System I

The Processor: Basic Principles

Haifeng Liu

Zhejiang University

Overview

- What is the architecture of computers?
- What is the inside of processor (CPU)?
- How to use CPU to solve problems?

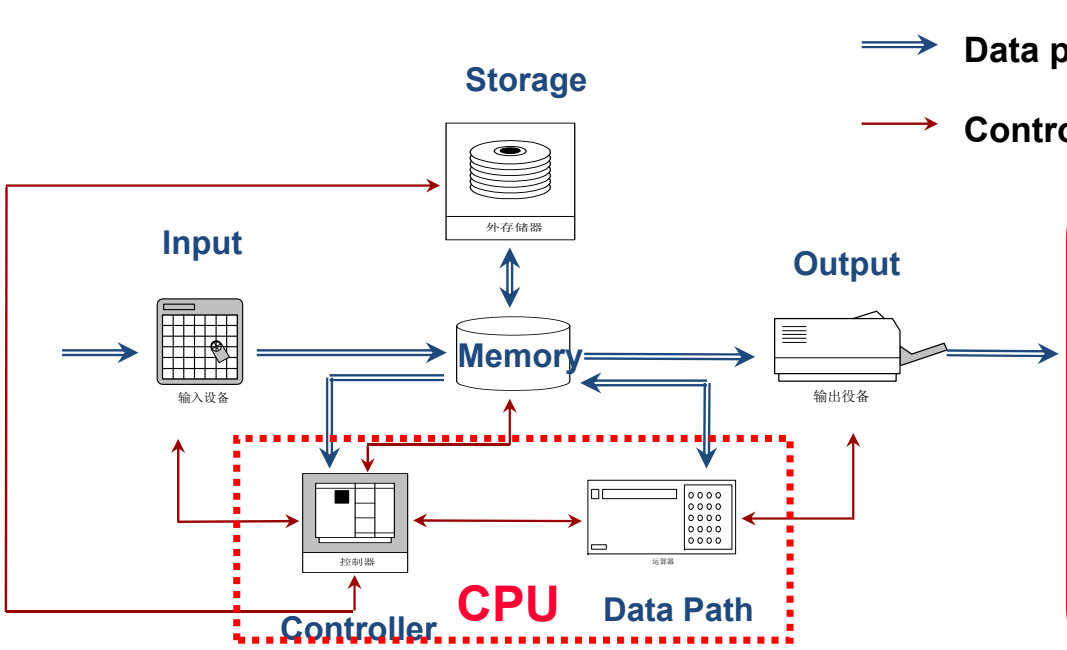
What is the architecture of computers?

浙江大学
NG UNIVERSITY

Computer Architecture!

problem
algorithm
program
runtime system
(VM, OS, MM)
ISA (architecture)
microarchitecture
logic
circuits
electrons

What is the architecture of computers?



Von Neumann Structure

- Computer
  - CPU
    - Control unit
    - Datapath
      - Path: multiplexors
      - ALU
      - Registers
      - ……
  - Memory
  - I/O interface

- Von Neumann structure: data and programs are in memory.
- CPU takes instructions and data from memory for operation and puts the results into memory.

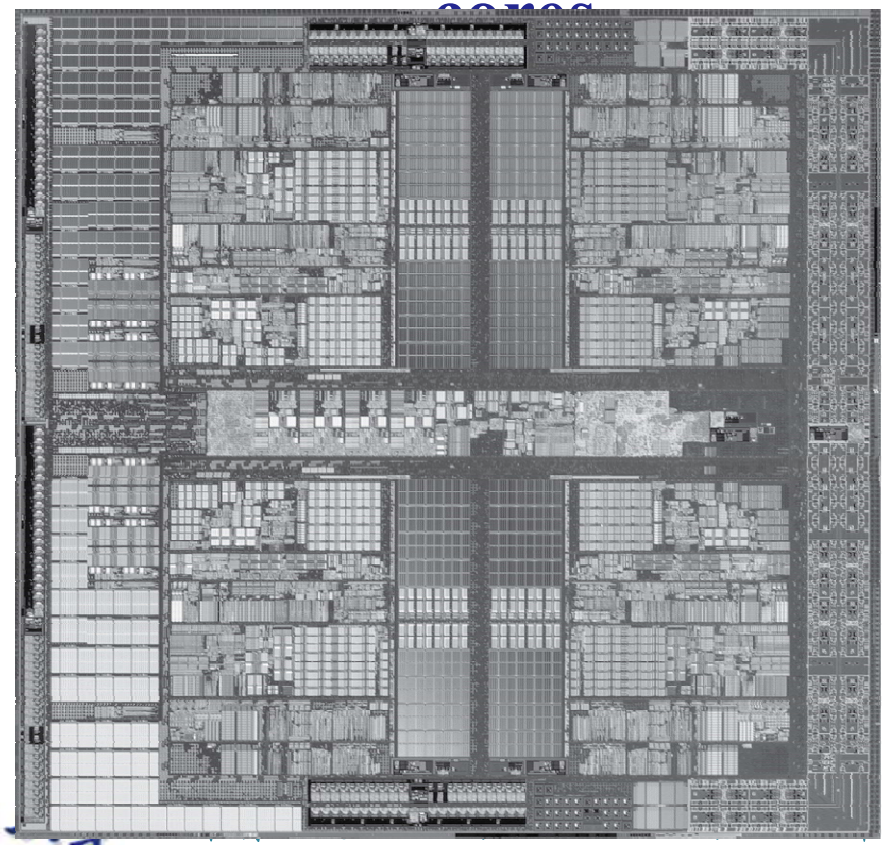
What is the inside of Processor ?

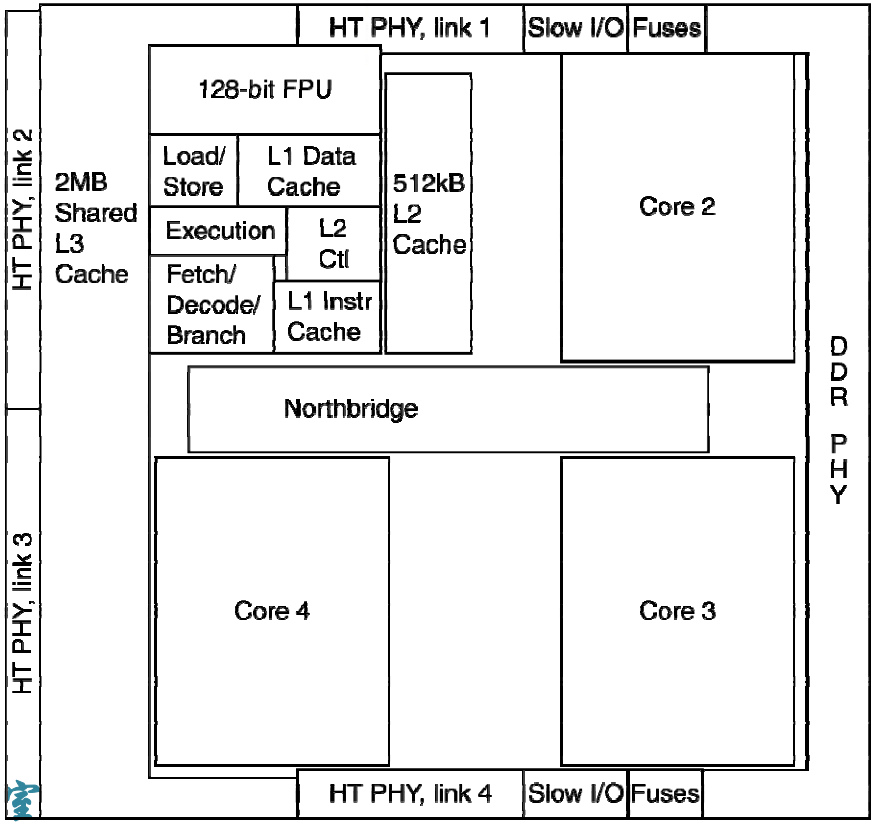
What is the inside of Processor (CPU)?

- **Datapath: performs operations on data**
- **Control: sequences datapath, memory, ...**
- **Cache memory**
  - Small fast SRAM memory for immediate access to data

What is the inside of Processor (CPU)?

AMD Barcelona: 4 processor

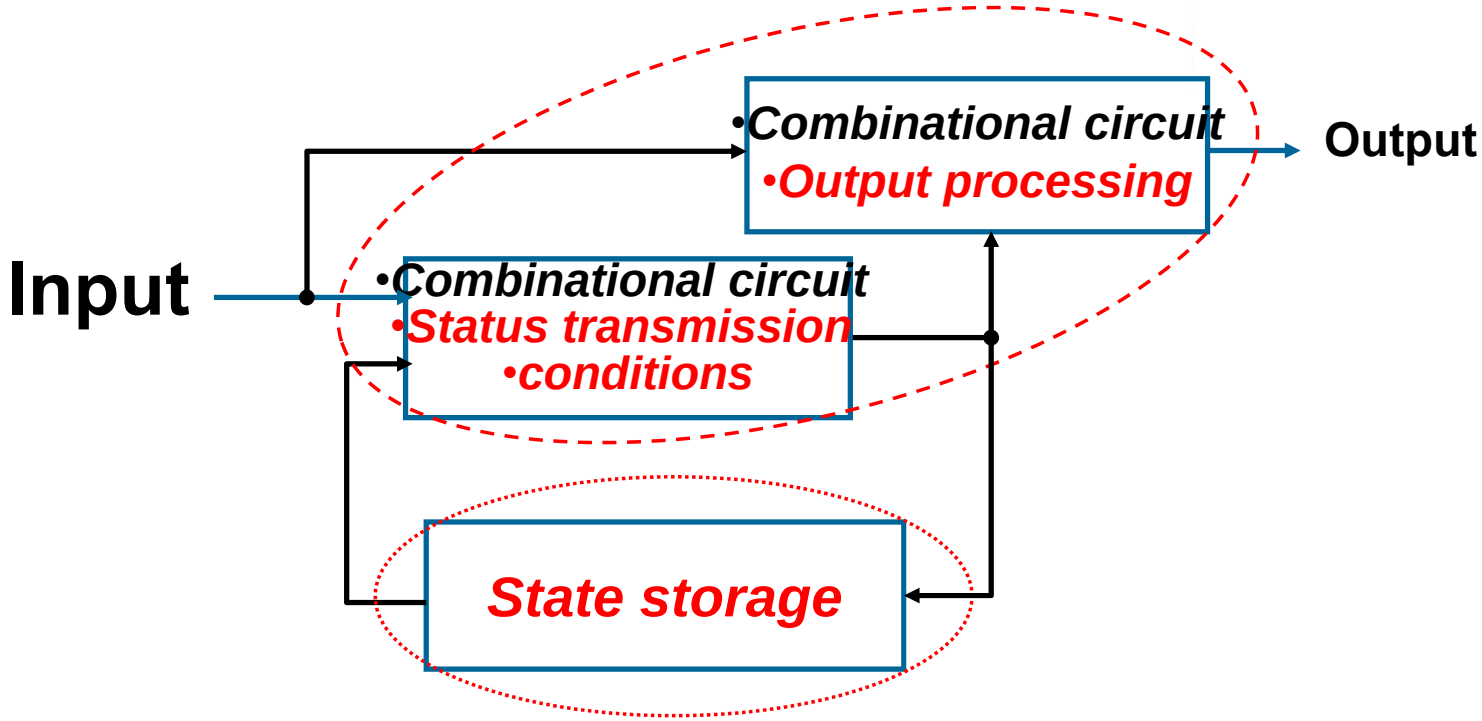




How to use CPU to solve problems?

Design methodology 1: Common digital system

◎Finite state machine（FSM）



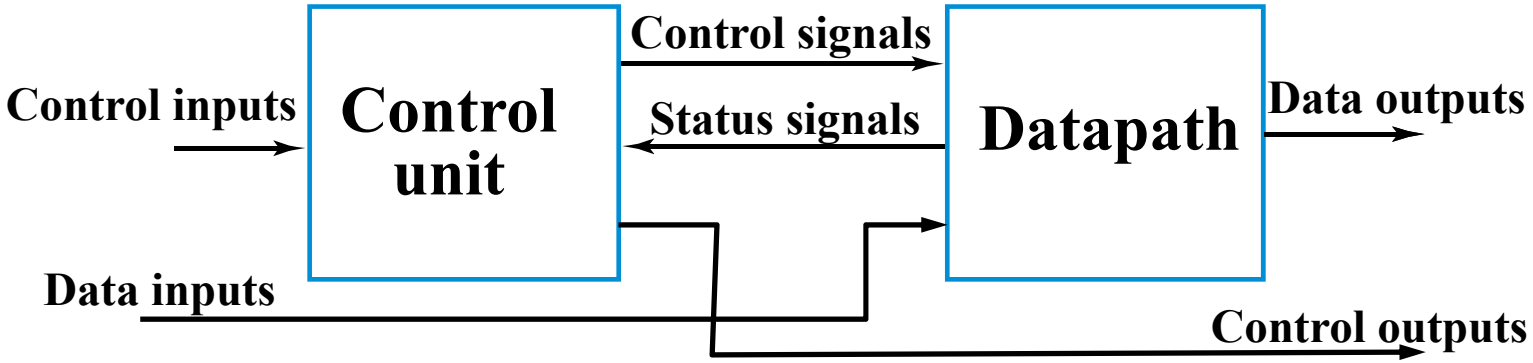
Design methodology 2: Control of Register Transfers

◎ Control of Register Transfers

- Register transfers performed on registers control that supervises the sequencing of the register transfers

◎ Three essential elements

- Set of registers: mostly in Datapath with some in Control Unit
- Basic operation (micromanipulation): Register transfers performed
- Control: that supervises the sequencing of the register transfers

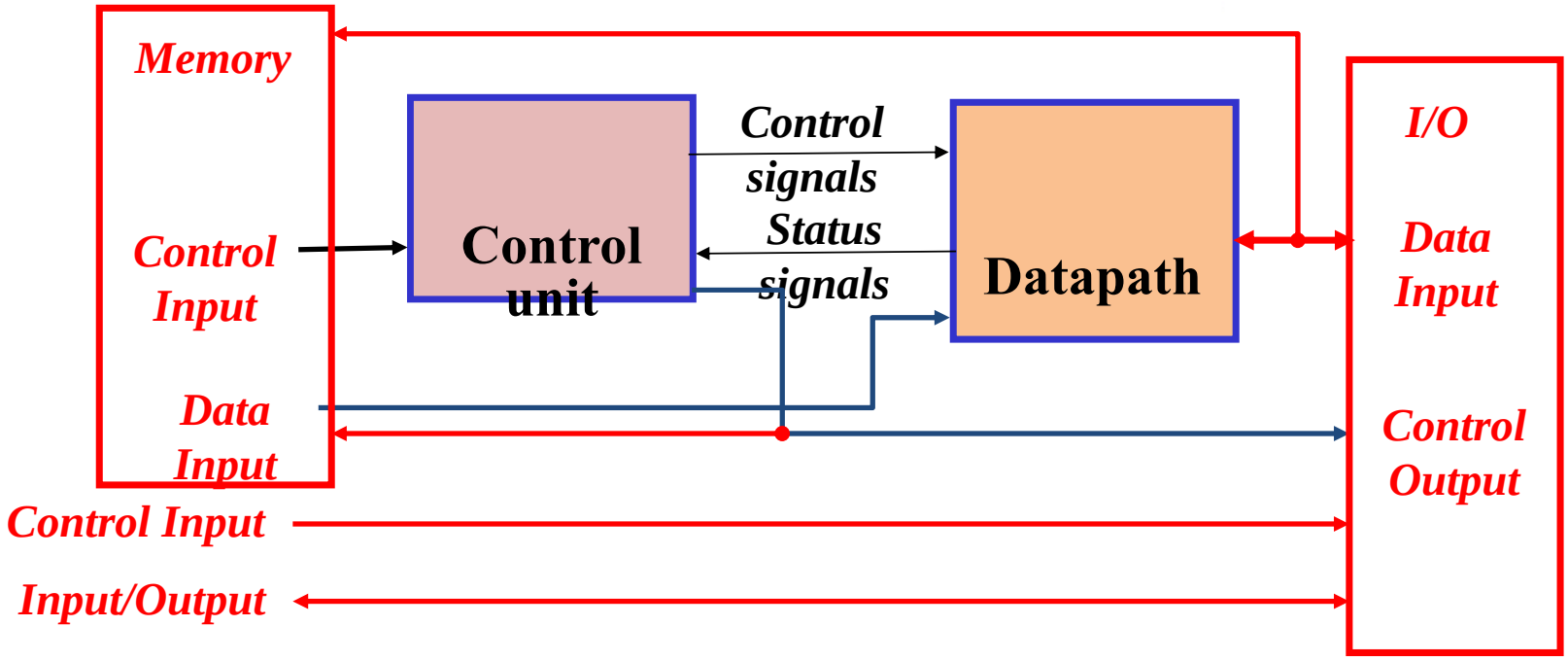


◎ Register Transfer Language: RTL

Design methodology 3: Control of Register Transfers

◎ General system

- Program state machine （PSM）



Specific and general system

◎ Non-programmable System: Specific System

- The control unit does not deal with fetching and executing instructions
- But contains all of the information for sequencing register transfers based on inputs and on status bits from the datapath

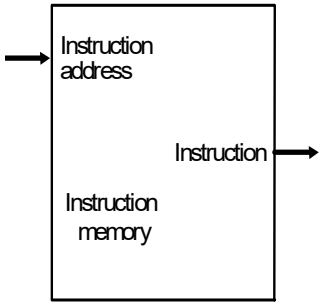
◎ Programmable System : General system

- A portion of the input consists of a sequence of *instructions* called a *program*,
- Typically stored in a memory and addressed by a *program counter*.
- The Control Unit is responsible for fetching and executing these instructions.

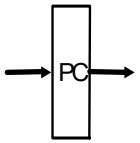
Implementation of CPU

- CPU
  - General digital system
  - A Turing Machine
    - Implemented by Register Transmission Control Technology
      - Datapath
        - The component of processor that performs arithmetic operations
      - Control
        - The component of processor that commands the datapath, memory, and I/O device according to the instructions of the program

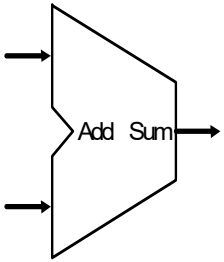
Simple Implementation



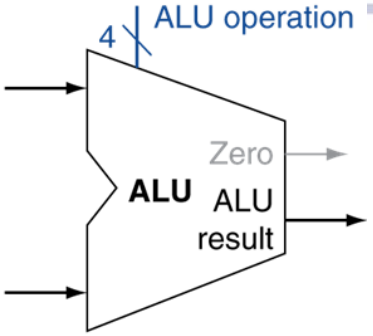
a. Instruction memory

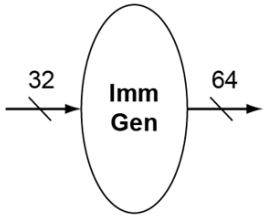


b. Program counter

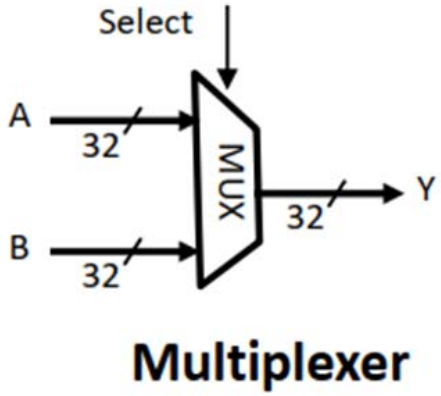


c. Adder

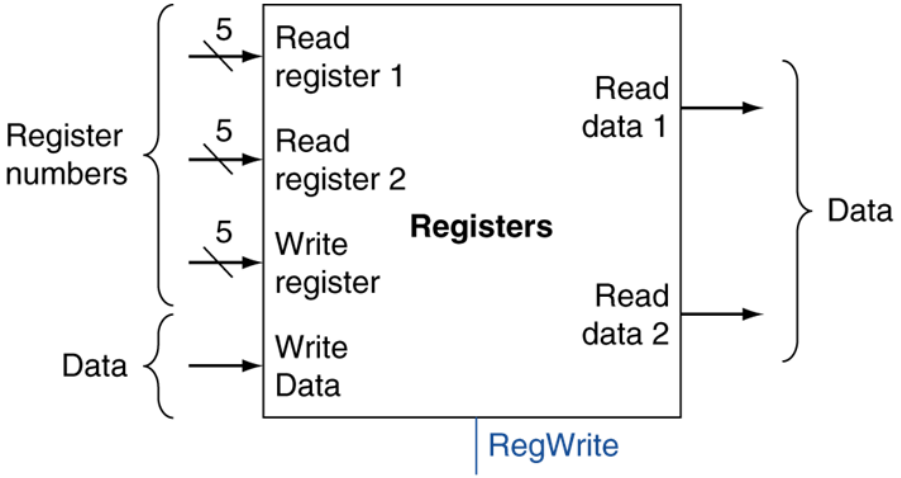




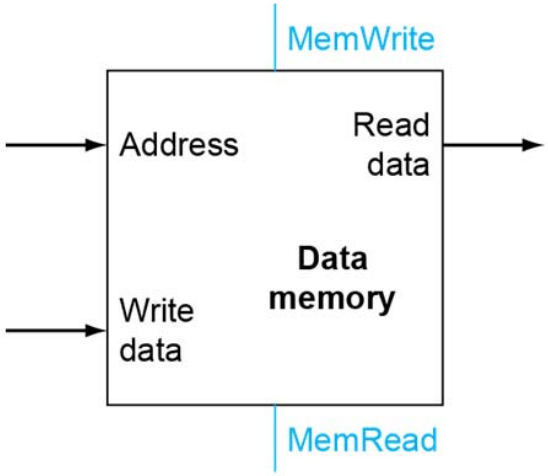
b. Immediate generation unit



Multiplexer



a. Registers



a. Data memory unit

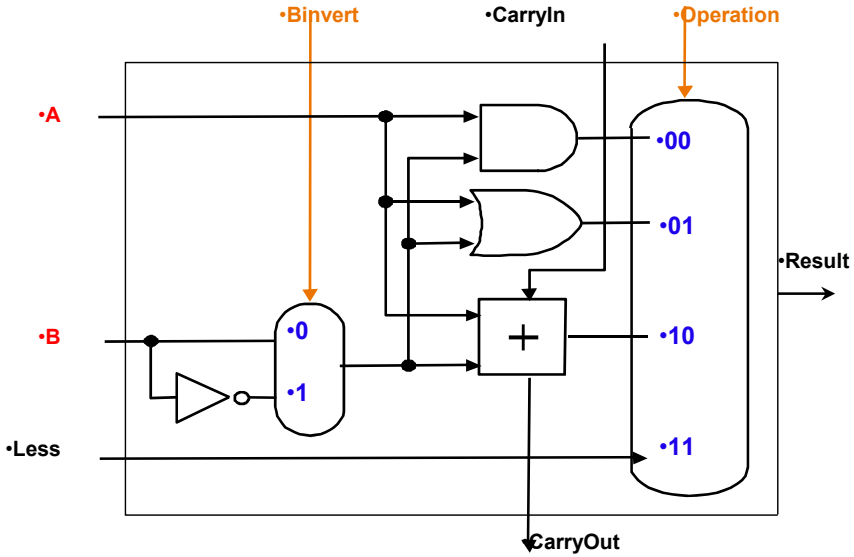
ALU

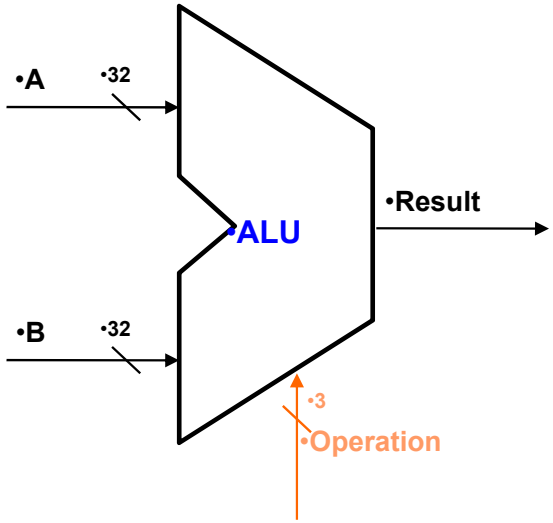
- 算术逻辑运算器ALU：即运算器
  - 5 Operations
  - “Set on less than”：

    if A<B then Result=1；

    else Result=0。

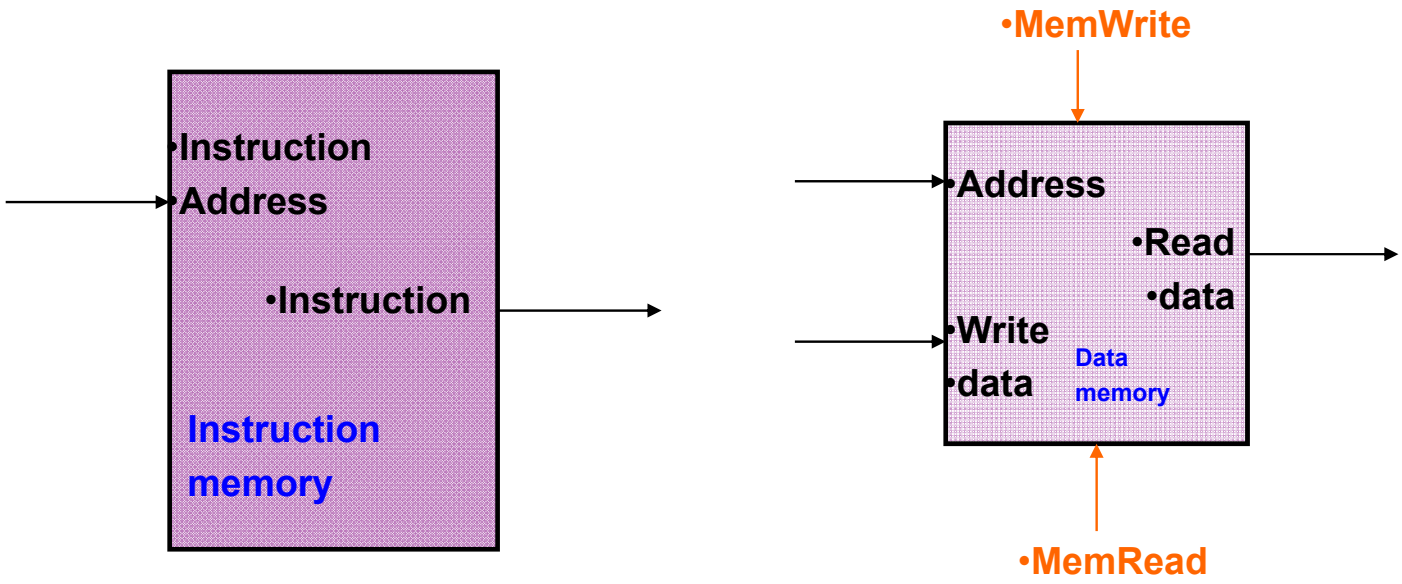
Operation	Function
000	And
001	Or
010	Add
110	Sub
111	Slt





Memory

- 存储器：
  - 可分为指令存储器与数据存储器；
  - 指令存储器设为只读；输入指令地址，输出指令。
  - 数据存储器可以读写，由MemRead和MemWrite控制。按地址读出数据输出，或将写数据写入地址所指存储器单元。



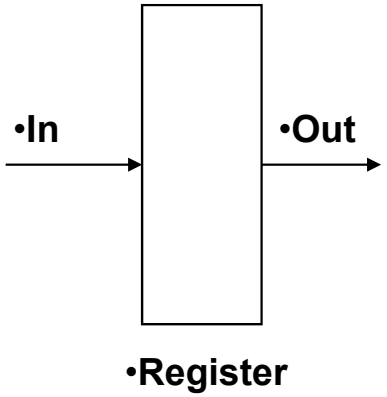
REGISTER

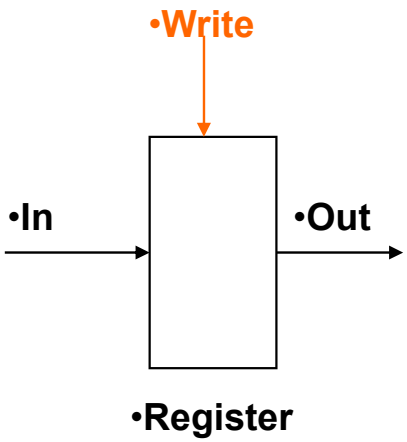
Register

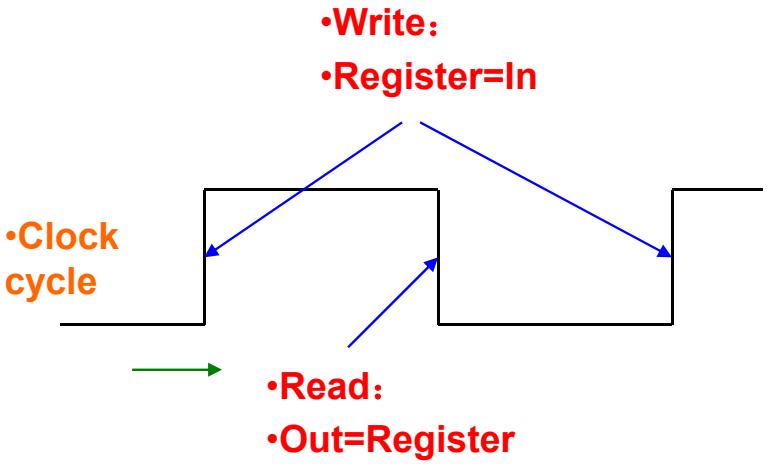
- State element。
- Can be controlled by Write signal.

