Review of Computer Architetcure

A Sahu
Deptt. of Comp. Sc. & Engg.
IIT Guwahati

Outline

- Computer organization Vs Architecture
- Processor architecture
- Pipeline architecture
  - Data, resource and branch hazards
- Superscalar & VLIW architecture
- Memory hierarchy
- Reference

Computer Organization vs Architecture

Comp Organization => Digital Logic Module

Logic and Low level

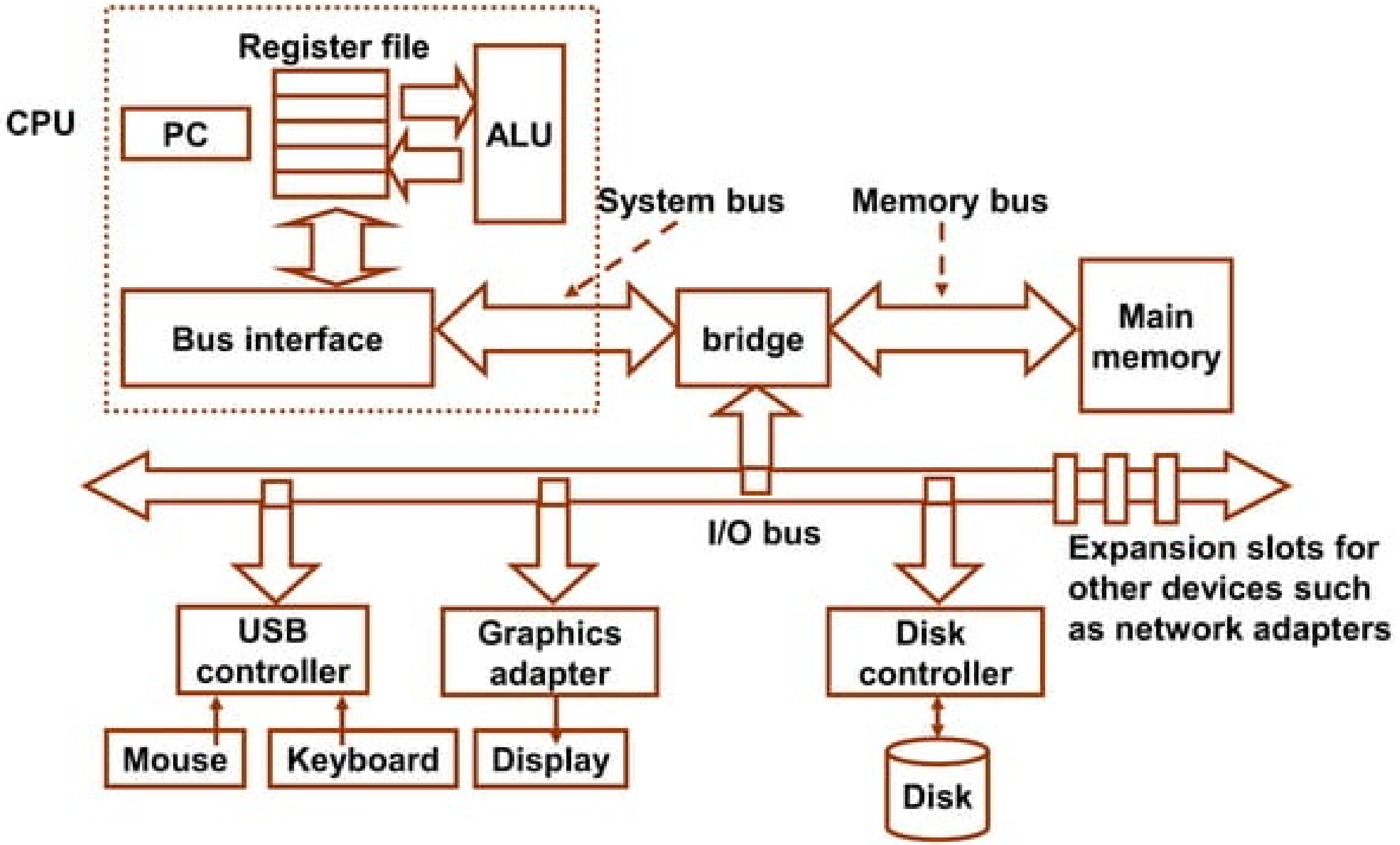
==========================

Comp Architecture = > ISA Design, MicroArch Design

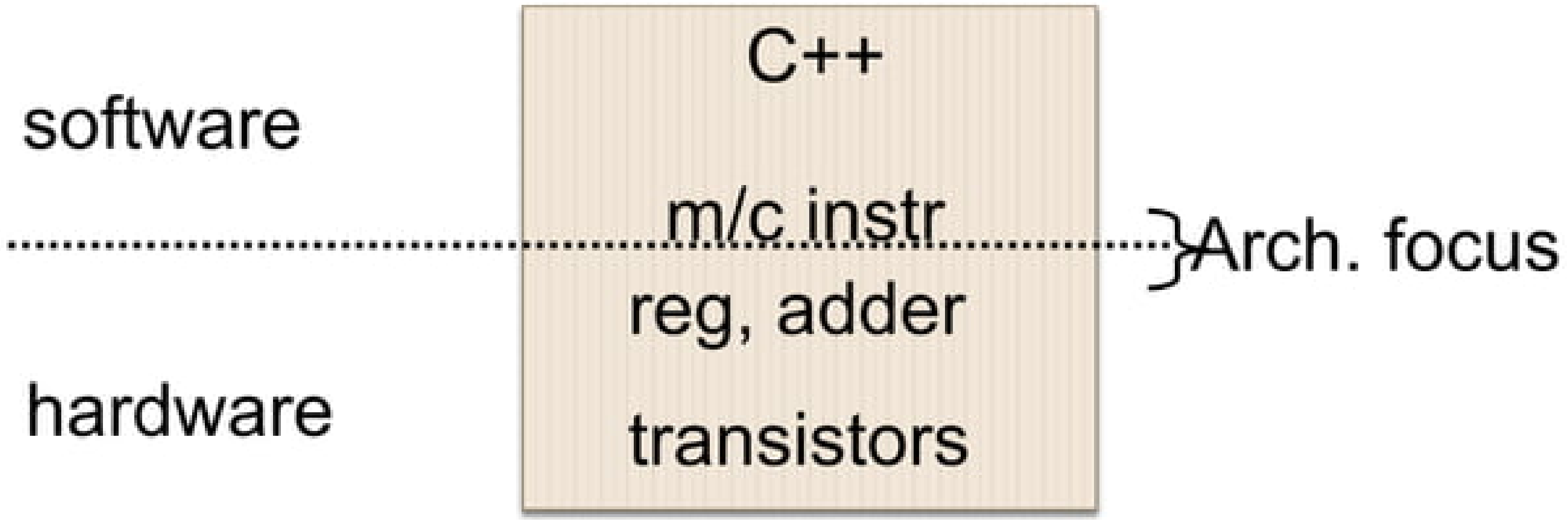
Algorithm for

- Designing best micro architecture,
- Pipeline model,
- Branch prediction strategy, memory management

Hardware abstraction



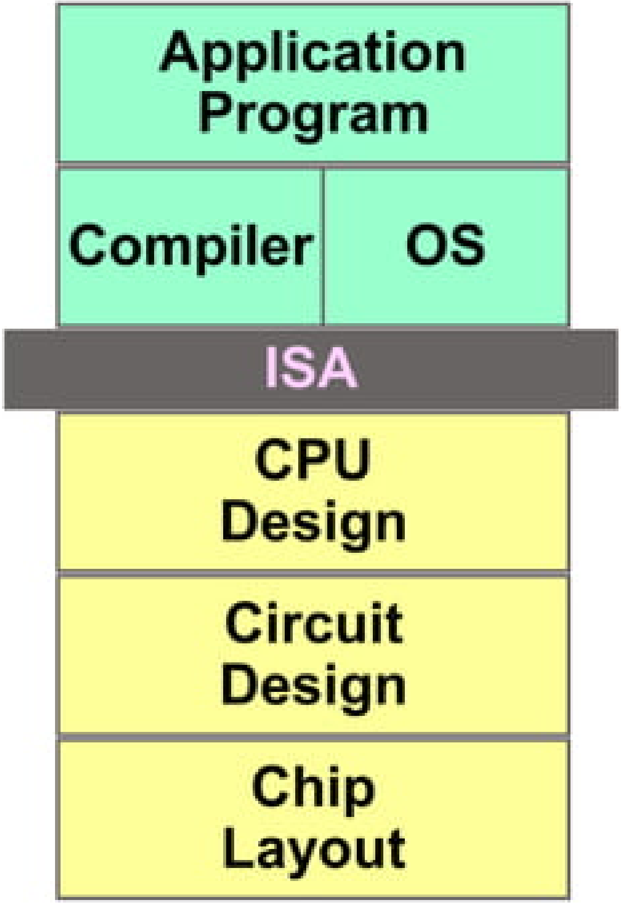
Hardware/software interface



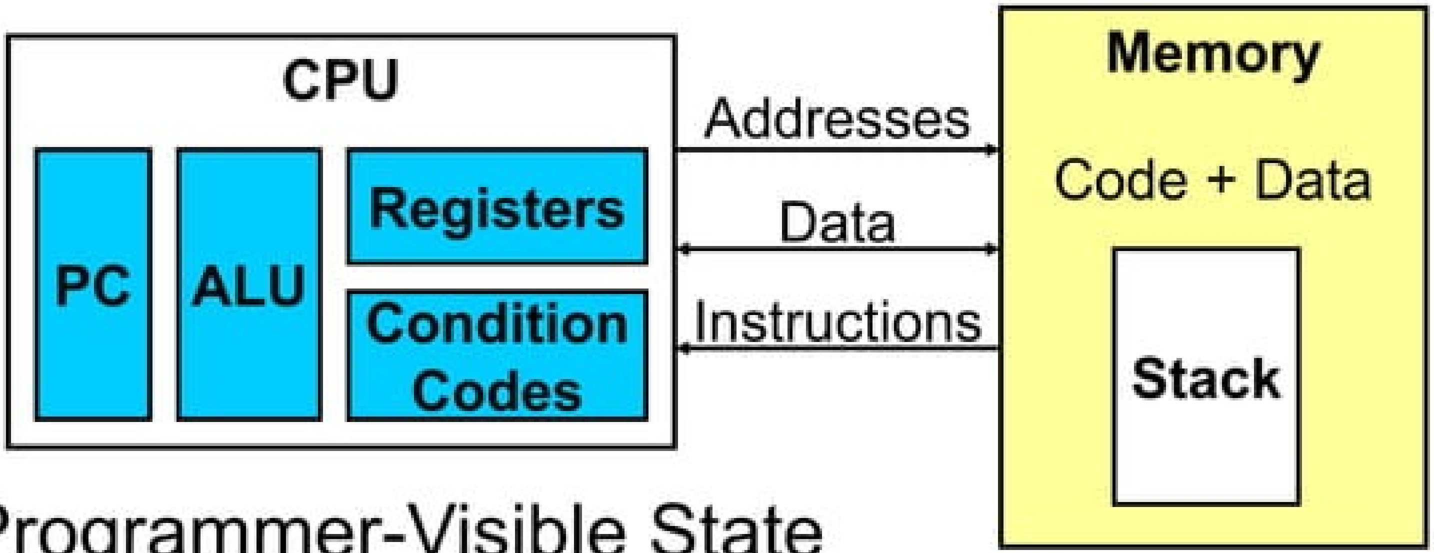
- Instruction set architecture
  - Lowest level visible to a programmer
- Micro architecture
  - Fills the gap between instructions and logic modules

Instruction Set Architecture

- Assembly Language View
  - Processor state (RF, mem)
  - Instruction set and encoding
- Layer of Abstraction
  - Above: how to program machine - HLL, OS
  - Below: what needs to be



The Abstract Machine



- Programmer-Visible State
  - PC Program Counter
  - Register File
    - heavily used data
  - Condition Codes

- Memory
  - Byte array
  - Code + data
  - stack

Instructions

- Language of Machine
- Easily interpreted
- primitive compared to HLLs

- Instruction set design goals
  - maximize performance,
  - minimize cost,
  - reduce design time

Instructions

- All MIPS Instructions: 32 bit long, have 3 operands
  - Operand order is fixed (destination first)
    Example:

```
    C code:          A = B + C
    MIPS code:       add $s0, $s1, $s2
```

(associated with variables by compiler)

- Registers numbers 0 .. 31, e.g., \$t0=8,\$t1=9,\$s0=16,\$s1=17 etc.
- 000000 10001 10010 01000 00000

Instructions LD/ST & Control

- Load and store instructions
- Example:

```
C code:              A[8] = h + A[8];
MIPS code:  lw    $t0, 32($s3)
            add $t0, $s2, $t0
            sw    $t0, 32($s3)
```

- Example: lw $t0, 32($s2)

```
35    18     9          32
op    rs     rt     16 bit number
```

- Example:

```
if (i != j)                  beq $s4, $s5, Lab1
   h = i + j;                add $s3, $s4, $s5
else                                 j Lab2
   h = i - j;     Lab1:         sub $s3, $s4, $s5
                  Lab2:             ...
```

What constitutes ISA?

