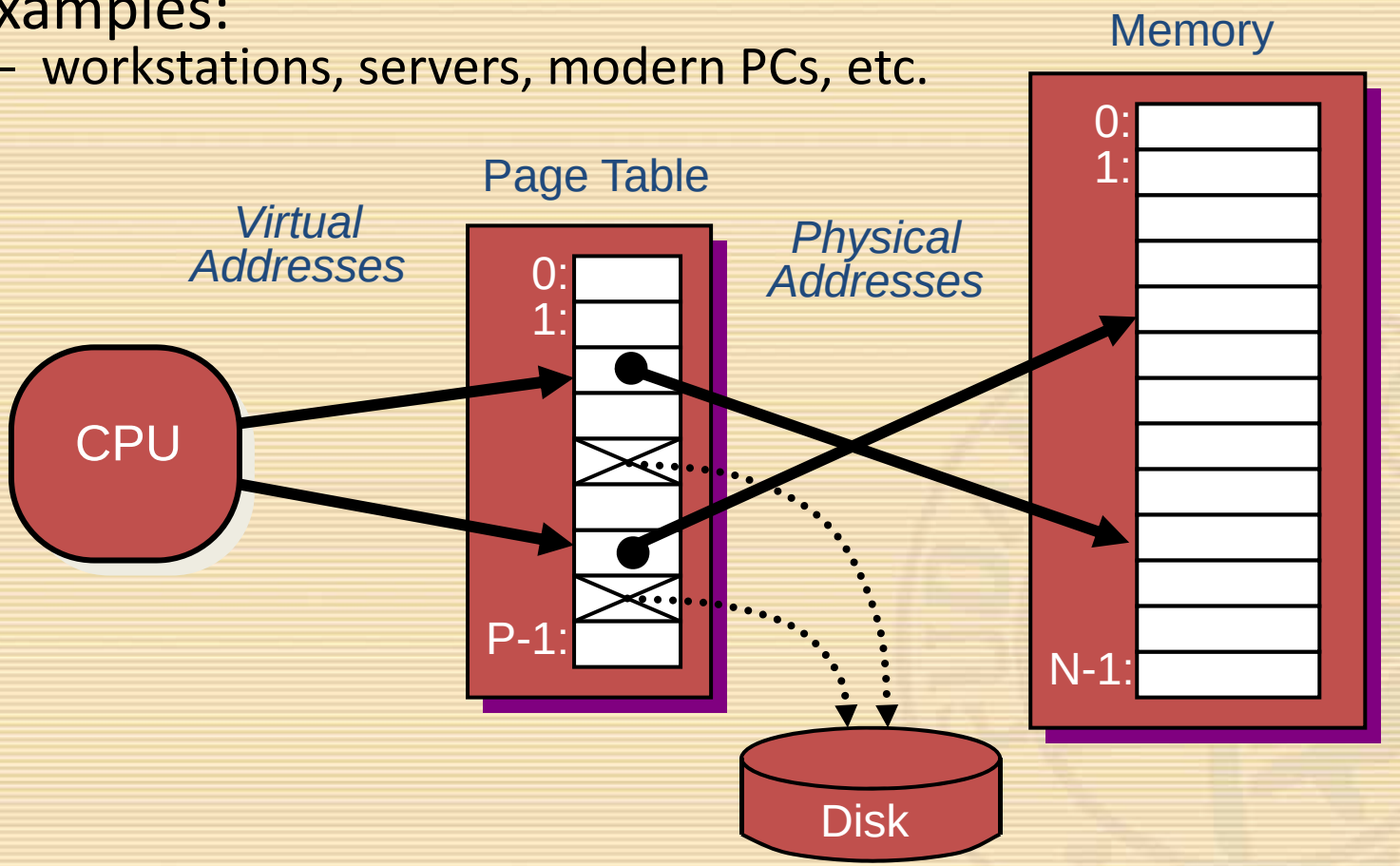
Locating an Object in Cache

- DRAM Cache
  - Each allocated page of virtual memory has entry in *page table*
  - Mapping from virtual pages to physical pages
    - From uncached form to cached form
  - Page table entry even if page not in memory
    - Specifies disk address
    - Only way to indicate where to find page
  - OS retrieves information