  置0，数据输出保持原状态不变

  置1，在有效时钟边沿到来，数据输出为数据输入值

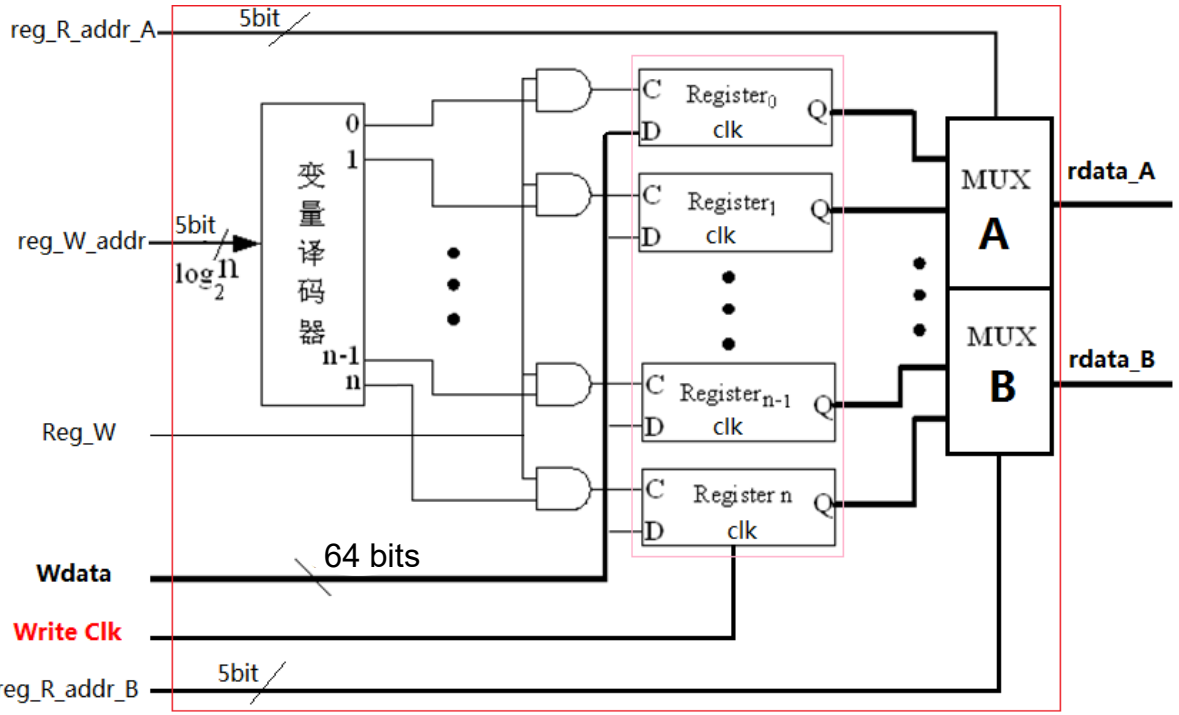






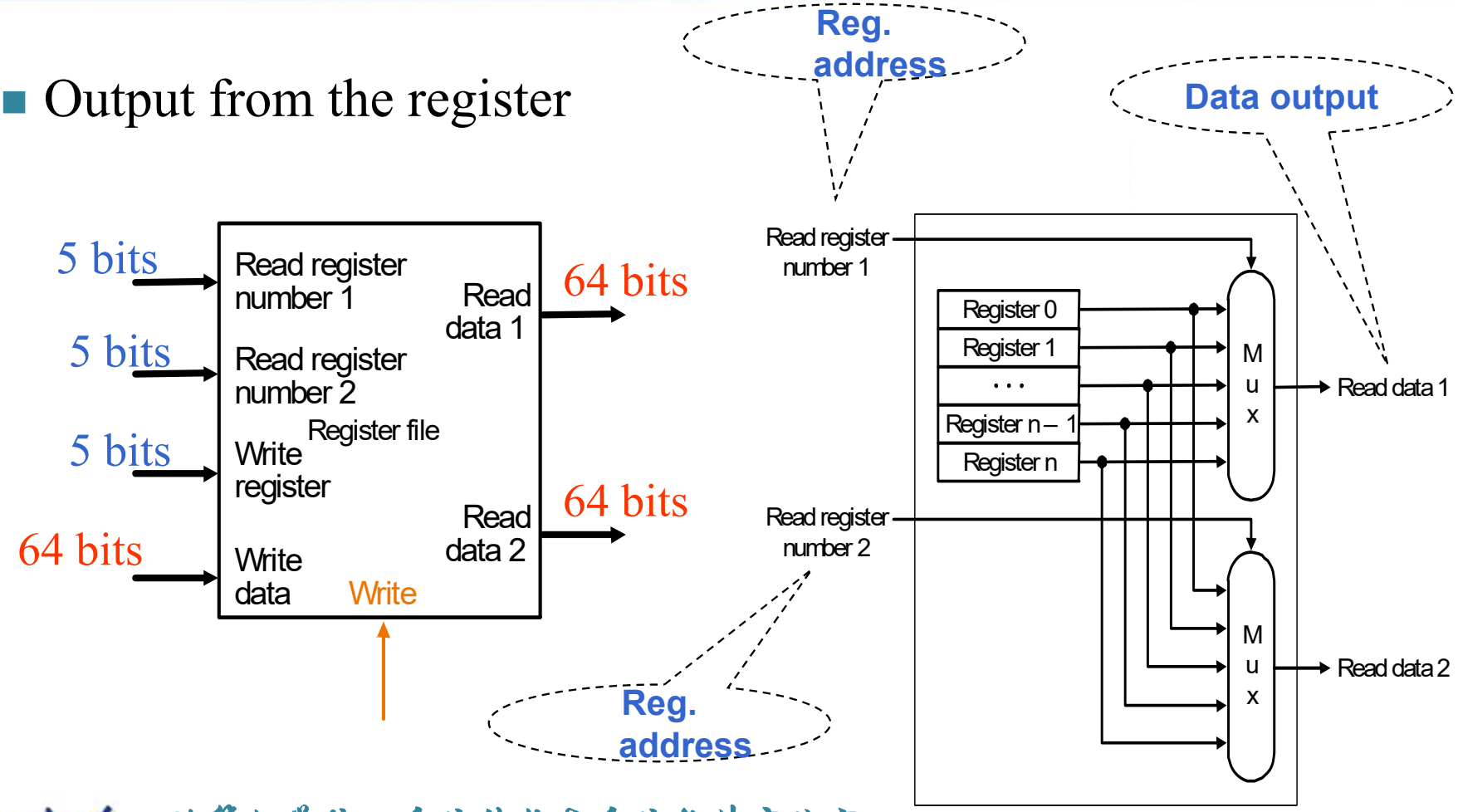
Register Files--Built using D flip-flops

- 32 64-bit Registers；
- Input: 2 5-bit register number/ one 5-bit register number and 64-bit data;
- Output: 64-bit data;
- Register write control。



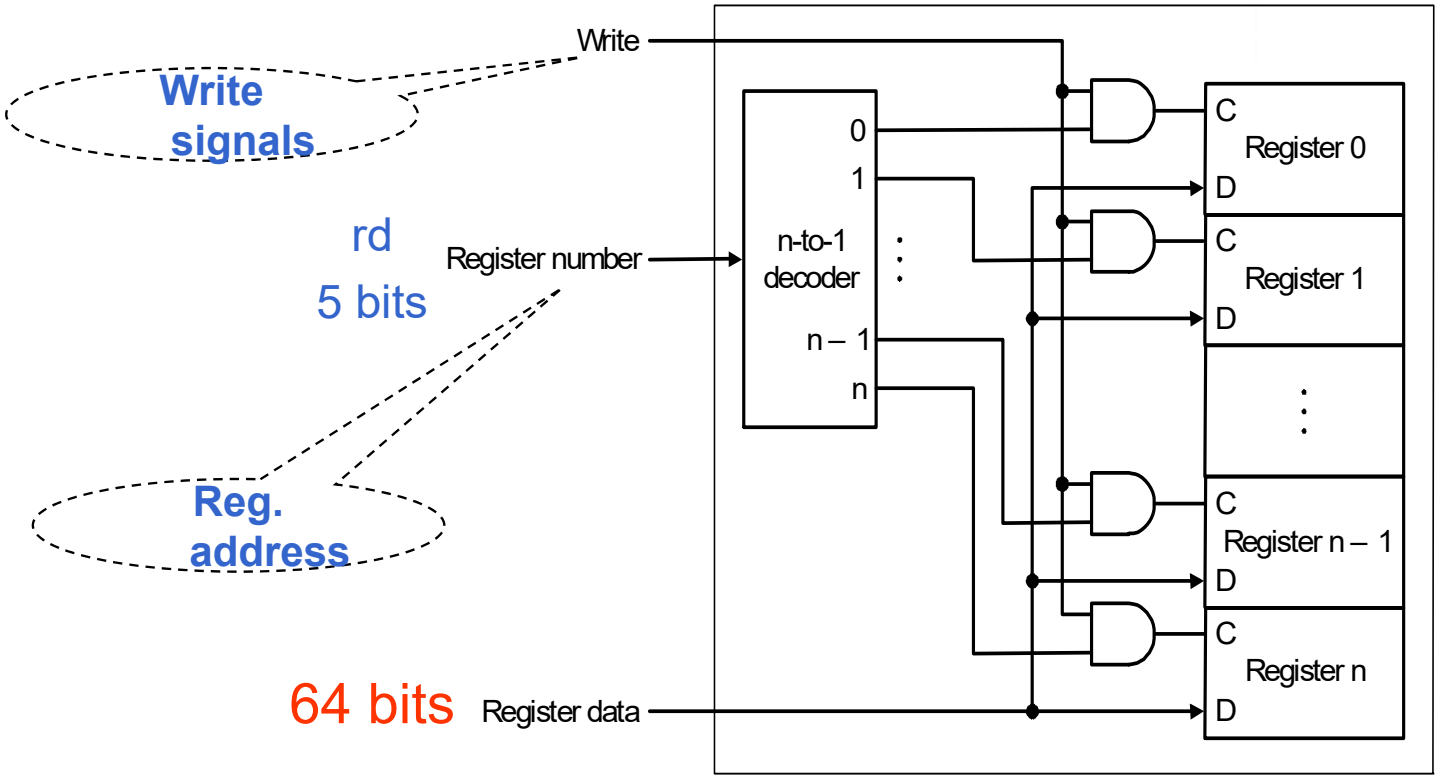
Register File: Read-Output

- Output from the register



Register File: Write-Input

- Written to the register



Description: 32×64bits Register files

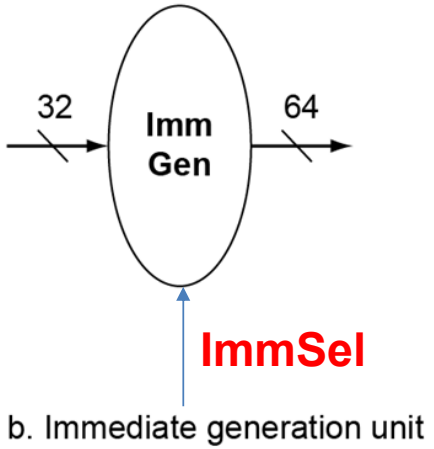
```
Module  regs( input              clk,  rst,  RegWrite,
              input    [4:0]    Rs1_addr, Rs2_addr, Wt_addr,
              input    [63:0]  Wt_data,
              output [63:0]  Rs1_data,  Rs2_data
            );
        reg [63:0] register [1:31];                    // r1 - r31
        integer i;

        assign rdata_A = (Rs1_addr== 0) ? 0 : register[Rs1_addr];              // read
        assign rdata_B = (Rs2_addr== 0) ? 0 : register[Rs2_addr];                 // read

        always @(posedge clk or posedge rst)
        begin   if (rst==1)
                        for (i=1; i<32; i=i+1)  register[i] <= 0;             // reset
                else if ((Wt_addr != 0) && (RegWrite == 1))
                        register[Wt_addr] <= Wt_data;                                 // write
        end
endmodule
```

The other elements

- Immediate generation unit：
  - 输入指令产生立即数的逻辑功能
    - 根据指令类型（加载，存储或者分支指令），产生相应的立即数
  - 转移指令偏移量左移位的功能
    - 立即数字段符号扩展为64位结果输出



b. Immediate generation unit

Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

Immediate generation

- Load: 0000011

  L_imm = {{52{inst[31]}}, Inst[31:20]};

- Save: 0100011

  S_imm = {{52{inst[31]}}, Inst[31:25], inst[11:7]};

- Branch: 1100011

  SB_imm = {{51{inst [31]}}, inst[31], inst [7], inst[30:25], inst[11:8],1'b0};

- Jal: 1101111

  UJ = {{43{inst[31]}}, inst [31], inst[19:12], inst[20], inst[30:21],1'b0};

ImmSel

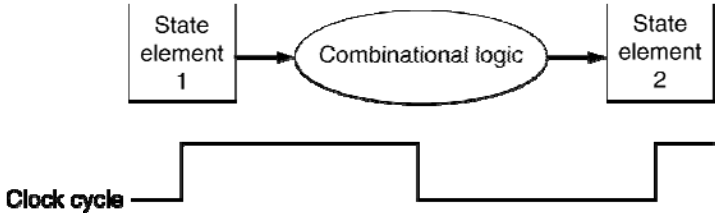
Instruction type	Instruction opcode[6:0]	Instruction operation	(sign-extend)immediate	ImmSel
I-type	0000011	Lw;lbu;lh; lb;lhu	(sign-extend) instr[31:20]	00
	0010011	Addi;slti;sltiu;xori;ori; andi;		
	1100111	jalr		
S-type	0100011	Sw;sb;sh	(sign-extend) instr[31:25],[11:7]	01
B-type	1100011	Beq;bne;blt;bge;bltu; bgeu	(sign-extend) instr[31],[7],[30:25],[11:8],1'b0	10
J-type	1101111	jal	(sign-extend) instr[31],[19:12],[20],[30:21],1'b0	11

Logic Design Conventions

- **Information encoded in binary**
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- **Combinational element**
  - Operate on data
  - Output is a function of input
- **State (sequential) elements**
  - Store information

Clocking Methodology

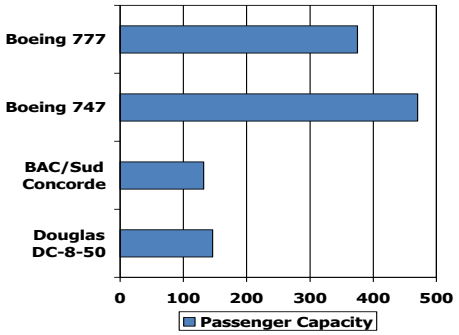
- An edge triggered methodology
- Typical execution:
  - read contents of some state elements,
  - send values through some combinational logic
  - write results to one or more state elements

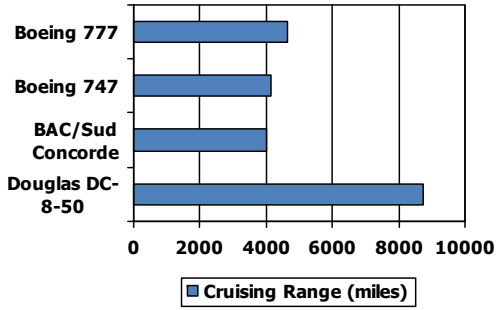


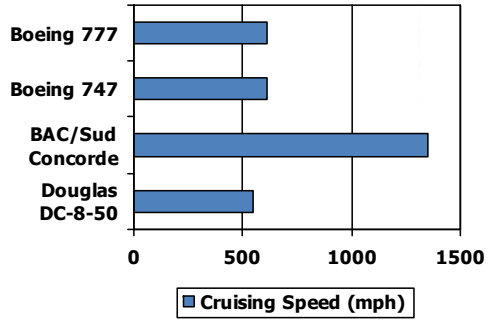
Performance?

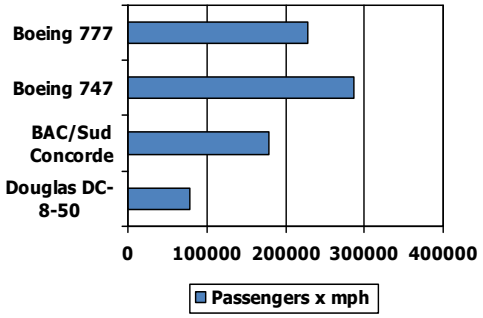
Defining Performance

Which airplane has the best performance?









Aircraft type	Passenger Capacity	Cruising Range(miles)	Cruising Speed(mph)	Passengers *mph
Boeing 777	375	4630	610	228,750
Boeing 747	470	4150	610	286,700
BAC/Sud Co ncorde	132	400 0	1350	178,200
Douglas DC-8-50	146	8720	544	79,424

Performance

Being able to gauge the relative performance of a computer is an important but tricky task. There are a lot of factors that can affect performance.

- Architecture
- Hardware implementation of the architecture
- Compiler for the architecture
- Operating system

Furthermore, we need to be able to define a measure of performance.

- Single users on a PC → a minimization of response time.
- Large data → a maximization of throughput

Response Time and Throughput

- **Latency (Response time)**
  - is the time between the start and completion of an event
  - How long it takes to do a task
- **Throughput (bandwidth)**
  - is the total amount of work done in a given period of time
  - Total work done per unit time
    - e.g., tasks/transactions/… per hour
- **How are response time and throughput affected by**
  - Replacing the processor in a computer with a faster processor
  - Adding more processors?
- **We'll focus on program response time for now**

Performance

Performance has an inverse relationship to execution time.

$$Performance = \frac{1}{Execution\ Time}$$

Comparing the performance of two machines can be accomplished by comparing execution times.

$$Performance_X > Performance_Y$$

$$\rightarrow \quad \frac{1}{Execution_X} > \frac{1}{Execution_Y}$$

$$\rightarrow \quad Execution_Y > Execution_X$$

Performance

- **"X is *n* time faster than Y"**

$$\text{Performance}_X / \text{Performance}_Y$$
$$= \text{Execution time}_Y / \text{Execution time}_X = n$$

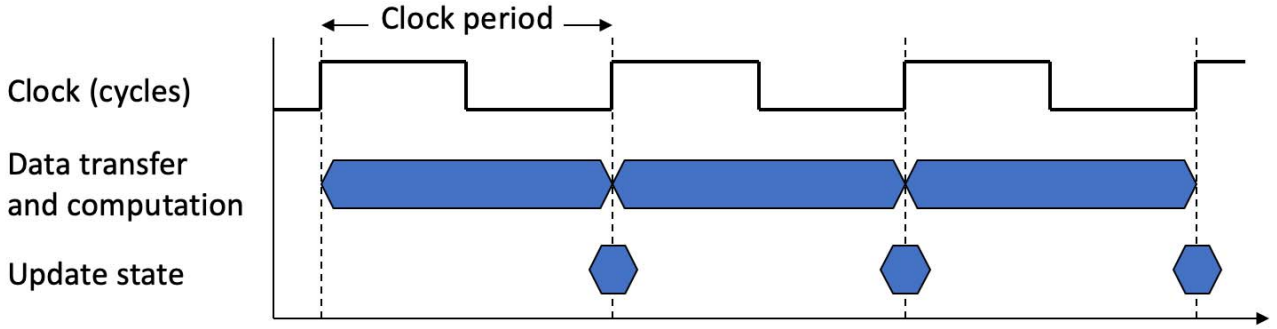
- **Example: time taken to run a program**
  - 10s on A, 15s on B
  - Execution Time_B / Execution Time_A
    = 15s / 10s = 1.5
  - So A is 1.5 times faster than B

Measuring Execution Time

- **Elapsed time**
  - Total response time, including all aspects
    - Processing, I/O, OS overhead, idle time
  - Determines system performance
- **CPU time**
  - Time spent processing a given job
    - Discounts I/O time, other jobs' shares
  - Comprises user CPU time and system CPU time
    - User CPU time : CPU time spent in the program itself
    - System CPU time: CPU time spent in the OS, performing tasks on behalf of the program.
  - Different programs are affected differently by CPU and system performance

CPU Clocking

- Operation of digital hardware governed by a constant-rate clock



- Clock period: duration of a clock cycle
  - e.g., 250ps = 0.25ns = 250×10^{-12}s
- Clock frequency (rate): cycles per second
  - e.g., 4.0GHz = 4000MHz = 4.0×10^{9}Hz

CPU Performance

In order to determine the effect of a design change on the performance experienced by the user, we can use the following relation:

$$CPU\ Execution\ Time = CPU\ Clock\ Cycles \times Clock\ Period$$

Alternatively,

$$CPU\ Execution\ Time = \frac{CPU\ Clock\ Cycles}{Clock\ Rate}$$

CPU Time

$$\text{CPU Time} = \text{CPU Clock Cycles} \times \text{Clock Cycle Time}$$

$$= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}$$

- **Performance improved by**
  - Reducing number of clock cycles
  - Increasing clock rate
  - Hardware designer must often trade off clock rate against cycle count

CPU Time Example

- Computer A: 2GHz clock, 10s CPU time
- Designing Computer B
  - Aim for 6s CPU time
  - Can do faster clock, but causes 1.2 × clock cycles
- How fast must Computer B clock be?

$$\text{Clock Rate}_B = \frac{\text{Clock Cycles}_B}{\text{CPU Time}_B} = \frac{1.2 \times \text{Clock Cycles}_A}{6\text{s}}$$

$$\text{Clock Cycles}_A = \text{CPU Time}_A \times \text{Clock Rate}_A$$

$$= 10\text{s} \times 2\text{GHz} = 20 \times 10^9$$

$$\text{Clock Rate}_B = \frac{1.2 \times 20 \times 10^9}{6\text{s}} = \frac{24 \times 10^9}{6\text{s}} = 4\text{GHz}$$

Metrics of CPU performance

The basic components of performance and how each is measured.

Component	Units of Measure
CPU Execution Time for a Program	Seconds for the Program
Instruction Count	Instructions Executed for the Program
Clock Cycles per Instruction	Average Number of Clock Cycles per Instruction
Clock Cycle Time (Clock Period)	Seconds per Clock Cycle

Instruction Count, CPI, and Clock Period combine to form the three important components for determining CPU execution time. *Just analyzing one is not enough!* Performance between two machines can be determined by examining non-identical components.