- **Set of basic/primitive operations**
  - Arithmetic, Logical, Relational, Branch/jump, Data movement
- **Storage structure – registers/memory**
  - Register-less machine, ACC based machine, A few special purpose registers, Several Gen purpose registers, Large number of registers
- **How addresses are specified**
  - Direct, Indirect, Base vs. Index, Auto incr and auto decr, Pre (post) incr/decr, Stack
- **How operand are specified**
  - 3 address machine $r1 = r2 + r3$, 2 address machine $r1 = r1 + r2$
  - 1 address machine $Acc = Acc + x$ (Acc is implicit)

RISC vs. CISC

- **RISC**
  - Uniformity of instructions,
  - Simple set of operations and addressing modes,
  - Register based architecture with 3 address instructions
- **RISC: Virtually all new ISA since 1982**
  - ARM, MIPS, SPARC, HP's PA-RISC, PowerPC, Alpha, CDC 6600
- **CISC :** Minimize code size, make assembly language easy

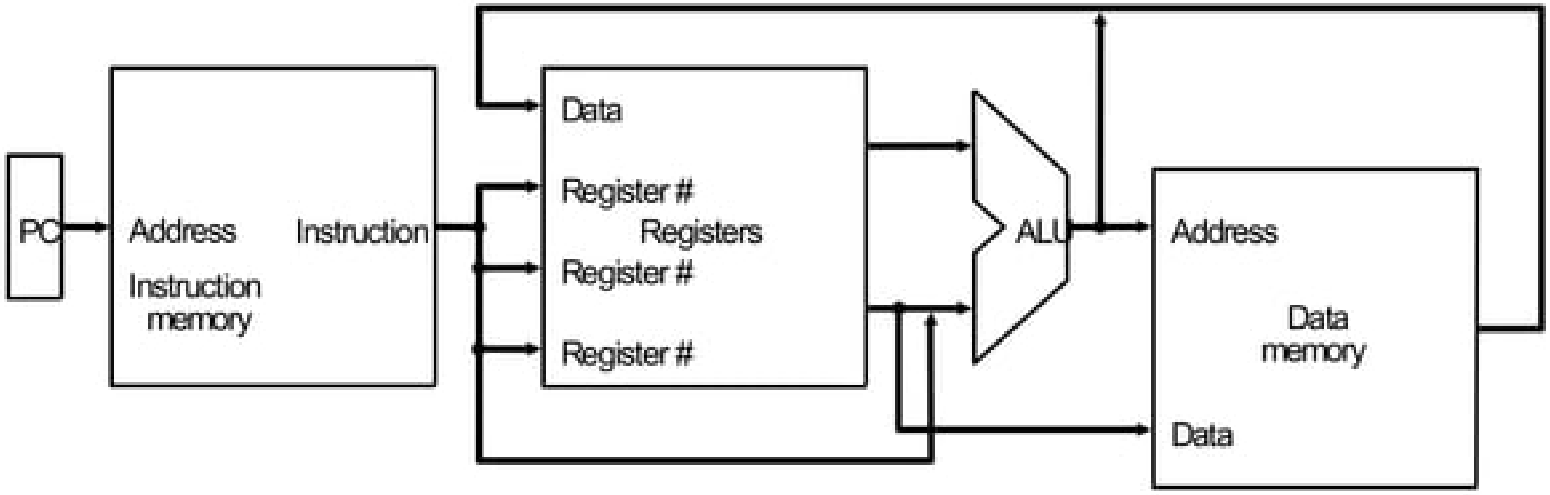
MIPS subset for implementation

- Arithmetic - logic instructions
  - add, sub, and, or, slt
- Memory reference instructions
  - lw, sw
- Control flow instructions
  - beq, j

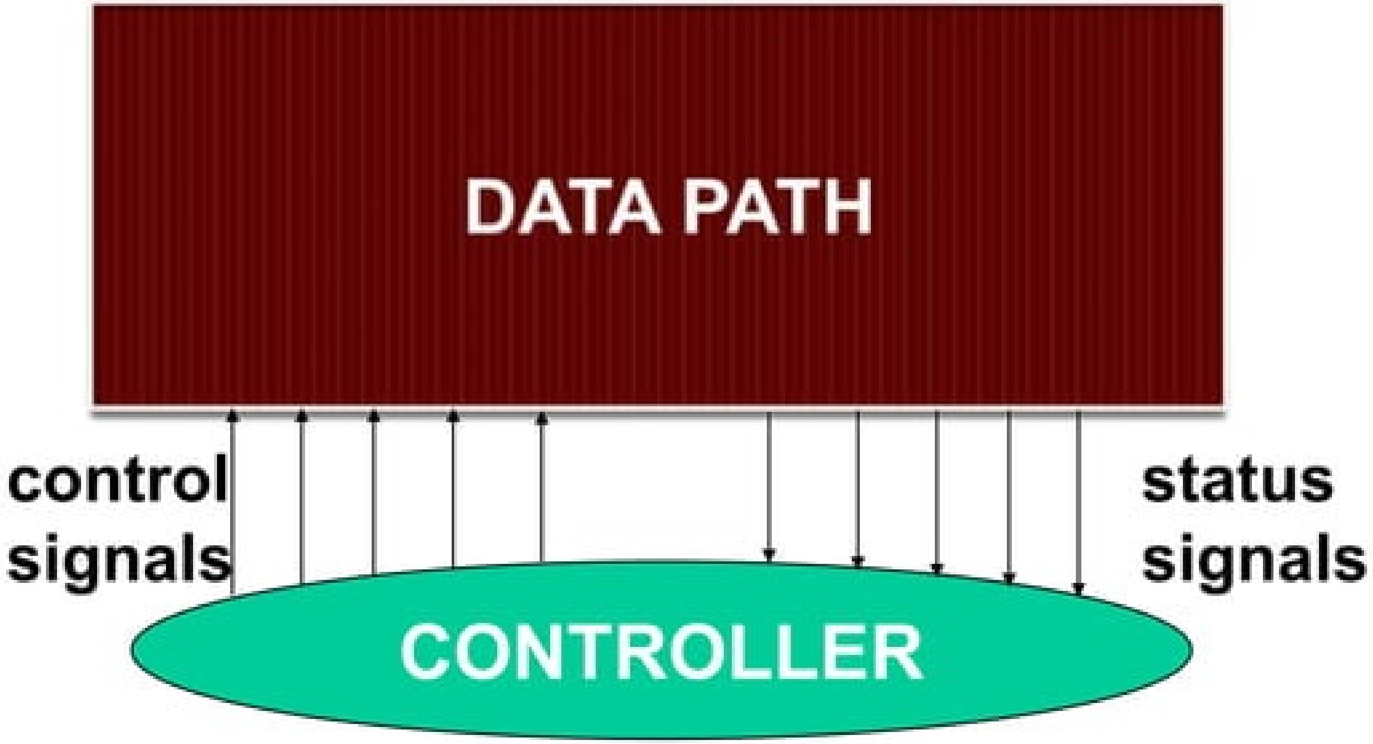
Incremental changes in the design to include other instructions will be discussed later

Design overview

- Use the program counter (PC) to supply instruction address
- Get the instruction from memory
- Read registers
- Use the instruction to decide exactly what to do



Division into data path and control



Building block types

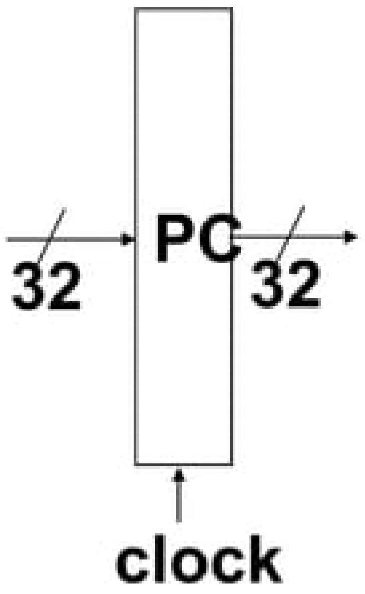
Two types of functional units:

- elements that operate on data values (combinational)
  - output is function of current input, no memory
  - Examples
    - gates: and, or, nand, nor, xor, inverter ,Multiplexer, decoder, adder, subtractor, comparator, ALU, array multipliers
- elements that contain state (sequential)
  - output is function of current and previous inputs
  - state = memory

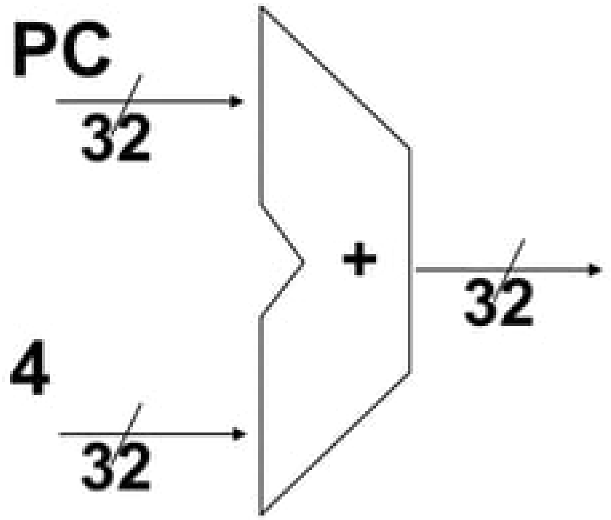
Components for MIPS subset

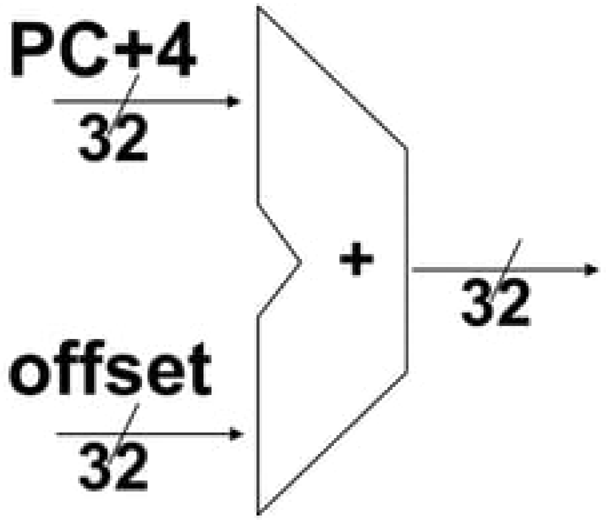
- Register,
- Adder
- ALU
- Multiplexer
- Register file
- Program memory
- Data memory
- Bit manipulation components

