Computer Architecture
计 算 机 体 系 结 构

Lecture 2. Instruction Set Architecture
第二讲、指令集架构

Chao Li, PhD.
李超 博士

SJTU-SE346, Spring 2019

Review

- ENIAC (1946) used decimal representation; 10 vacuum tubes per digit; could store 20 numbers of 10 digits each
- From a transistor to integrated circuit
- The feature size of a fabrication process : minimum lateral dimensions of the transistors. Moore’s Law revisited
- From a single server to a data center
- Scalability issue (scale out/up). Exponential growth of computing vs. Skyrocketing power/energy demand

Outlines

- Instruction Set Architecture
- ISA: Programmer's View
- ISA: Microarchitecture Level Design
  – Simple Pipeline Implementation

Aspects of Computer Design

- **Architecture** (instruction set architecture)
  - Compiler designer's view: functional appearance (visible state) to its immediate user or programmer
- **Implementation** (micro-architecture)
  - CPU designer's view: logical structure or organization (e.g., pipeline design) that implements the architecture
- **Physical Design** (chip realization)
  - IC designer's view: physical structure (with design optimization) the embodies the implementation

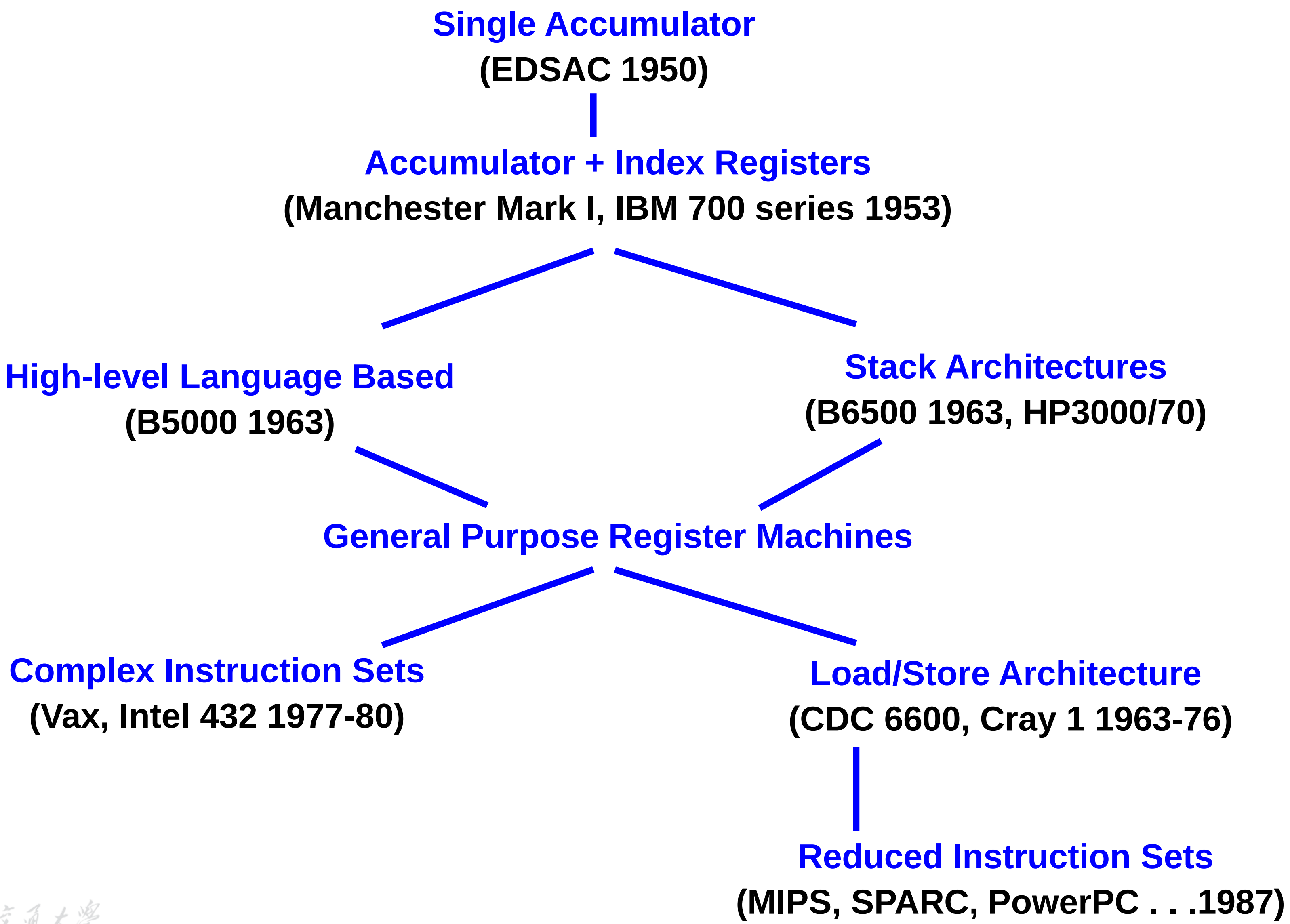
Instruction Set Architecture

- Basic view of ISA: **it defines data and control flow**
  - Storage resources that hold data (e.g., memory and their addressing)
  - Instructions that transform data (e.g., arithmetic/logic, floating point)

- Another view of ISA:
  - Defines set of programmer visible state
  - Defines instruction format and semantics

- Many possible implementations exist for a single ISA
  - IBM 360 implementations: model 30 (1964), z900 (2001)
  - x86 implementations: 8086 (1978), 80186, 286, 386, ...
  - MIPS implementations: R2000, R4000, R10000, …

Architecture Evolution


Single Accumulator
(EDSAC 1950)
Accumulator + Index Registers
(Manchester Mark I, IBM 700 series 1953)
High-level Language Based
(B5000 1963)
Stack Architectures
(B6500 1963, HP3000/70)
General Purpose Register Machines
Complex Instruction Sets
(Vax, Intel 432 1977-80)
Load/Store Architecture
(CDC 6600, Cray 1 1963-76)
Reduced Instruction Sets
(MIPS, SPARC, PowerPC . . .1987)

Four Classic Architectures

Stack

Operations that perform $i{+}{+}$

```
PUSH   &i ;
PUSH   0x01;
ADD;
```

Accumulator

Operations that perform $i{+}{+}$

```
LDA    &i ;
ADDA   0x01;
STA    &i;
```

Register-Memory

Operations that perform $i{+}{+}$

```
MOVE   d0, &i ;
INC    d0;
MOVE   &i, d0;
```

Register-Oriented

Operations that perform $i{+}{+}$

```
LD     r7, &i ;
ADD    r7, 0x01, r7;
ST     r7, &i;
```

Complex Instruction Set Computer (CISC)

- Dominant style through mid-80’s
- Easy for compiler, fewer code bytes
- Stack-oriented instruction set
  - Use stack to pass arguments, save program counter
  - Explicit push and pop instructions
- Register memory architecture:
  - Arithmetic instructions can access memory
- Condition codes:
  - Set as side effect of arithmetic and logical instructions

Reduced Instruction Set Computer (RISC)

- Modern version of RISC dates back to the 1980s
- Fewer, simpler instructions
  - Might take more to get given task done
  - Can execute on small and fast hardware
- Register-oriented instruction set
  - Many registers (typical 32) for arguments, return pointer, etc.
- Load-store architecture:
  - Only load/store instruction can access memory
- No condition codes

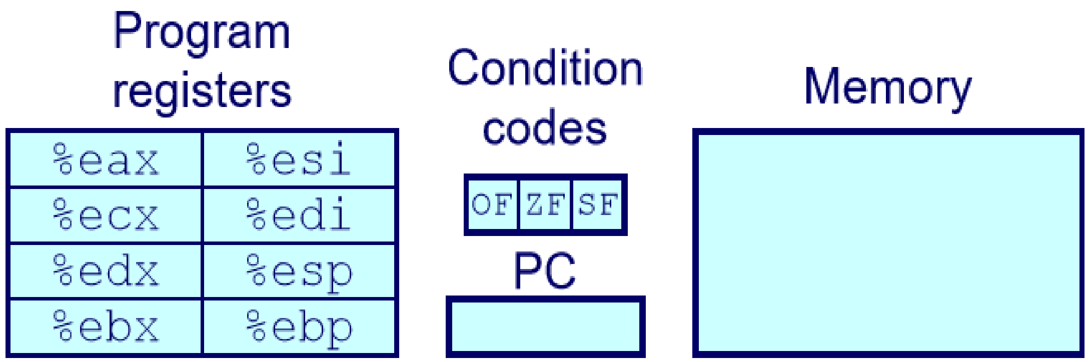
Example of CISC: IA32 Processors

- Intel-compatible processors (called “IA32” by Intel or “x86” colloquially. IA32 is short for “Intel Architecture, 32-bit”
- Starting in 1978 with 8086, dominate market today
  - More features are added as time goes on
  - x86 has not yet been competitive in the personal mobile device.

Comparison of Different Concept

- ➢ X86-64: 64-bit version of the x86 instruction set
- ➢ IA-64: a radically new ISA for high performance
  - ❖ for Itanium family of 64-bit Intel microprocessors

IA32 Registers



General-purpose register

- can be used as operand, such as integer and FP

Special-purpose register

- specific defined function, such as condition codes, processor status, PC, etc.

- 80386 CPU contains 8 GPRs, each 32 bits
- EIP (Intel’s PC) stores the address of the next instruction
- Condition codes store status information about the most recently executed arithmetic operation

Processor State

- Processor State: the information held in the processor at the end of an instruction to provide the processing context for the next instruction.
  - PC, general register values, special register values, etc.