Instruction Count and CPI

- **Instruction Count for a program**
  - Determined by program, ISA and compiler
- **Average cycles per instruction (CPI)**
  - Determined by CPU hardware
  - If different instructions have different CPI
    - Average CPI affected by instruction mix

$$CPI = \frac{CPU\ Clock\ Cycles}{Instruction\ Count}$$

$$CPU\ Clock\ Cycles = Instructions\ for\ a\ Program \times Average\ Clock\ Cycles\ Per\ Instruction$$

$$CPU\ Time = Instruction\ Count \times CPI \times Clock\ Period$$

$$CPU\ Time = \frac{Instruction\ Count \times CPI}{Clock\ Rate}$$

CPI Example

- Computer A: Cycle Time = 250ps, CPI = 2.0
- Computer B: Cycle Time = 500ps, CPI = 1.2
- Same ISA
  - Which is faster, and by how much?

$$\begin{aligned}
\text{CPU Time}_A &= \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A \\
&= I \times 2.0 \times 250\text{ps} = I \times 500\text{ps} \\
\text{CPU Time}_B &= \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B \\
&= I \times 1.2 \times 500\text{ps} = I \times 600\text{ps} \\
\frac{\text{CPU Time}_B}{\text{CPU Time}_A} &= \frac{I \times 600\text{ps}}{I \times 500\text{ps}} = 1.2
\end{aligned}$$

CPI in More Detail

- If different instruction classes take different numbers of cycles

$$\text{Clock Cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{Instruction Count}_i)$$

- Weighted average CPI

$$\text{CPI} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^{n} \left(\text{CPI}_i \times \underbrace{\frac{\text{Instruction Count}_i}{\text{Instruction Count}}}_{\text{Relative frequency}} \right)$$

Weighted CPI Example

- Alternative compiled code sequences using instructions in classes A, B, C

Class	A	B	C
CPI for class	1	2	3
IC in sequence 1	2	1	2
IC in sequence 2	4	1	1

- Sequence 1: IC = 5 Clock Cycles = 2×1 + 1×2 + 2×3 = 10
  - Avg. CPI = 10/5 = 2.0
- Sequence 2: IC = 6 Clock Cycles = 4×1 + 1×2 + 1×3 = 9
  - Avg. CPI = 9/6 = 1.5

Performance Summary

The BIG Picture

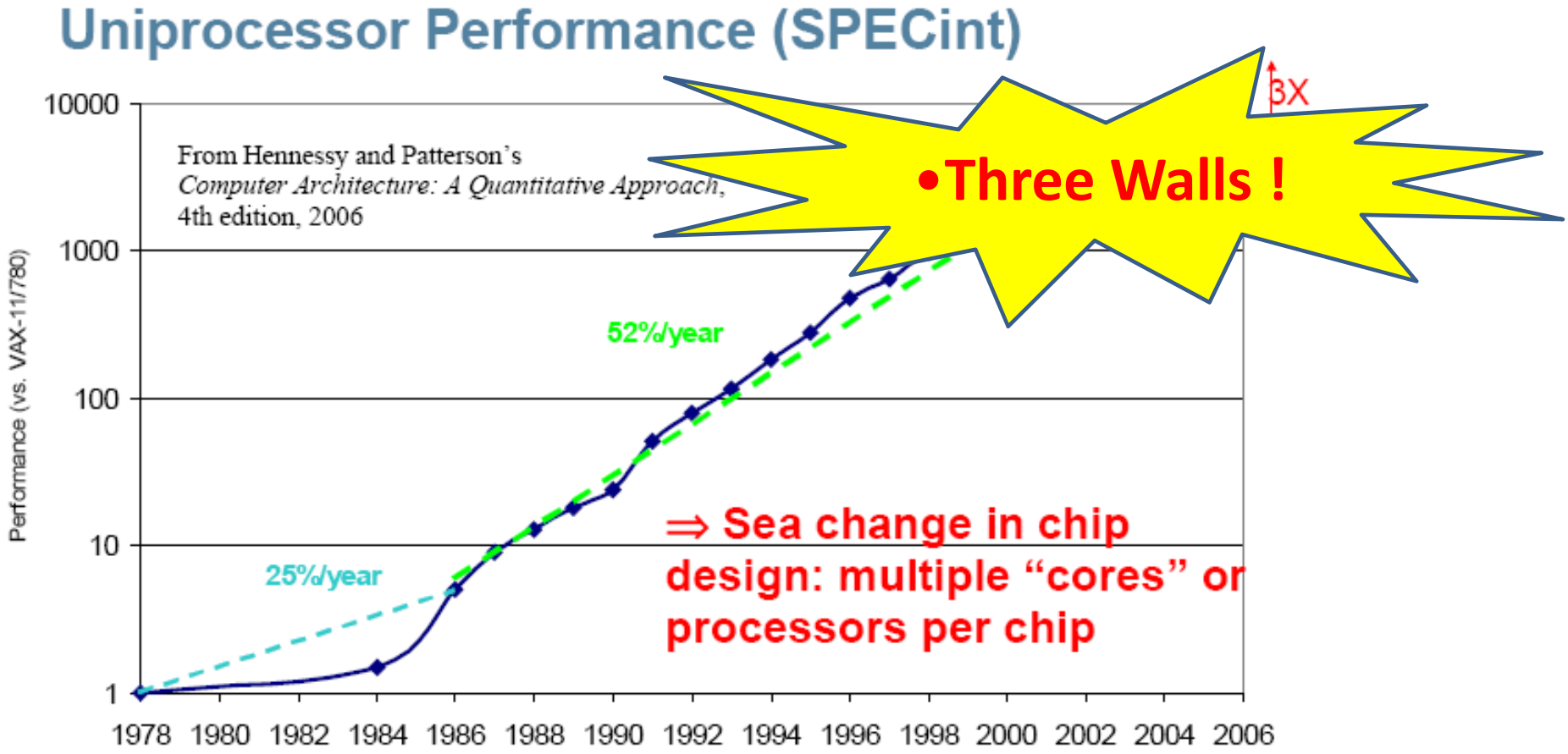
$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- **Performance depends on**
  - Algorithm: affects IC, possibly CPI
  - Programming language: affects IC, CPI
  - Compiler: affects IC, CPI
  - Instruction set architecture: affects IC, CPI, T_c

Improvement

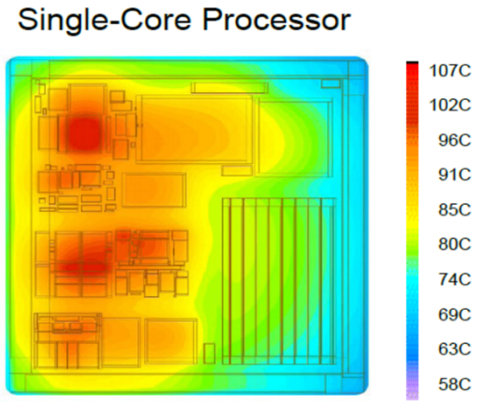
- **Improvement of input / output**
- **The development of memory organization structure**
  - Associative memory and associated processor
  - General register group
  - Cache
- **Two directions of instruction set development:**
  - CISC
  - RISC
- **Parallel processing technology**
  - How to develop parallelism in traditional machines?
  - Develop parallel technologies at different levels.
  - For example, micro operation level, instruction level, thread level, process level, task level, etc.

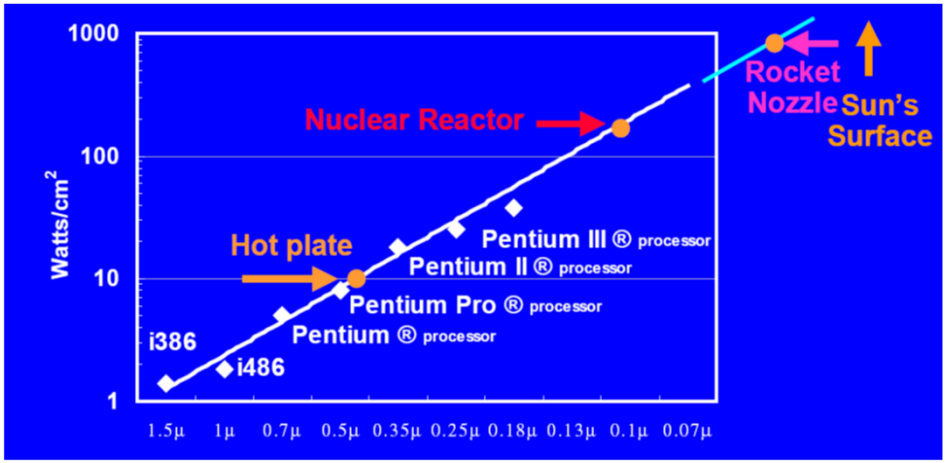
Incredible performance improvement



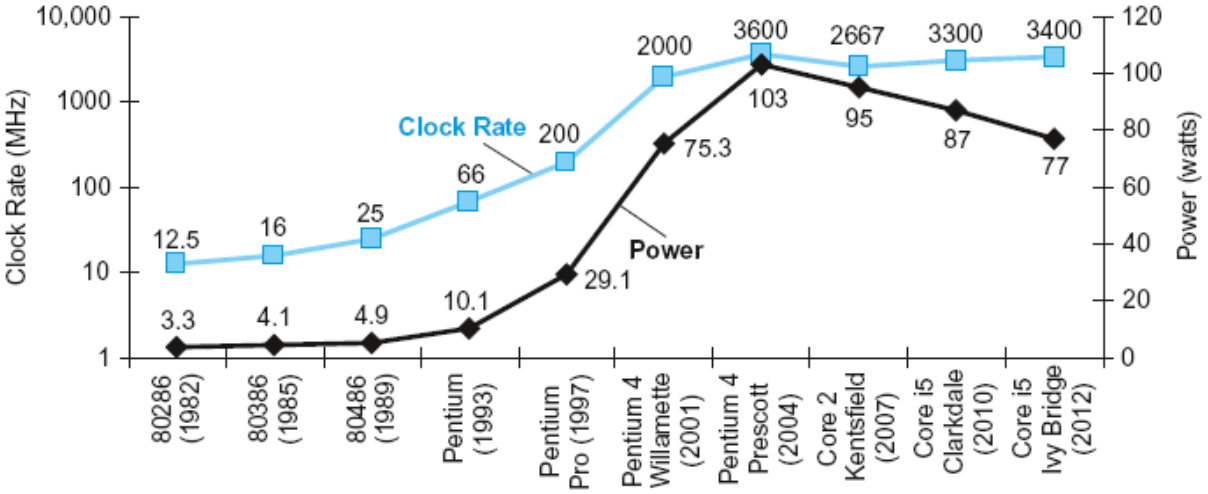
Power Wall

- **Power efficiency decrease**
  - **the trend of consuming double the power with each doubling of operating frequency**
- **Hot-spot**
- **Power leakage**





Power Trends



◎In CMOS IC technology

$$\text{Power} = \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency}$$

×30　　5V → 1V　　×1000

Reducing Power

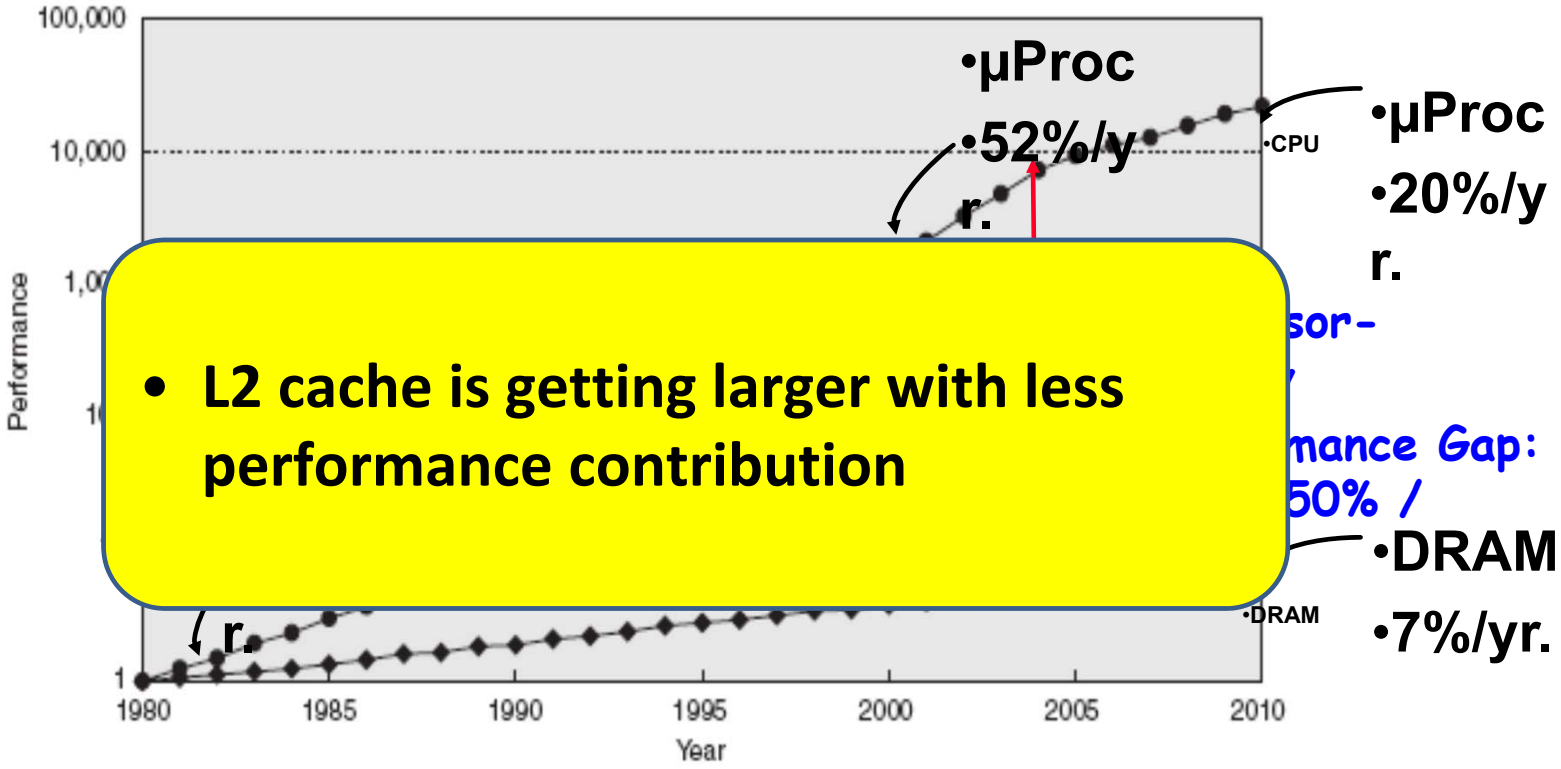
◎Suppose a new CPU has

- 85% of capacitive load of old CPU
- 15% voltage and 15% frequency reduction

$$\frac{P_{new}}{P_{old}} = \frac{C_{old} \times 0.85 \times (V_{old} \times 0.85)^2 \times F_{old} \times 0.85}{C_{old} \times V_{old}^{\ 2} \times F_{old}} = 0.85^4 = 0.52$$

- **The power wall**
  - **We can't reduce voltage further**
  - **We can't remove more heat**
- **How else can we improve performance?**

Memory Wall



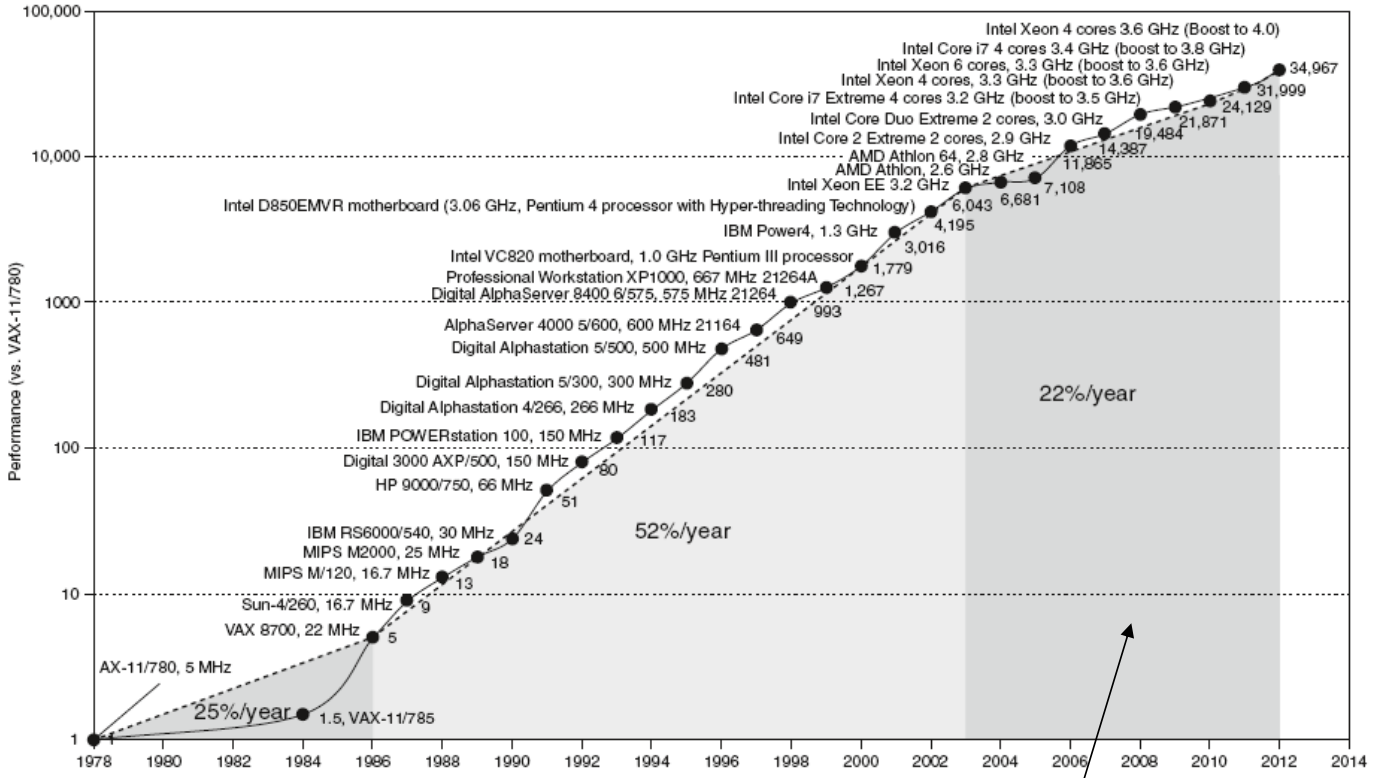
❑1980: no cache in µproc; 2001: 2-level cache on chip
(1989 first Intel µproc with a cache on chip)

ILP Wall

- *"ILP wall"* refers to increasing difficulty to find enough parallelism in the instructions stream of a single process to keep higher performance processor cores busy.

- **20%: little ILP left to exploit due to power dissipation and memory gap**
  - ILP => TLP and DLP

Uniprocessor Performance



Constrained by power, instruction-level parallelism, memory latency

Advanced - Multiprocessors

- **Multicore microprocessors**
  - More than one processor per chip
- **Requires explicitly parallel programming**
  - Compare with instruction level parallelism
    - Hardware executes multiple instructions at once
    - Hidden from the programmer
  - Hard to do
    - Programming for performance
    - Load balancing
    - Optimizing communication and synchronization

Amdahl's Law

Amdahl's Law states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.

Amdahl's Law depends on two factors:

- The fraction of the time the enhancement can be exploited.
- The improvement gained by the enhancement while it is exploited.

$$Improved\ Execution\ Time = \frac{Affected\ Execution\ Time}{Amount\ of\ Improvement} + Unaffected\ Execution\ Time$$

Make the common case fast!

Amdahl's Law

- **Improving an aspect of a computer and expecting a proportional improvement in overall performance**

$$T_{improved} = \frac{T_{affected}}{improvement\ factor} + T_{unaffected}$$

- **Example: multiply accounts for 80s/100s**
  - How much improvement in multiply performance to get 5× overall?

$$20 = \frac{80}{n} + 20$$

Can't be done!

Amdahl's Law

- The system performance acceleration rate is limited by the percentage of the execution time of the component to the total execution time in the system.
- Amdahl's law defines the *speedup* that can be gained by using a particular feature.

$$\text{Speedup} = \frac{\text{Performance for entire task}_{\text{Using Enhancement}}}{\text{Performance for entire task}_{\text{Without Enhancement}}}$$

$$= \frac{\text{Total Execution Time}_{\text{Without Enhancement}}}{\text{Total Execution Time}_{\text{Using Enhancement}}}$$

Amdahl's Law

- **Based on the basic idea that:**

$$\text{Execution time}_{\text{new}} = \text{Execution time}_{\text{can not be enhance}} + \text{Execution time}_{\text{enhanced}}$$

$$\text{Execution time}_{\text{new}} = \text{Execution time}_{\text{old}} \times \left((1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right)$$

$$\text{Speedup}_{\text{overall}} = \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

Amdahl's Law

- **Improved ratio: In the system before the improvement, the ratio of the execution time of the improvement part to the total execution time.**
  - It is always less than or equal to 1.
  - For example: a program that needs to run for 60 seconds has 20 seconds of calculation that can be accelerated,
    Then the ratio is 20/60.
- **Component speedup ratio: The multiple that can be improved after some improvements. It is the ratio of the execution time before the improvement to the execution time after the improvement.**
  - Under normal circumstances, the component acceleration ratio is greater than 1.
  - For example: if the system is improved, the execution time of the improved part is 2 seconds, Before the improvement, its execution time was 5 seconds, and the component acceleration ratio was 5/2.

Amdahl's Law

- **Example 1.1 Increasing the processing speed of a certain function in the computer system to *20 times* the original, but the processing time of this function only accounts for *40%* of the running time of the entire system. After adopting this method to improve performance, how much can the performance of the entire system improve?**

Answer:

- Fraction $_{enhanced} = 40\%$
- Speedup $_{enhanced} = 20$

$$\text{Speedup} = \frac{1}{0.6 + \frac{0.4}{20}} = 1.613$$

Amdahl's Law

- **Example 1.2 After a computer system adopts floating-point arithmetic components, the floating-point arithmetic speed is increased by *20 times*, and the overall performance of a certain program of the system is increased by *5 times*. Try to calculate the proportion of the floating-point operations in this program.**

Answer:

- Speedup $_{overall} = 5$
- Speedup $_{enhanced} = 20$

$$\frac{1}{(1 - Fraction) + \frac{Fraction}{20}} = 5$$

Fraction = 84.2%

Amdahl's Law

- A decreasing rule for performance improvement
  If only a part of the computing task is improved, the more the improvement, the more limited the overall performance improvement.

- Important inference: If only a part of the whole task is improved and optimized, the speedup obtained will not exceed:

  1/ (1-the ratio can be improved)

System I

The Processor: Design

Haifeng Liu

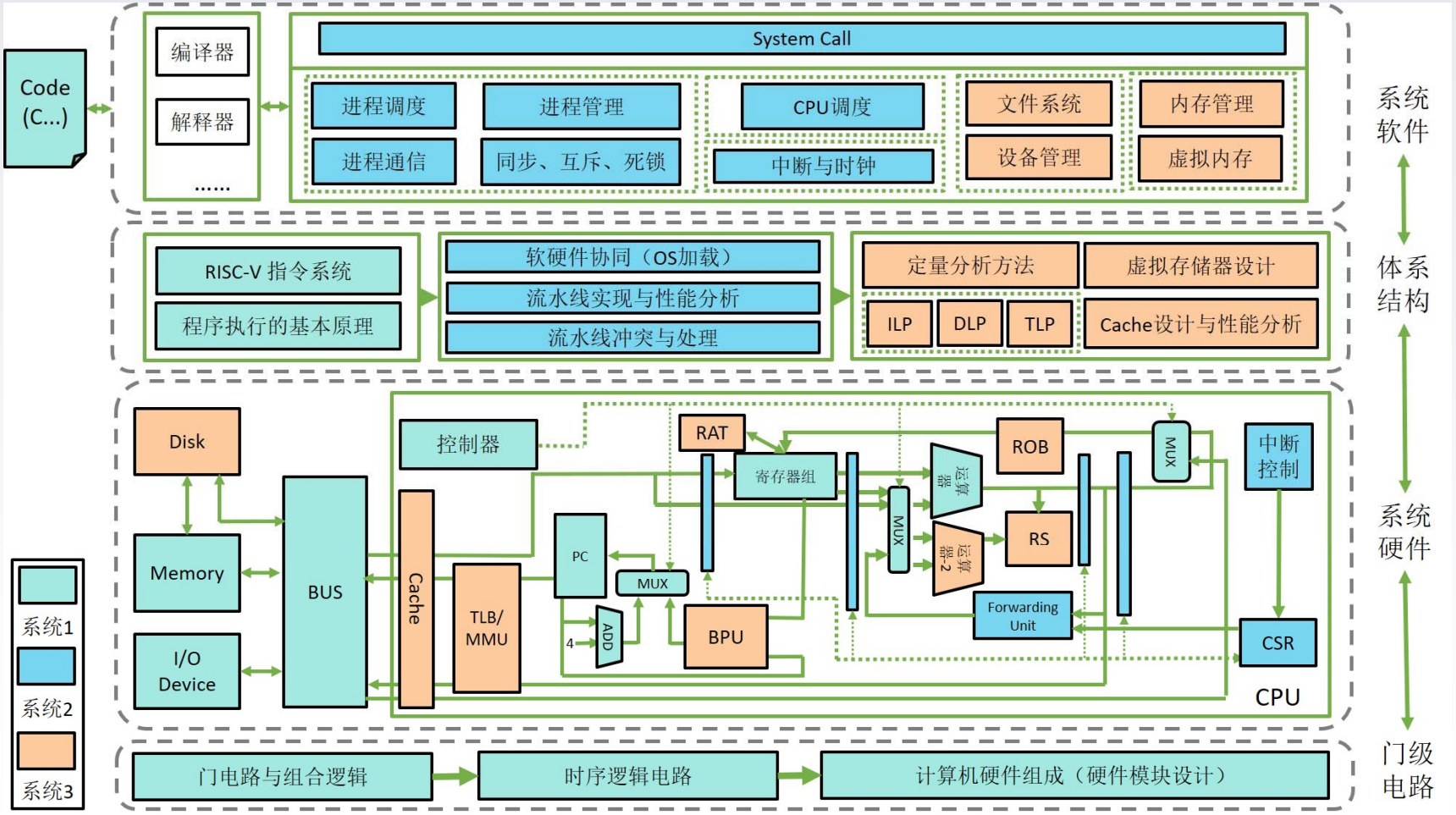
Zhejiang University

CPU we will be doing

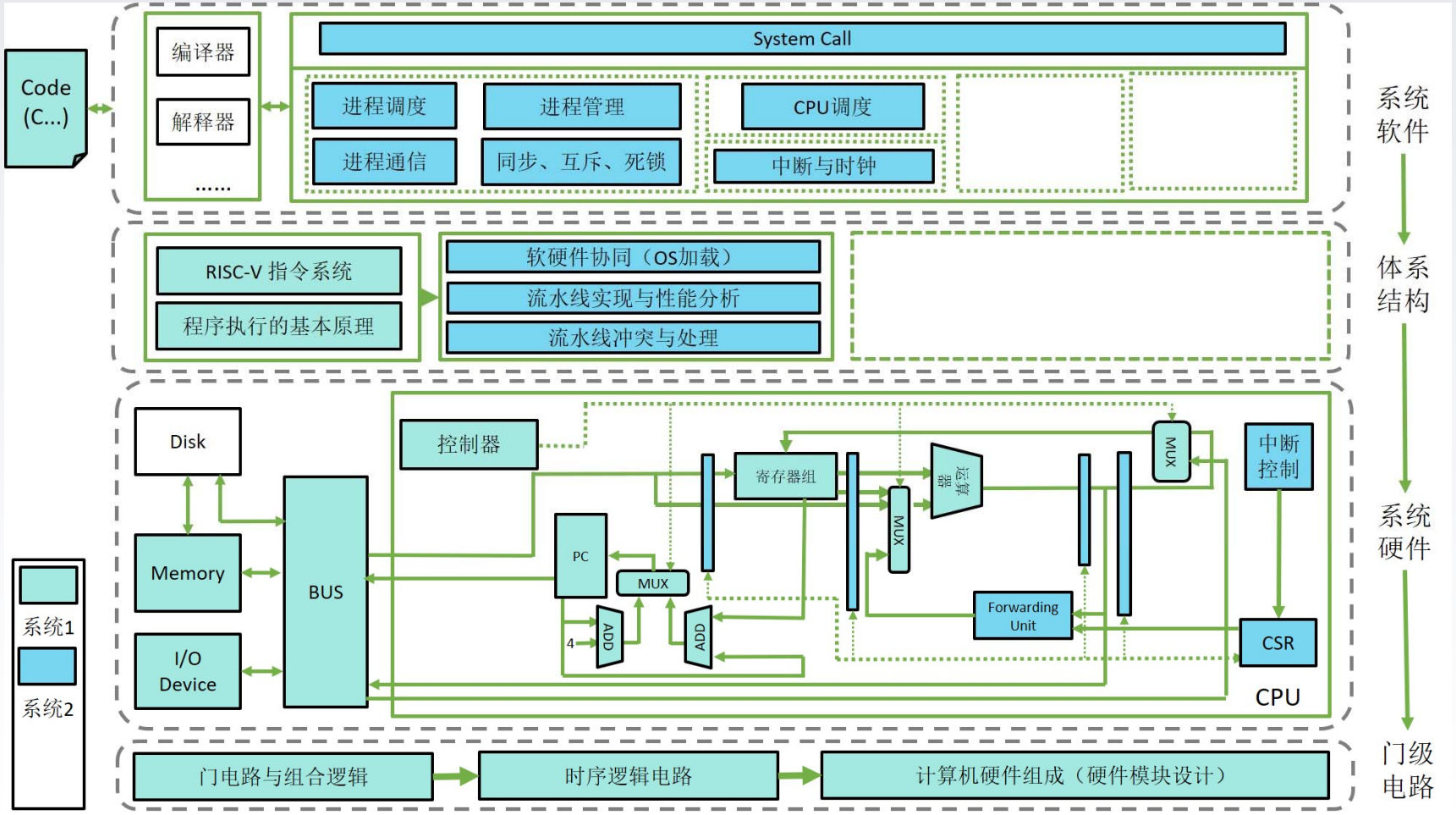
- We'll look at an implementation of RISC-V
- Simplified to contain only:
  - memory-reference instructions: **lw, sw**
  - arithmetic-logical instructions: **add, sub, and, or, slt**
  - control flow instructions: **beq, jal**
- An overview of the implementation
  - For every instruction, the first two step are identical
    1. Fetch the instruction from the memory
    2. Decode and read the registers
  - Next steps depend on the instruction class
    - **Memory-reference Arithmetic-logical branches**