Components - register

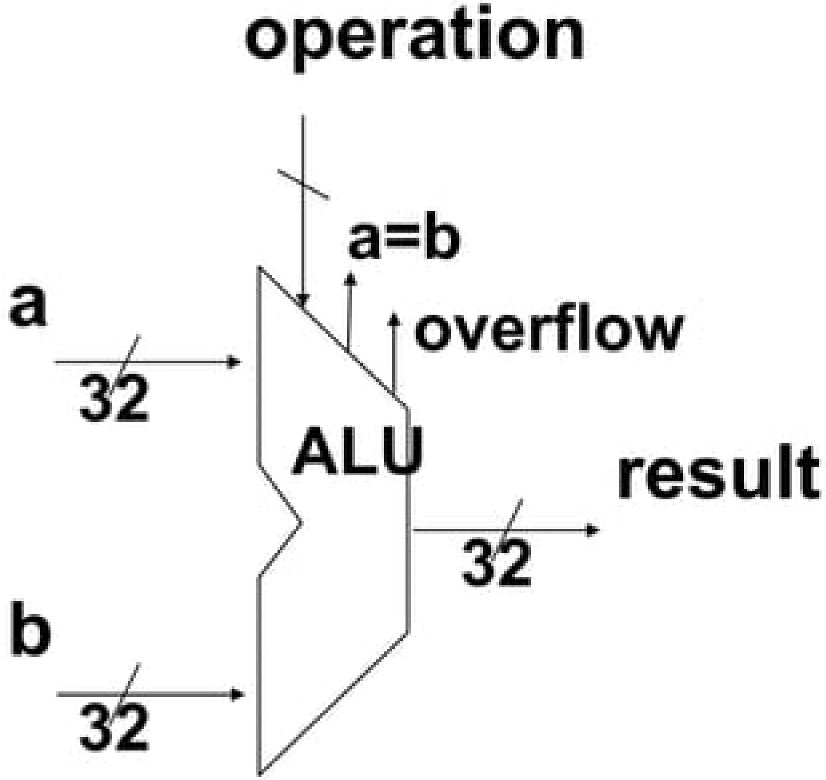


Components - adder

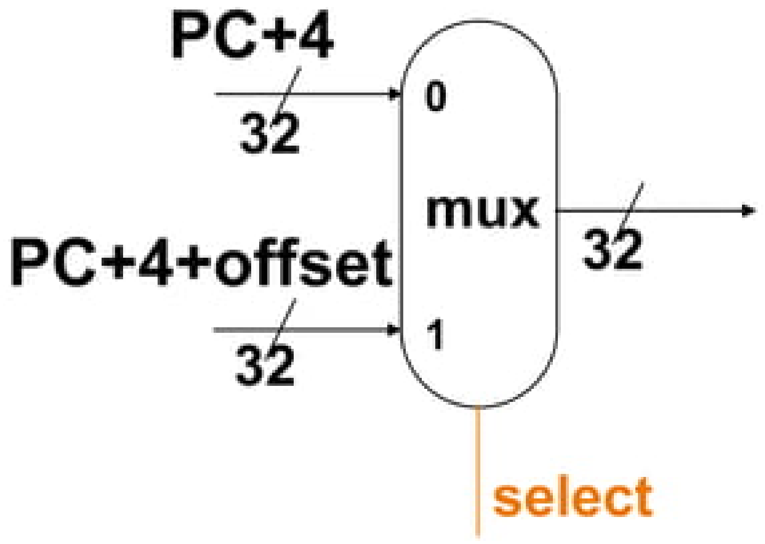




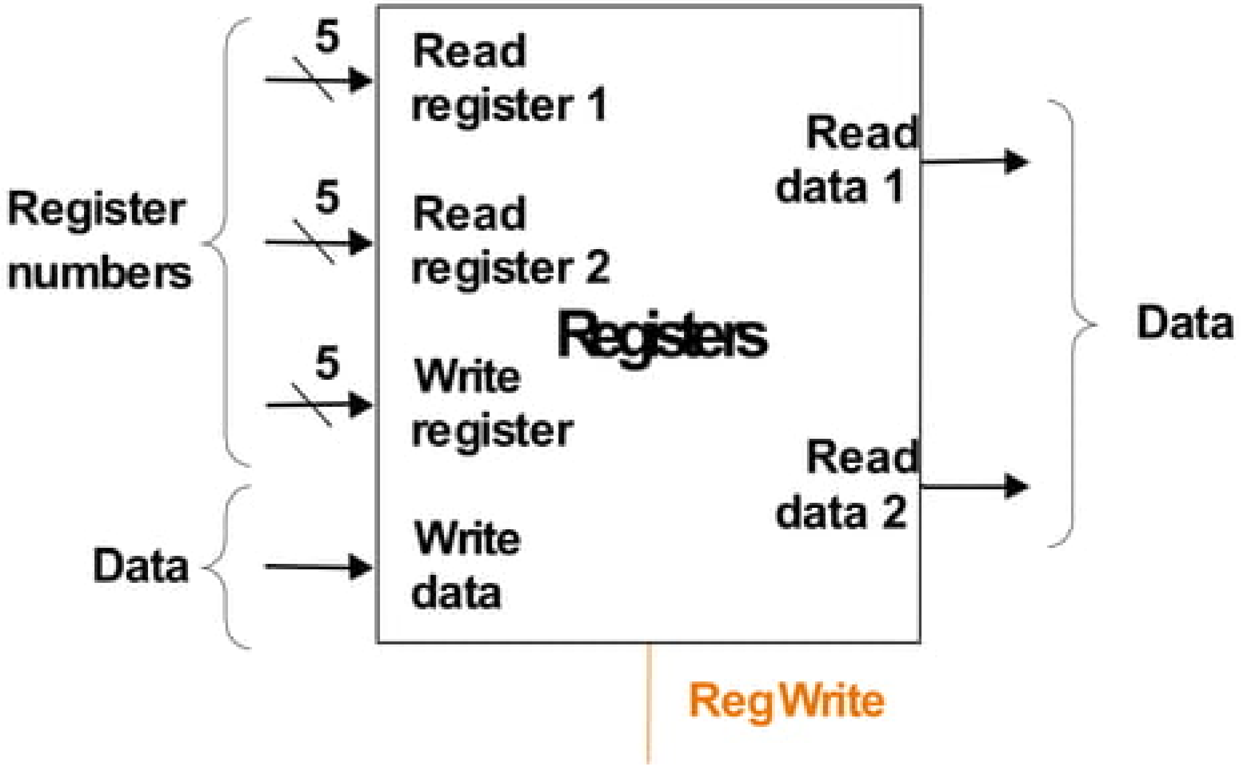
Components - ALU



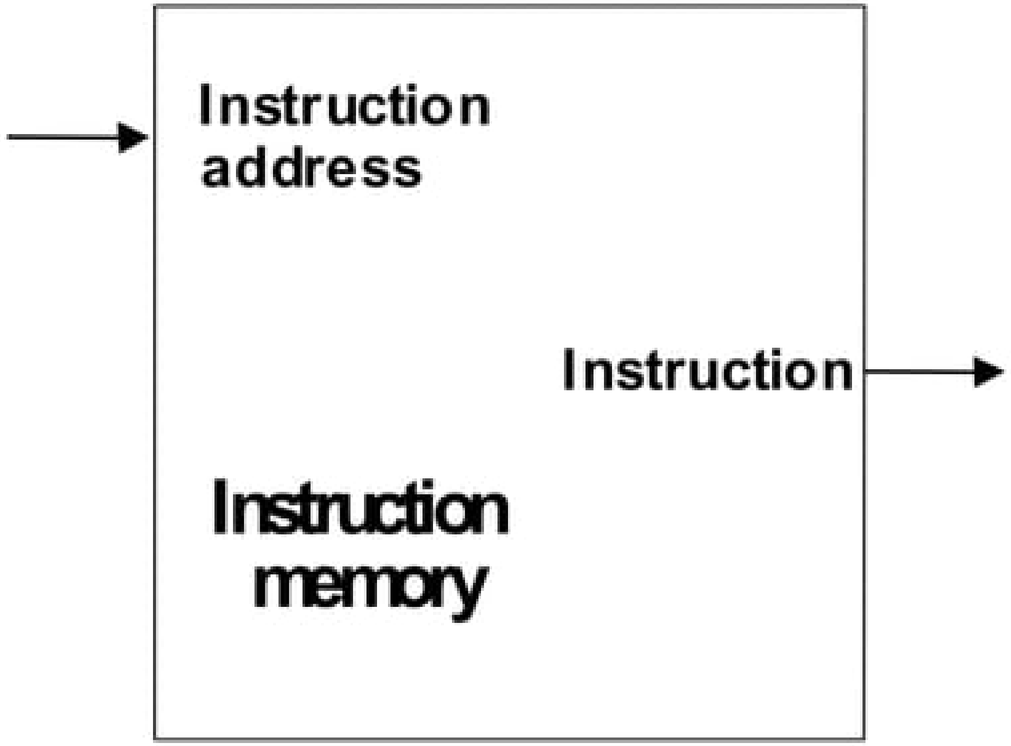
Components - multiplexers



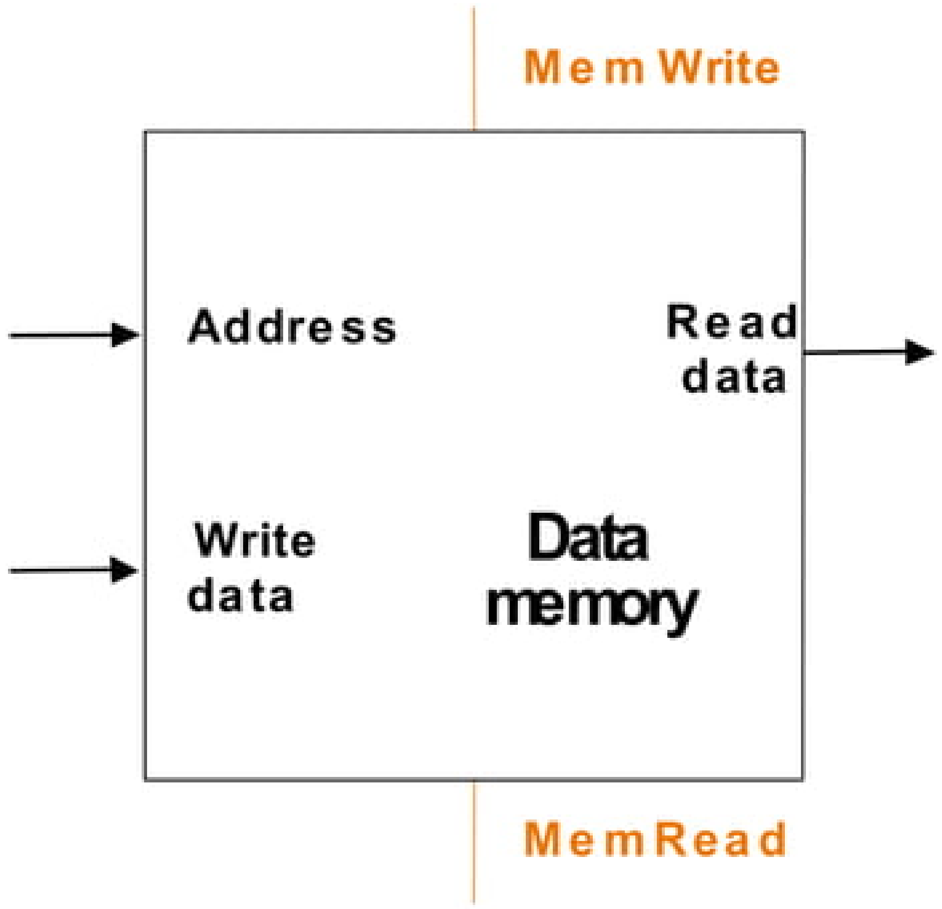
Components - register file



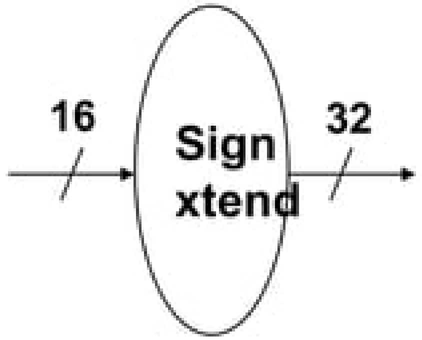
Components - program memory

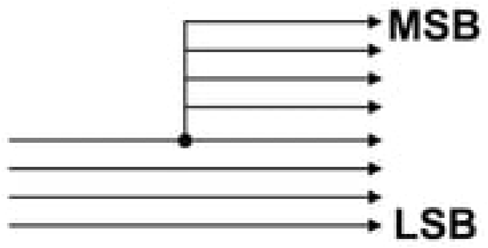


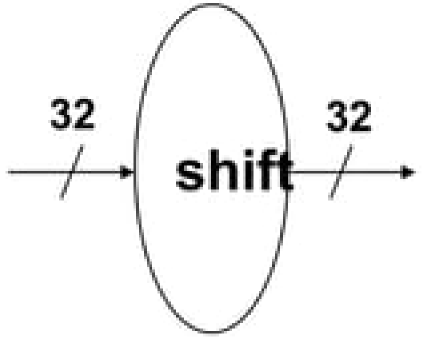
MIPS components - data memory

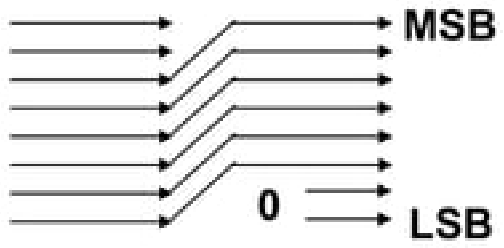


Components - bit manipulation circuits









MIPS subset for implementation

- Arithmetic - logic instructions
  - add, sub, and, or, slt
- Memory reference instructions
  - lw, sw
- Control flow instructions
  - beq, j

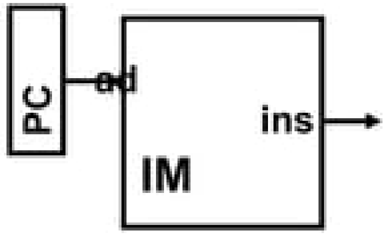
Datapath for add,sub,and,or,slt

- Fetch instruction
- Address the register file
- Pass operands to ALU actions
- Pass result to register file required
- Increment PC