A System with Virtual Memory

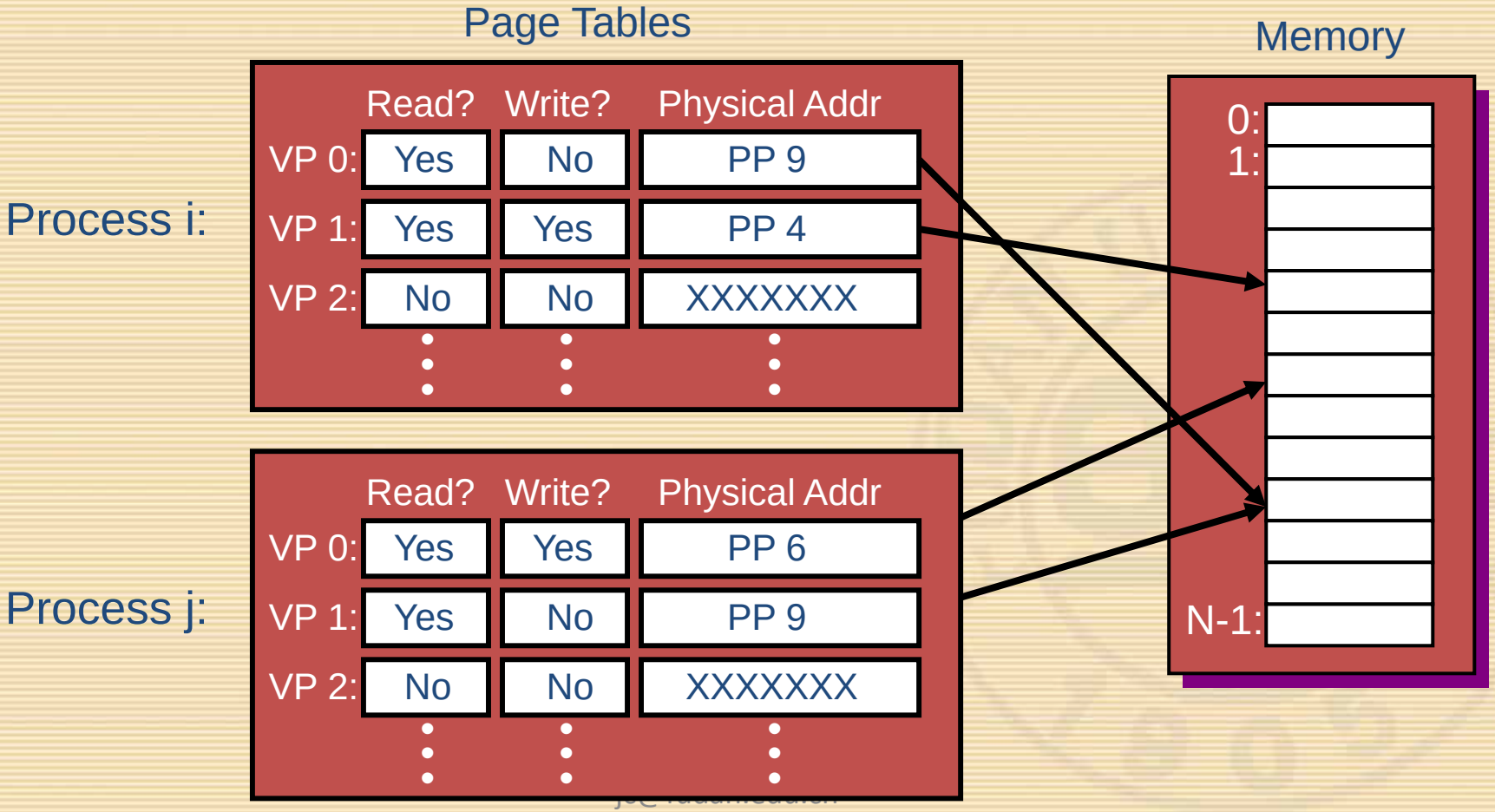
- Examples:
  - workstations, servers, modern PCs, etc.



- Address Translation: Hardware converts virtual addresses to physical addresses via OS-managed lookup table (page table)

Protection

- Page table entry contains access rights information
  - hardware enforces this protection (trap into OS if violation occurs)

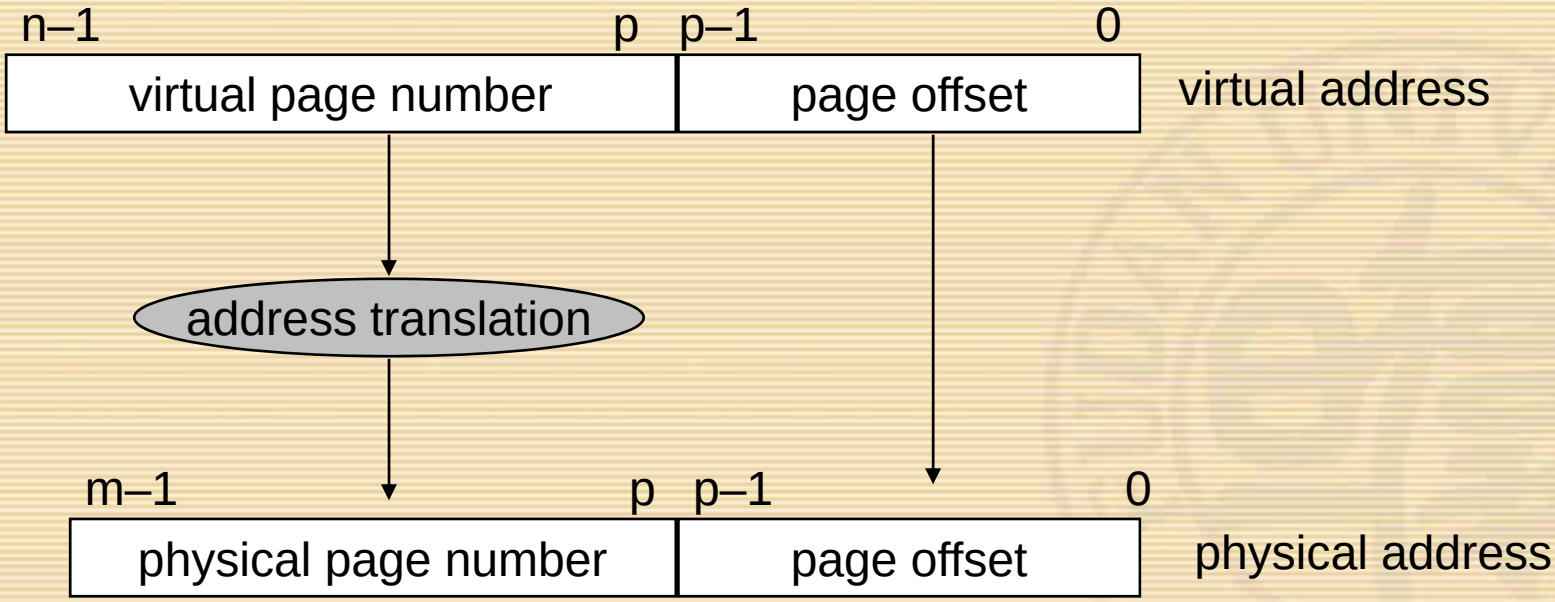


VM Address Translation

- Virtual Address Space
  - V = {0, 1, …, N–1}
- Physical Address Space
  - P = {0, 1, …, M–1}
  - M < N
- Address Translation
  - MAP: V $\rightarrow$ P U {$\varnothing$}
  - For virtual address a:
    - MAP(a) = a' if data at virtual address a at physical address a' in P
    - MAP(a) = $\varnothing$ if data at virtual address a not in physical memory
      - Either invalid or stored on disk