- Programmer visible state to the processor (and memory) plays a central role in computer organization for both hardware and software
  - If the processing of an instruction can be interrupted then the hardware must save and restore the state transparently

- Not to be confused with Processor Power State

Example of RISC: MIPS Processors

- Example of RISC
- Originally an acronym for “Microprocessor without Interlocked Pipeline Stages”
- Elegant example of the ISA designed since the 1980s
  – In 1981, a team led by John Hennessy at Stanford University started work on what would become the first MIPS processor
- Widely studied in the CA course around the world ☺

MIPS Registers

$0	$0	Constant 0
$1	$at	Reserved Temp.
$2	$v0	Return Values
$3	$v1	
$4	$a0	Procedure arguments
$5	$a1	
$6	$a2	
$7	$a3	
$8	$t0	Caller Save Temporaries: May be overwritten by called procedures
$9	$t1	
$10	$t2	
$11	$t3	
$12	$t4	
$13	$t5	
$14	$t6	
$15	$t7	

$16	$s0	Callee Save Temporaries: May not be overwritten by called procedures
$17	$s1	
$18	$s2	
$19	$s3	
$20	$s4	
$21	$s5	
$22	$s6	
$23	$s7	
$24	$t8	Caller Save Temp
$25	$t9	
$26	$k0	Reserved for Operating Sys
$27	$k1	
$28	$gp	Global Pointer
$29	$sp	Stack Pointer
$30	$s8	Callee Save Temp
$31	$ra	Return Address

MIPS Instruction Overview

opcode (6)	rs (5)	rt (5)	rd (5)	00000	func (6)

Register-Register: rd ← (rs) func (rt)

```
add $s1, $s2, $s3          # register add: $s1 = $s2 + $s3
```

opcode (6)	rs (5)	rt (5)	offset (16)

Load/Store: rt ← Mem[(rs) + offset]

```
lw $s1, 10($s2)            # load word: $s1 = Mem[$s2 + 20]
```

opcode (6)	rs (5)	rt (5)	offset (16)

Branch: go to offset×4 if (rs) equal to (rt)

```
beq $s1, $s2, 25           # if ($s1==$s2) then go to [PC+4+100]
```

opcode (6)	target (26)

Jump: go to target×4 (append 10000 to PC <31:28> to get new address

```
j   2500                   # unconditional absolute jump to new address
```

Outlines

- Instruction Set Architecture
- ISA: Programmer's View
- ISA: Microarchitecture Level Design
  - Simple Pipeline Implementation

Compilation to Machine Code

Compiler:

- The variable x and a are assigned to $s1 and $s2,respectively
- The base address of the array A is in $s0

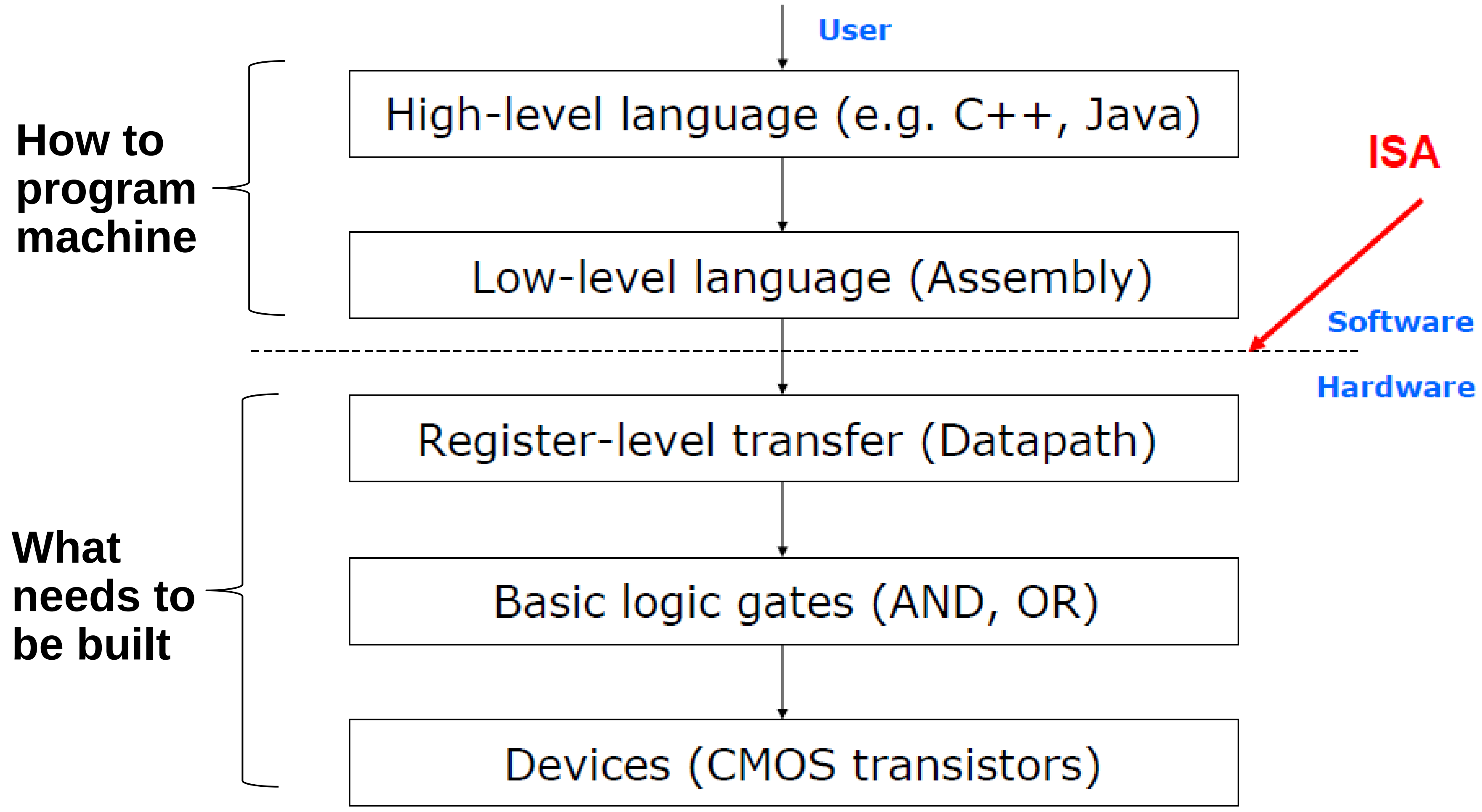
Source code	*Assembly language code*
x = a + A[2]	lw $t0, 8($S0)
	add $s1, $s2, $t0

Assembler:

Converts each assembly language instruction into a bit pattern (machine code) that hardware understands

HW/SW Interface

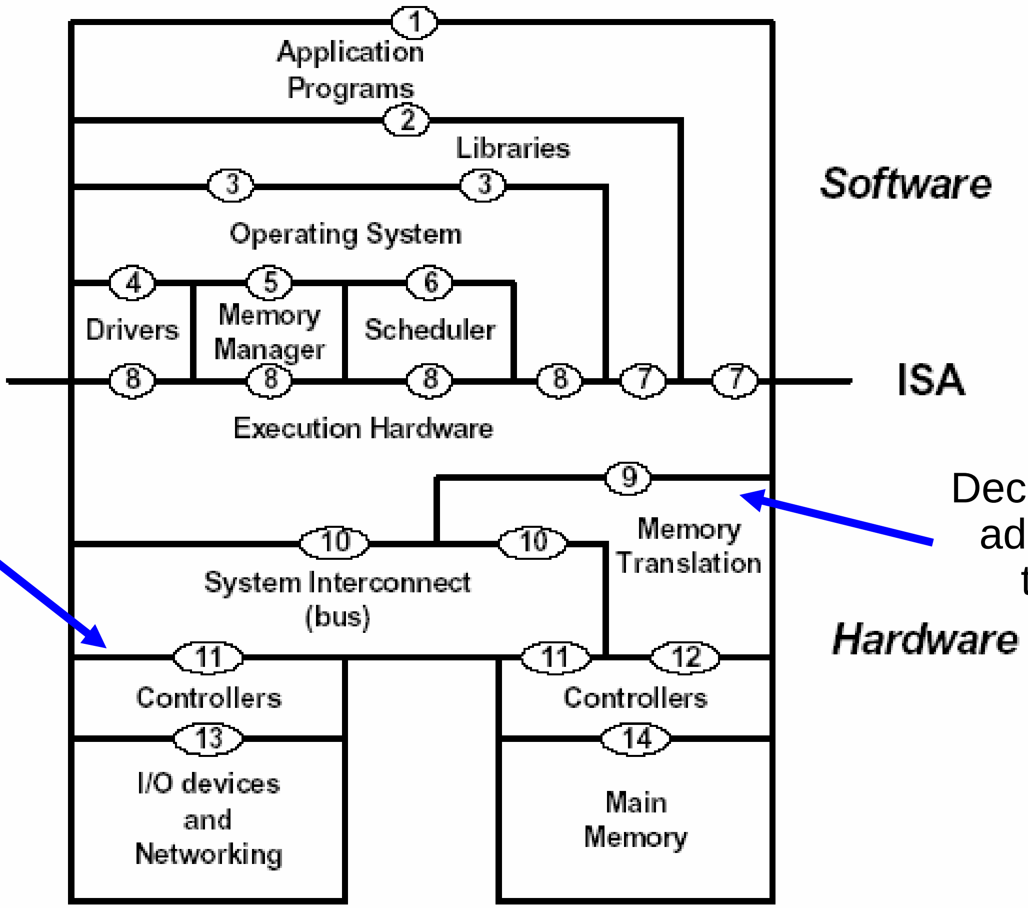
ISA defines the interface between software and hardware


User
How to program machine
High-level language (e.g. C++, Java)
Low-level language (Assembly)
ISA
Software
Hardware
What needs to be built
Register-level transfer (Datapath)
Basic logic gates (AND, OR)
Devices (CMOS transistors)

Layers of Computer System Architecture

- **An interface declares a set of operations**
  - Different layers communicate vertically via the shown interfaces

Declares the signals that drive I/O device controllers



Declares the way addresses are translated

User ISA and System ISA

- **ISA Typically divided into two parts:**

- **User ISA**: Gets application's work done
  - This is the subset of an ISA targeted by compilers when mapping an algorithm specified in a high-level language to machine instructions

- **System ISA**: Manages (shared) resources
  - This is the subset of an ISA carefully programmed in assembly language for low-level O/S subsystems (e.g., scheduler, virtual memory, device drivers)

The “User” ISA

- Refers to those aspects of the instruction set that are visible to an application program
  - Data flow
  - ALU operations
  - Control flow