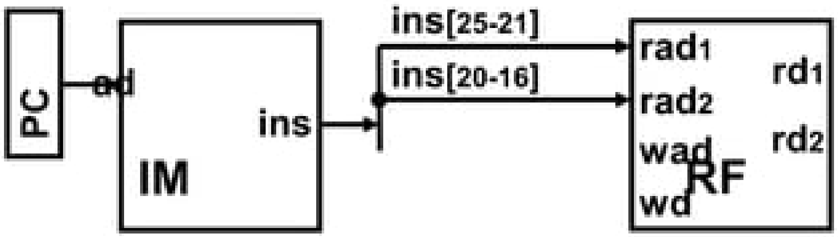
Format: add $t0, $s1, $s2

000000	10001	10010	01000	00000	100000

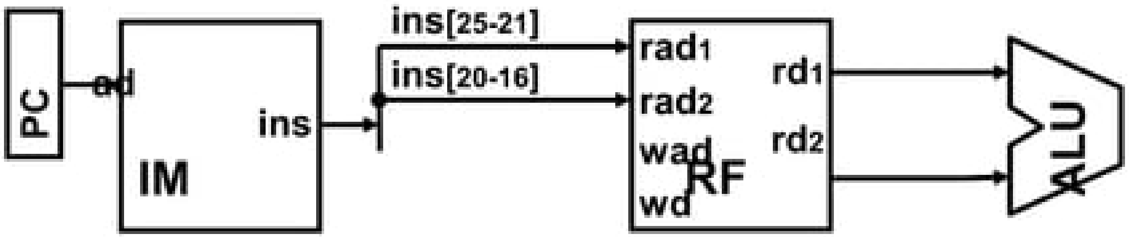
Fetching instruction



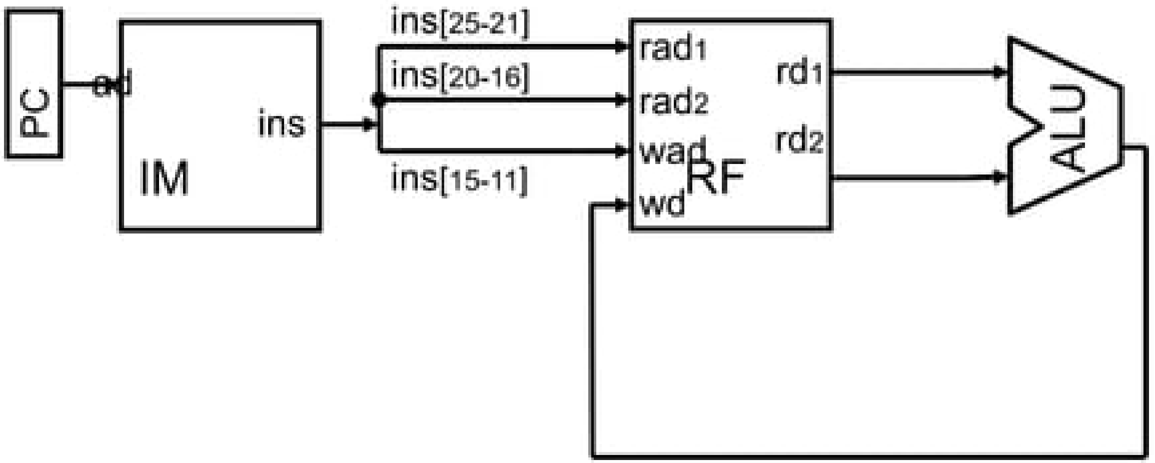
Addressing RF



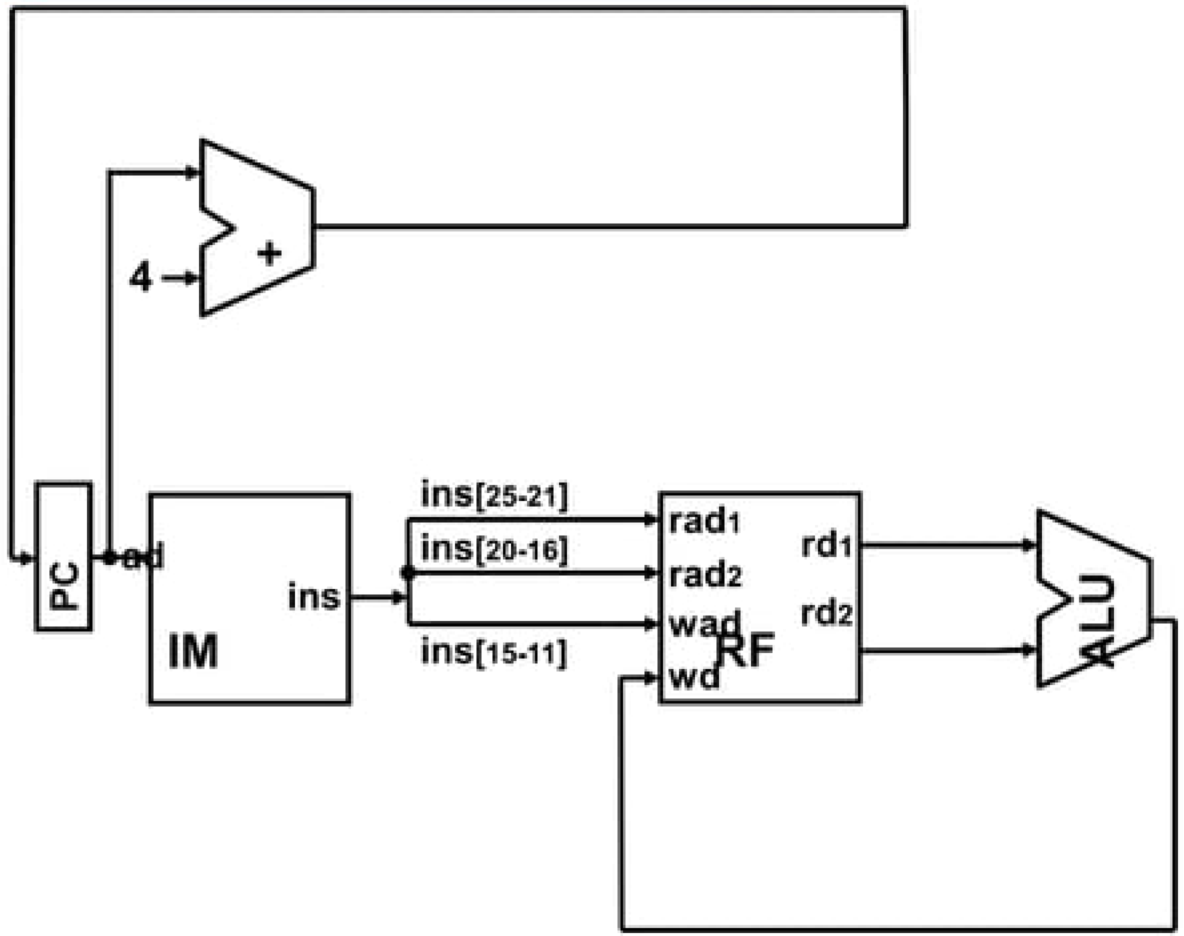
Passing operands to ALU



Passing the result to RF



Incrementing PC



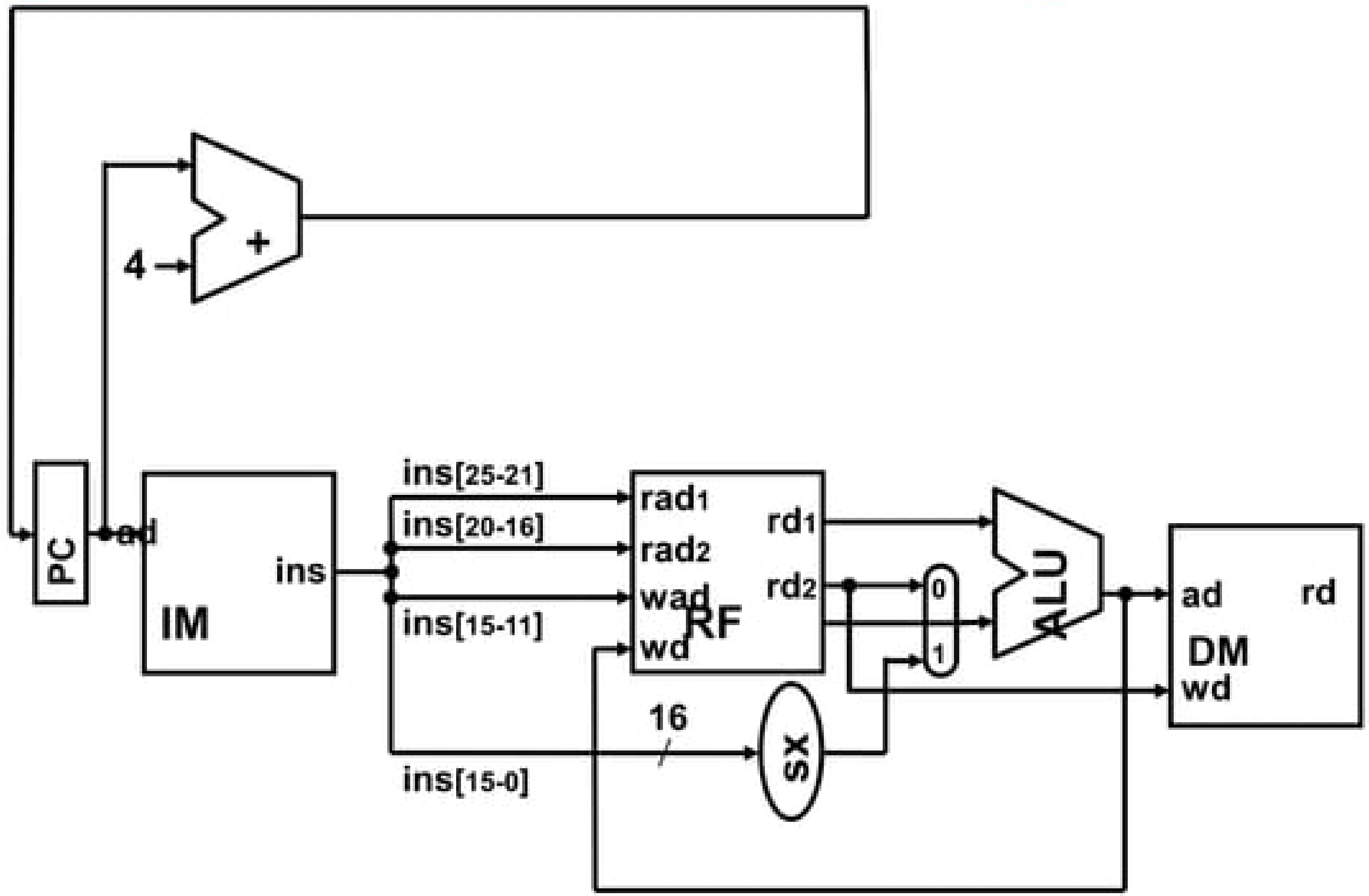
Load and Store instructions

- format : I

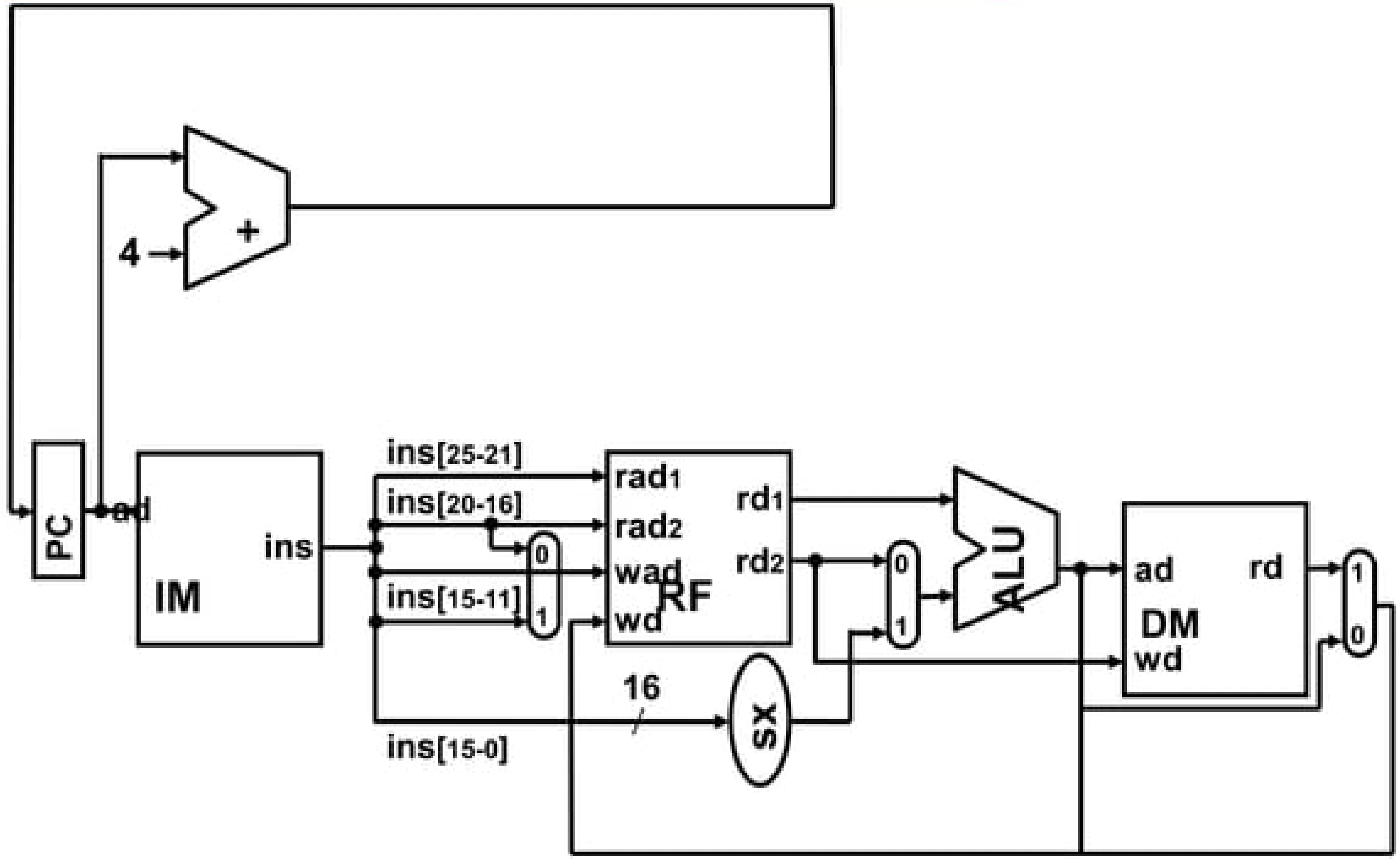
- Example: lw $t0, 32($s2)