- ***What are the steps?***
- ***How many functions***

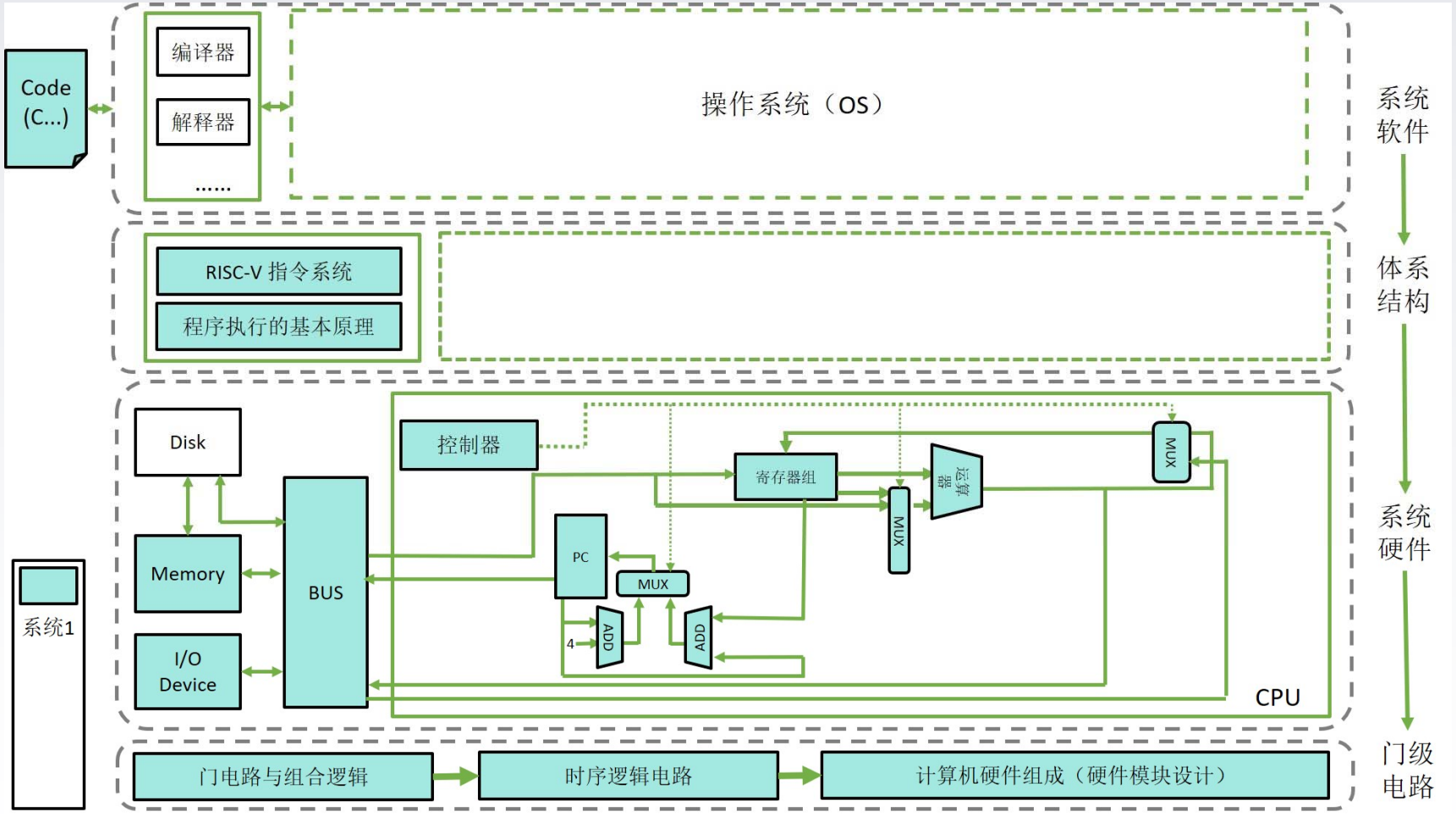
CPU we will be doing



CPU we will be doing



CPU we will be doing



Instruction Execution Overview

- **For every instruction, the first two steps are identical**
  - **Fetch the instruction from the memory**
  - **Decode and read the registers**
- **Next steps depend on the instruction class**
  - **Memory-reference Arithmetic-logical branches**
- **Depending on instruction class**
  - Use ALU to calculate
    - Arithmetic result
    - Memory address for load/store
    - Branch comparison
  - Access data memory for load/store
  - PC ← target address or PC + 4

Contents

- **Building a datapath**
- **A Simple Implementation Scheme**

Building a datapath

- **Datapath**
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux's, memories, …
- **We will build a RISC-V datapath incrementally**
  - Refining the overview design

RISC-V fields (format)

Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

- ***opcode***: *basic operation of the instruction.*
- ***rs1***: *the first register source operand.*
- ***rs2***: *the second register source operand.*
- ***rd***: *the register destination operand.*
- ***funct***: *function,this field selects the specific variant of the operation in the op field.*
- ***Immediate***: *address or immediate*

Name	Example	Comments
32 registers	x0-x31	Fast locations for data. In RISC-V, data must be in registers to perform arithmetic. Register x0 always equals 0.
2^{61} memory words	Memory[0], Memory[8] , …… , Memory[18446744073709551608]	Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential doubleword accesses differ by 8. Memory holds data structures, arrays, and spilled registers.

Name	Register name	Usage	Preserved On call?
x0	0	The constant value 0	n.a.
x1(ra)	1	Return address(link register)	yes
x2(sp)	2	Stack pointer	yes
x3(gp)	3	Global pointer	yes
x4(tp)	4	Thread pointer	yes
x5-x7	5-7	Temporaries	no
x8-x9	8-9	Saved	yes
x10-x17	10-17	Arguments/results	no
x18-x27	18-27	Saved	yes
x28-x31	28-31	Temporaries	no

RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add x5,x6,x7	x5=x6 + x7	Add two source register operands
	subtract	sub x5,x6,x7	x5=x6 - x7	First source register subtracts second one
	add immediate	addi x5,x6,20	x5=x6+20	Used to add constants
Data transfer	load doubleword	ld x5, 40(x6)	x5=Memory[x6+40]	doubleword from memory to register
	store doubleword	sd x5, 40(x6)	Memory[x6+40]=x5	doubleword from register to memory
Logical	and	and x5, x6, 3	x5=x6 & 3	Arithmetic shift right by register
	inclusive or	or x5,x6,x7	x5=x6 | x7	Bit-by-bit OR
Conditional Branch	branch if equal	beq x5, x6, 100	if(x5 == x6) go to PC+100	PC-relative branch if registers equal
	branch if not equal	bne x5, x6, 100	if(x5 != x6) go to PC+100	PC-relative branch if registers not equal
Unconditional Branch	jump and link	jal x1, 100	x1 = PC + 4; go to PC+100	PC-relative procedure call
	jump and link register	jalr x1, 100(x5)	x1 = PC + 4; go to x5+100	procedure return; indirect call

R-type Instructions	funct7	rs2	rs1	funct3	rd	opcode	Example
add (Add)	0000000	00011	00010	000	00001	0110011	add x1,x2,x3
sub (Sub)	010000	00011	00010	000	00001	0110011	sub x1,x2,x3
I-type Instructions	**immediate**		**rs1**	**funct3**	**rd**	**opcode**	**Example**
addi (Add Immediate)	001111101000		00010	000	00001	0010011	addi x1,x2, 1000
ld (Load doubleword)	001111101000		00010	011	00001	0000011	ld x1,1000 (x2)
S-type Instructions	**immed -iate**	**rs2**	**rs1**	**funct3**	**immed -iate**	**opcode**	**Example**
sd (Store doubleword)	0011111	00001	00010	011	01000	0100011	sd x1,1000(x2)

SB-type instruction	**111110**	**01011**	**01010**	**001**	**10000**	**1100011**

bne x10, x11, 2000

UJ-type instruction	**01111101000000000000**	**00000**	**1101111**

jal x0, 2000

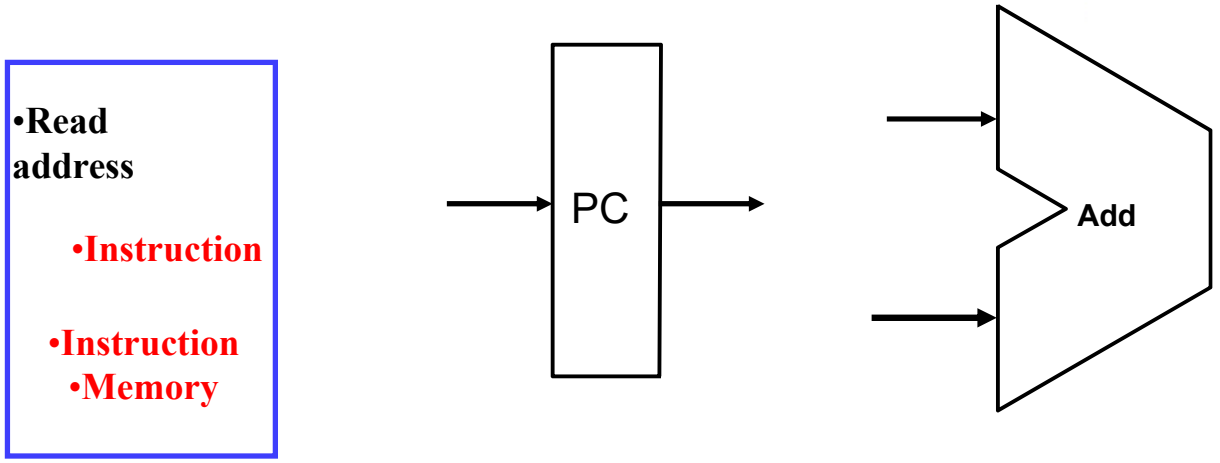
Instruction execution in RISC-V

- **Fetch :**
  - Take instructions from the instruction memory
  - Modify PC to point the next instruction
- **Instruction decoding & Read Operand:**
  - Will be translated into machine control command
  - Reading Register Operands, whether or not to use
- **Executive Control:**
  - Control the implementation of the corresponding ALU operation
- **Memory access:**
  - Write or Read data from memory
  - Only ld/sd
- **Write results to register:**
  - If it is R-type instructions, ALU results are written to rd
  - If it is I-type instructions, memory data are written to rd
- **Modify PC** for branch instructions

Instruction fetching three elements

Data Stream of Instruction fetching

How to connect? Who?

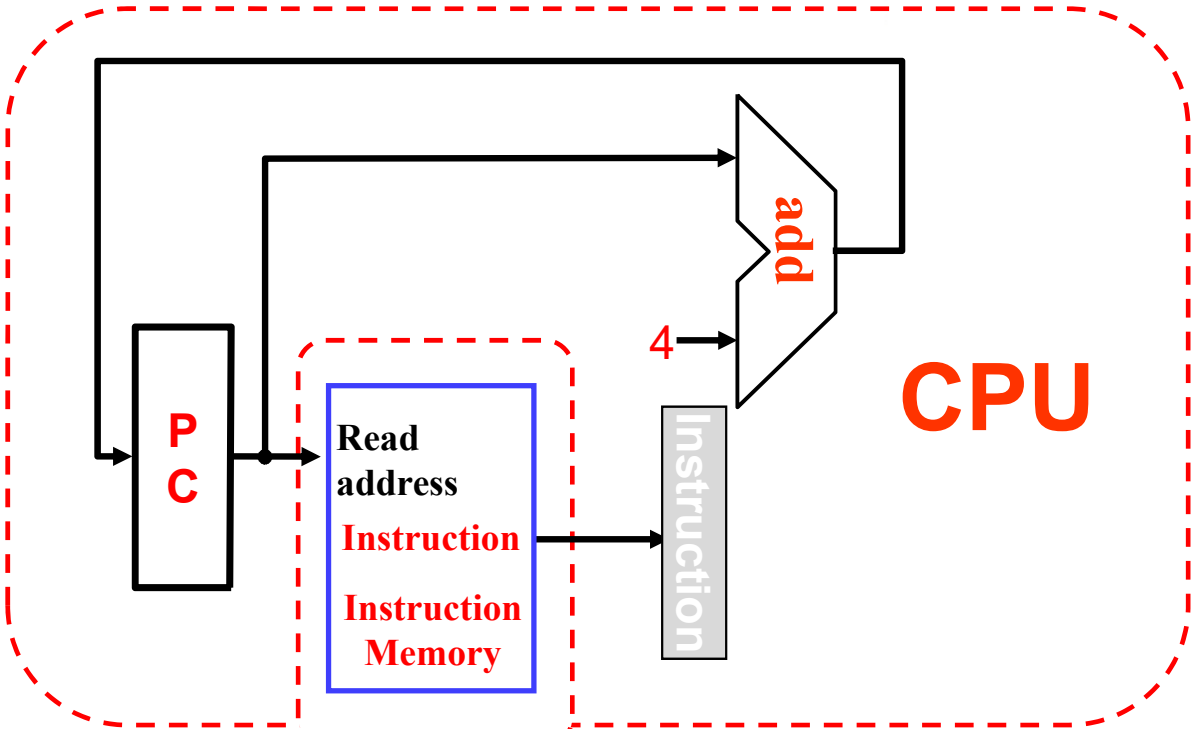


Instruction memory Program counter Adder

Instruction fetching unit

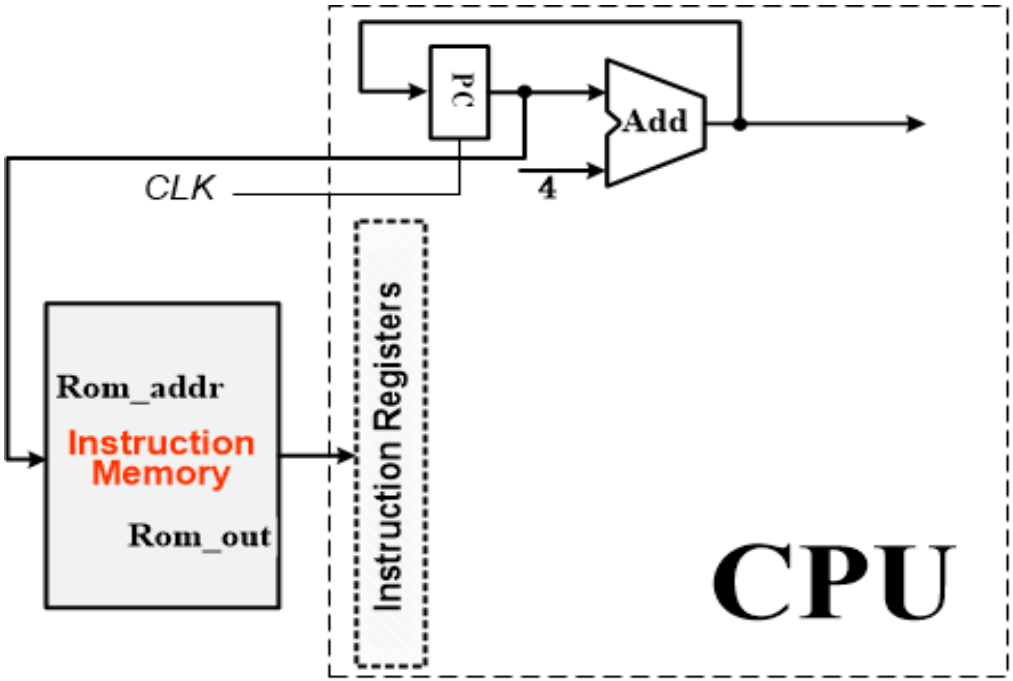
Instruction Register

- Can you omit it?



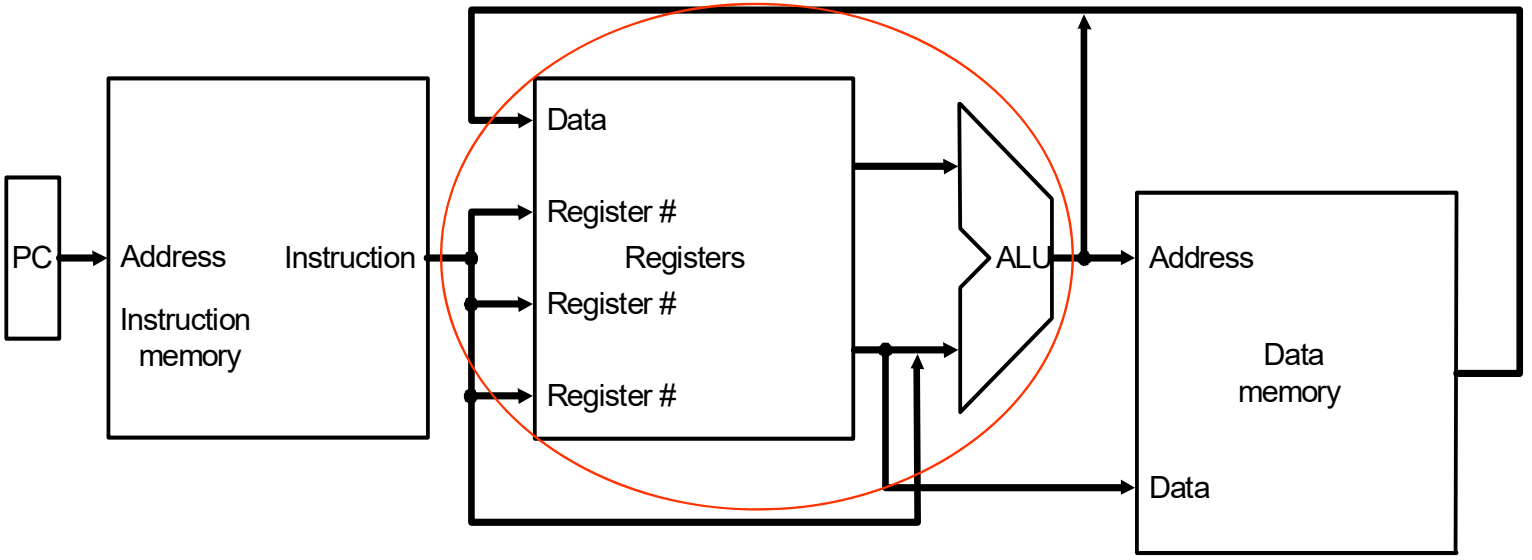
How simple is!

- Why PC+4?



More Implementation Details

Abstract / Simplified View:



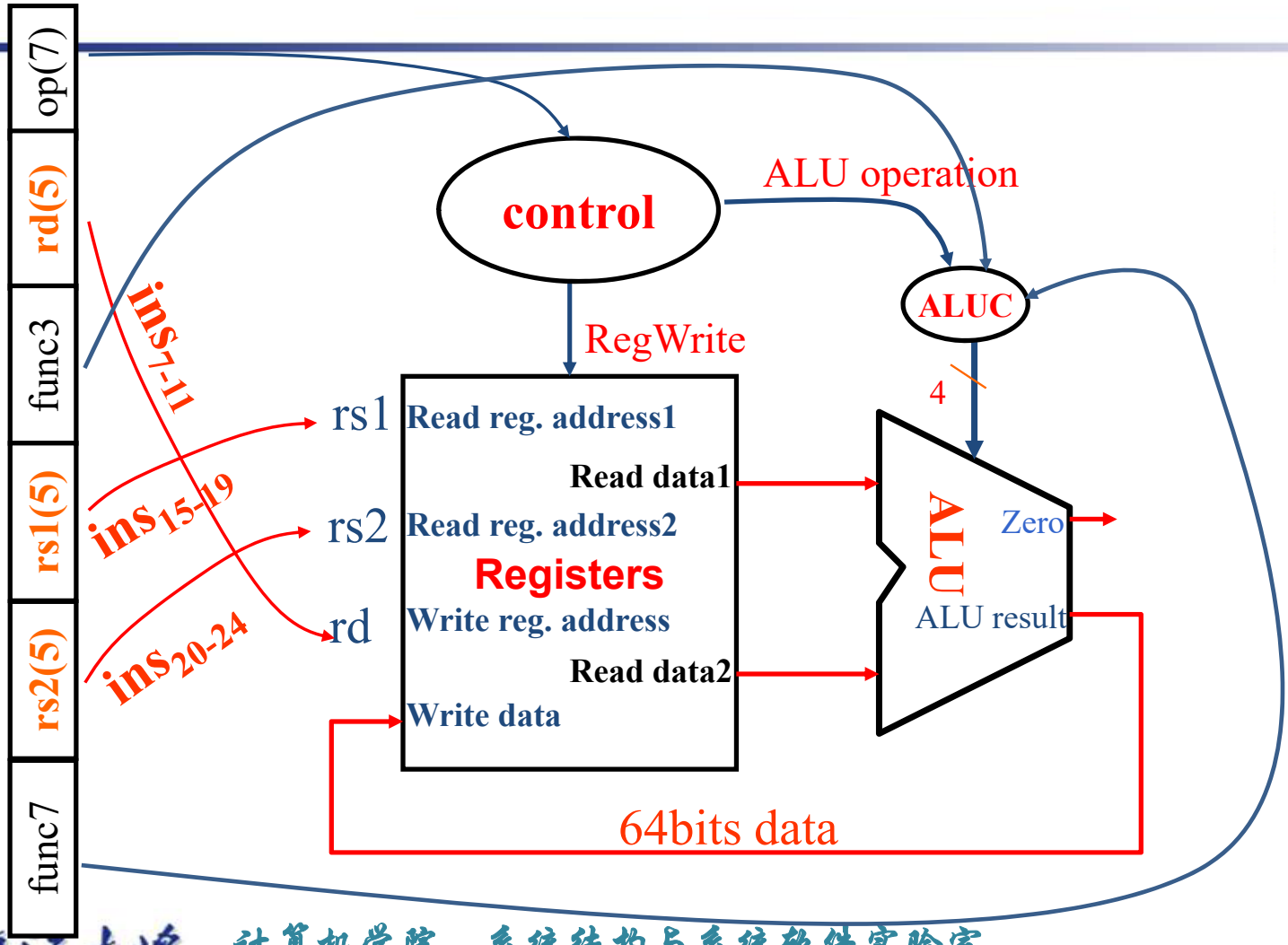
Path Built using Multiplexer

Data Stream of Instruction executing

- R-type instruction Datapath
- I-type instruction Datapath
  - For ALU
  - For load
- S-type (store) instruction Datapath
- SB-type (branch) instruction Datapath
- UJ-type instruction Datapath
  - For Jump