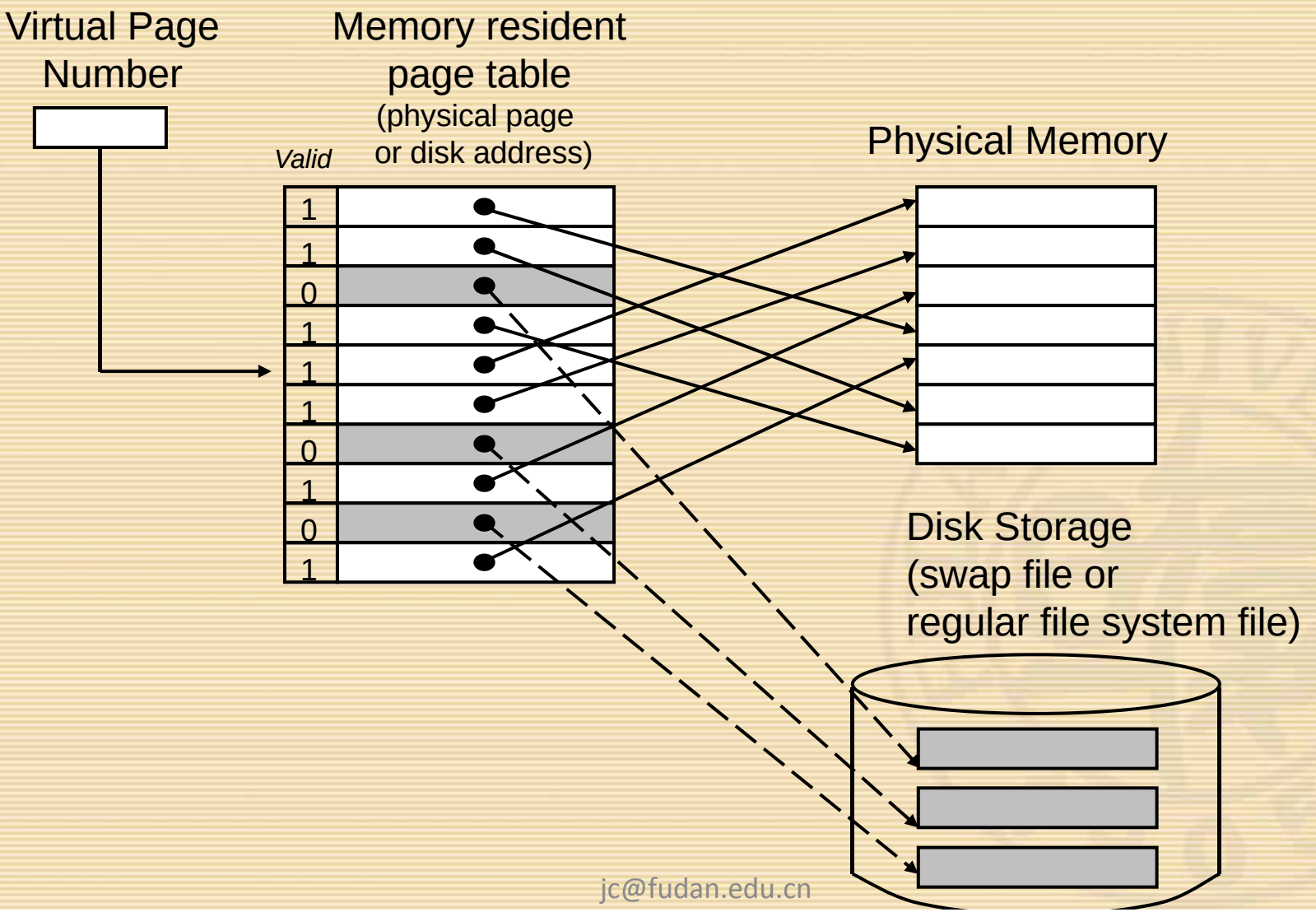
VM Address Translation

- Parameters
  - $P = 2^p$ = page size (bytes).
  - $N = 2^n$ = Virtual address limit
  - $M = 2^m$ = Physical address limit