35	18	9	32
op	rs	rt	16 bit number

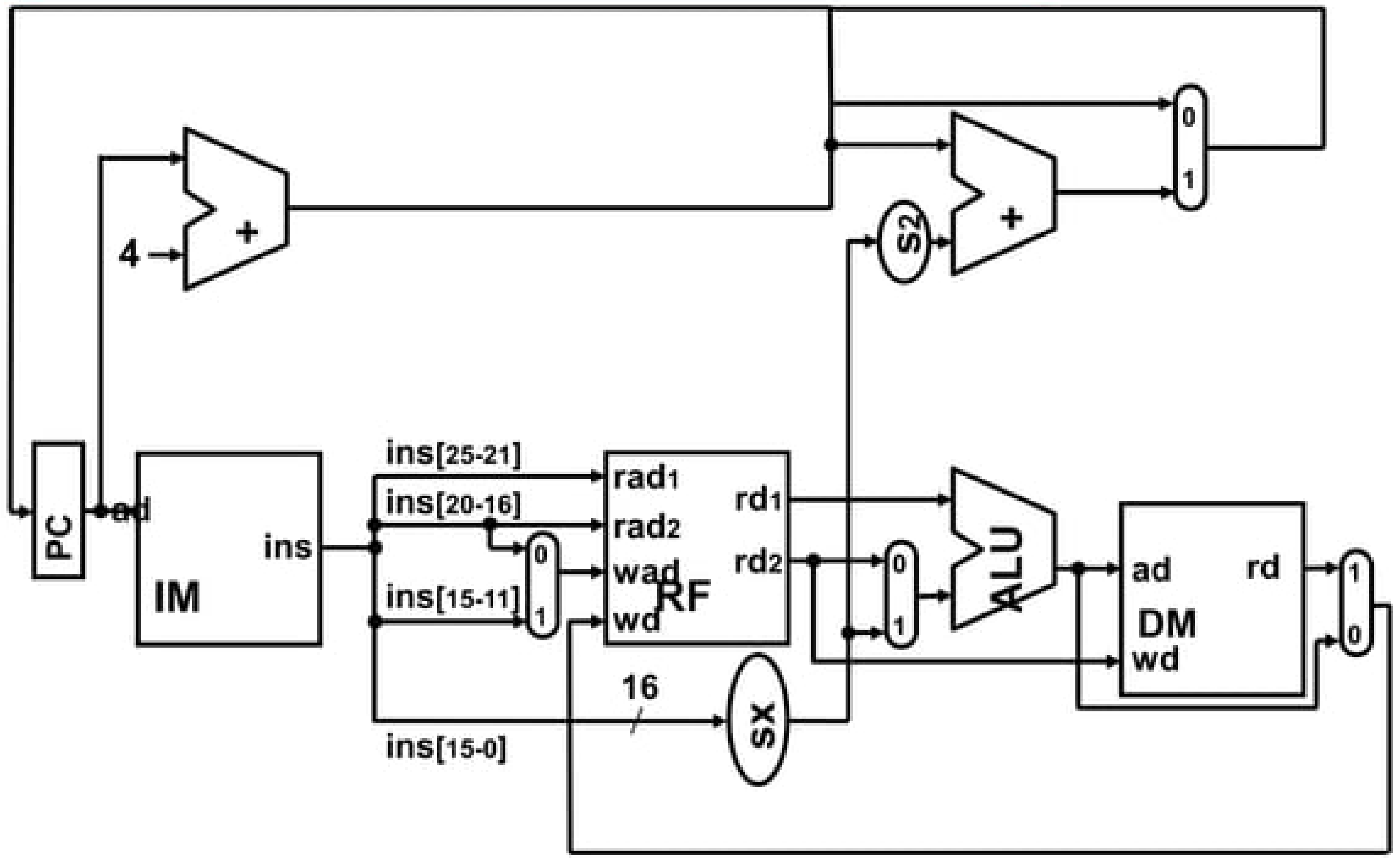
Adding "sw" instruction



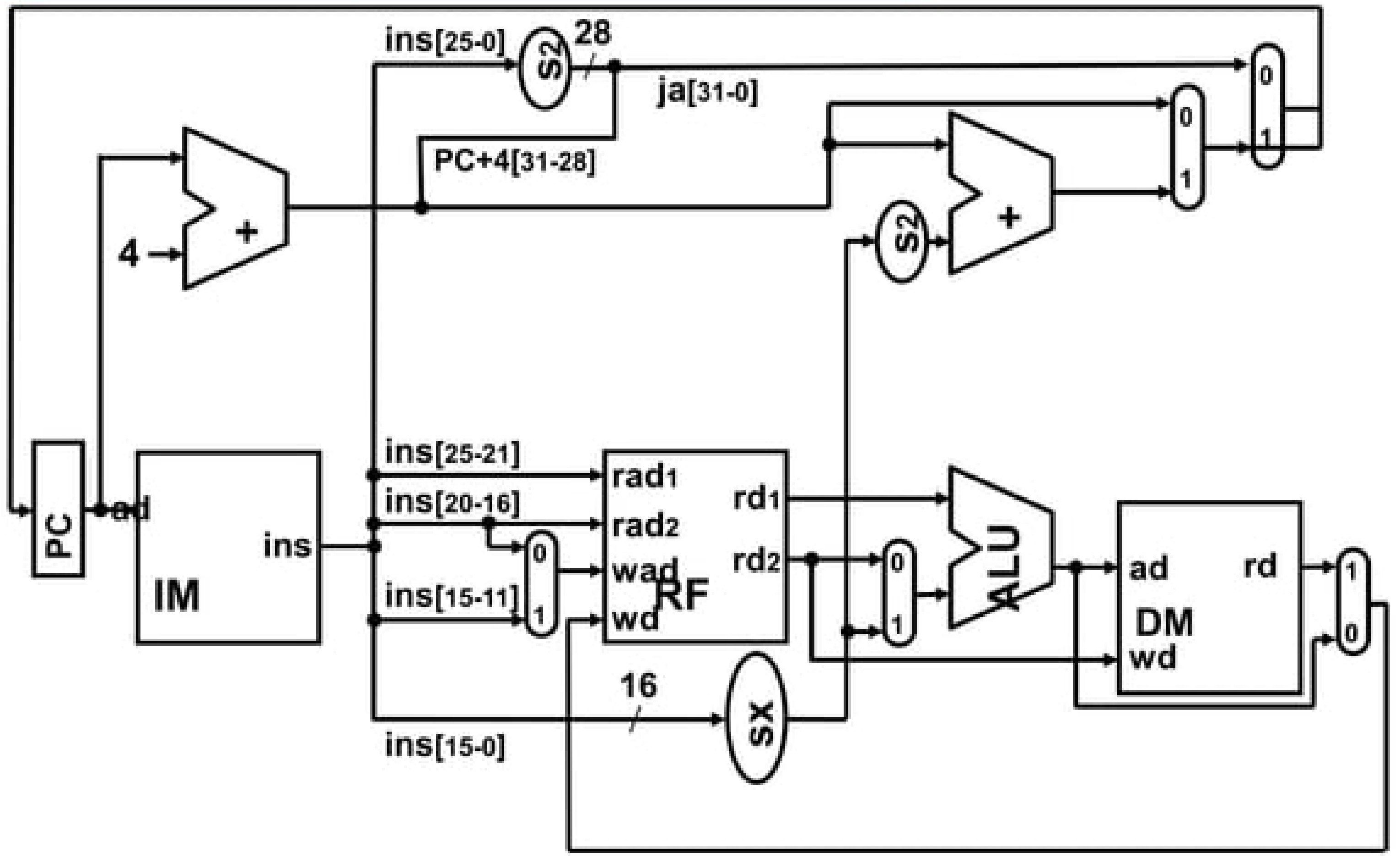
Adding "lw" instruction



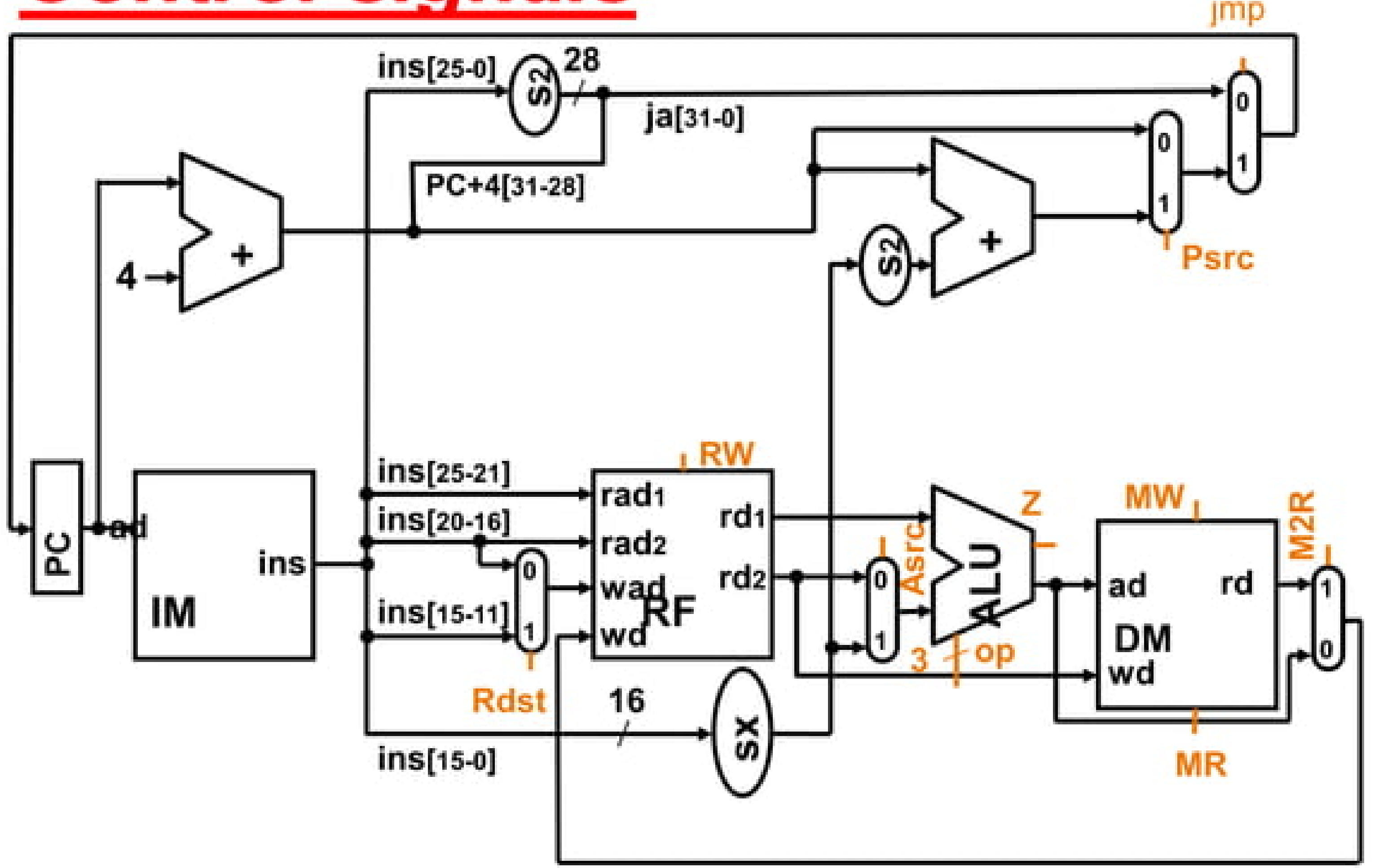
Adding "beq" instruction



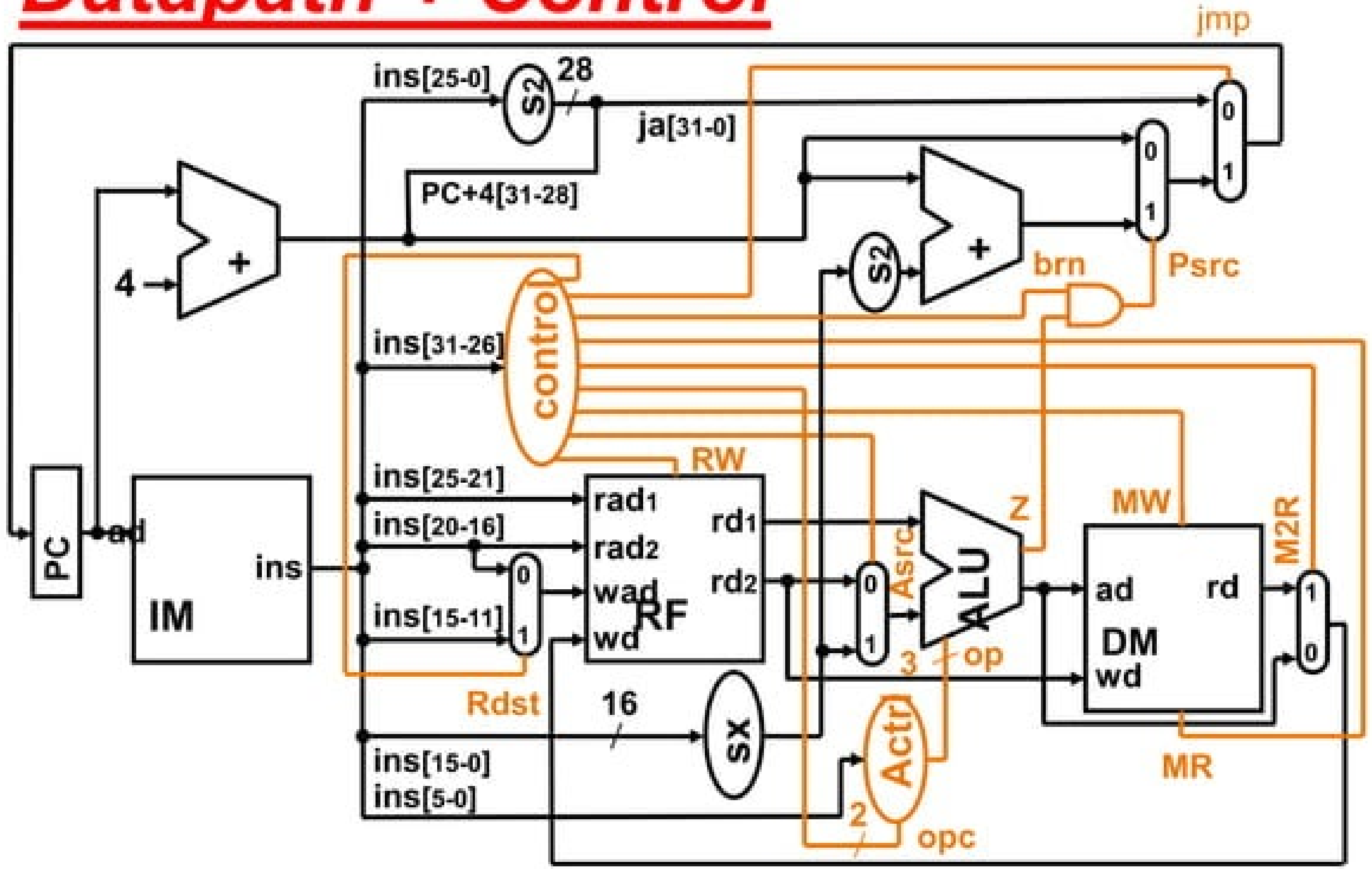
Adding "j" instruction



Control signals



Datapath + Control

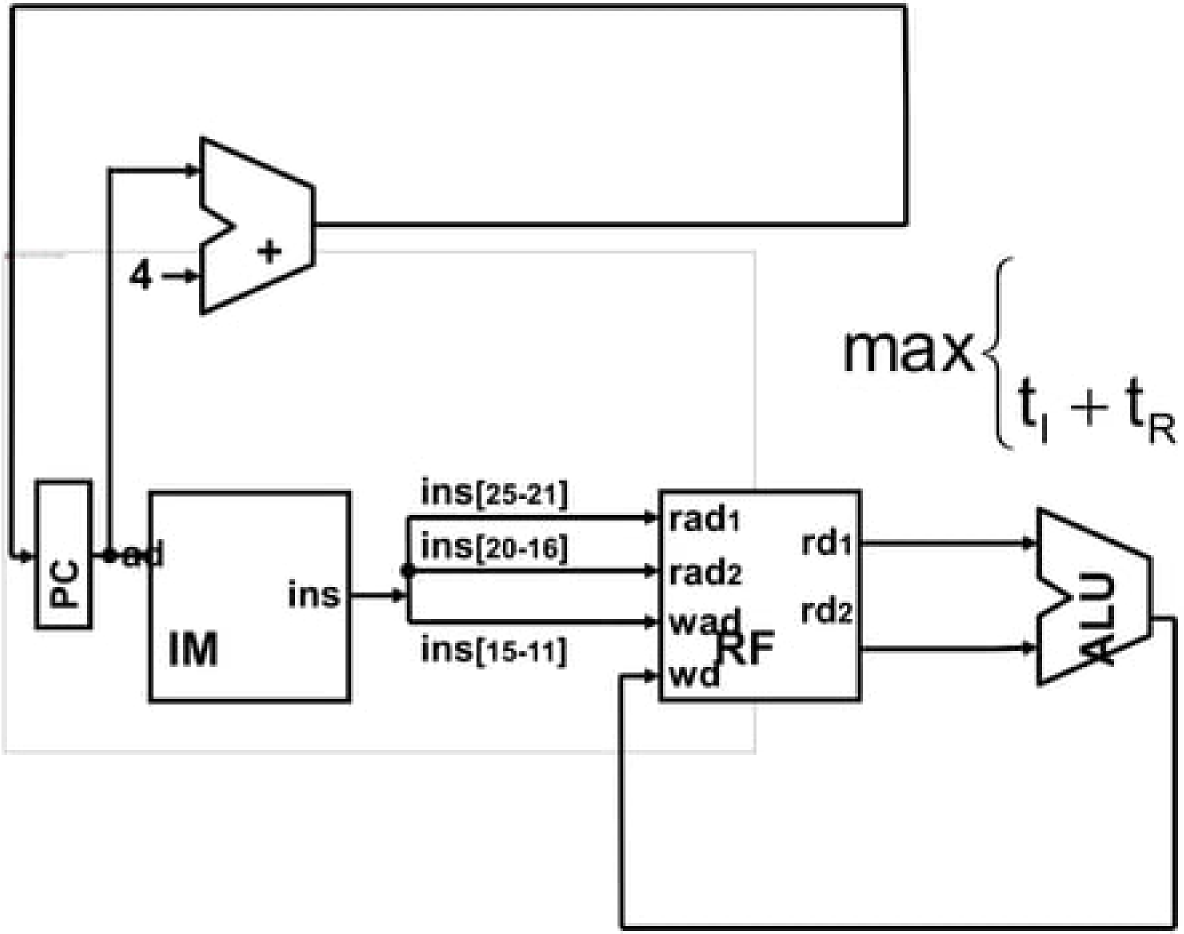