- First, Look at the data flow within instruction execution

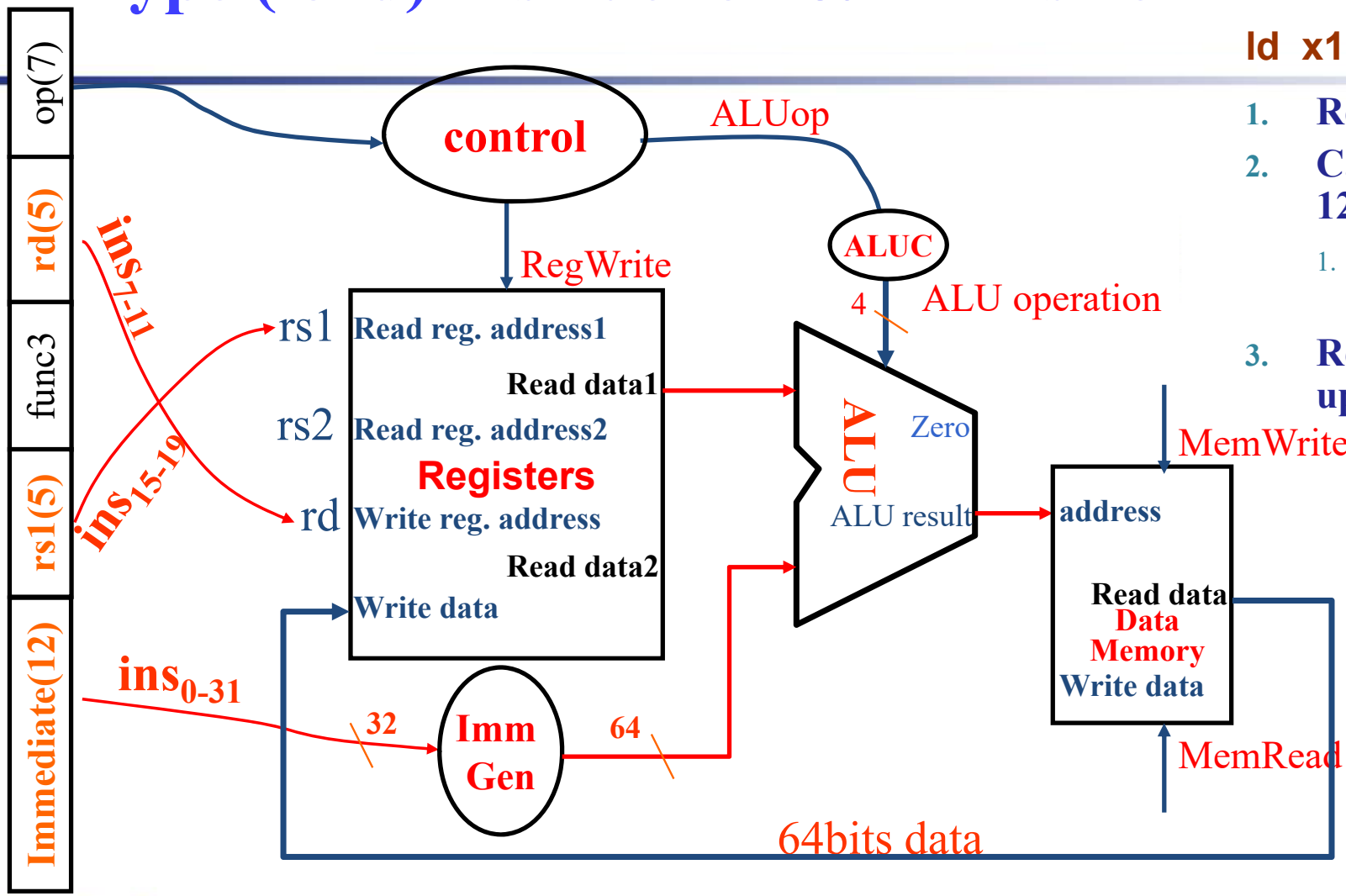
R type Instruction & Data stream



add x9, x20, x21

1. **Read two register operands**
2. **Perform arithmetic/logical operation**
3. **Write register result**

I type (load) Instruction & Data stream



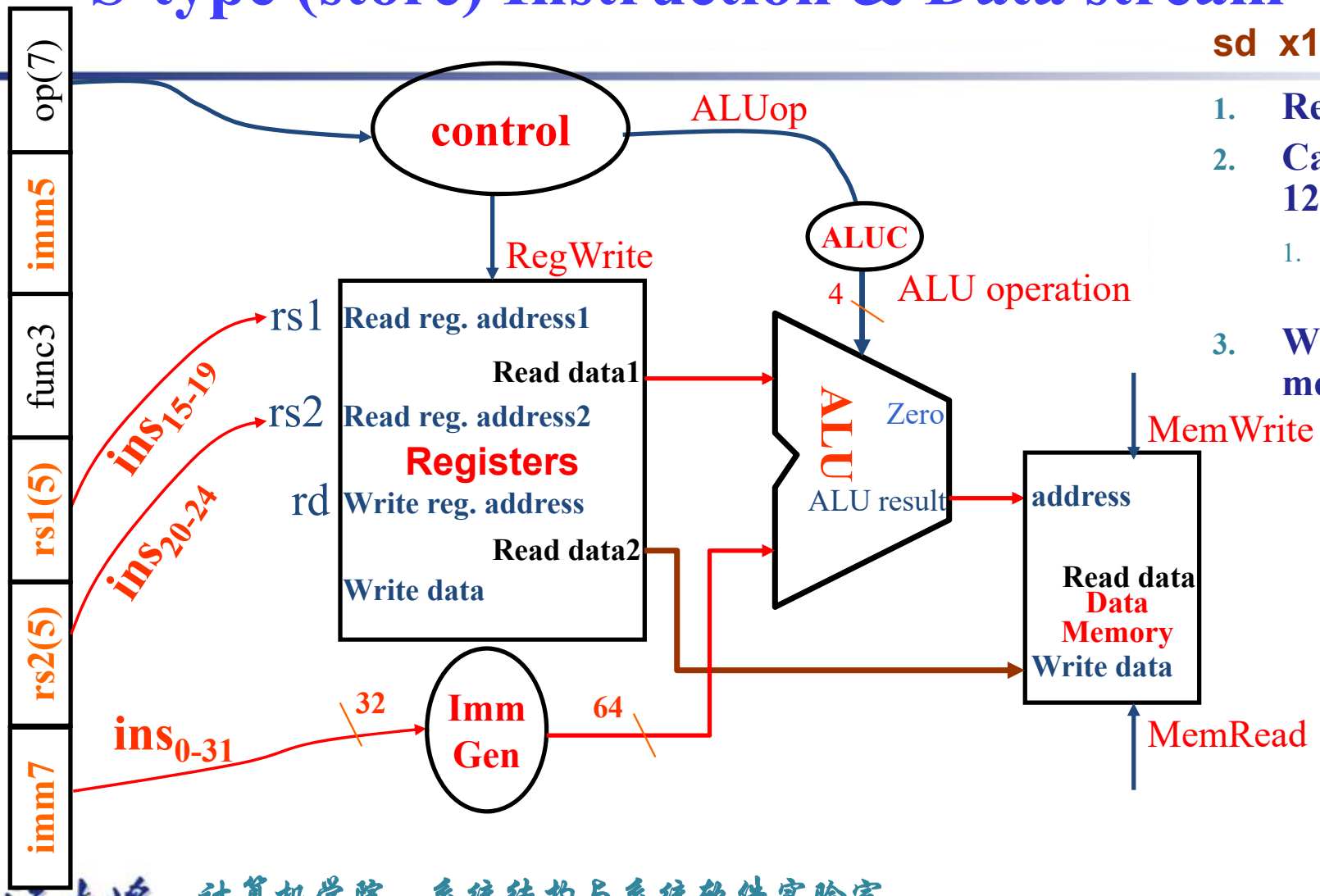
ld x1, 200(x2)

1. Read register operands
2. Calculate address using 12-bit offset
   1. Use ALU, but sign-extend offset
3. Read memory and update register

addi x1, x2, 4?

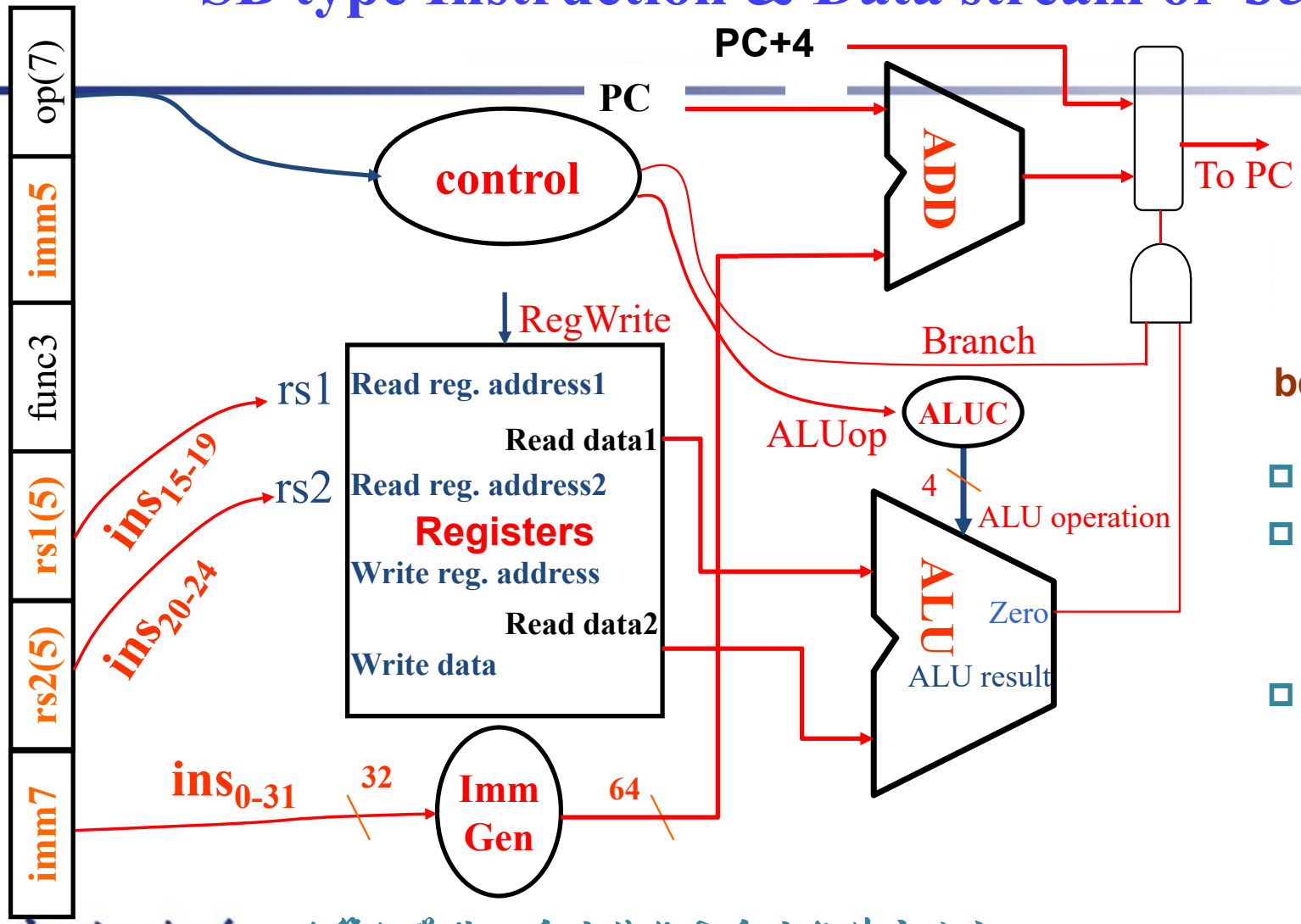
S-type (store) Instruction & Data stream

sd x1, 200(x2)



1. Read register operands
2. Calculate address using 12-bit offset
   1. Use ALU, but sign-extend offset
3. Write register value to memory

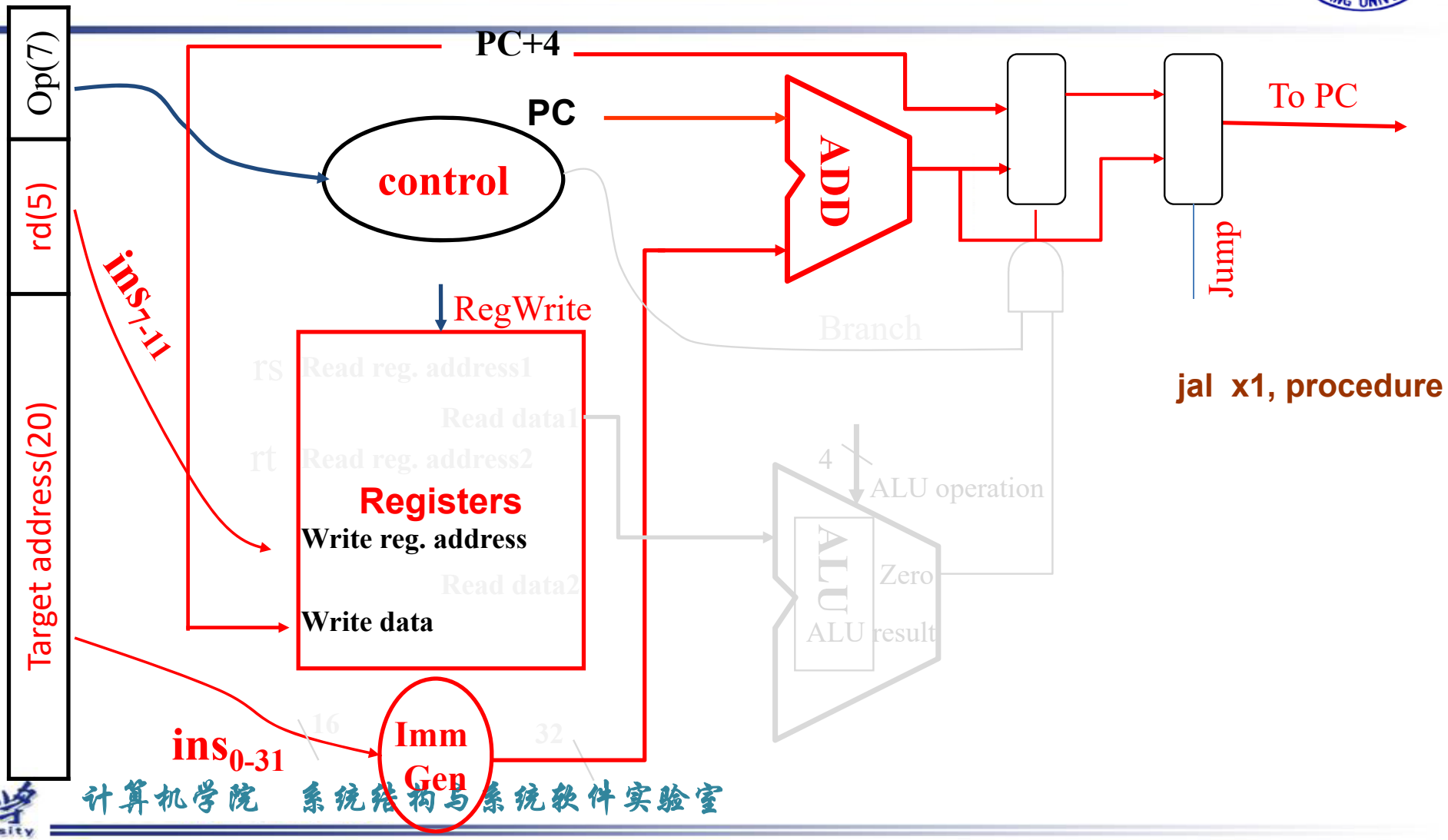
SB type Instruction & Data stream of *beq*



beq x1, x2，200

- **Read register operands**
- **Compare operands**
  - Use ALU, subtract and check Zero output
- **Calculate target address**
  - Sign-extend displacement
  - Shift left 1 place (halfword displacement)
  - Add to PC value

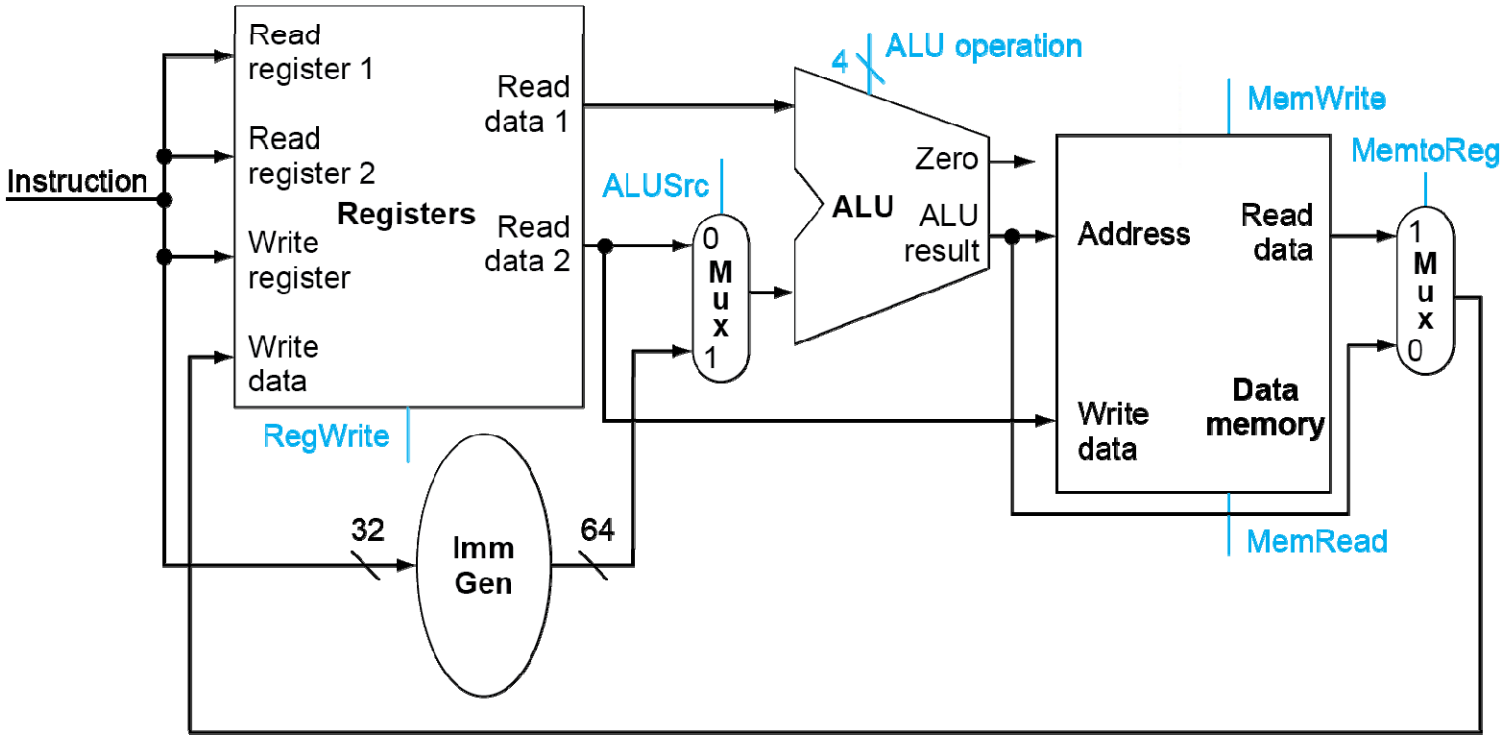
UJ type Instruction



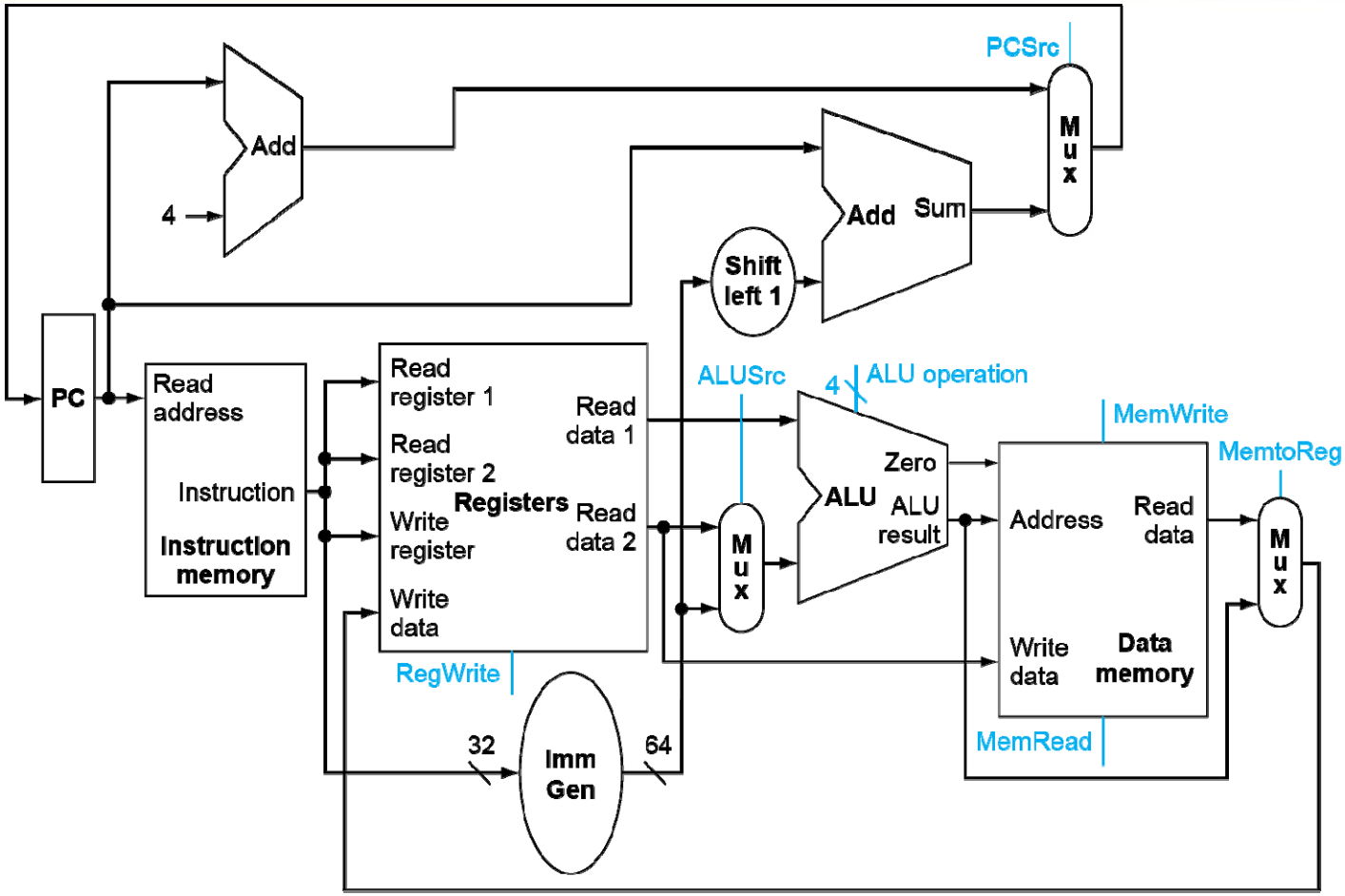
Composing the Elements

- **First-cut data path does an instruction in one clock cycle**
  - Each datapath element can only do one function at a time
  - Hence, we need separate instruction and data memories
- **Use multiplexers where alternate data sources are used for different instructions**