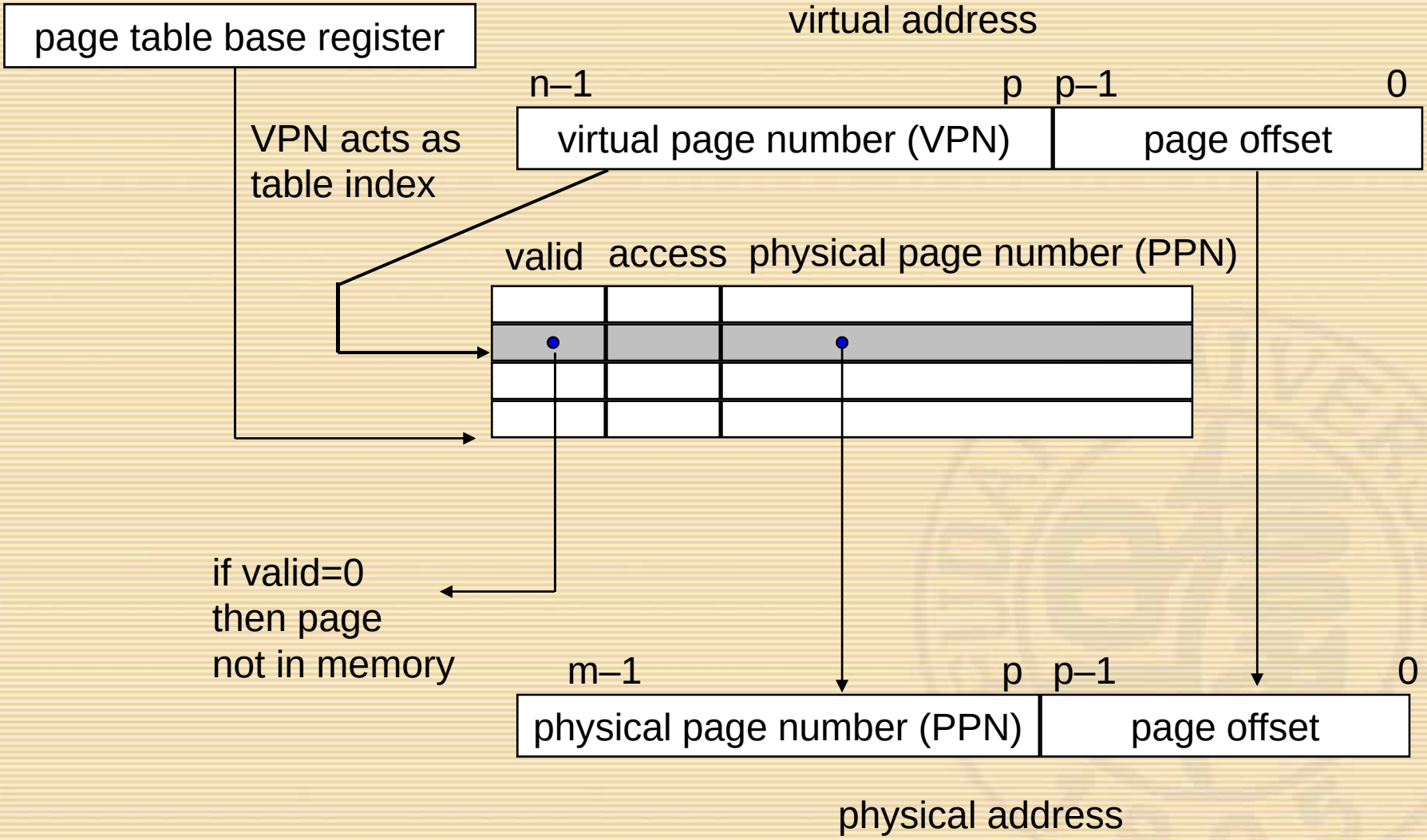


Page offset bits don't change as a result of translation

Page Tables

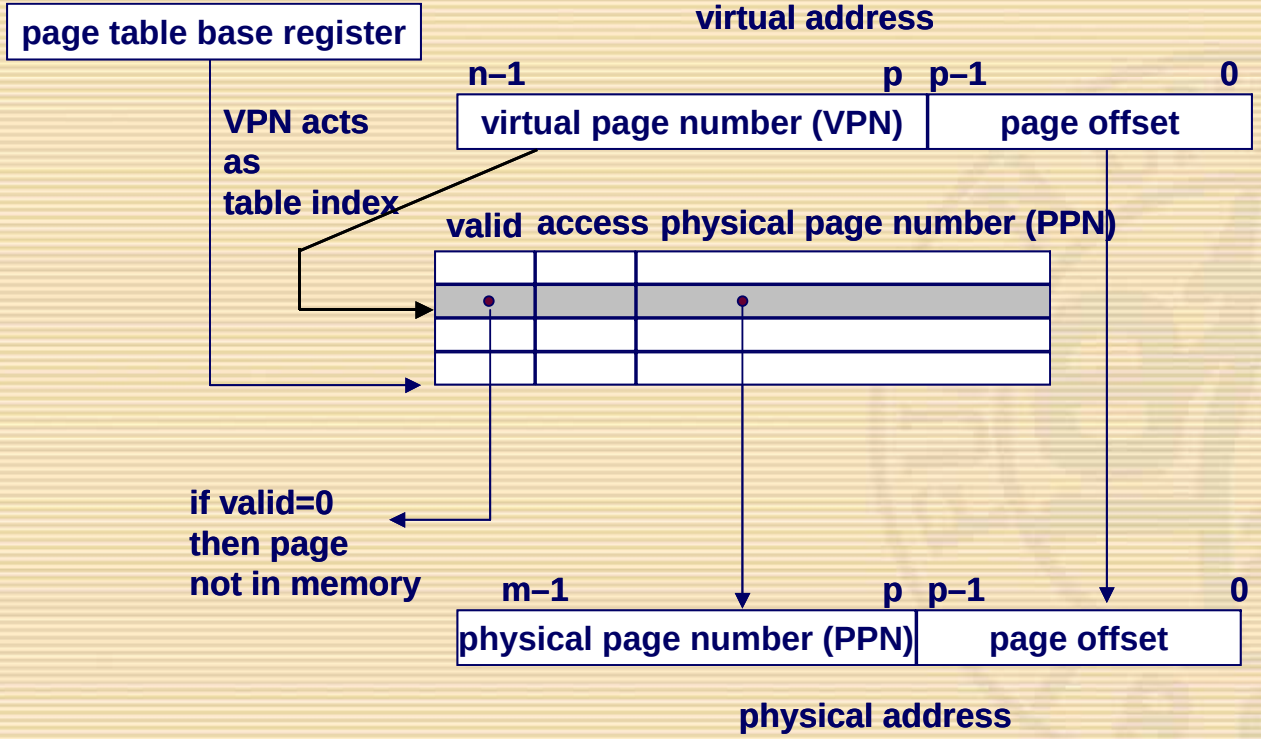

Virtual Page Number
Memory resident page table
(physical page or disk address)
Valid
1
1
0
1
1
1
0
1
0
1
Physical Memory
Disk Storage
(swap file or
regular file system file)