Analyzing performance

Component delays

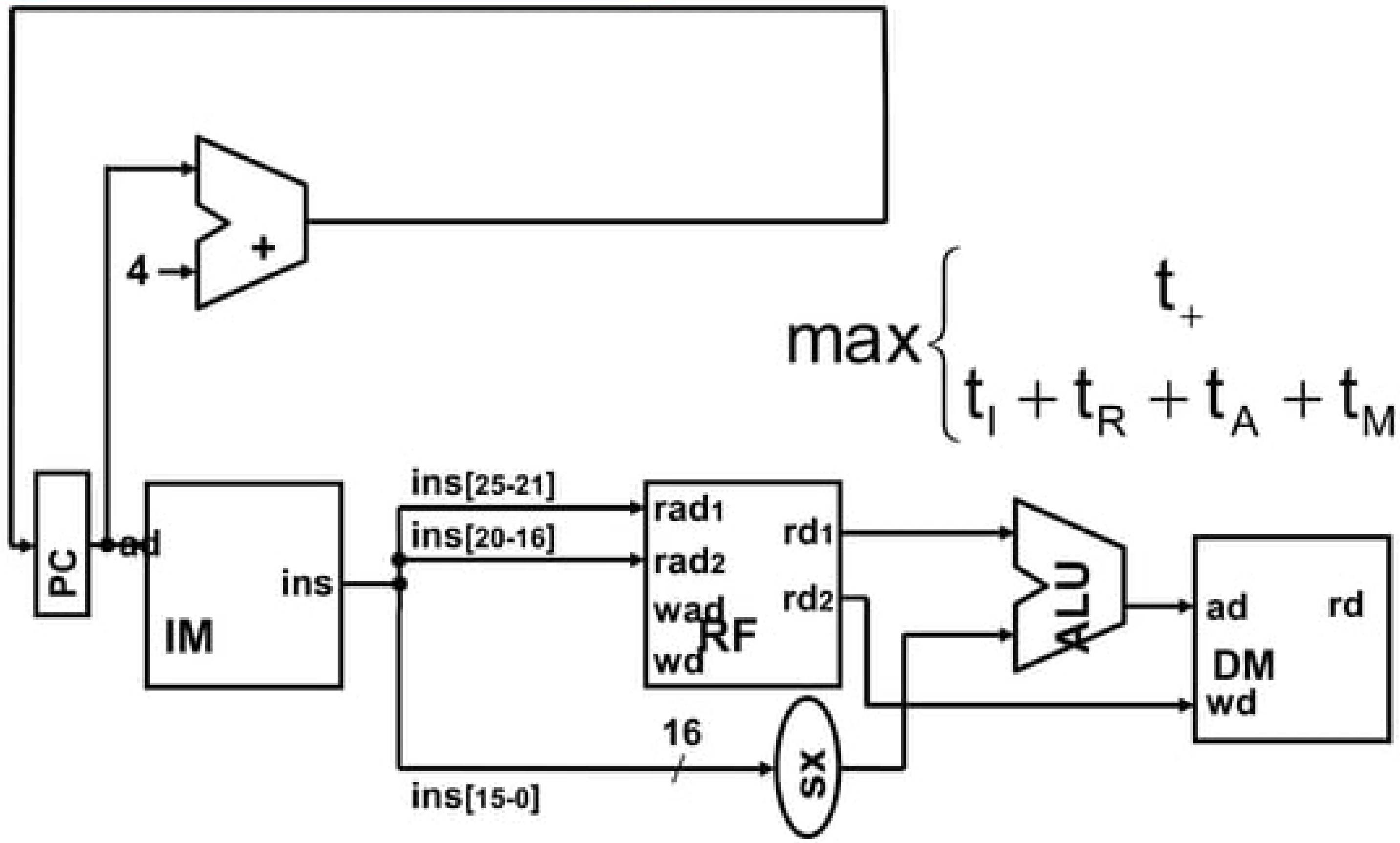
Register	0
Adder	t_+
ALU	t_A
Multiplexer	0
Register file	t_R
Program memory	t_I
Data memory	t_M
Bit manipulation components	0

Delay for {add, sub, and, or, slt}



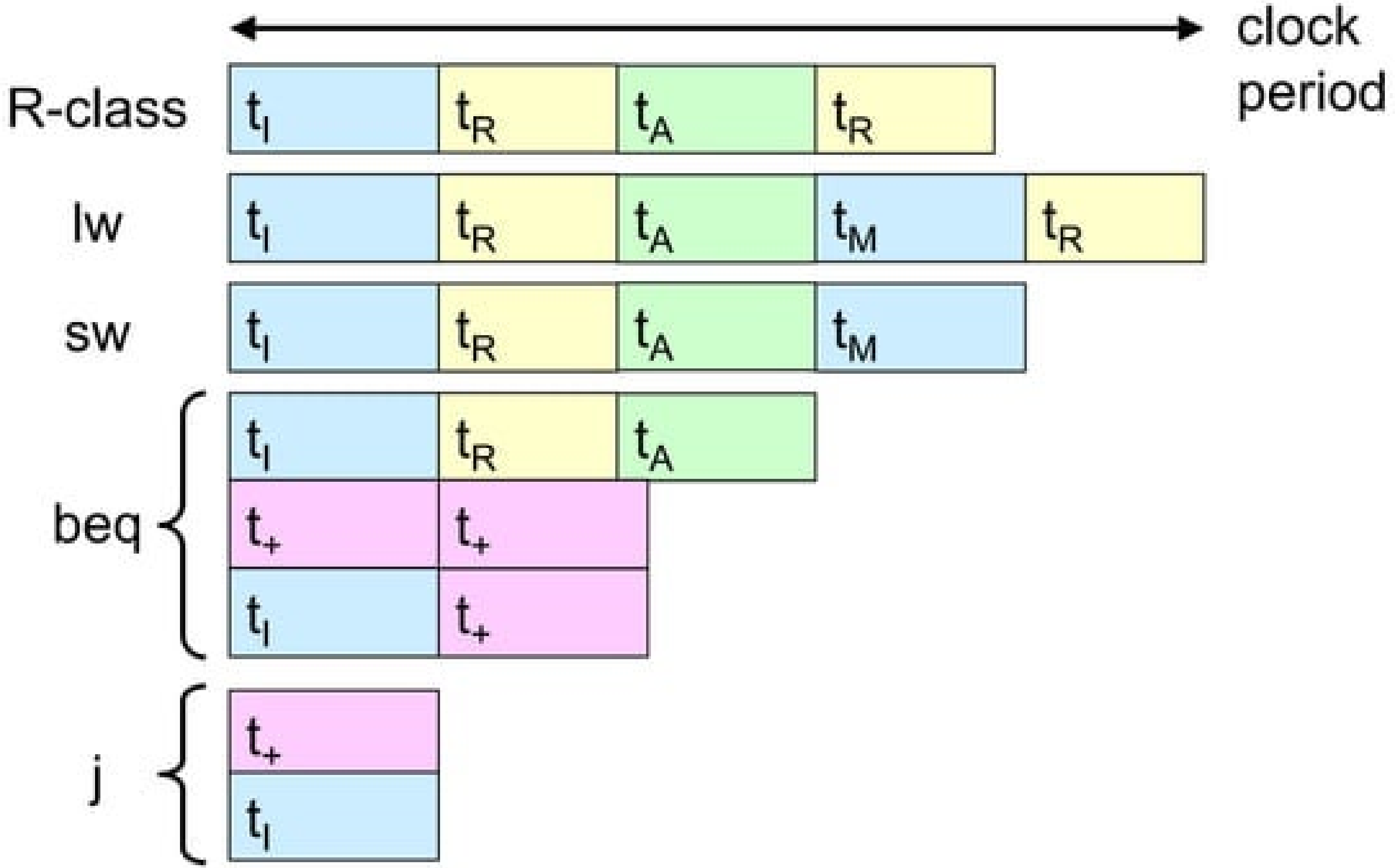
$$\max\begin{cases} t_+ \\ t_I + t_R + t_A + t_R \end{cases}$$