Overview of User ISA

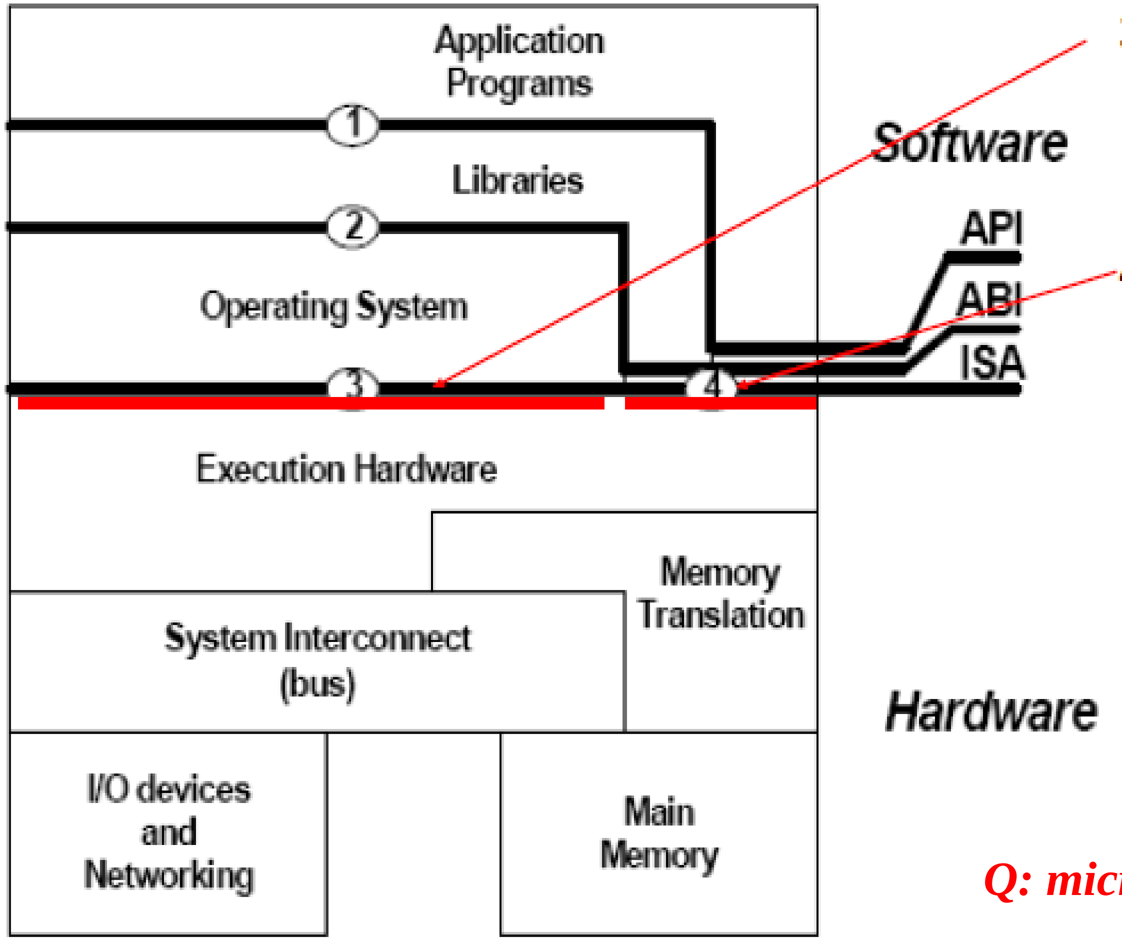
Integer	**Memory**	**Control Flow**	**Floating Point**
Add	Load byte	Jump	Add single
Sub	Load word	Jump equal	Mult. double
And	Store multiple	Call	Sqrt double
Compare	Push	Return	…
…	…	…	

The “System” ISA

- Refers to those aspects of the instruction set that are visible to supervisor software, such as the O/S, which is responsible for managing hardware resources

- **Overview of System ISA:**
  - Privilege levels
  - Control registers
  - Instructions that manage key resources
    - Processor (scheduling, time-sharing)
    - Memory (isolated address spaces)
    - I/O (e.g., isolated disk storage space)

Different Interfaces



3: "system" ISA
I/O, memory mgmt, CPU
intercept & emulate

4: "user" ISA
ALU, branch, load/store
direct execution

Q: micro-architecture an interface ?

上海交通大學 SHANGHAI JIAO TONG UNIVERSITY

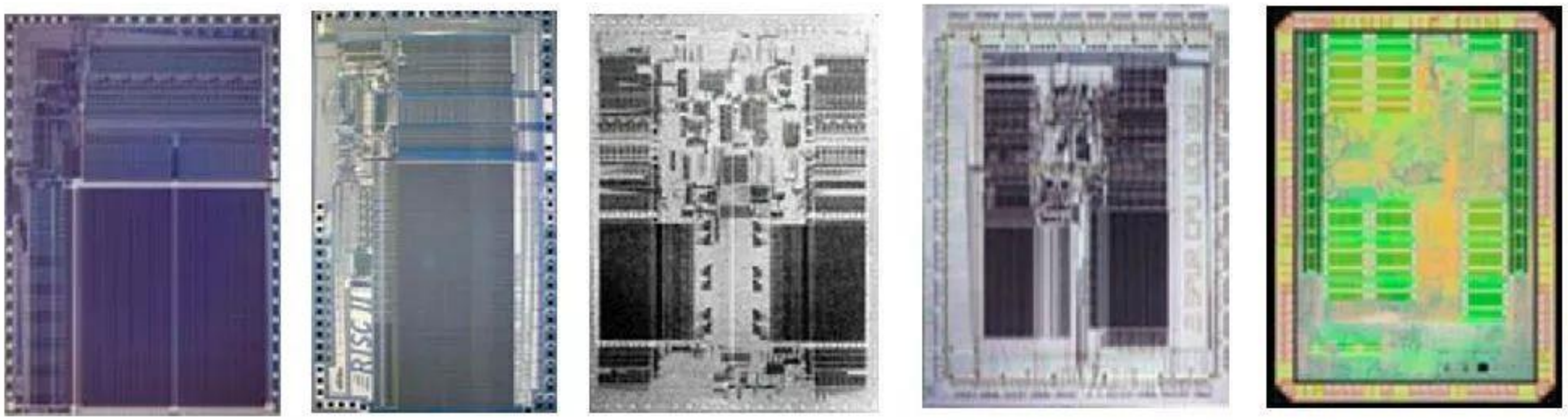
Interface Design

- **“Ideal” instruction set changes continually**
  - Technology allows larger CPUs over time
  - Technology constraints change (e.g., power)
  - Compiler technology improves (e.g., register allocation)
  - Programming styles change (assembly, object-oriented, …)
  - Applications change (e.g., multimedia, deep learning, ...)

- **A good instruction set**
  - Last through many implementations (compatibility)
  - Used in many different ways (generality)
  - Provides convenient functionality to higher levels
  - Permits an efficient implementation at lower levels

RISC-V: An open, Free ISA

- A RISC ISA that can be **freely used** for any purpose
- Support small, fast, and low-power system designs



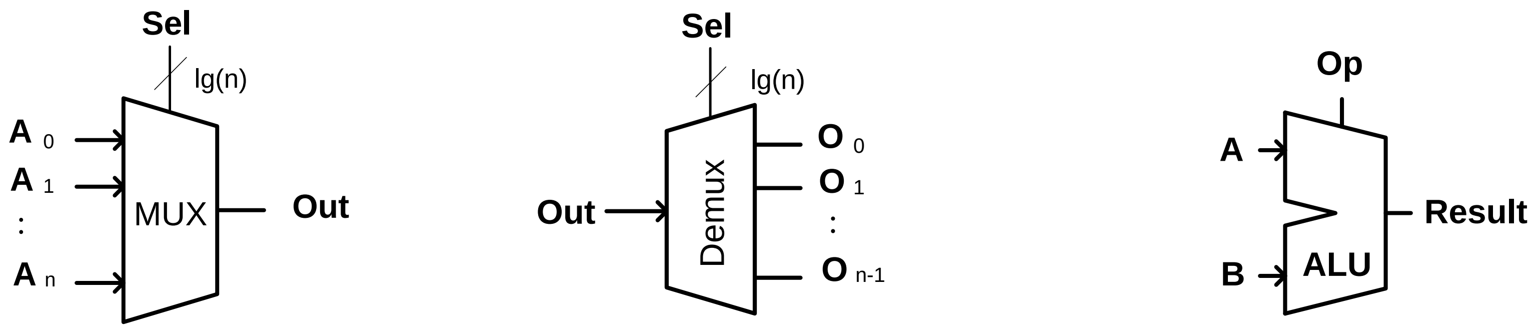
RISC-I 1981 | RISC-II 1983 | RISC-III (SOAR) 1984 | RISC-IV (SPUR) 1988 | RISC-V 2013

Outlines

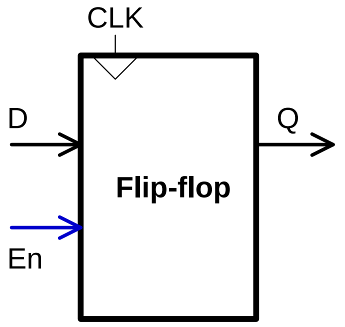
- Instruction Set Architecture
- ISA: Programmer’s View
- ISA: Microarchitecture Level Design
  - Simple Pipeline Implementation

Hardware Elements

- Combinational logic:
  - Output is a function of the present input only：Out

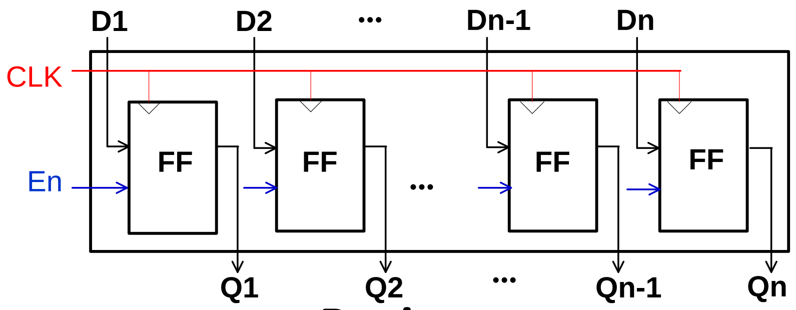


- Synchronous sequential logic
  - Edge-triggered: data is sampled at the clock edge



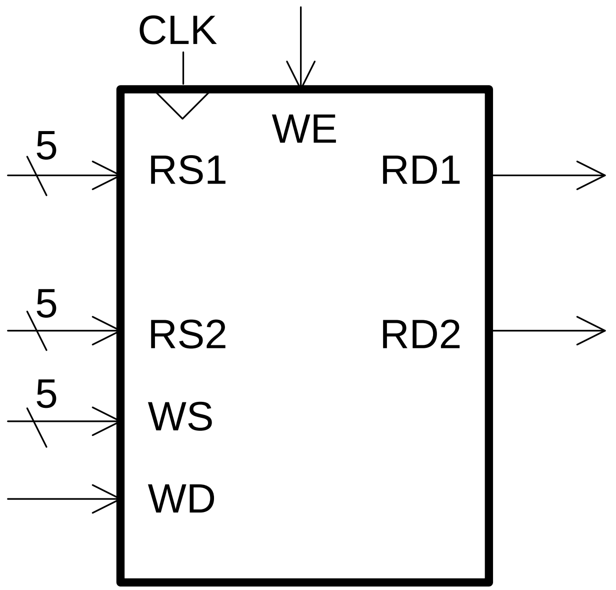
D	CLK	Q
0	Rising	0
1	Rising	1
-	0	Last Q
-	1	Last Q