Address Translation via Page Table

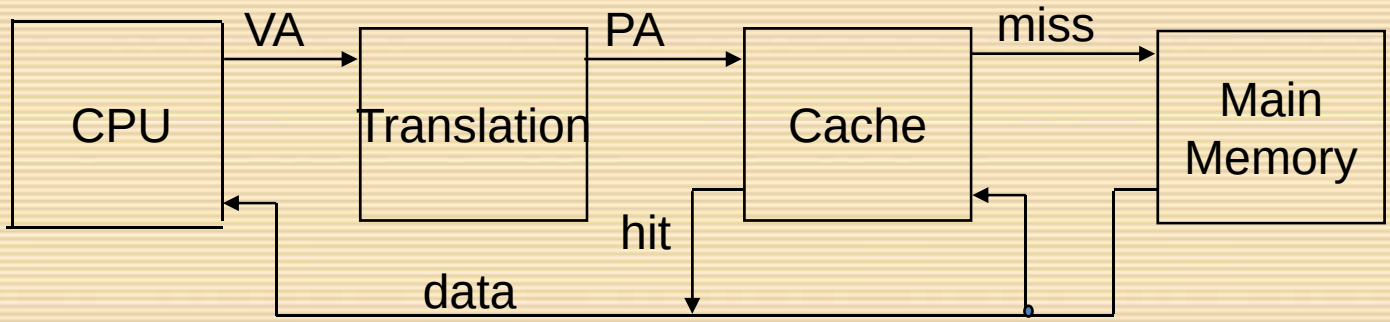


Page Table Operation

- Translation
  - Separate (set of) page table(s) per process
  - VPN forms index into page table (points to a page table entry)