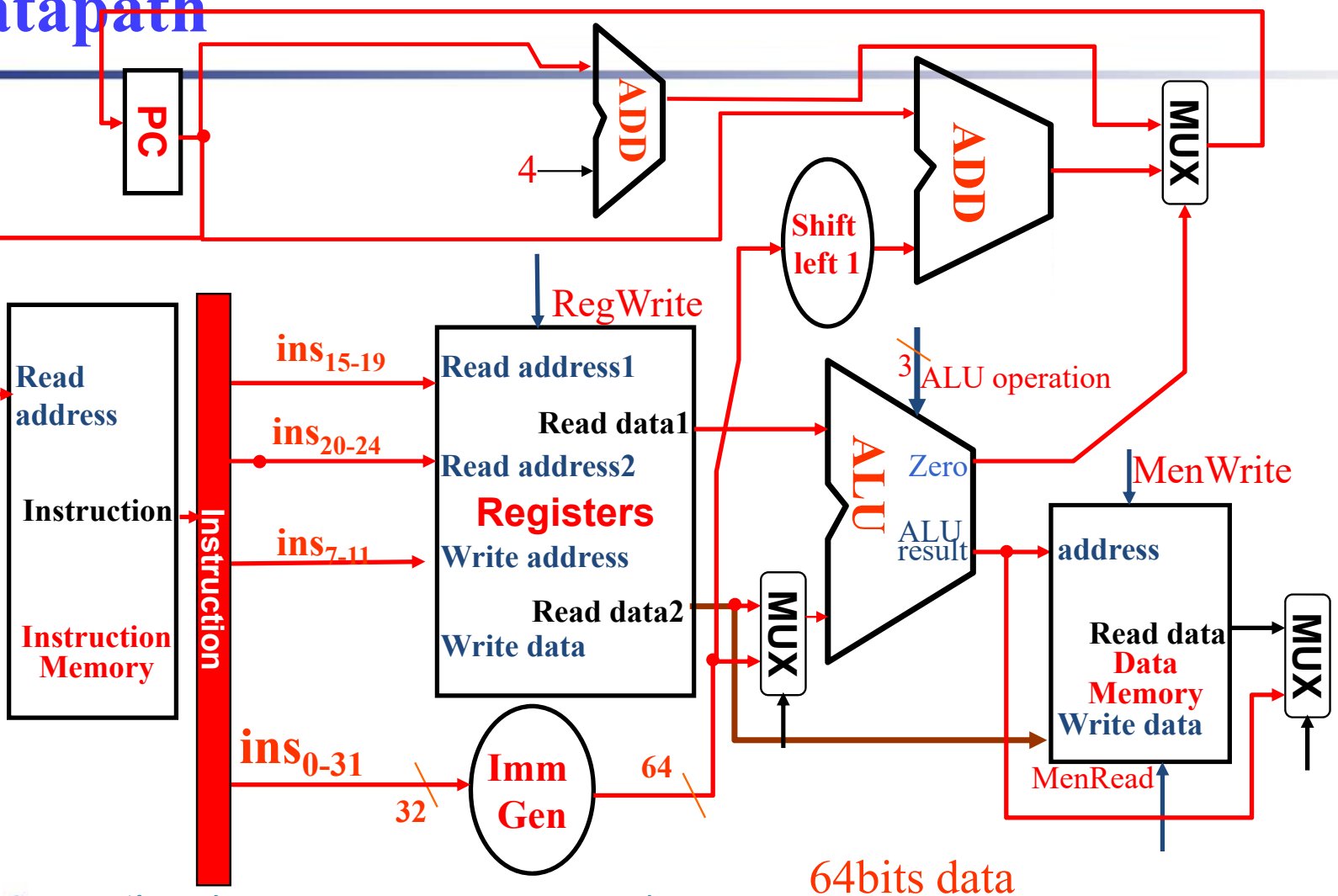
R-Type/Load/Store Datapath



Full Datapath

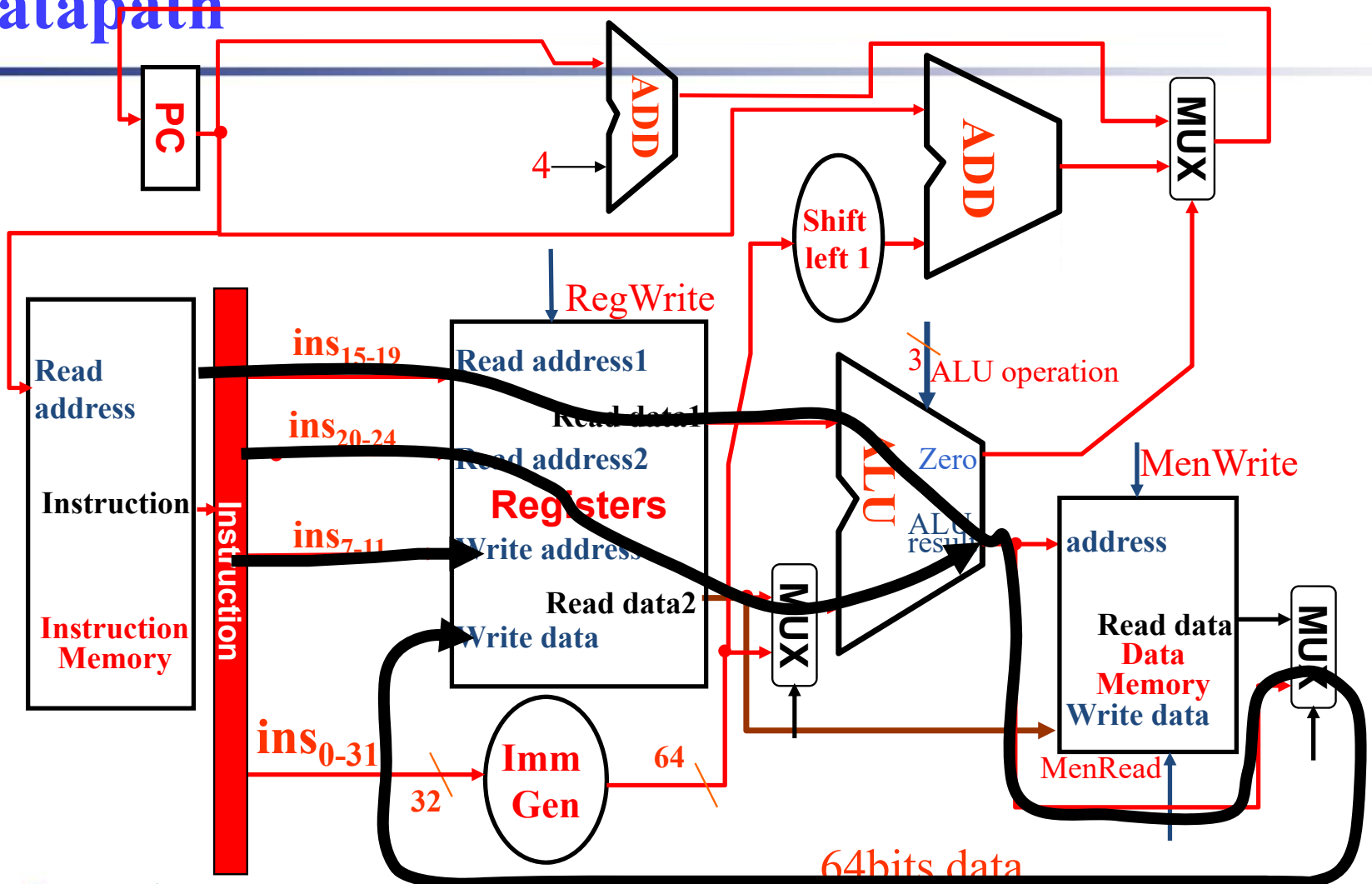


Full datapath



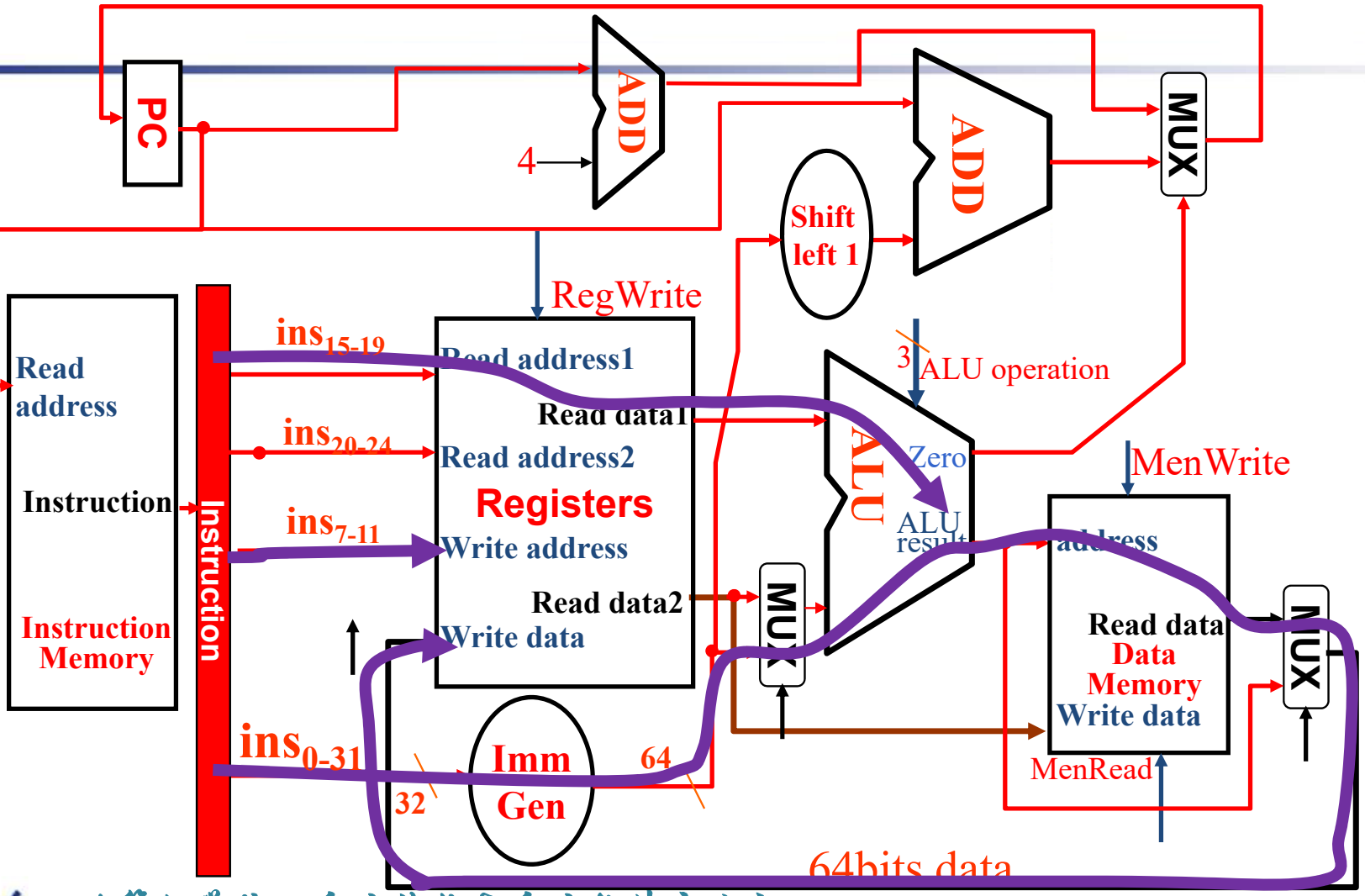
Full datapath

R



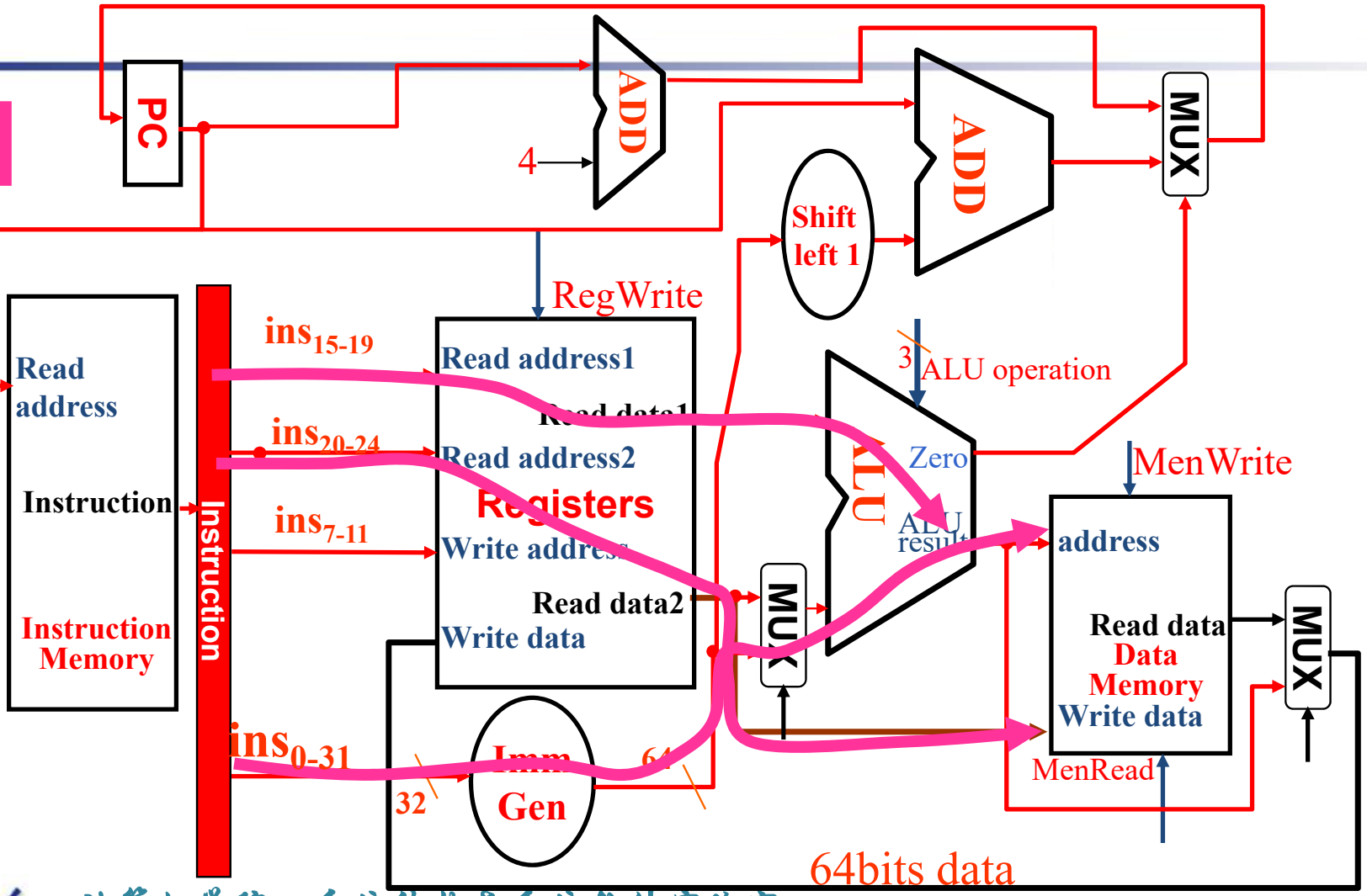
Full datapath

I-ld



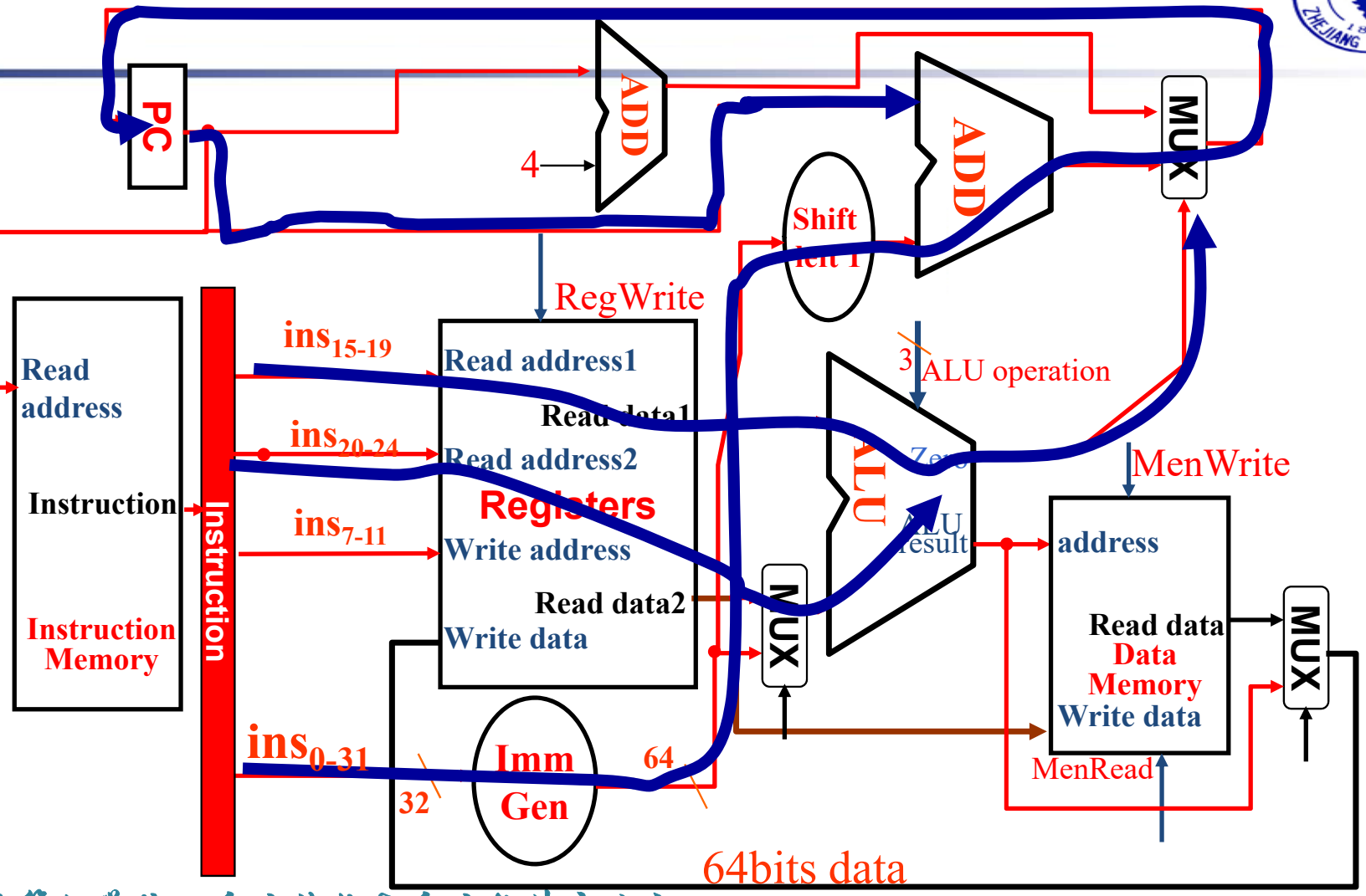
Full datapath

S-sd



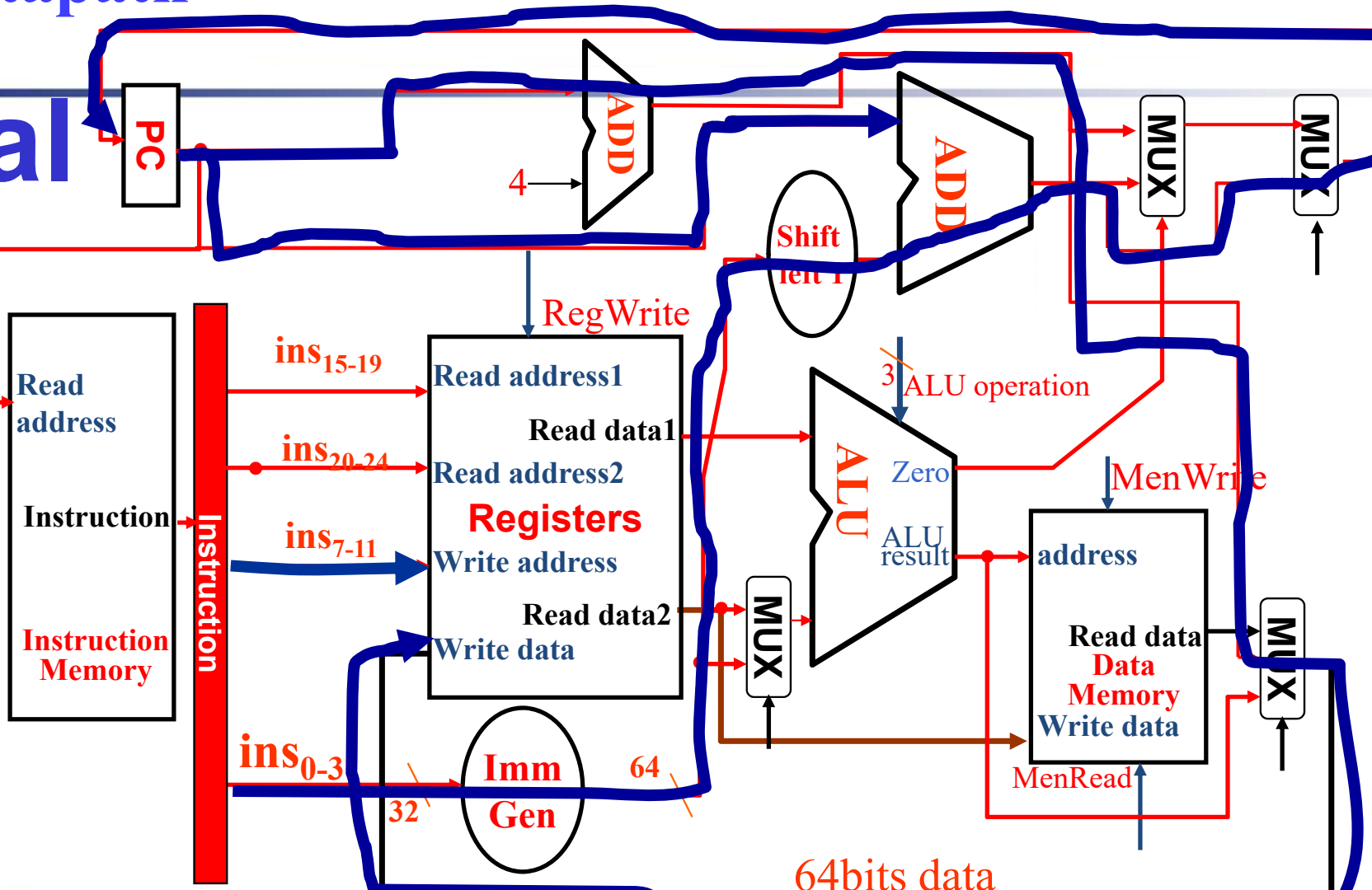
Full datapath

SB-beq

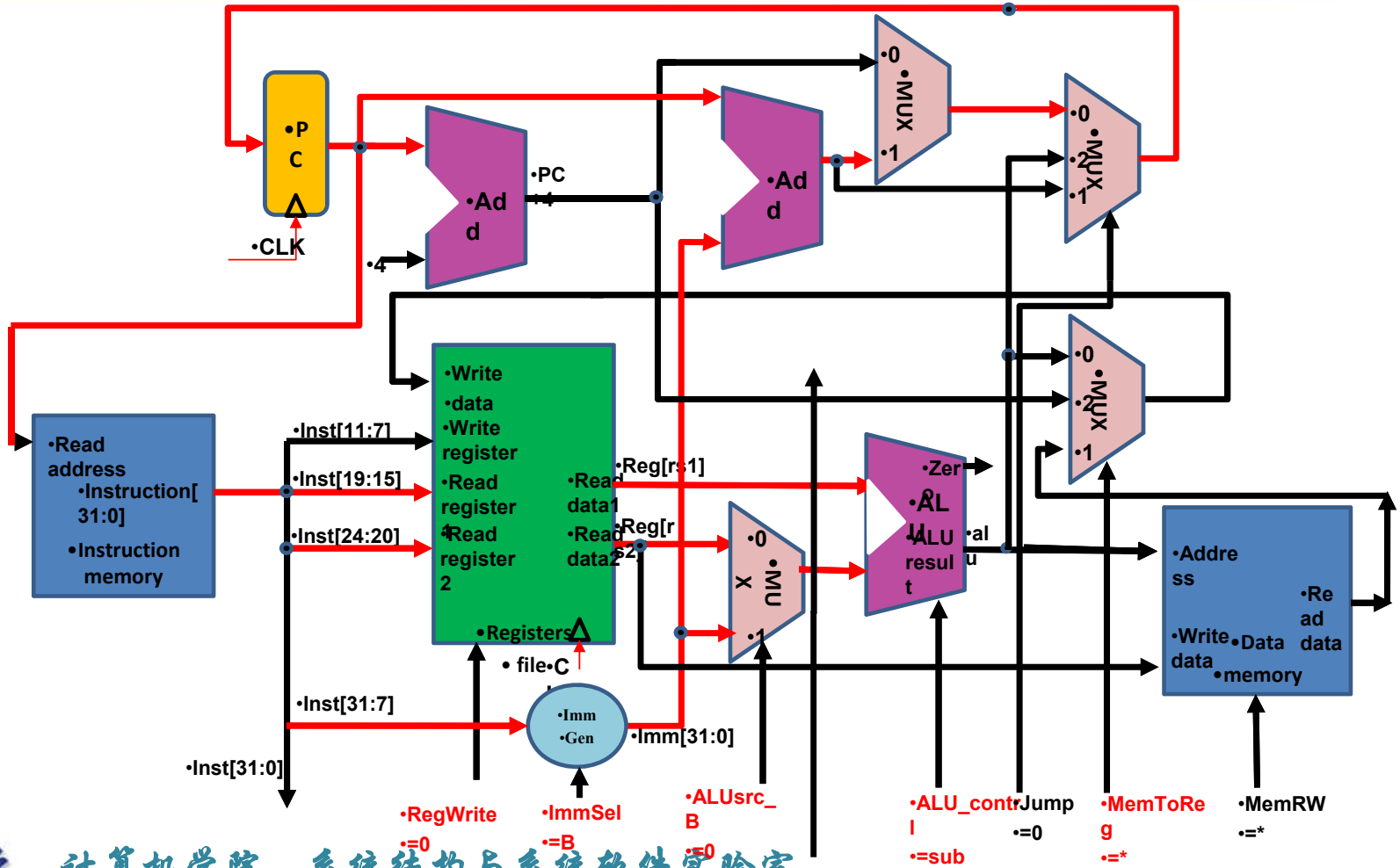


Full datapath

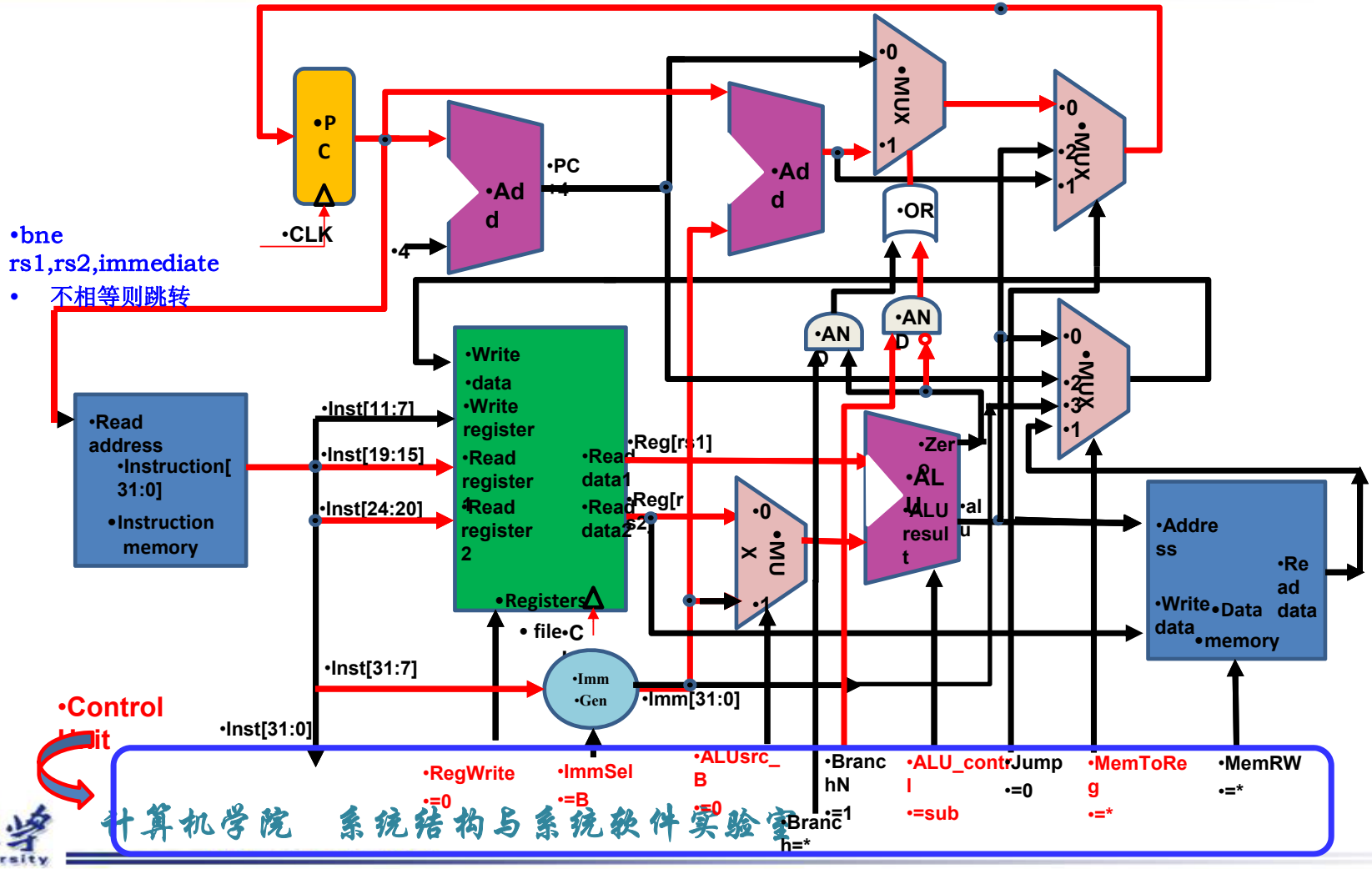
UJ-jal



单周期数据通路结构（add jalr)



单周期数据通路结构 (add lui, bne)