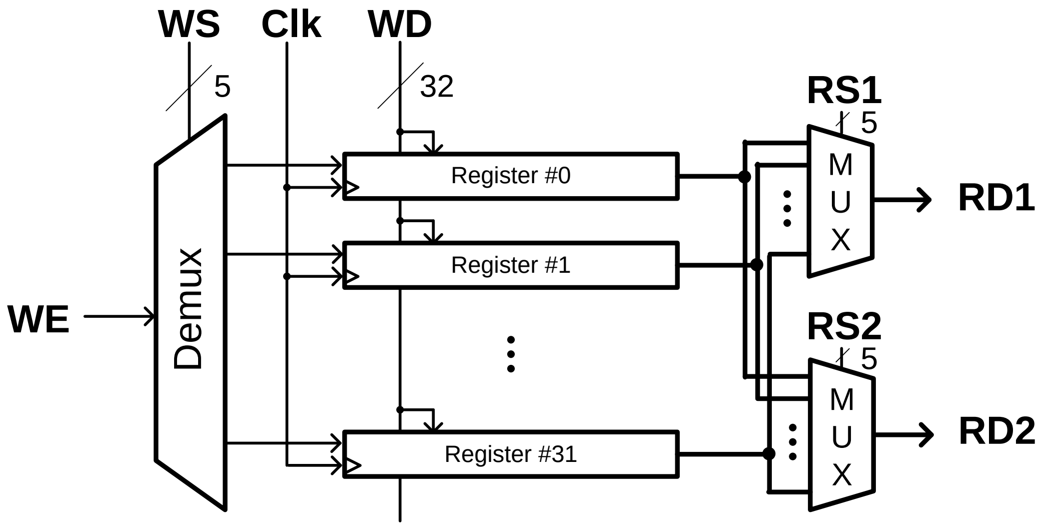
Latch



Register

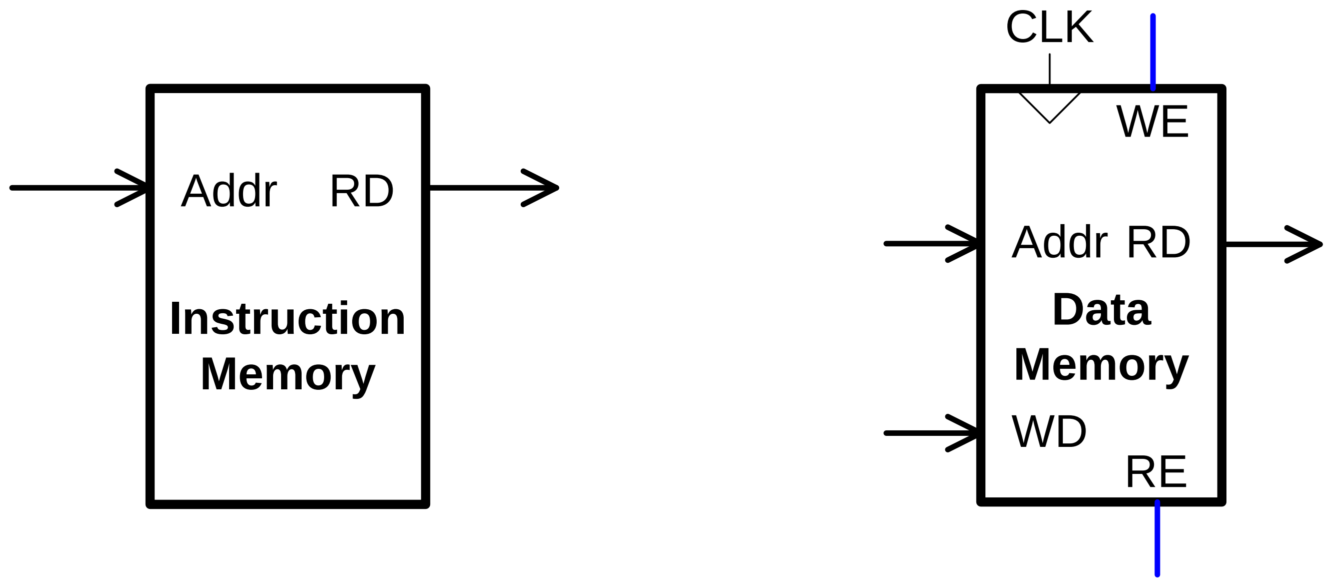
Register Files





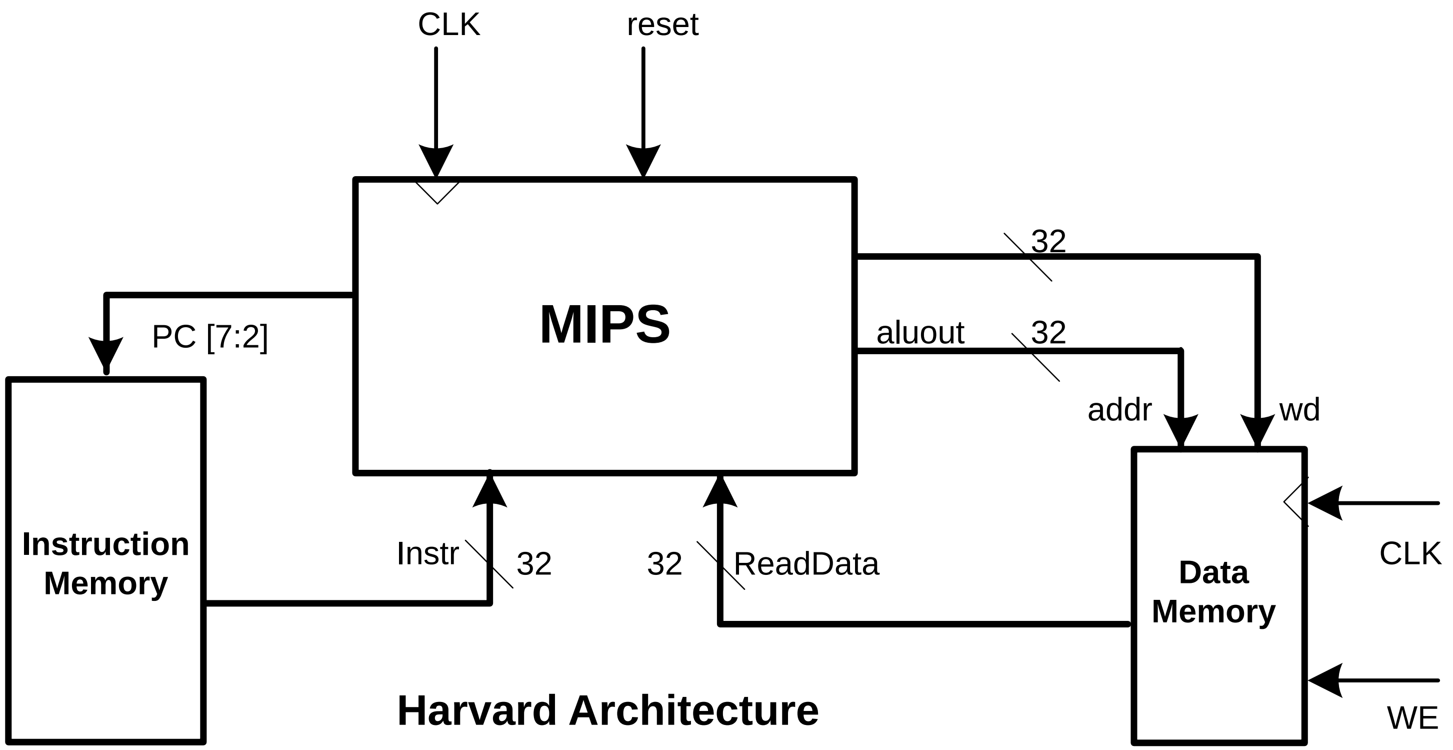
- The processor's general-purpose registers are stored in a structure called a **register file**

A Simple Memory Model



- a Read can be done any time (i.e. combinational logic)
- a Write is performed at the rising clock edge

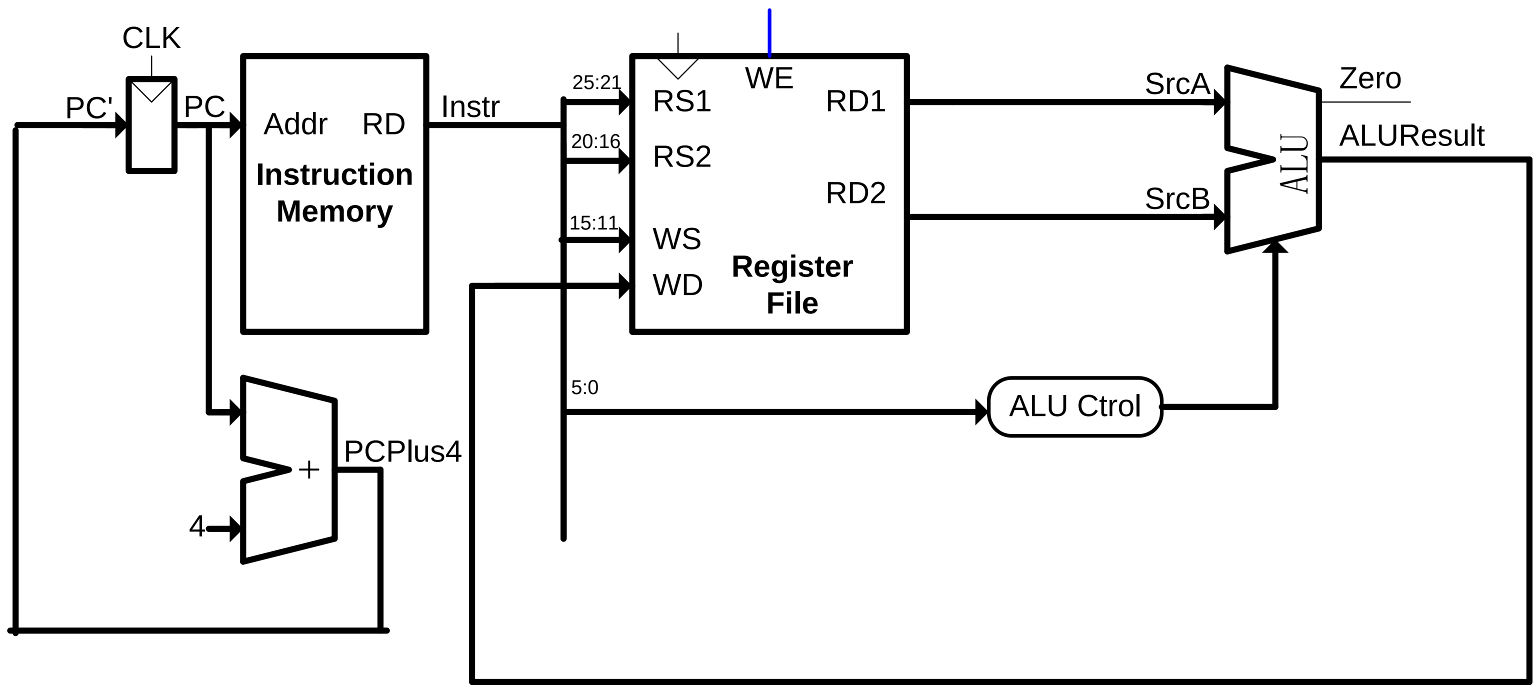
Harvard Architecture



Harvard Architecture

- **Harvard Style**: Physically separate storage and signal pathways for instructions and data
- **Princeton Style**: The same pathways (von Neumann Model)