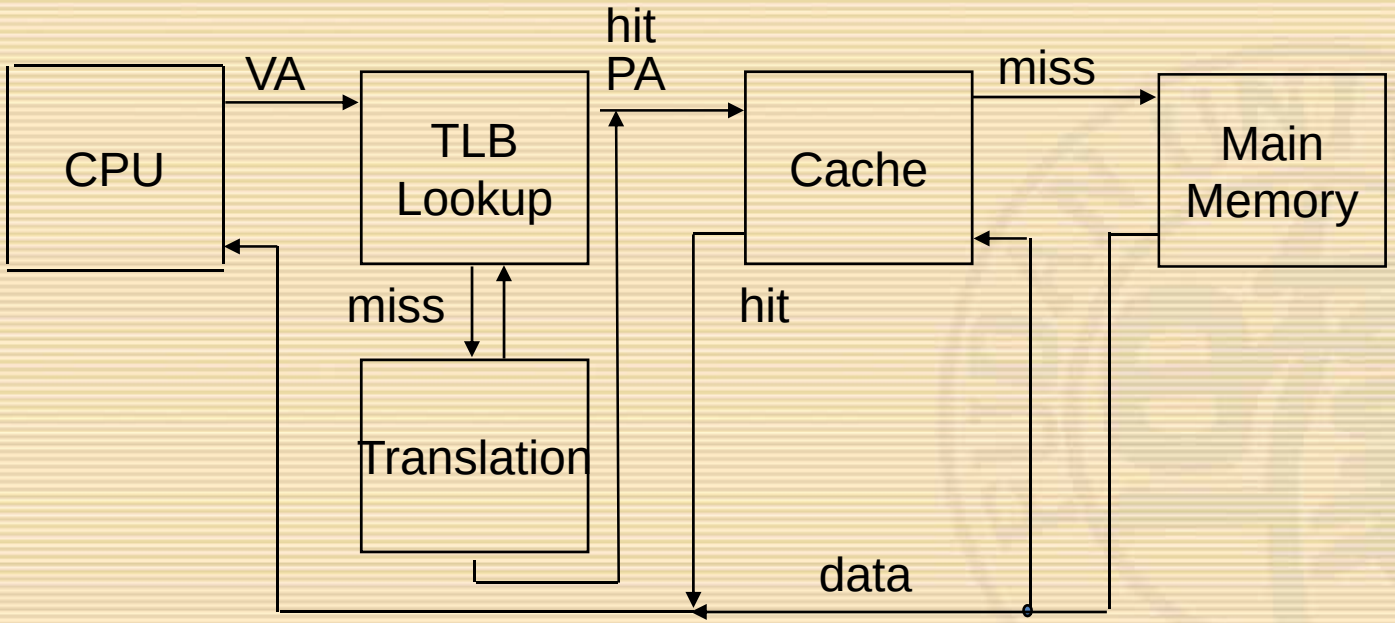
Integrating VM and Cache



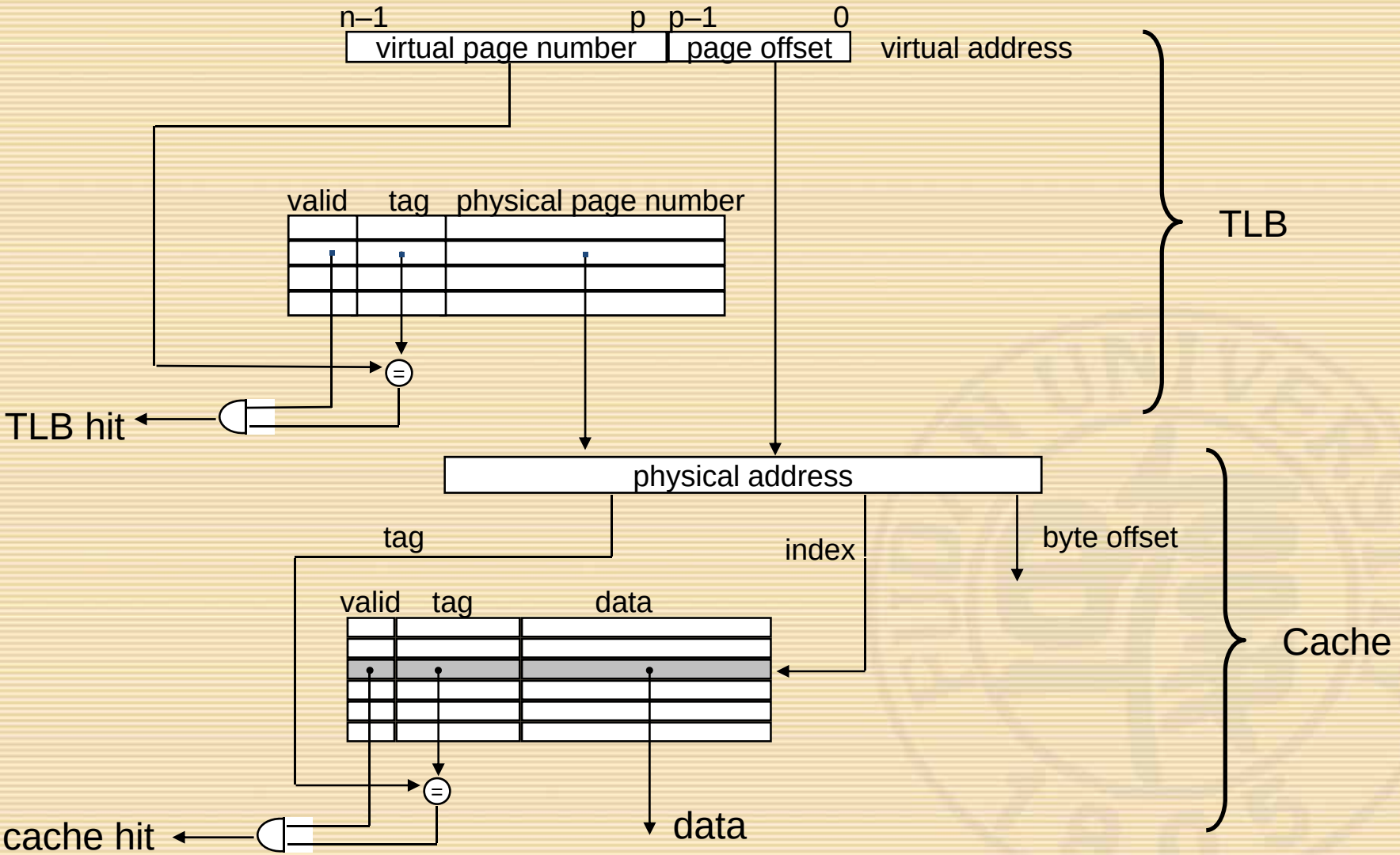
- Most Caches "Physically Addressed"
  - Accessed by physical addresses
  - Allows multiple processes to have blocks in cache at same time
  - Allows multiple processes to share pages
  - Cache doesn't need to be concerned with protection issues
    - Access rights checked as part of address translation
- Perform Address Translation Before Cache Lookup
  - But this could involve a memory access itself (of the PTE)
  - Of course, page table entries can also become cached

Speeding up Translation with a TLB

- "Translation Lookaside Buffer" (TLB)
  - Small hardware cache in MMU
  - Maps virtual page numbers to physical page numbers
  - Contains complete page table entries for small number of pages