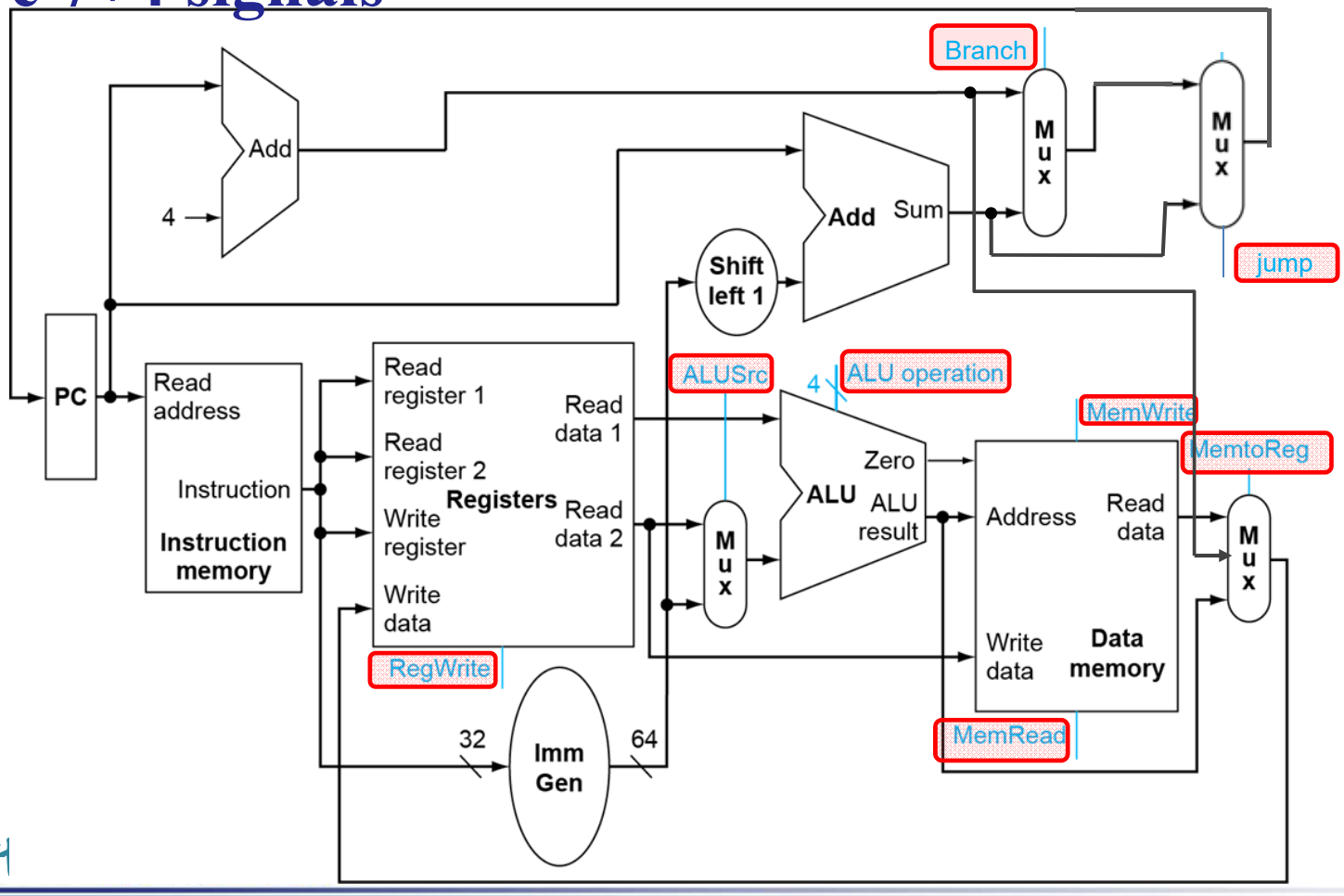


Contents

- Building a datapath
- **A Simple Implementation Scheme**

Building the Datapath & Controller

- There are 7+4 signals



Building Controller

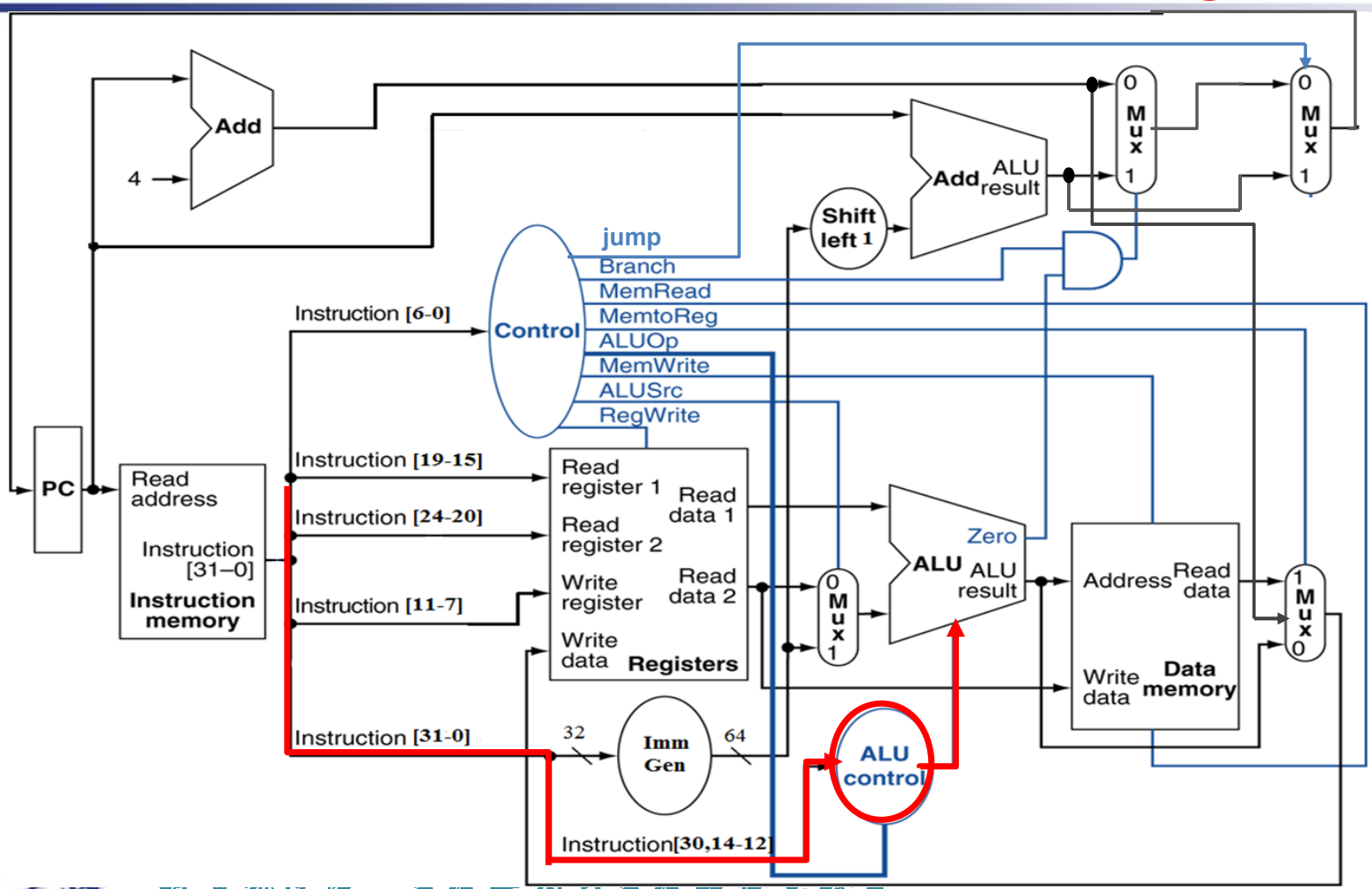
Analyse for cause and effect

- Information comes from the 32 bits of the instruction
- Selecting the operations to perform (ALU, read/write, etc.)
- Controlling the flow of data (multiplexor inputs)
- ALU's operation based on instruction type and function code

Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

The ALU control is where and other signals(7)

Output signals



What should ALU do ?

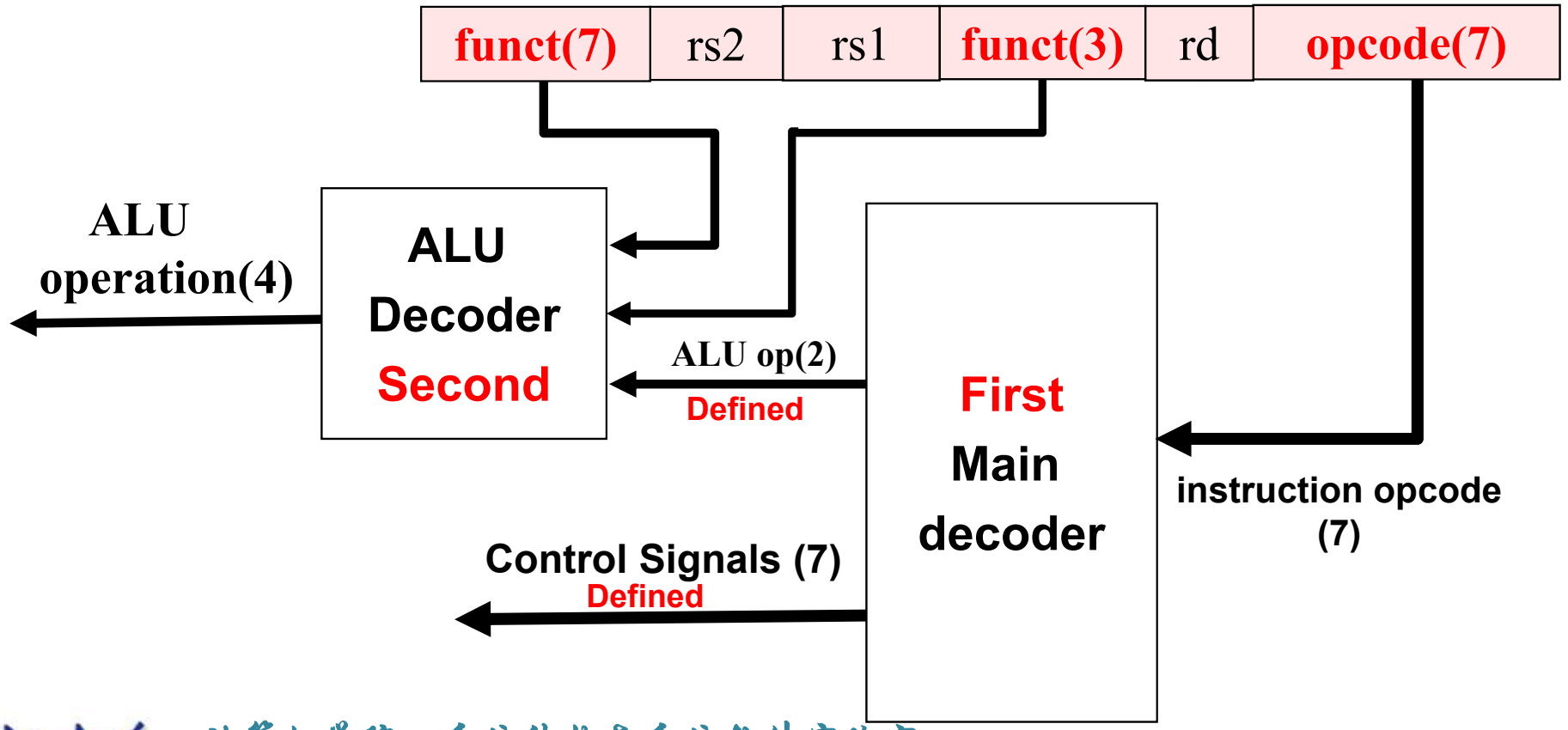
- **ALU used for**
  - Load/Store: F = add
  - Branch: F = subtract
  - R-type: F depends on opcode
- **Assume 2-bit ALUOp derived from opcode**
  - Combinational logic derives ALU control

Operation	Function
000	And
001	Or
010	Add
110	Sub
111	Slt

opcode	ALUOp	Operation	Funct7	funct3	ALU function	ALU control
ld	00	load register	XXXXXXX	xxx	add	0010
sd	00	store register	XXXXXXX	xxx	add	0010
beq	01	branch on equal	XXXXXXX	xxx	subtract	0110
R-type	10	add	0000000	000	add	0010
		subtract	0100000	000	subtract	0110
		AND	0000000	111	AND	0000
		OR	0000000	110	OR	0001
		SLT	0000000	010	Slt	0111

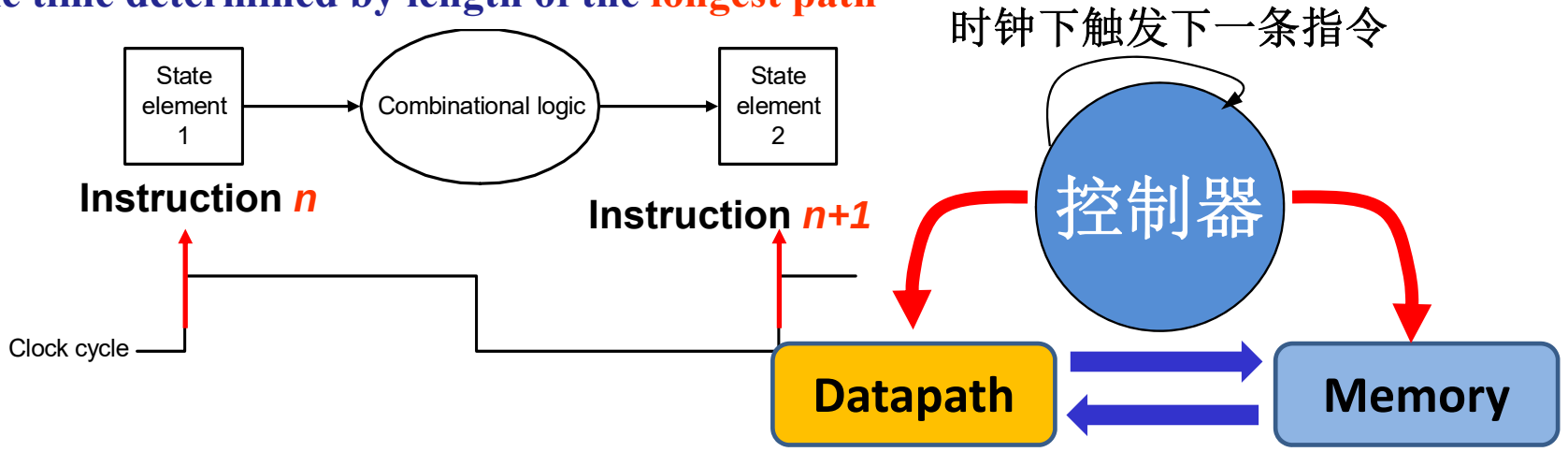
Scheme of Controller

- 2-level decoder



Our Simple Control Structure

- All of the logic is combinational
- We wait for everything to settle down, and the right thing to be done
  - ALU might not produce right answer? right away
  - we use write signals along with clock to determine when to write
- Cycle time determined by length of the longest path

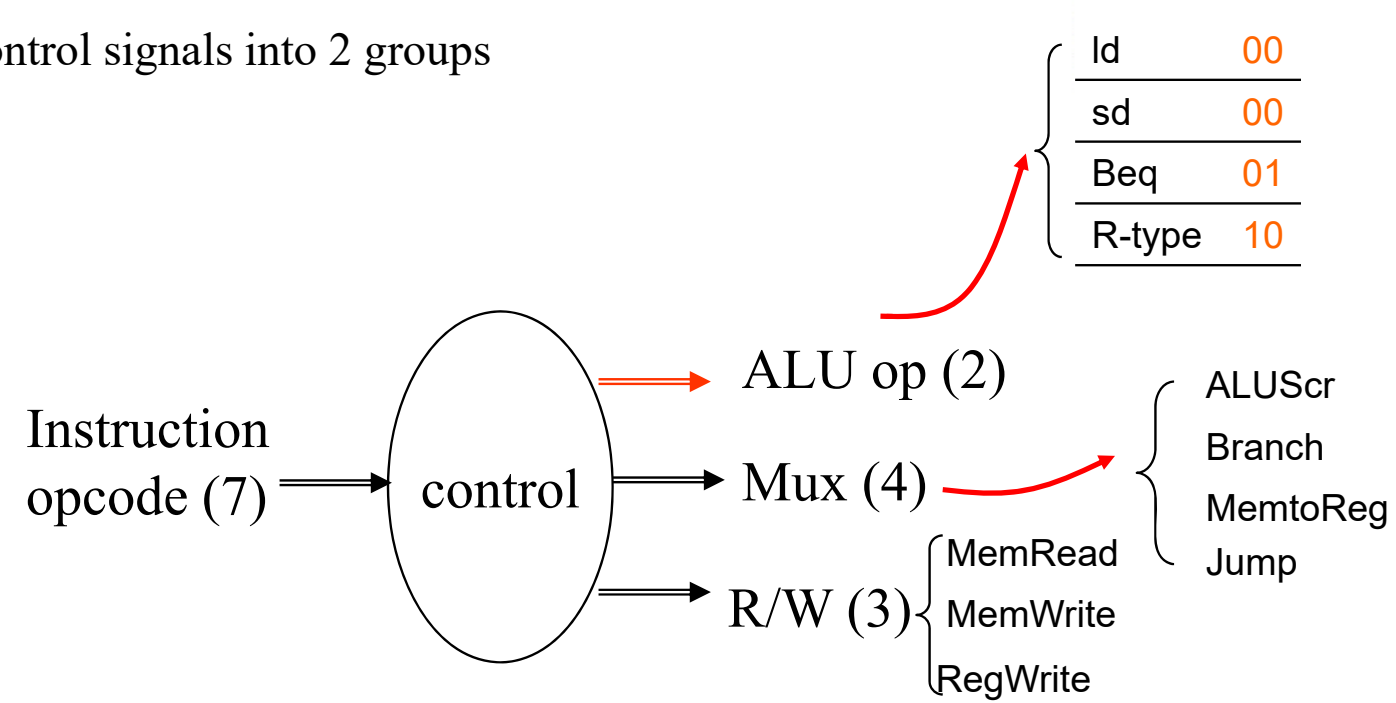


We are ignoring some details like setup and hold times

Designing the Main Control Unit

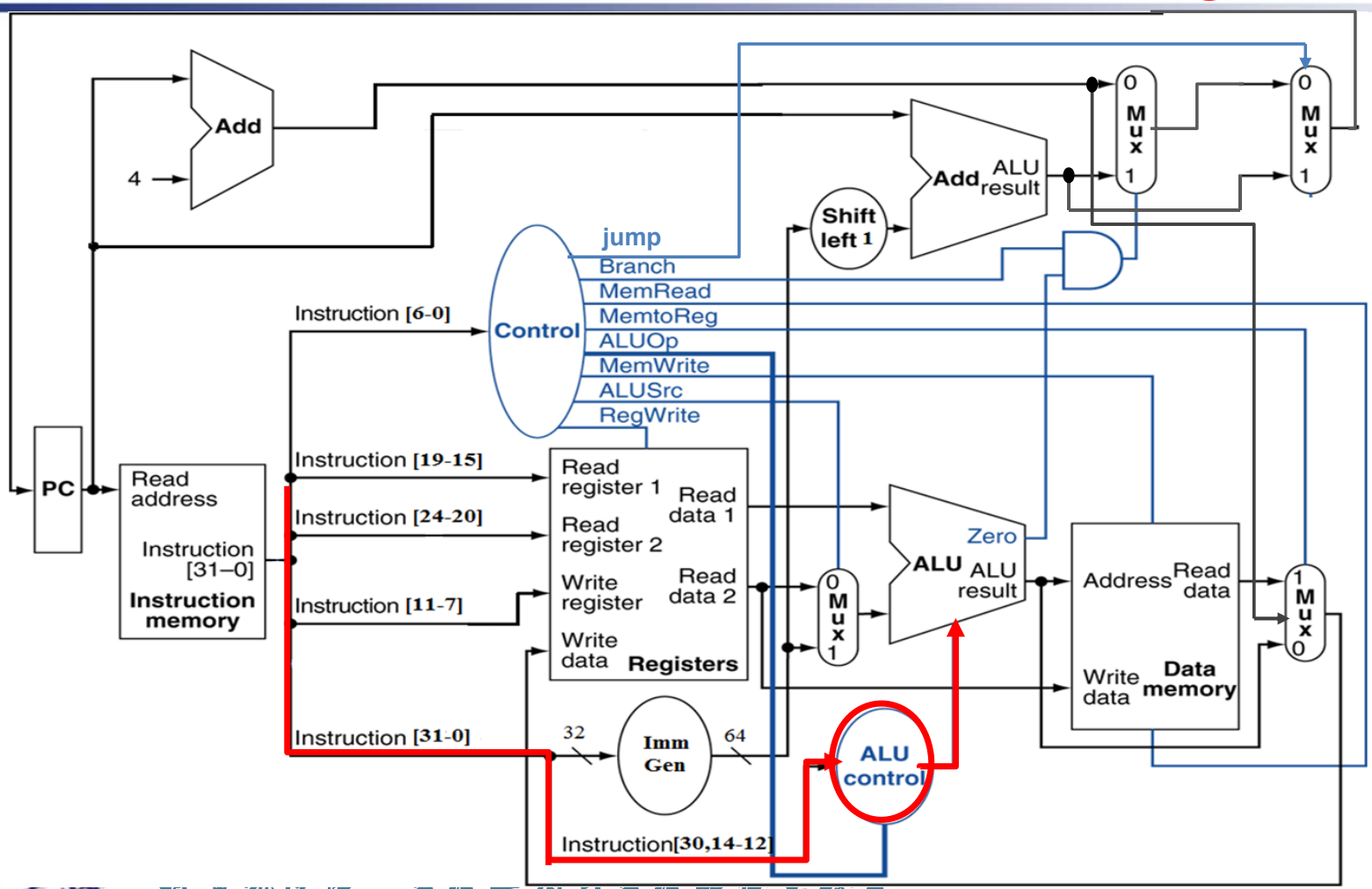
First level

- **Main Control Unit function**
  - ALU op (2)
  - Divided 7 control signals into 2 groups
    - 4 Mux
    - 3 R/W



The ALU control is where and other signals(7)

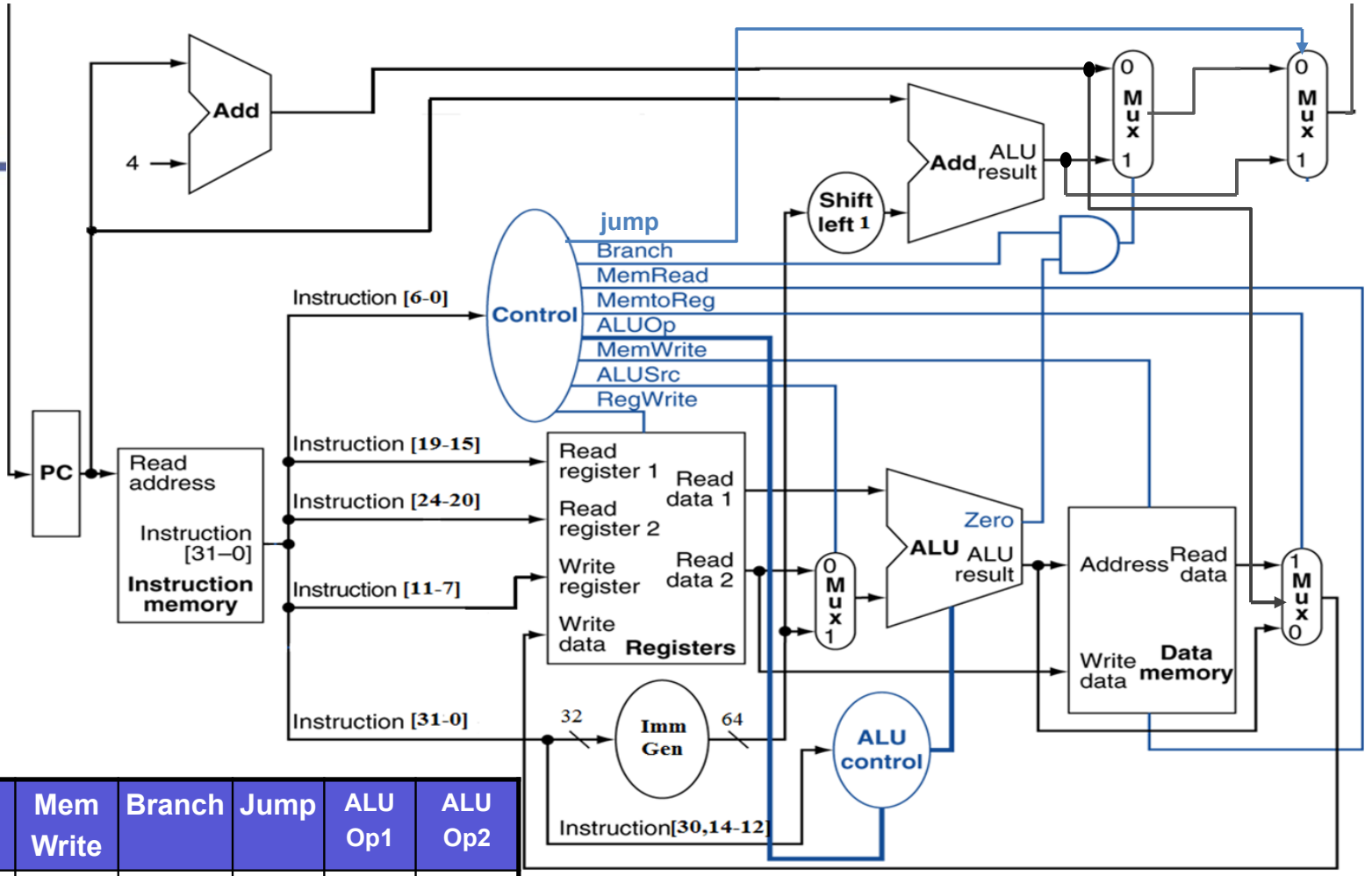
Output signals



Signals for datapath Defined 7 control signals

Signal name	Effect when deasserted(=0)	Effect when asserted(=1)
RegWrite	None	Register destination input is written with the value on the Write data input
ALUScr	The second ALU operand come from the second register file output (Read data 2)	The second ALU operand comes from the output of the Immediate Generator
Branch（PCSrc）	The PC is replaced by the output of the adder that computers the value PC+4	The PC is replaced by the output of the adder that computers the branch target.
Jump	The PC is replaced by PC+4 or branch target	The PC is updated by jump address computed by adder
MemRead	None	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None	Data memory contents designated by the address input are replaced by value on the Write data input.
MemtoReg (2位)	00：The value fed to register Write data input comes from the Alu	01：The value fed to the register Write data input comes from the data memory.
		10：The value fed to the register Write data input comes from PC+4

Truth Table for Main decoder



Instruction	ALUSrc	Memto Reg	Reg Write	Mem Read	Mem Write	Branch	Jump	ALU Op1	ALU Op2
R-format									
Ld									
Sd									
beq									
Jal									

Truth tables & Circuitry of main Controller

输入		输出								
Instruction	OPCode	ALUSrcB	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	Jump	ALU Op1	ALU Op0
R-format	0110011	0	00	1	0	0	0	0	1	0
Ld(I-Type)	0000011	1	01	1	1	0	0	0	0	0
sd(S-Type)	0100011	1	X	0	0	1	0	0	0	0
beq(SB-Type)	1100111	0	X	0	0	0	1	0	0	1
Jal(UJ-Type)	1101111	X	10	1	0	0	0	1	X	X