Datapath: R-Type Instructions

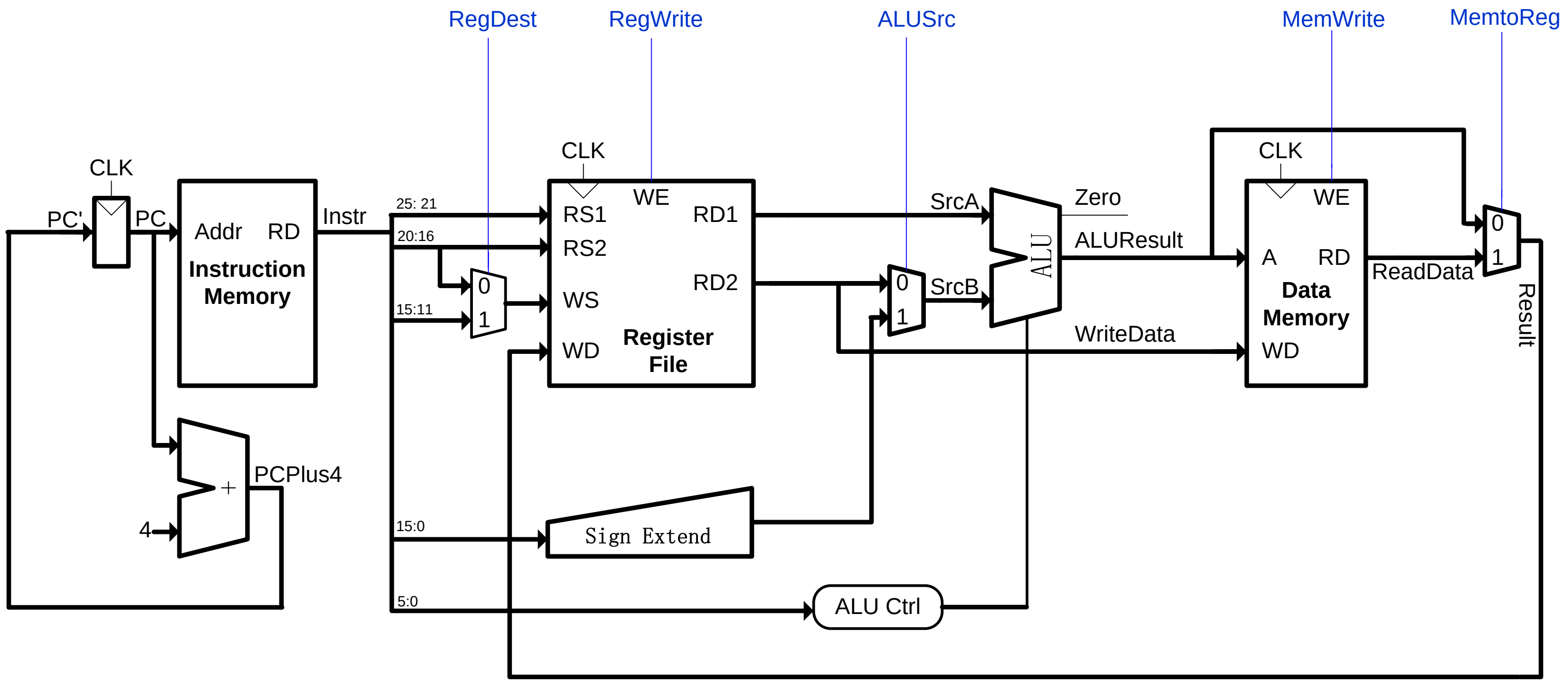


6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
op	rs	rt	rd	shamt	func

rd ← (rs) func (rt)

Q: CLK for RF reading ?

Datapath: Load/Store Instructions

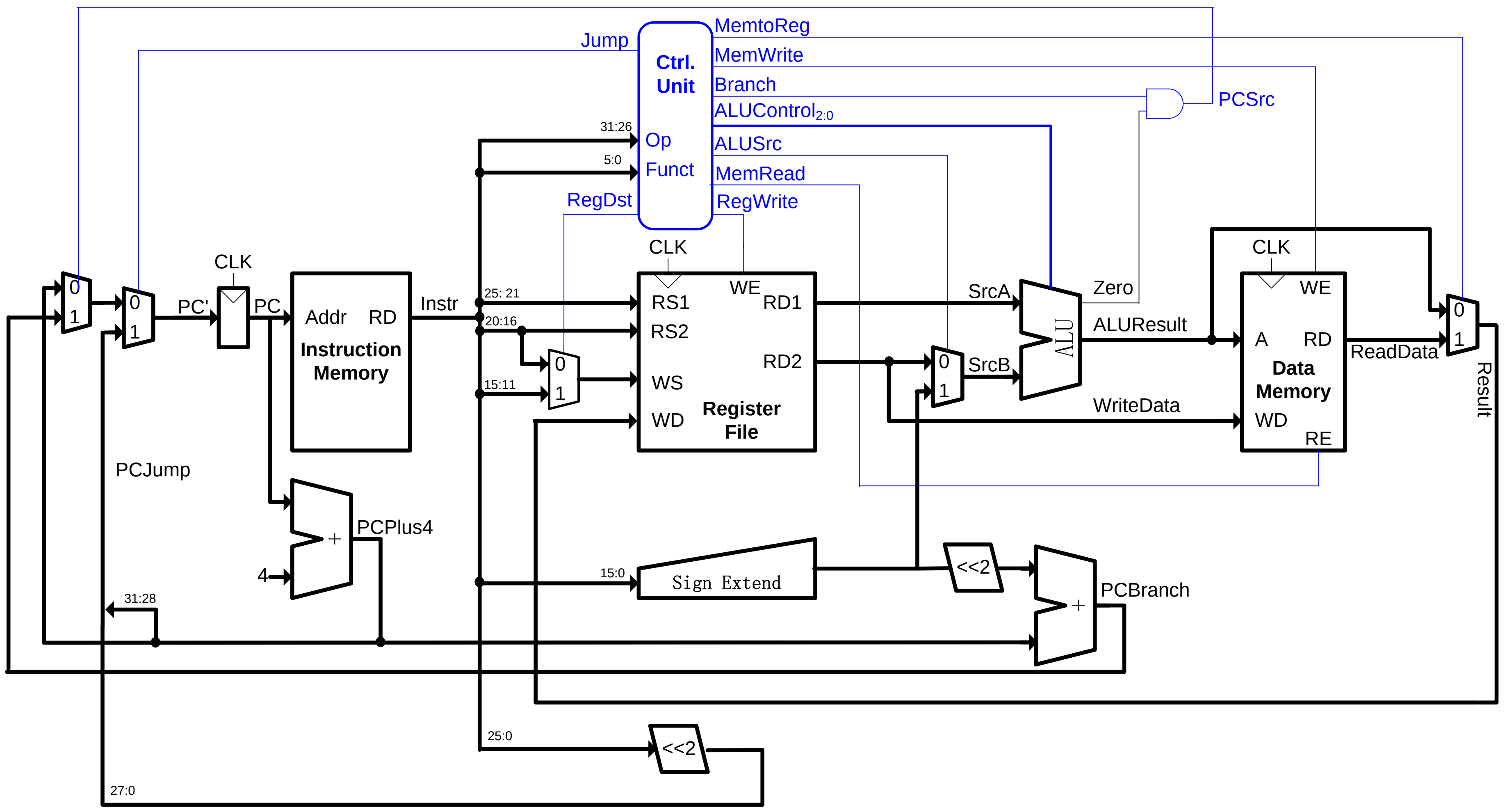


6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
op	rs	rt	address		

rs is the base register

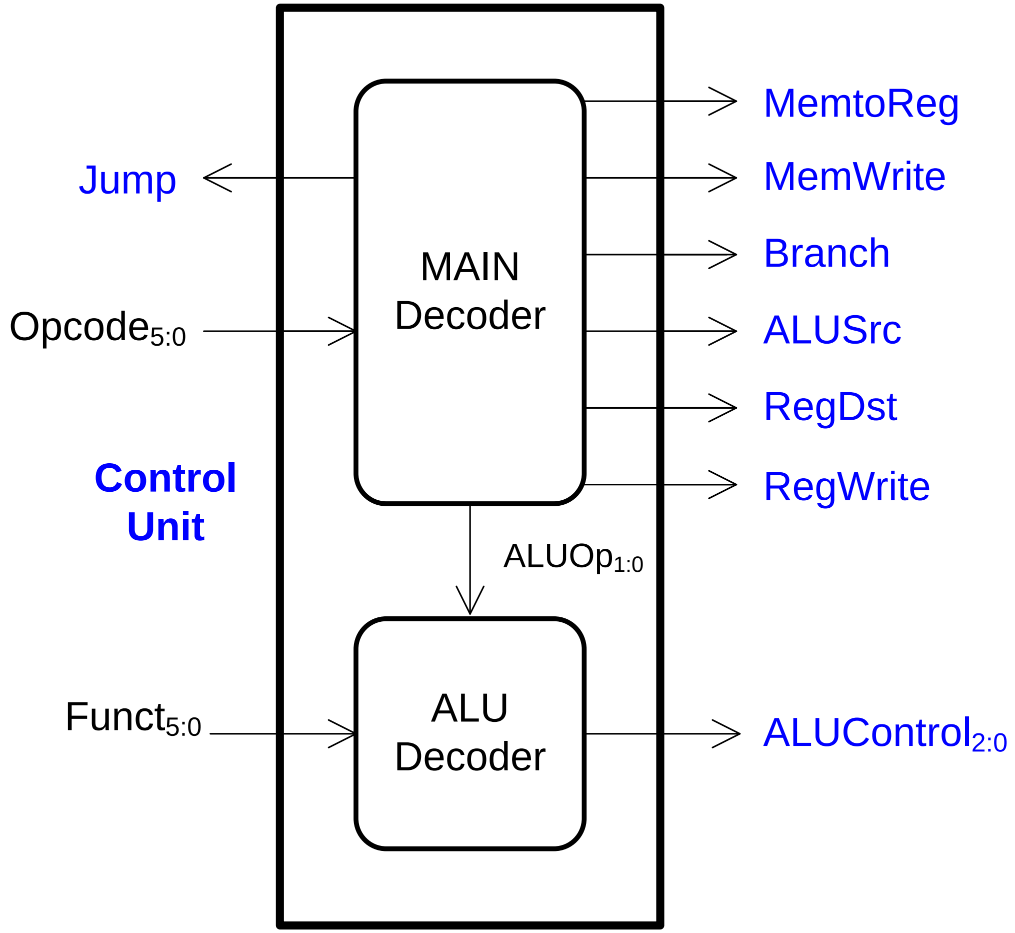
rt is the destination of Load or the source for a Store