Delay for {sw}

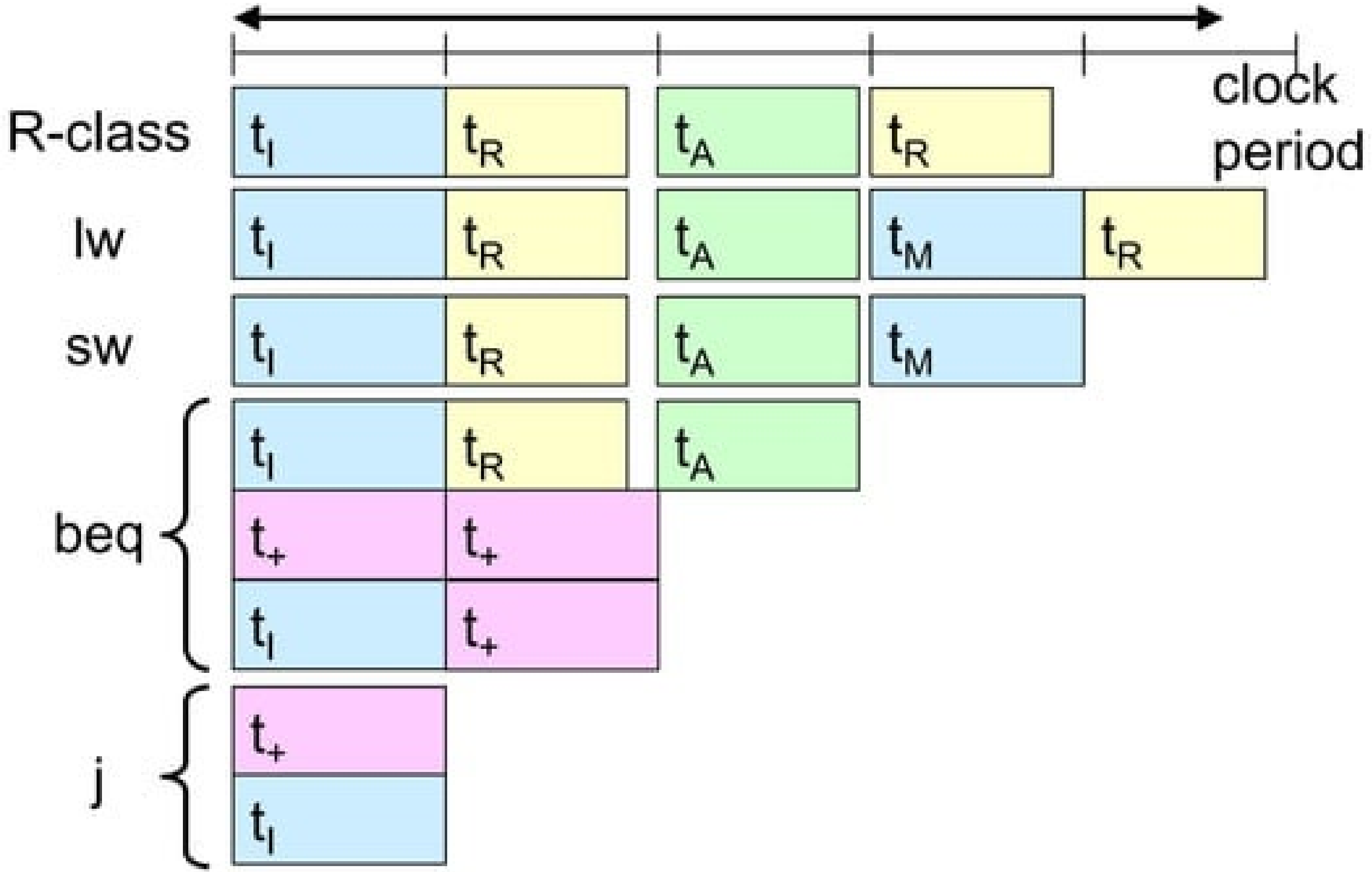


$$\max\begin{cases} t_{+} \\ t_I + t_R + t_A + t_M \end{cases}$$

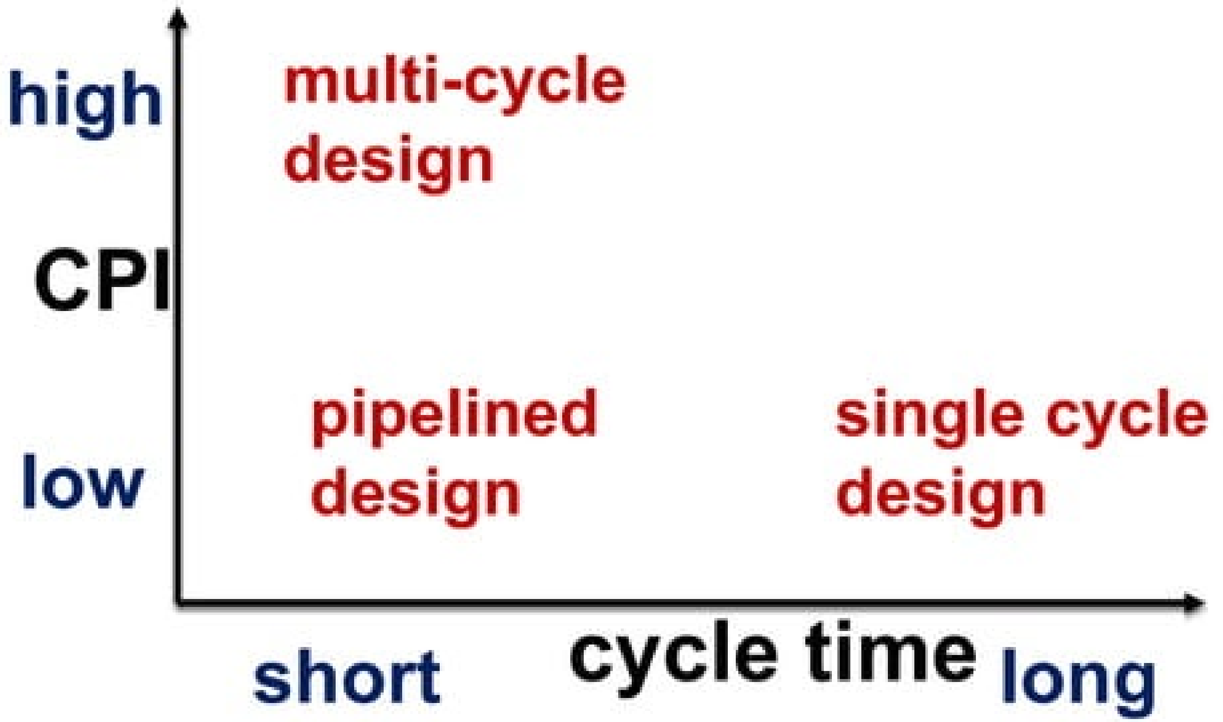
Clock period in single cycle design



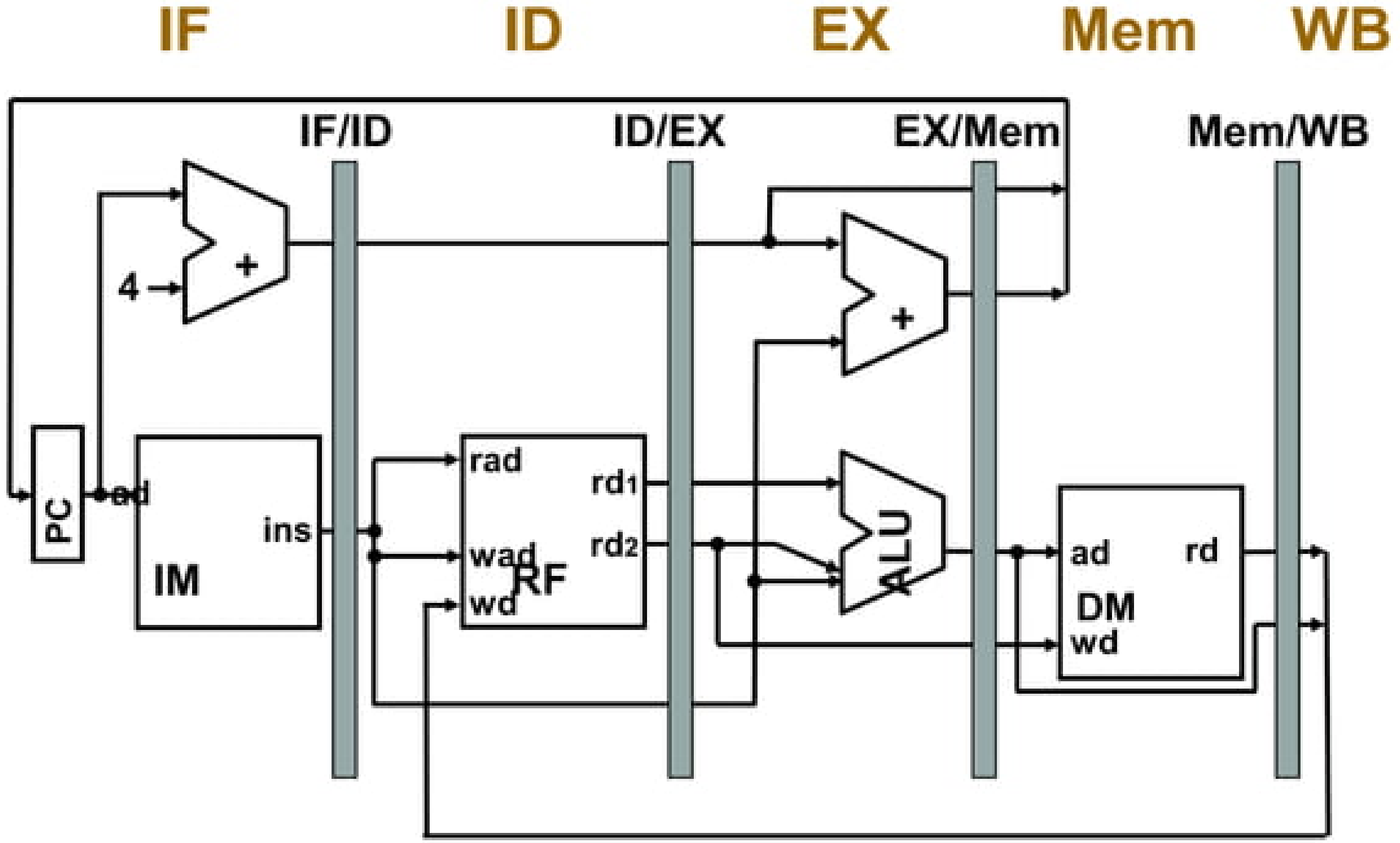
Clock period in multi-cycle design



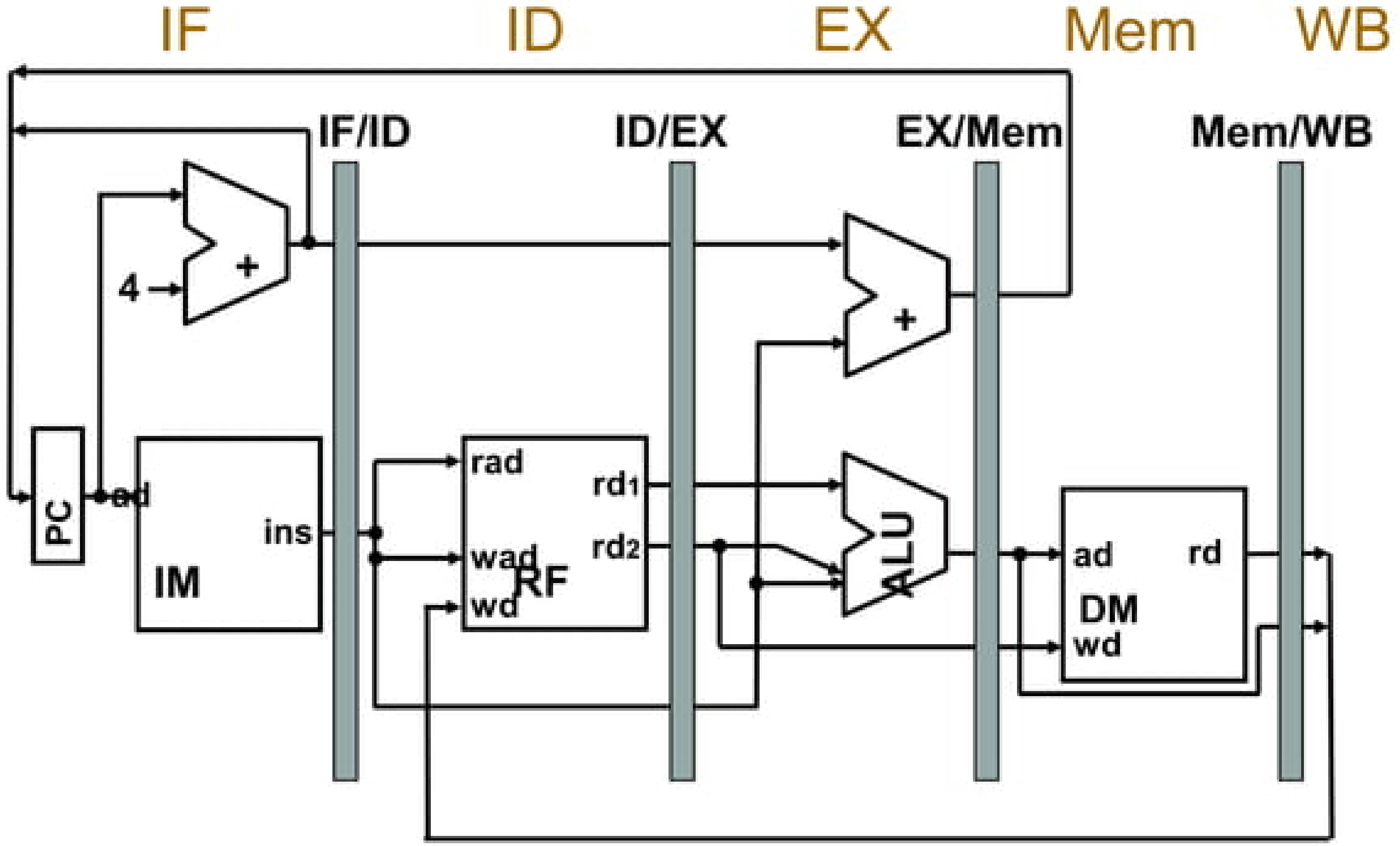
Cycle time and CPI



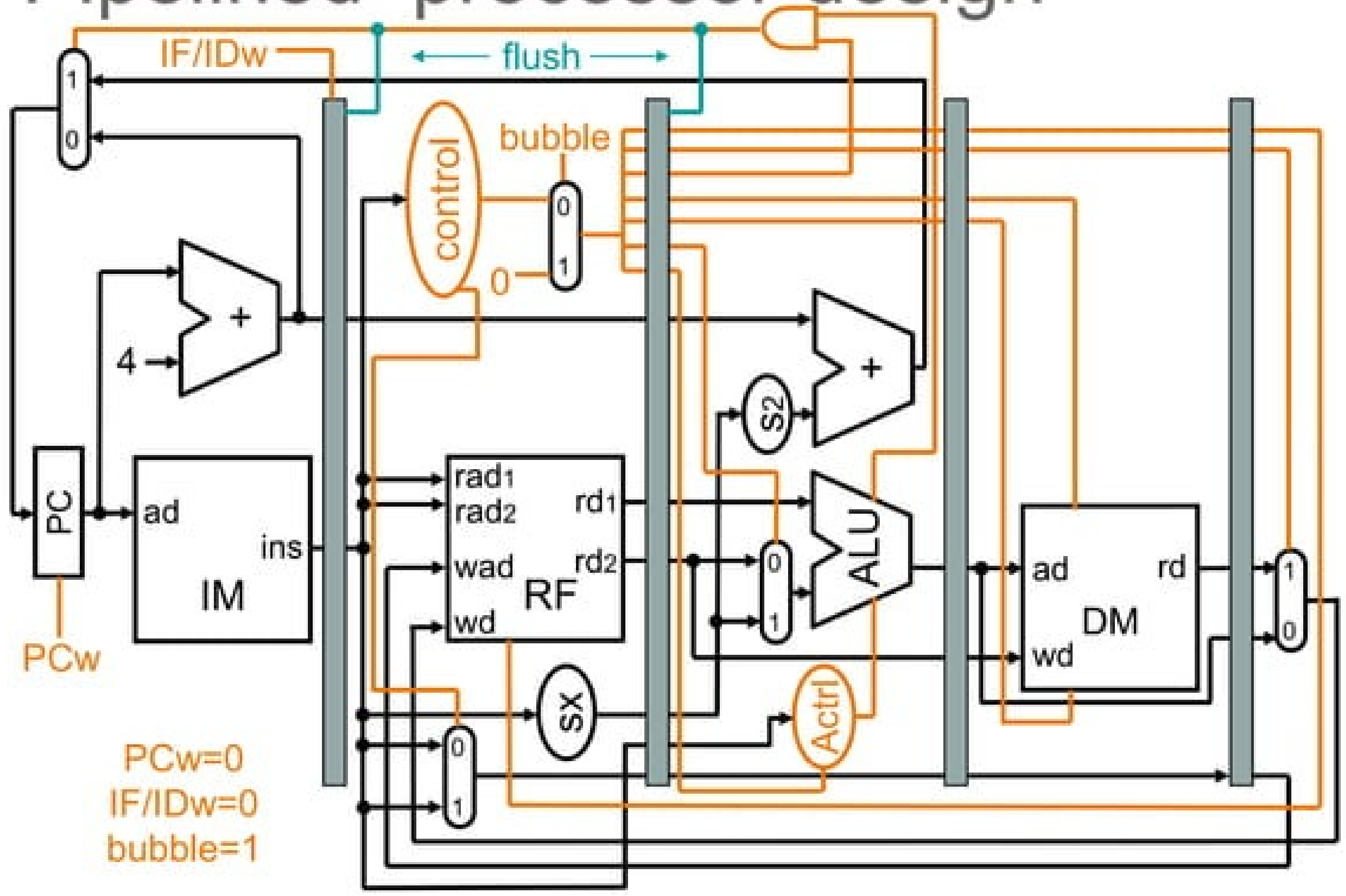
PIpelined datapath (abstract)



Fetch new instruction every cycle



Pipelined processor design

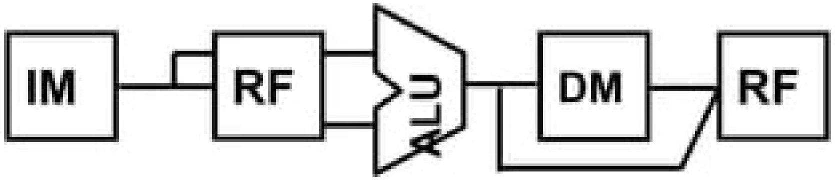


PCw=0
IF/IDw=0
bubble=1

Graphical representation

5 stage pipeline

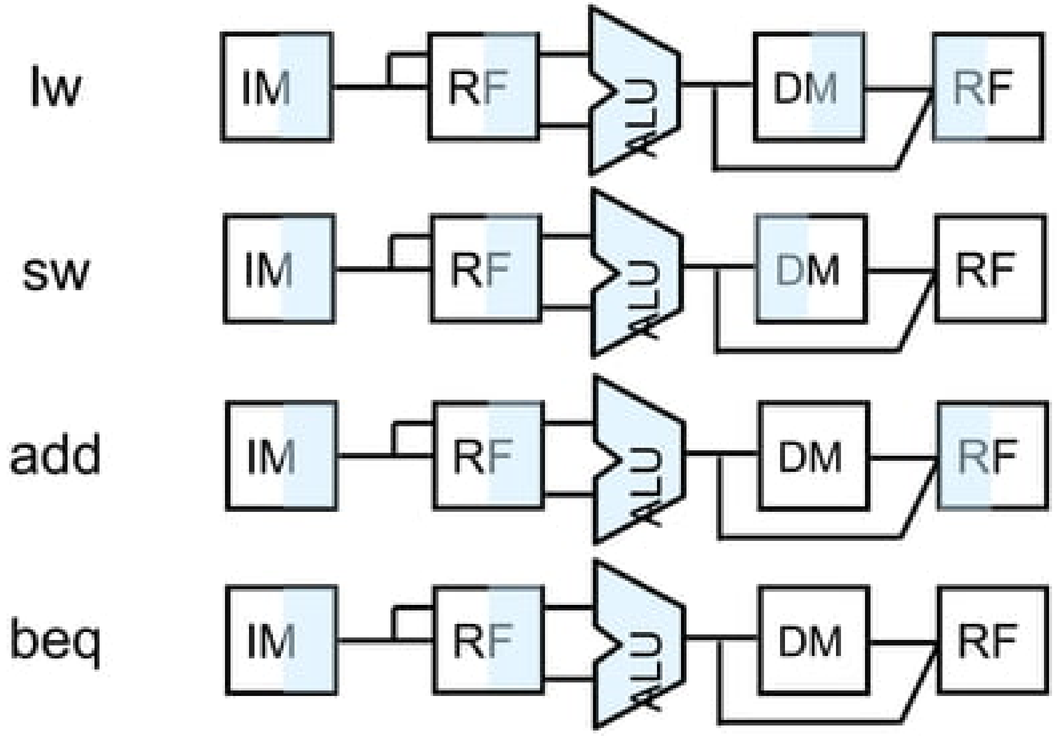
stages



actions

IF	ID	EX	Mem	WB

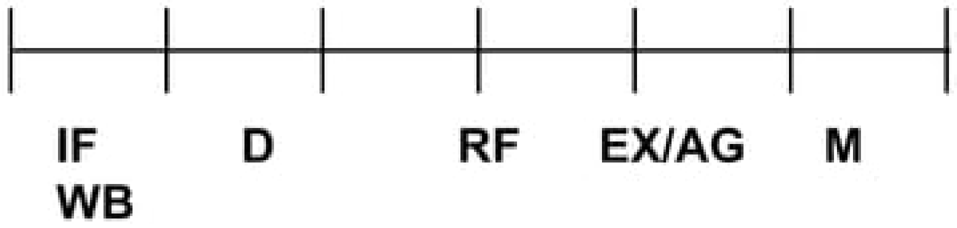
Usage of stages by instructions



Pipelining

Simple multicycle design :