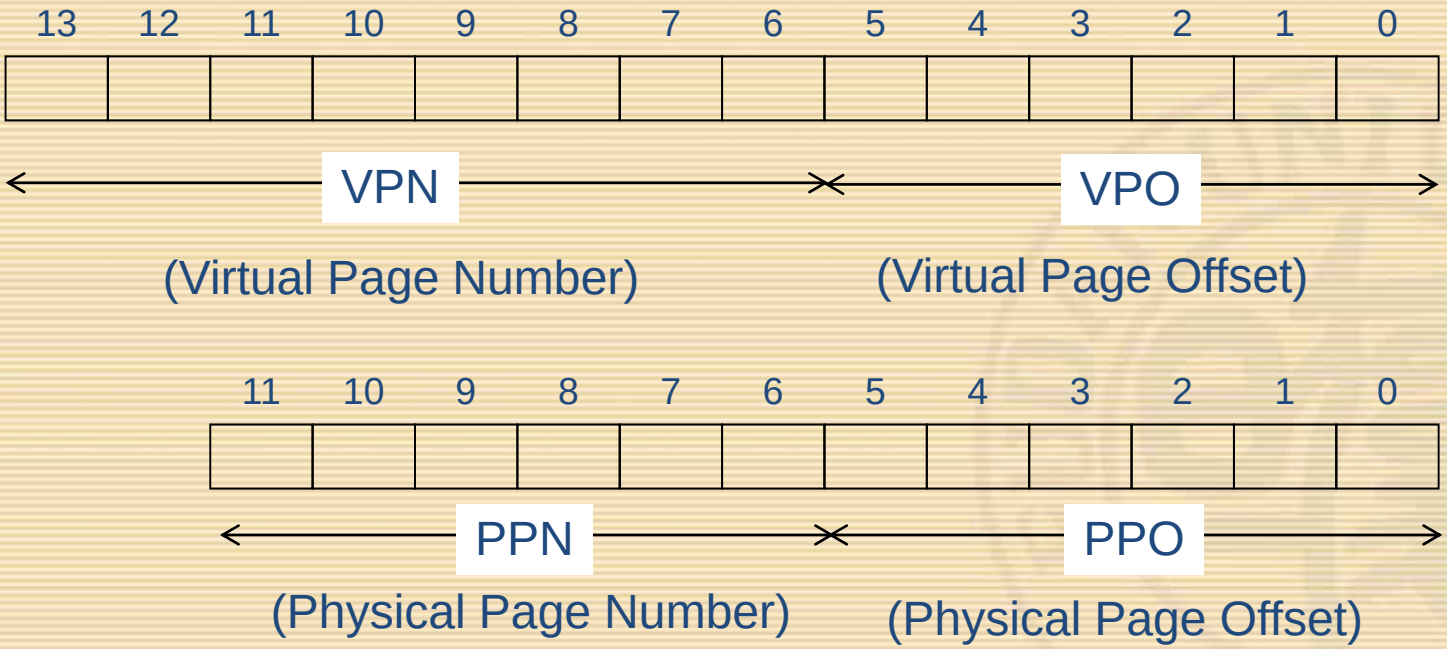


Address Translation with a TLB



Simple Memory System Example

- Addressing
  - 14-bit virtual addresses
  - 12-bit physical address
  - Page size = 64 bytes



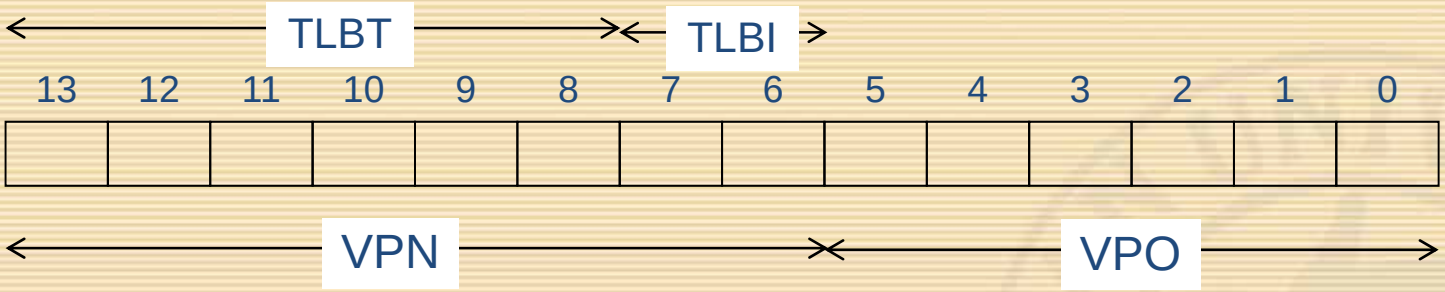
Simple Memory System Page Table

– Only show first 16 entries

VPN	PPN	Valid	VPN	PPN	Valid
00	28	1	08	13	1
01	–	0	09	17	1
02	33	1	0A	09	1
03	02	1	0B	–	0
04	–	0	0C	–	0
05	16	1	0D	2D	1
06	–	0	0E	11	1
07	–	0	0F	0D	1

Simple Memory System TLB

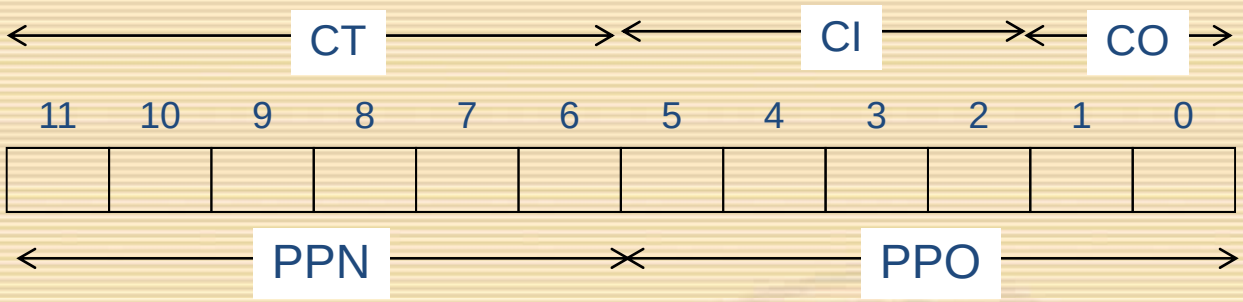
- TLB
  - 16 entries
  - 4-way associative



Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Simple Memory System Cache

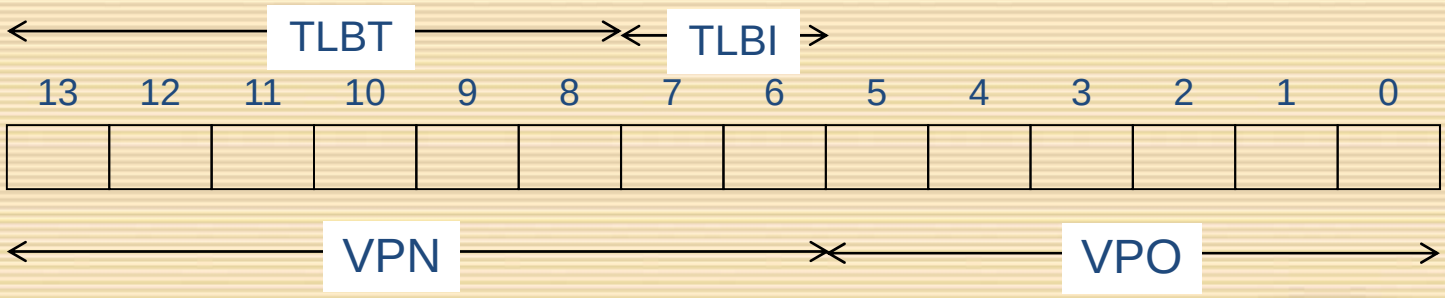
- Cache
  - 16 lines
  - 4-byte line size
  - Direct mapped