Control and Datapath with Branch/Jump Support



Control Logic

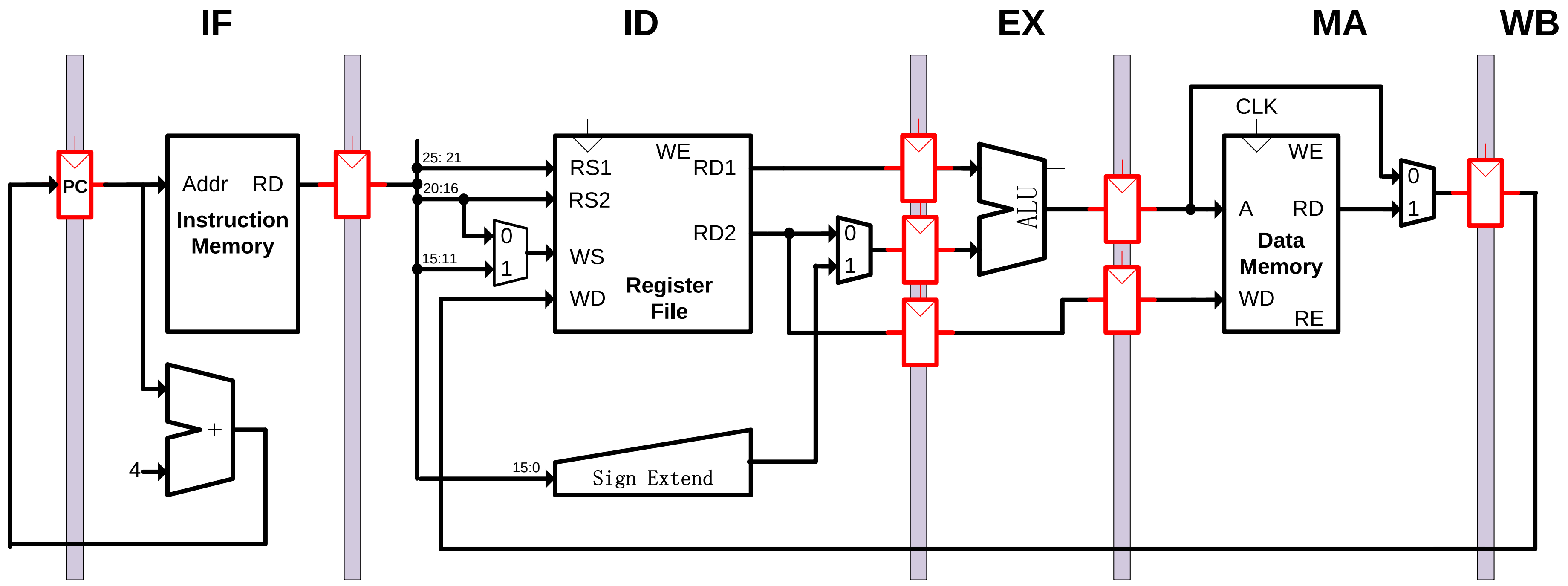
Focus on the design of the state machines to decode instructions and generate the sequence of control signals



The setting of the control lines is completely determined by the opcode fields of the instruction

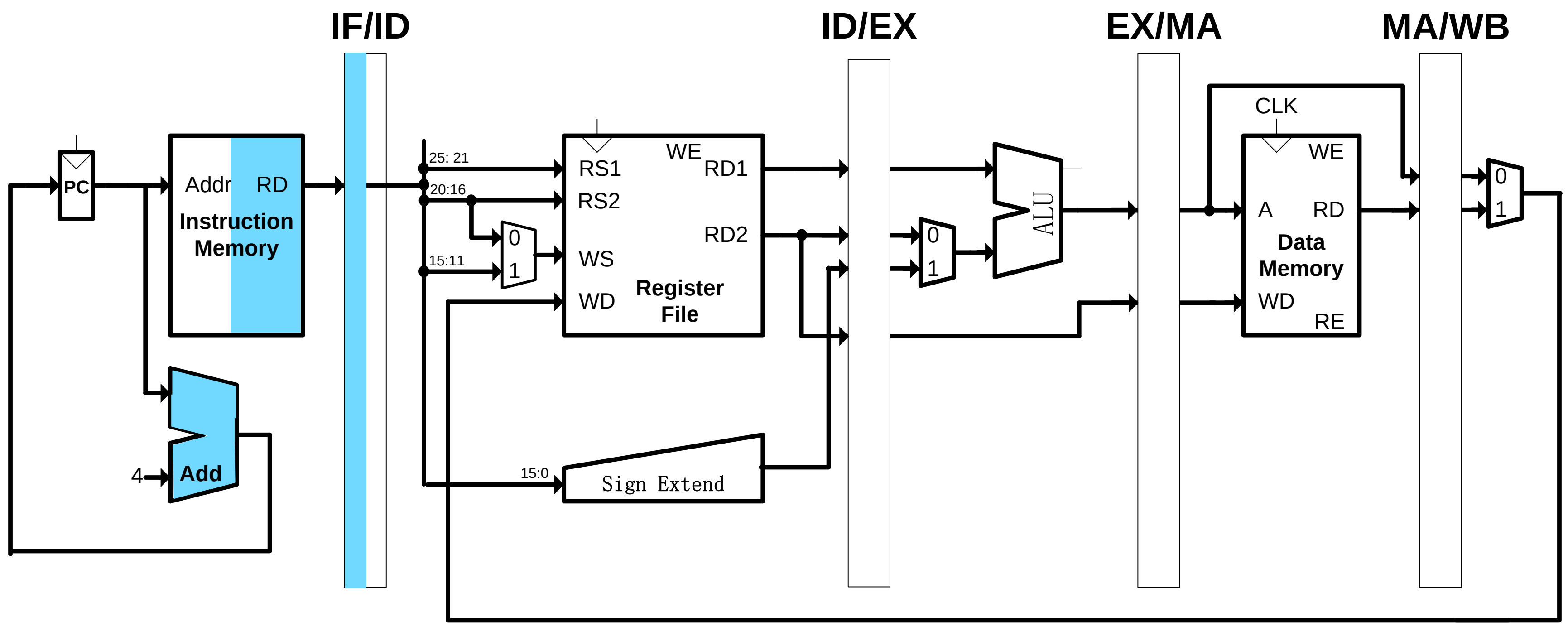
Generic 5-Stage Pipeline

- Pipelining: An implementation technique whereby multiple instructions are overlapped in execution
- Instruction-level Parallelism: Exploit parallelism among instructions (statistically & dynamically).