Main Controller Code

指令译码器参考描述

```
`define CPU_ctrl_signals {ALUSrc_B,MemtoReg,RegWR,MemWrite,Branch,Jump,ALUop}
        always @* begin
                case(OPcode)
                5‘b01100: begin CPU_ctrl_signals = ?;  end        //ALU
                5’b00000: begin CPU_ctrl_signals = ?;   end       //load
                5’b01000: begin CPU_ctrl_signals = ?;   end       //store
                5’b11000: begin CPU_ctrl_signals = ?;   end       //beq
                5’b11011: begin CPU_ctrl_signals = ?;   end       //jump
                5’b00100: begin CPU_ctrl_signals = ?;   end       //ALU(addi;;;;)
                                        ……
                default:     begin  CPU_ctrl_signals = ?;  end
                endcase
        end
```

Design the ALU Decoder second level

- **ALU operation is decided by 2-bit ALUOp derived from opcode, and funct7 & funct3 fields of the instruction**
  - Combinational logic derives ALU control

opcode	ALUOp	Operation	Funct7	funct3	ALU function	ALU control
ld	00	load register	XXXXXXX	xxx	add	0010
sd	00	store register	XXXXXXX	xxx	add	0010
beq	01	branch on equal	XXXXXXX	xxx	subtract	0110
R-type	10	add	0000000	000	add	0010
		subtract	0100000	000	subtract	0110
		AND	0000000	111	AND	0000
		OR	0000000	110	OR	0001
		SLT	0000000	010	Slt	0111

ALU Controller Code

- **ALU Control HDL Description**

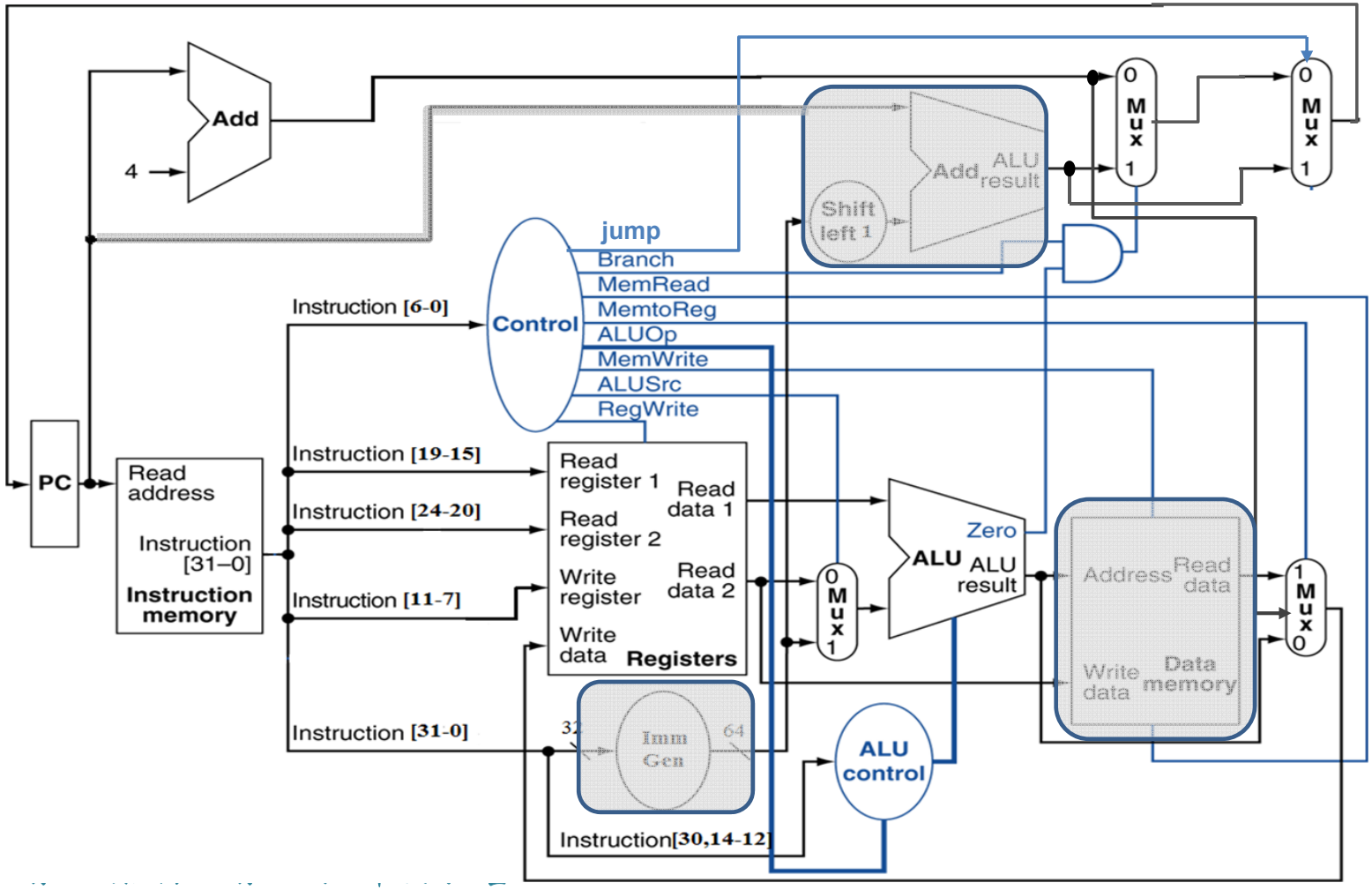
```
assign Fun = {Fun3,Fun7};
always @* begin
    case(ALUop)
    2’b00: ALU_Control = ? ;                    //add计算地址
    2’b01: ALU_Control = ? ;                    //sub比较条件
    2’b10:
        case(Fun)
            4'b0000: ALU_Control = 3'b010 ;     //add
            4’b0001: ALU_Control = ? ;  //sub
            4’b1110: ALU_Control = ? ;  //and
            4’b1100: ALU_Control = ? ;  //or
            4’b0100: ALU_Control = ? ;  //slt
            4’b1010: ALU_Control = ? ;  //srl
            4’b1000: ALU_Control = ? ;  //xor
                    ……
            default:   ALU_Control=3'bx;
            endcase
    2’b11:
        case(Fun3)
                ……
    endcase
```

R-Type Instruc

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

add x9, x20, x21

1. Read two register operands
2. Perform arithmetic/logical operation
3. Write register result

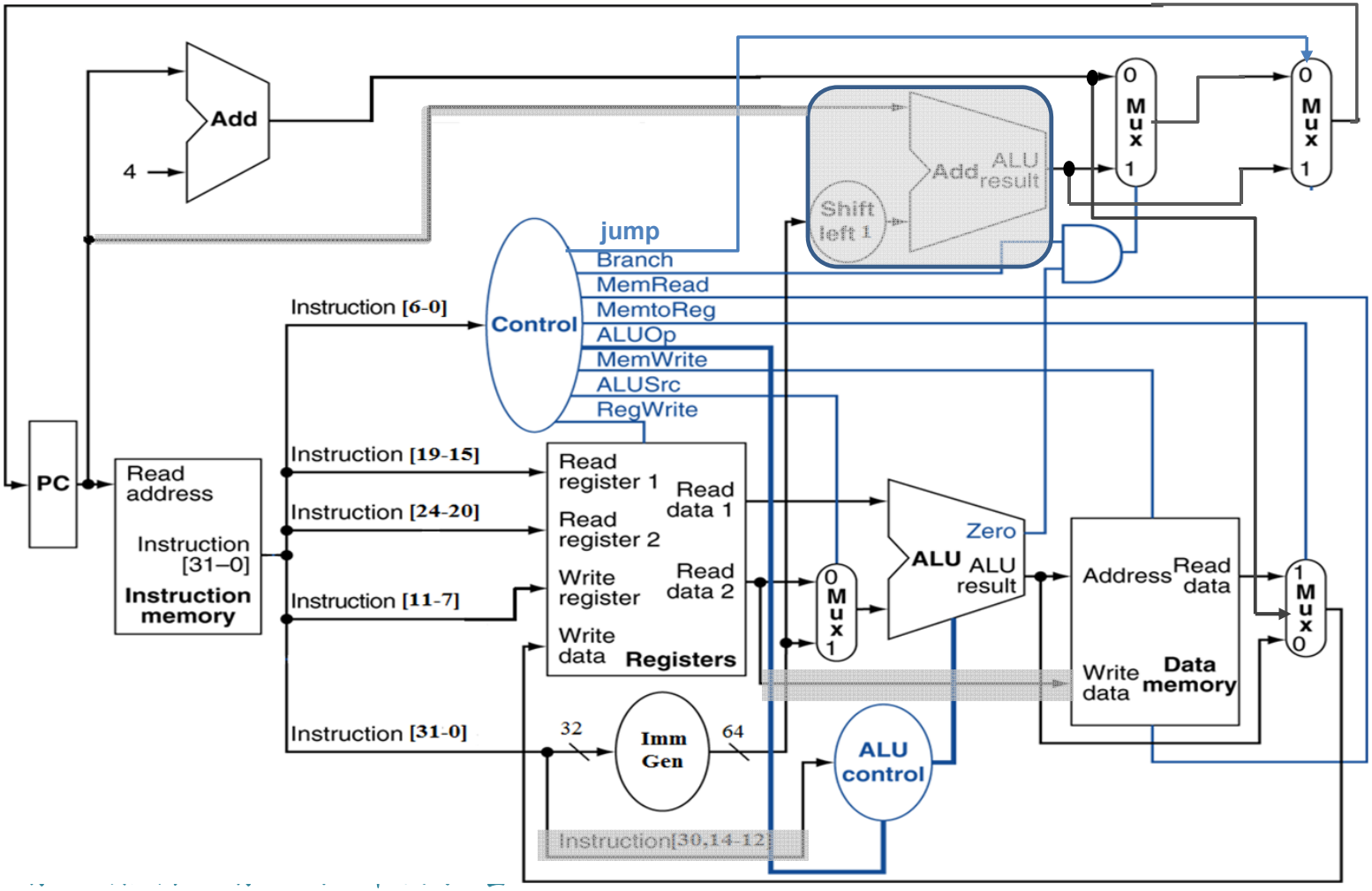


Load Instruction

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

ld x1, 200(x2)

1. **Read register operands**
2. **Calculate address using 12-bit offset**
   1. Use ALU, but sign-extend offset
3. **Read memory and update register**

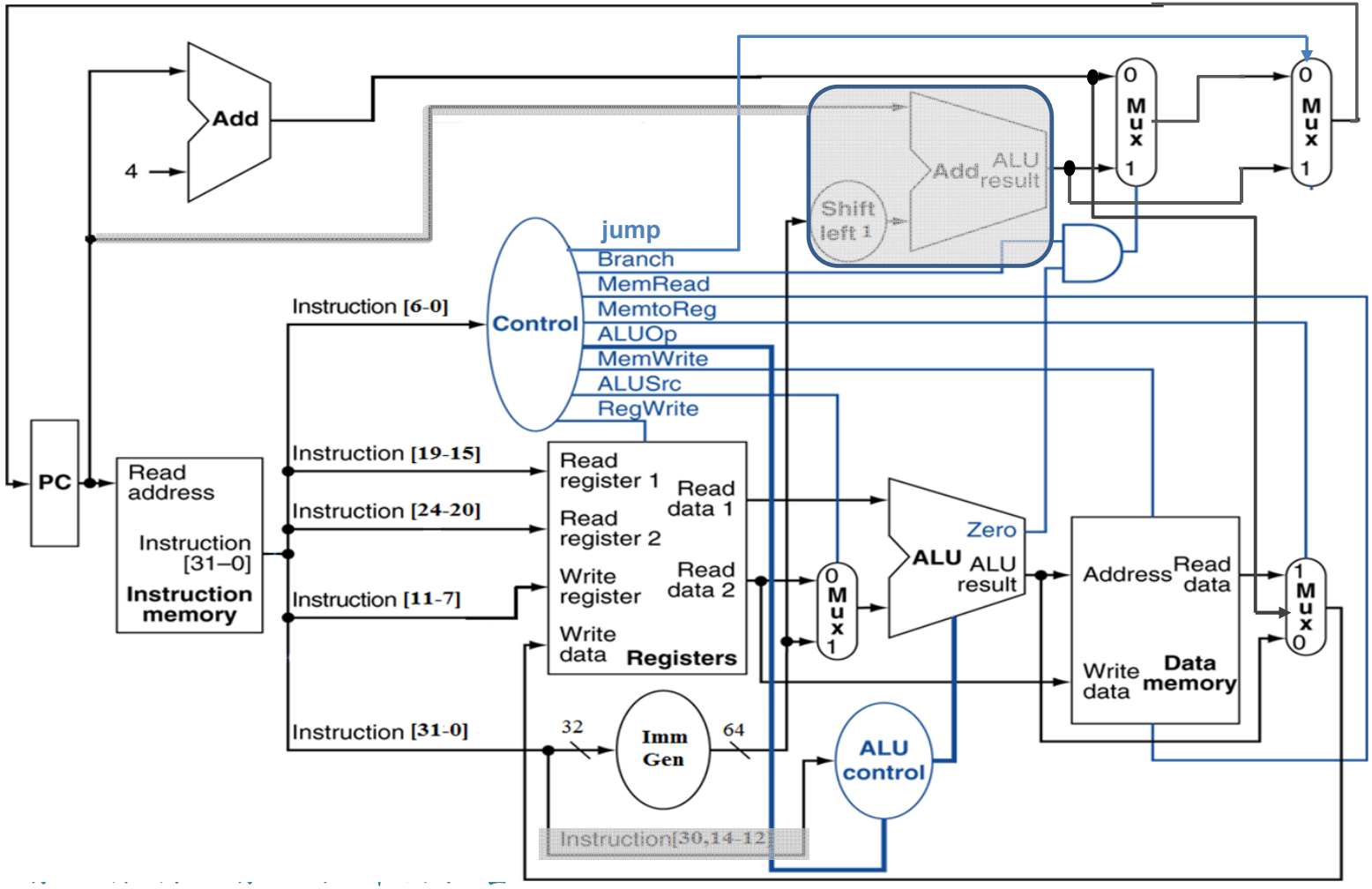


Save Instructi

immediate[11:5]	rs2	rs1	funct3	immediate[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

sd x1, 200(x2)

1. **Read register operands**
2. **Calculate address using 12-bit offset**
   1. Use ALU, but sign-extend offset
3. **Write register value to memory**

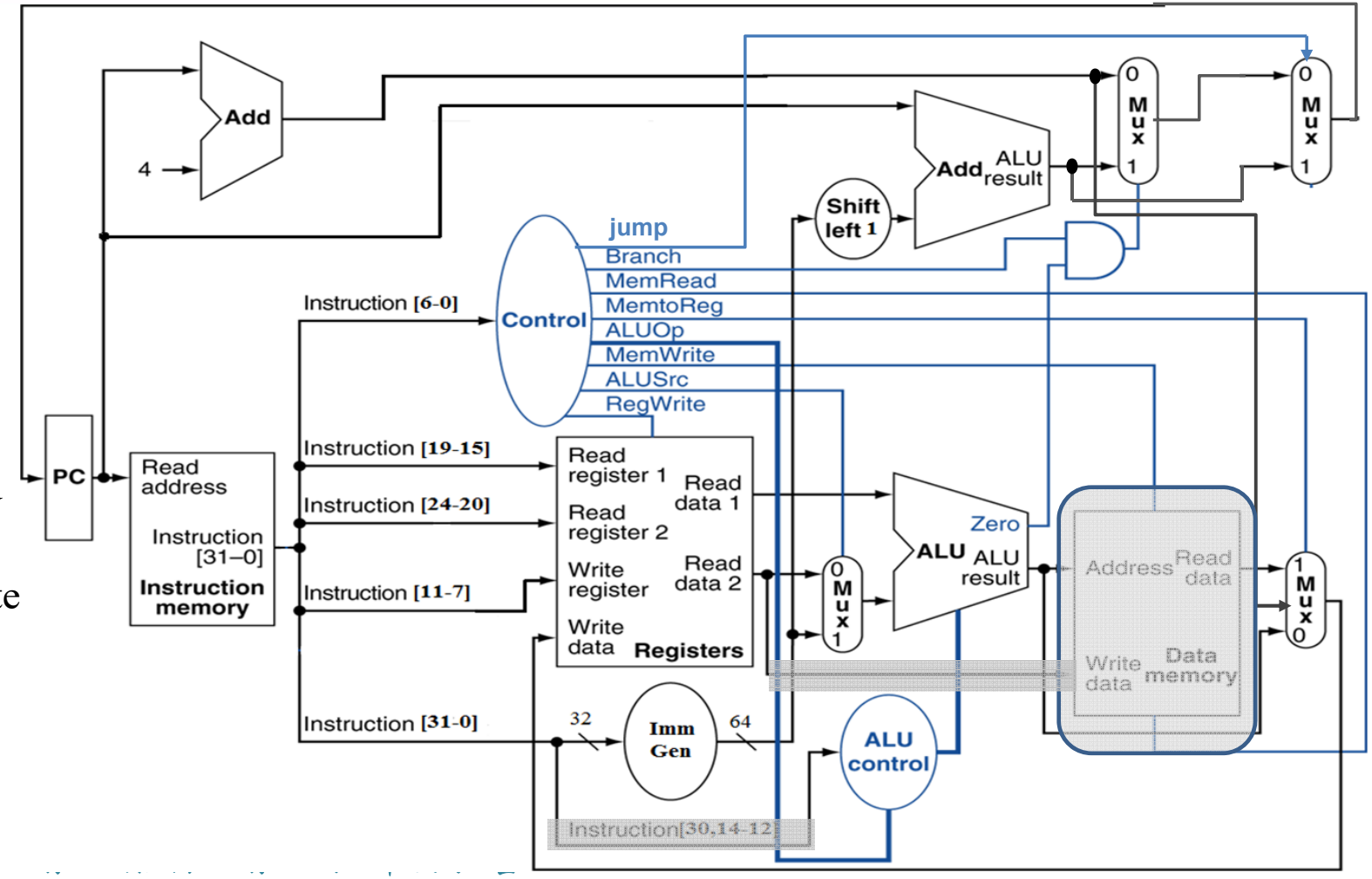


BEQ Instructio

imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode

beq x1, x2，200

- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 1 place (halfword displacement)
  - Add to PC value and update PC

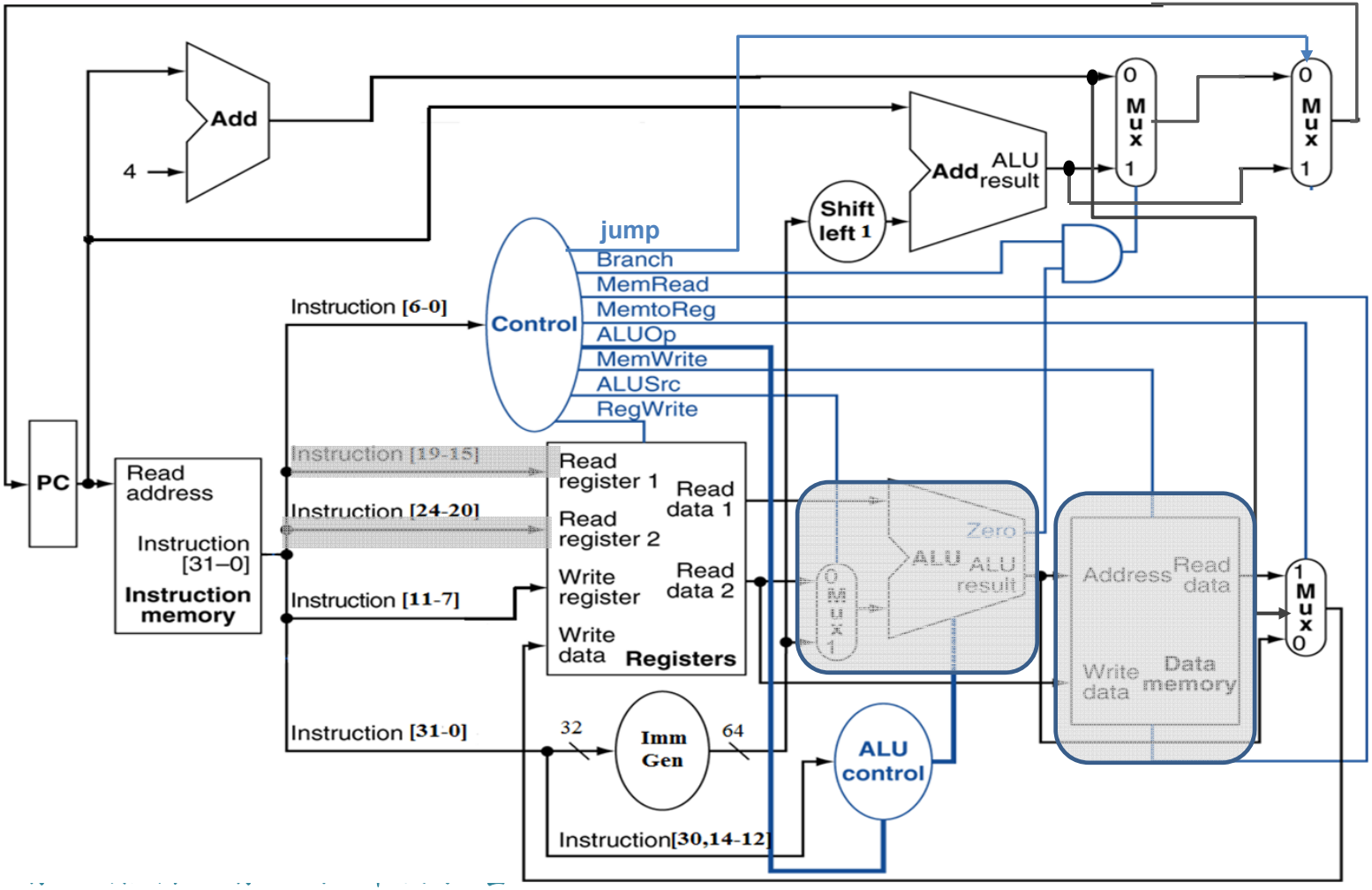


Jal Instruction

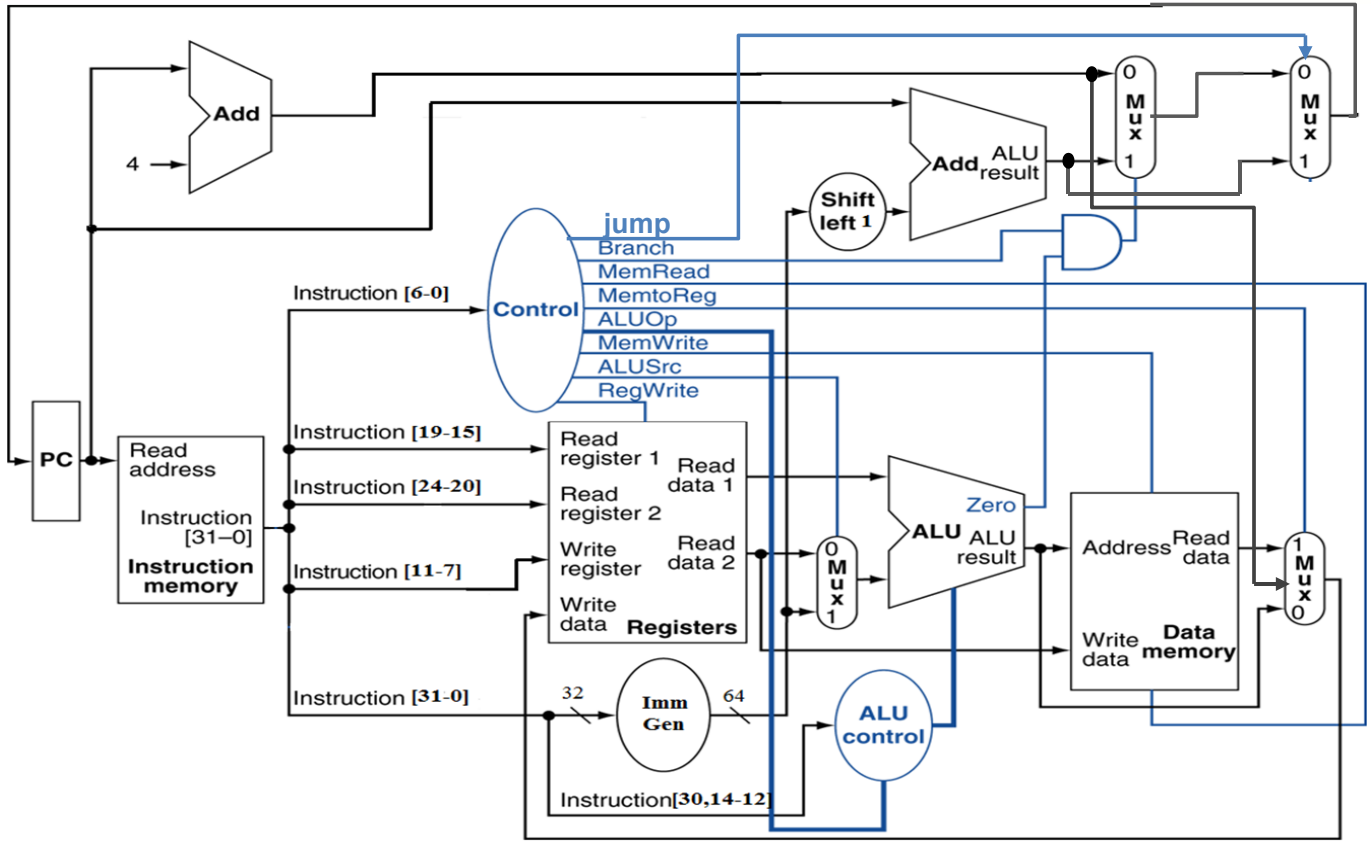
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode

jal x1, procedure

- Write PC+4 to rd
- Calculate target address
  - Sign-extend displacement
  - Shift left 1 place (halfword displacement)
  - Add to PC value and update PC



Single Cycle Implementation performance for lw



200ps 100+100=200ps 200ps 200ps

- Calculate cycle time assuming negligible delays except:
  - memory (200ps), ALU and adders (200ps), register file access (100ps)

Performance in Single Cycle Implementation

- **Let's see the following table:**

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
ld	200ps	100 ps	200ps	200ps	100 ps	800ps
sd	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

- **The conclusion:**

  Different instructions needs different time.

  The clock cycle must meet the need of the slowest instruction.

  So, some time will be wasted.

Performance Issues

- **Longest delay determines clock period**
  - Critical path: load instruction
  - Instruction memory → register file → ALU → data memory → register file
- **Wasteful of area. If the instruction needs to use some functional unit multiple times.**
  - E.g., the instruction 'mult' needs to use the ALU repeatedly. So, the CPU will be very large.
- **Violates design principle**
  - Making the common case fast
- **We will improve performance by pipelining**


END