Idx	Tag	Valid	B0	B1	B2	B3	Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11	8	24	1	3A	00	51	89
1	15	0	–	–	–	–	9	2D	0	–	–	–	–
2	1B	1	00	02	04	08	A	2D	1	93	15	DA	3B
3	36	0	–	–	–	–	B	0B	0	–	–	–	–
4	32	1	43	6D	8F	09	C	12	0	–	–	–	–
5	0D	1	36	72	F0	1D	D	16	1	04	96	34	15
6	31	0	–	–	–	–	E	13	1	83	77	1B	D3
7	16	1	11	C2	DF	03	F	14	0	–	–	–	–

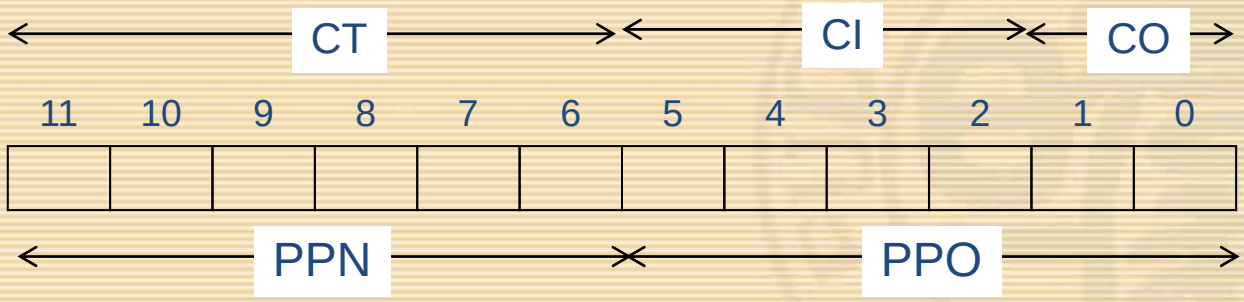
Address Translation Example #1

- Virtual Address `0x03D4`



VPN ___ TLBI ___ TLBT ____ TLB Hit? __ Page Fault? __ PPN: ____

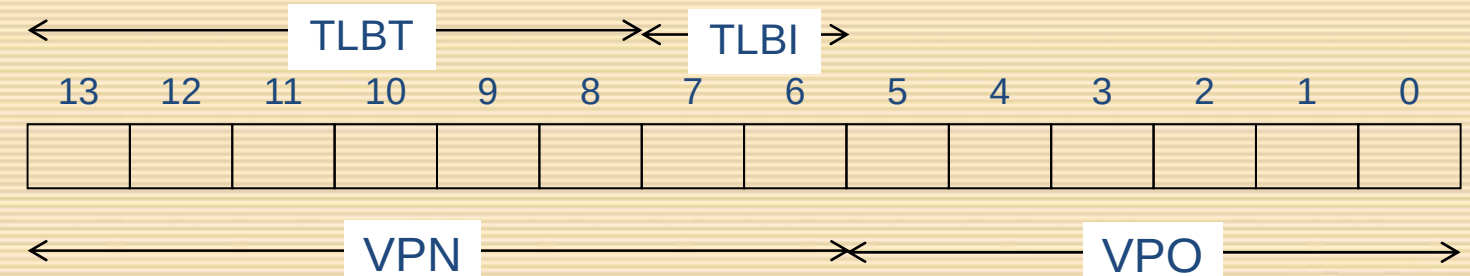
- Physical Address



Offset ___ CI___ CT ____ Hit? __ Byte: ____

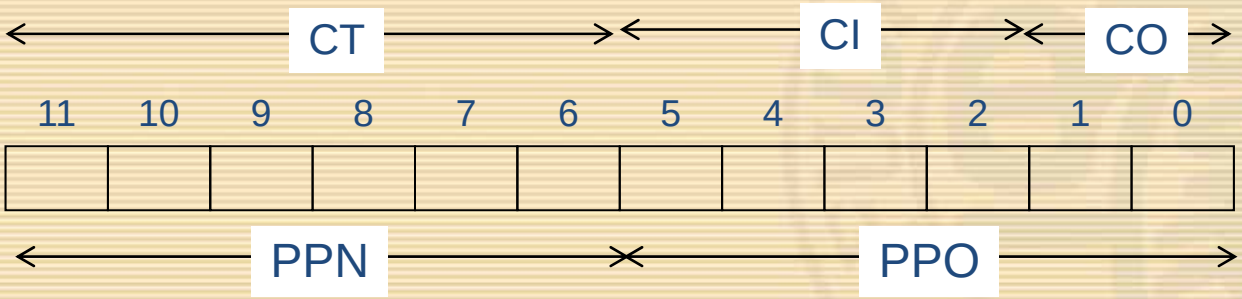
Address Translation Example #2

- Virtual Address `0x0B8F`



VPN ___ TLBI ___ TLBT ____ TLB Hit? __ Page Fault? __ PPN: ____

- Physical Address



Offset ___ CI___ CT ____ Hit? __ Byte: ____