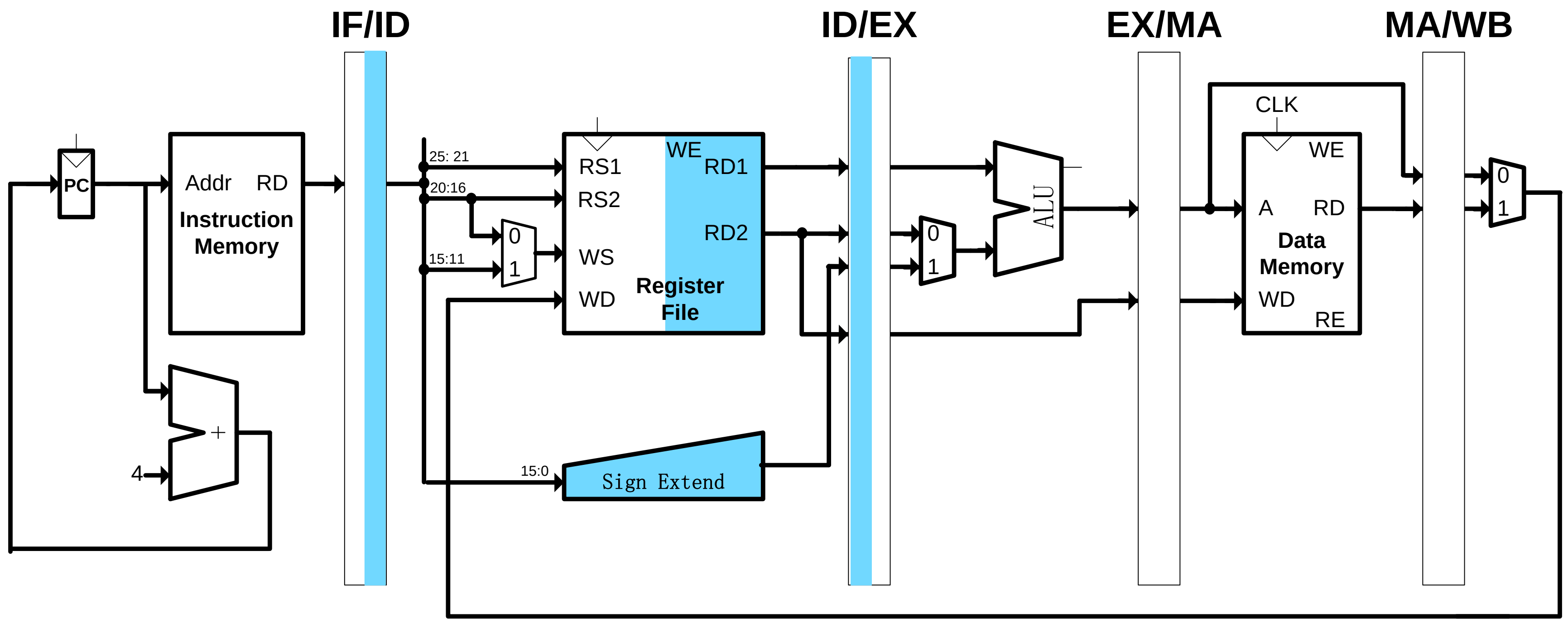
Pipelined Load

- Instruction Fetch



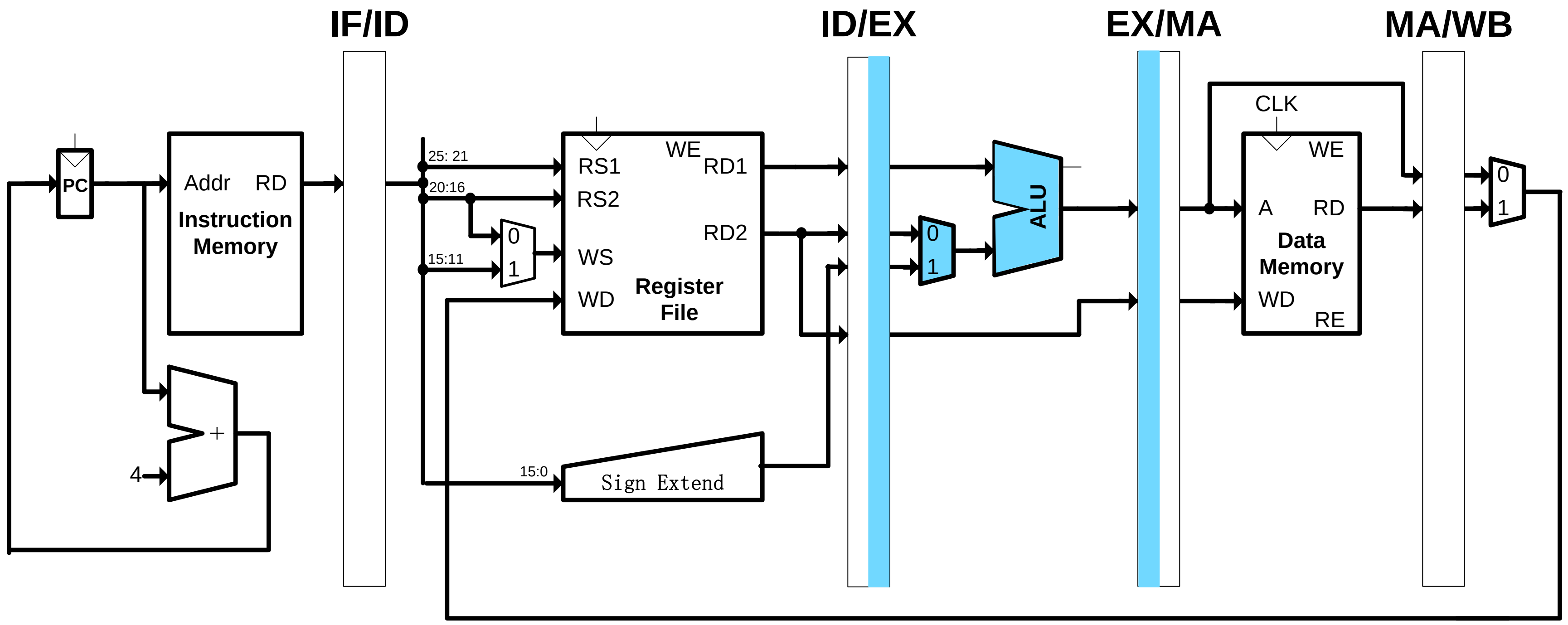
Pipelined Load

- Instruction decode and register file read



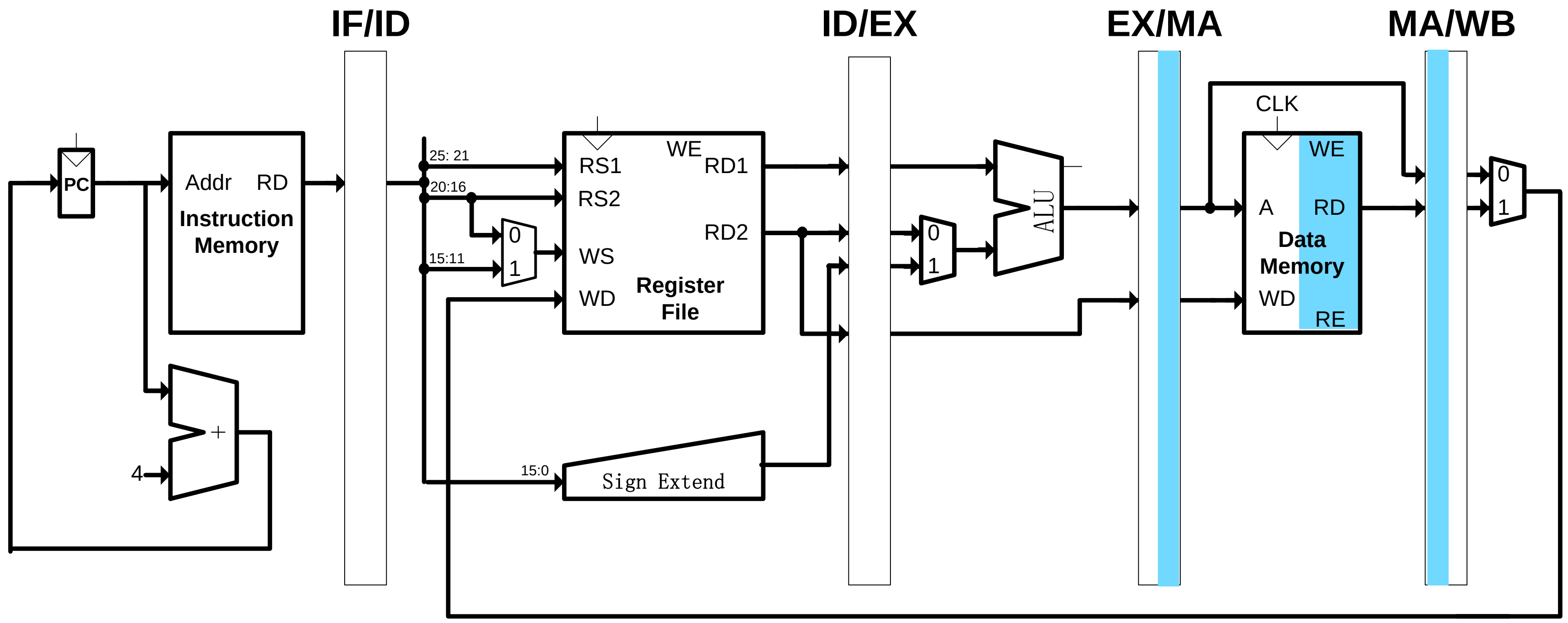
Pipelined Load

- Execute or address calculation



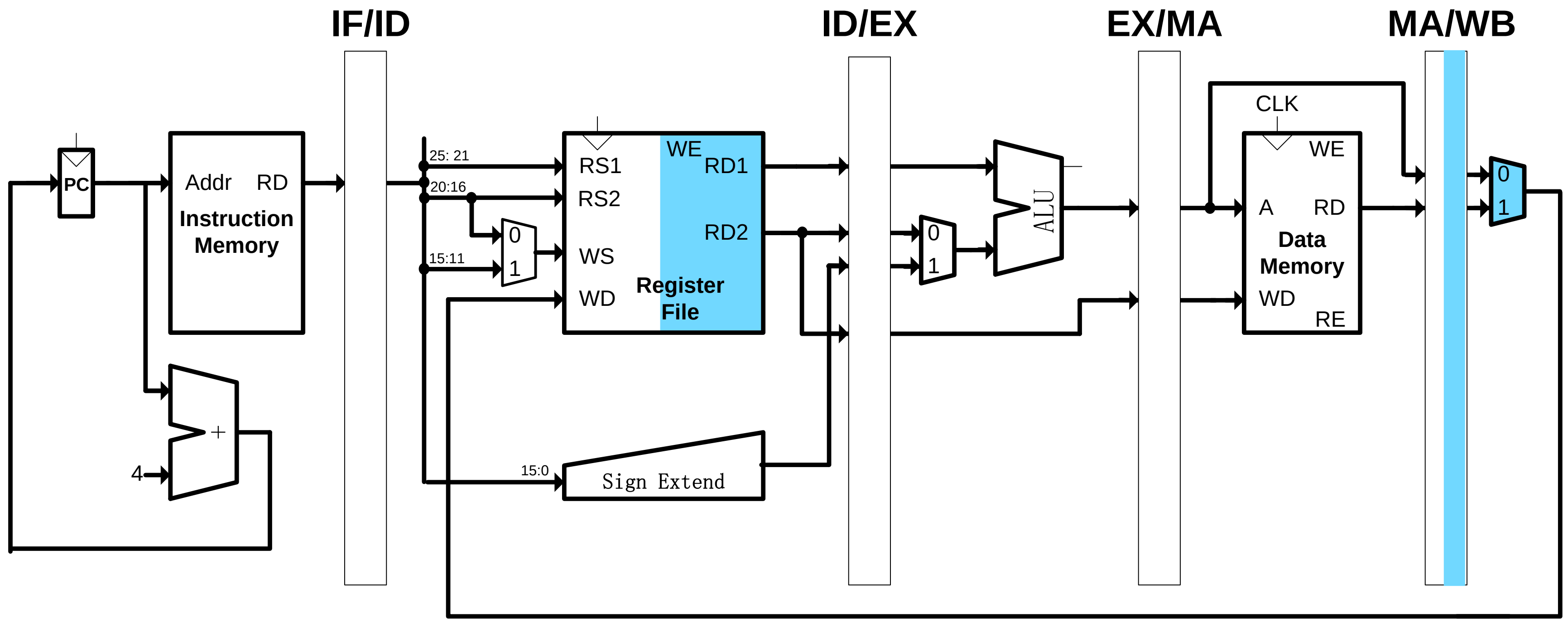
Pipelined Load

- Memory access



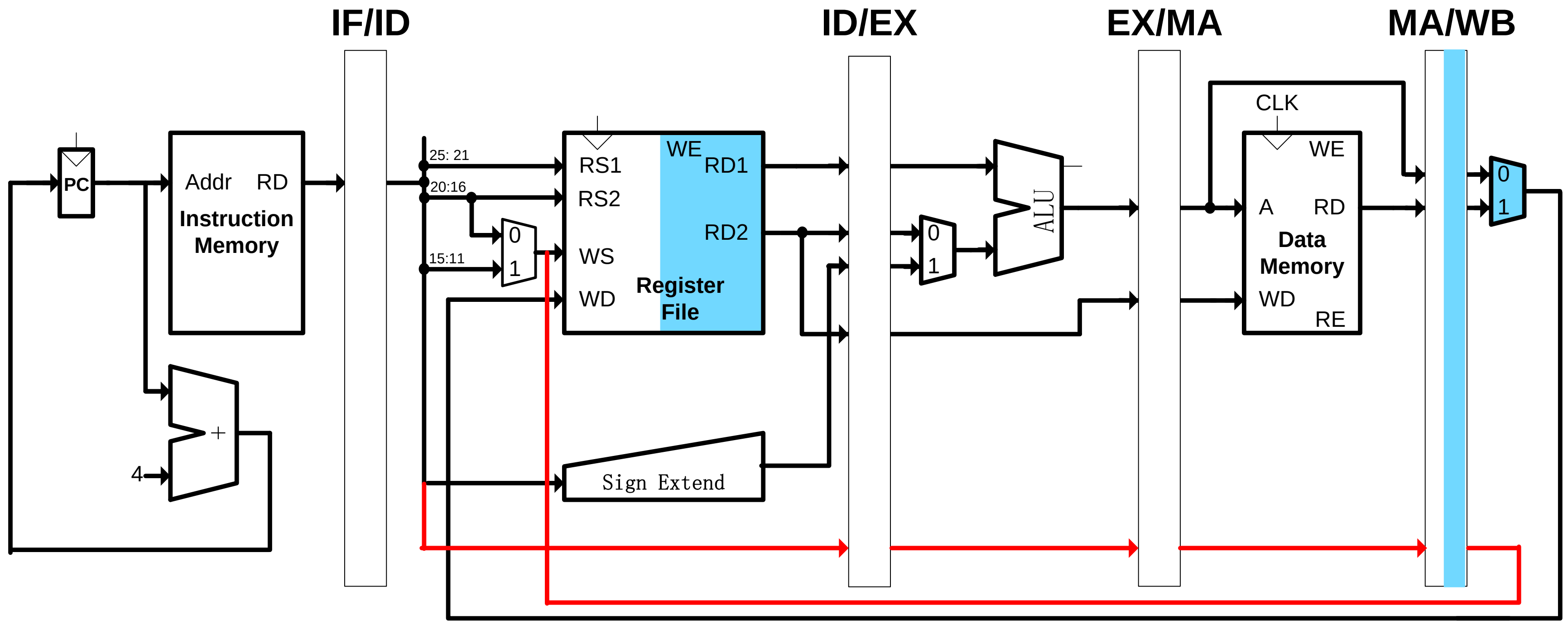
Pipelined Load

- Write-back



Pipelined Load (Fixed)

- The write register number must propagate



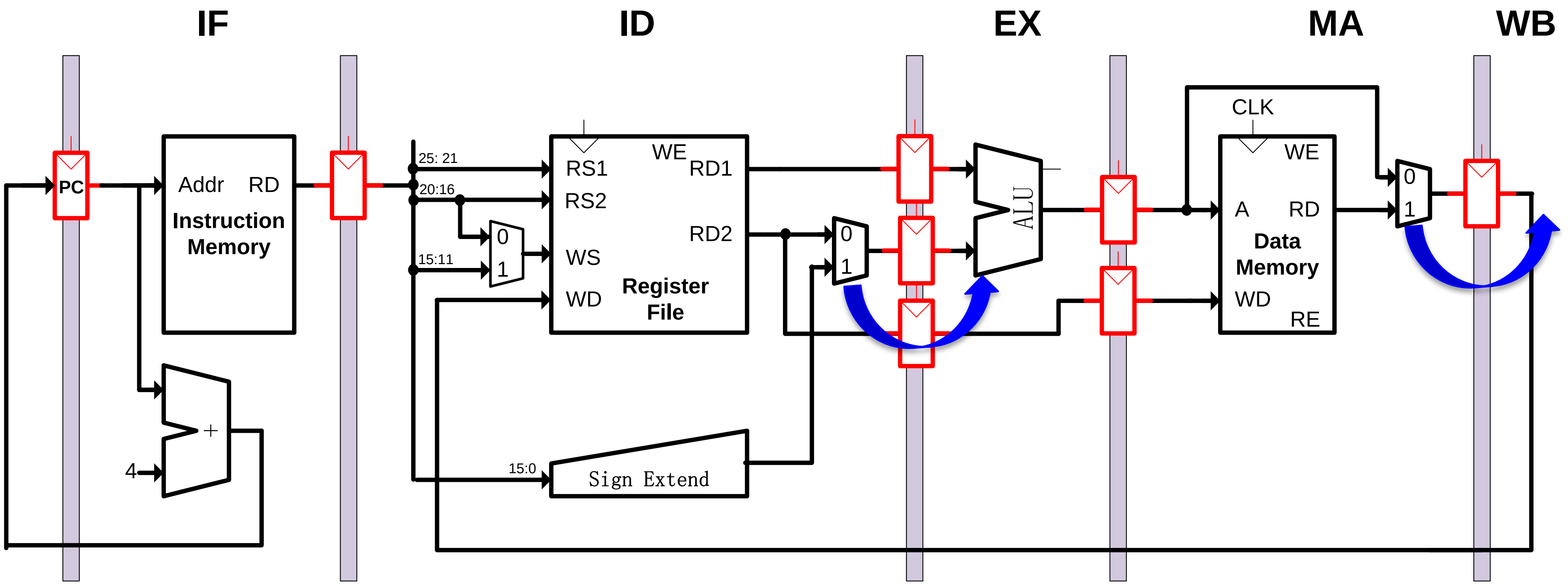
Pipeline Speedup

$$\text{Time between instruction}_{\text{pipelined}} = \frac{\text{Time between instruction}_{\text{Non-pipelined}}}{\text{Number of stages}}$$

- Speedup comes from increased throughput
- The latency of instruction does not decrease
- Pipeline rate limited by the slowest pipeline stage
- Ideally, the speed-up from pipelining is approximately equal to the number of pipe stages

An Ideal Pipeline

- Uniform sub-computations => balancing pipeline states
- Identical computations => unifying instruction types
- Independent computations => minimizing pipeline stalls

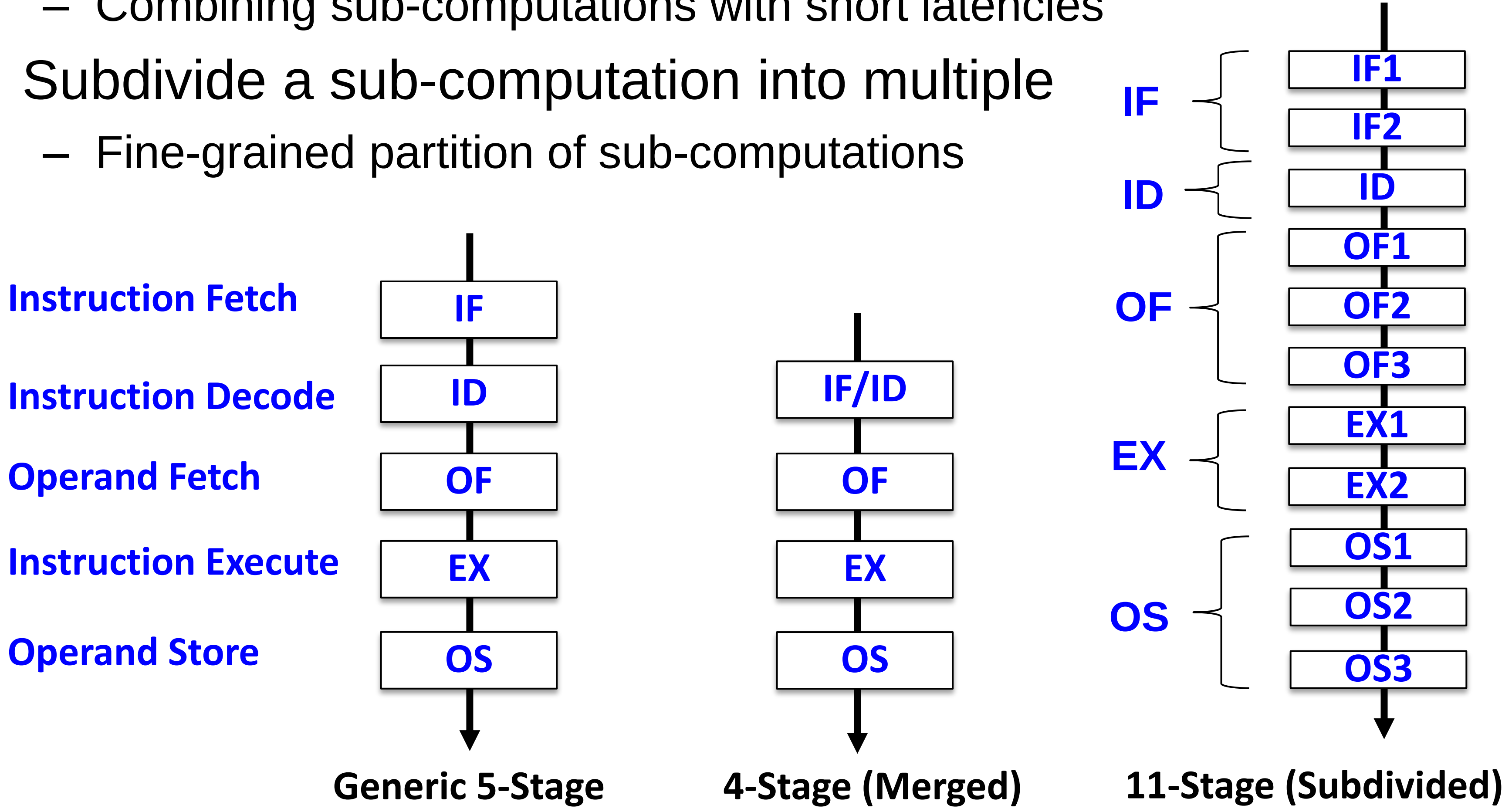


The slowest stage determines the clock: re-organize stages

Q: why identical computation is the ideal case ?

Stage Quantization

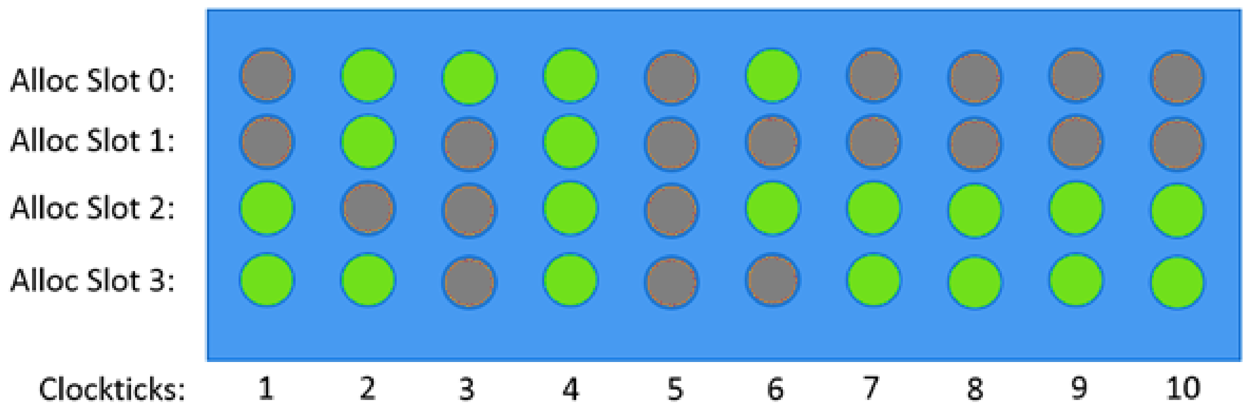
- Merge multiple sub-computations into one
  - Combining sub-computations with short latencies
- Subdivide a sub-computation into multiple
  - Fine-grained partition of sub-computations



Q: the more (stages), the better ?

Pipeline Slots

- A pipeline slot represents hardware resources needed to process one uOp.
  - Front-end
  - Back-end
  - Retiring
  - Bad Speculation



Front-End and Back End

- Front-End denotes the first part of the processor core responsible for fetching operations that are executed later on by the Back-End part.
  - a branch predictor predicts the next address to fetch
  - cache-lines are fetched and parsed into instructions
  - decoded into micro-ops (uOps)
- Front-End Bound metric represents a slots fraction where the processor's Front-End undersupplies its Back-End.

Summary

- Architecture vs. microarchitecture
- Evolution of instruction sets
- CISC (IA32) vs. RISC (MIPS)
- Machine interfaces
- User/System ISA
- MIPS instruction field
- Single-cycle MIPS
- Ideal pipeline
- Stage quantization
- Pipeline slot

HW-1

- List the advantages and disadvantages of the four classic computer architecture types in relation to each other.
- What is the motivation of RISC-v?
- Give your own summary of the key design principles of an ideal pipeline.