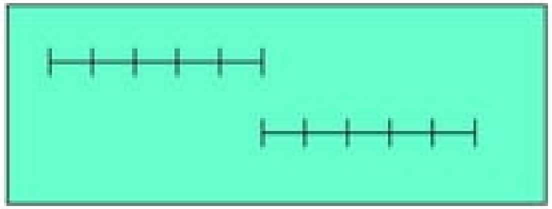
- Resource sharing across cycles
- All instructions may not take same cycles



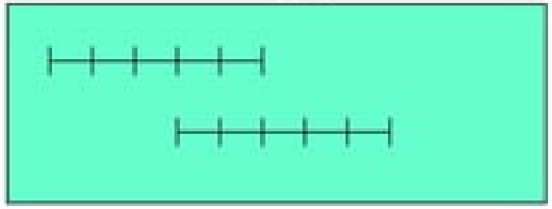
- Faster throughput with pipelining

Degree of overlap   Depth

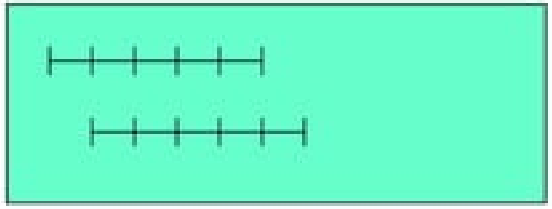
Serial



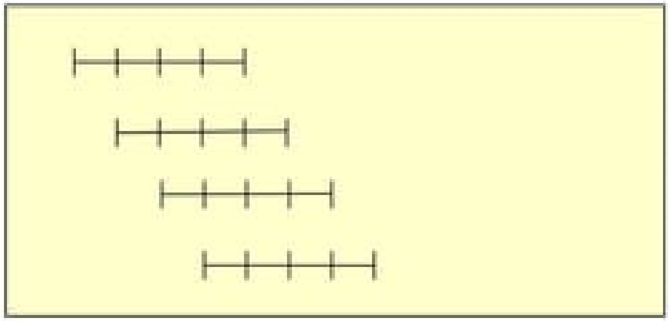
Overlapped



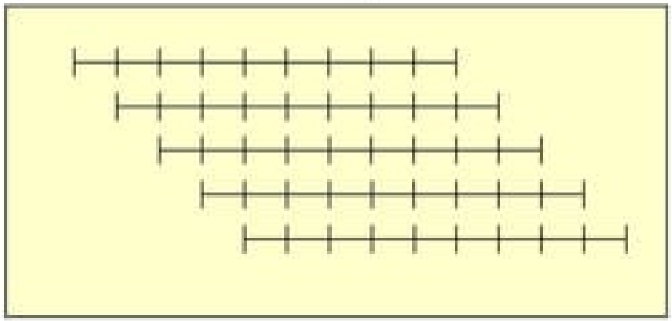
Pipelined



Shallow



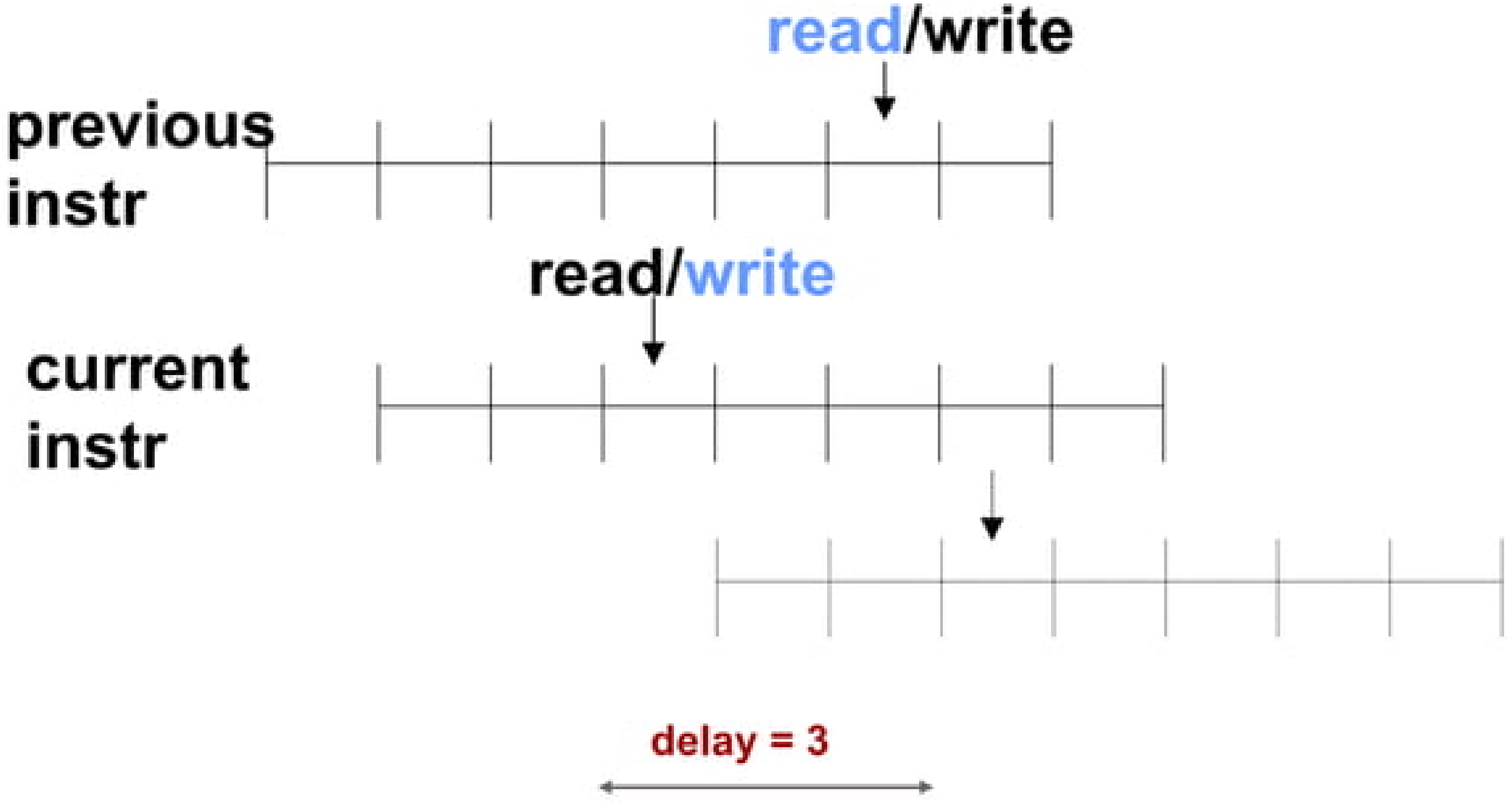
Deep



Hazards in Pipelining

- Procedural dependencies => Control hazards
  - cond and uncond branches, calls/returns
- Data dependencies => Data hazards
  - RAW (read after write)
  - WAR (write after read)
  - WAW (write after write)
- Resource conflicts => Structural hazards
  - use of same resource in different stages

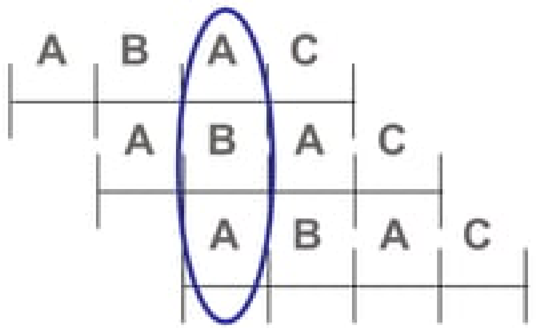
Data Hazards



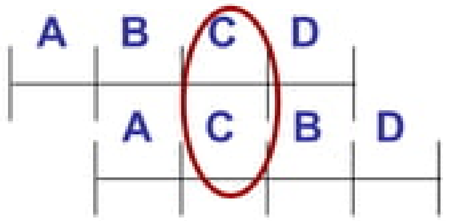
Structural Hazards

Caused by Resource Conflicts

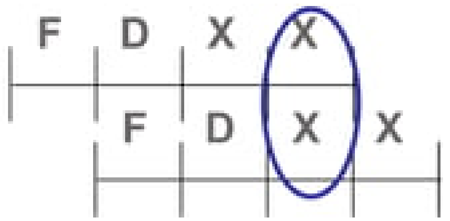
- Use of a hardware resource in more than one cycle



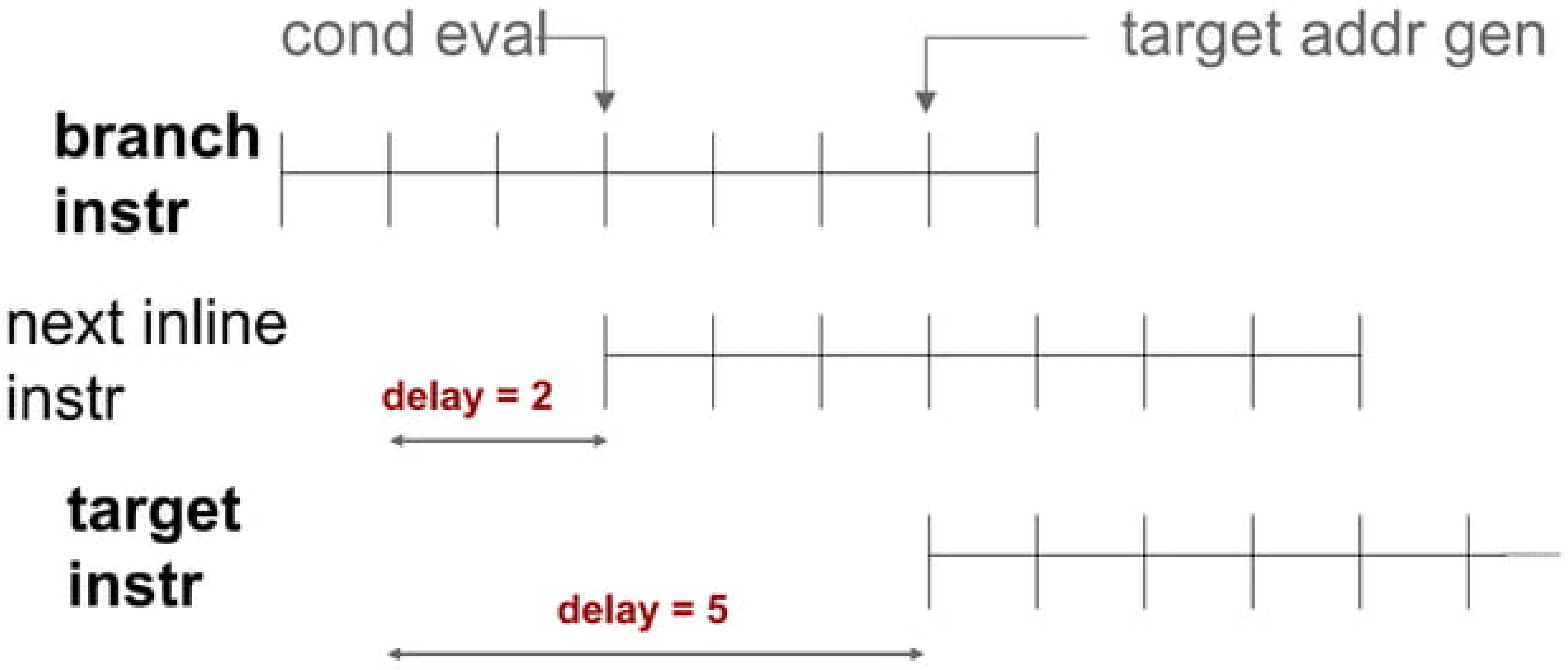
- Different sequences of resource usage by different instructions



- Non-pipelined multi-cycle resources

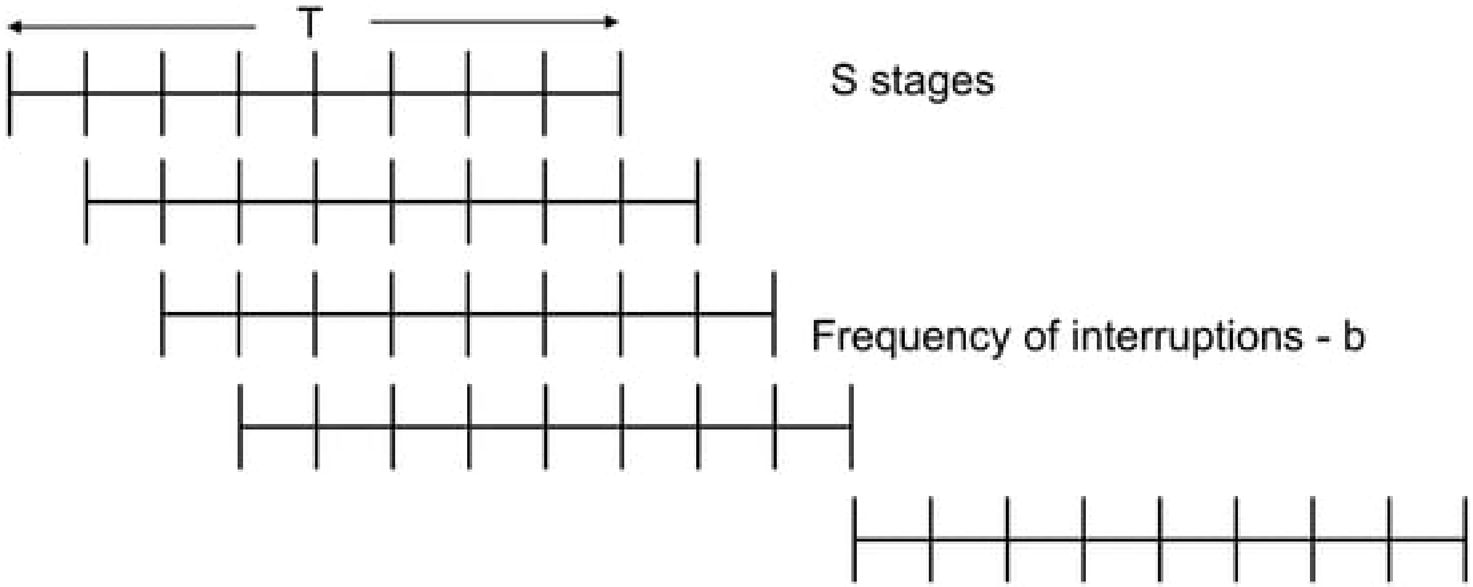


Control Hazards



- the order of *cond eval* and *target addr gen* may be different
- *cond eval* may be done in previous instruction

Pipeline Performance



CPI = 1 + (S - 1) * b

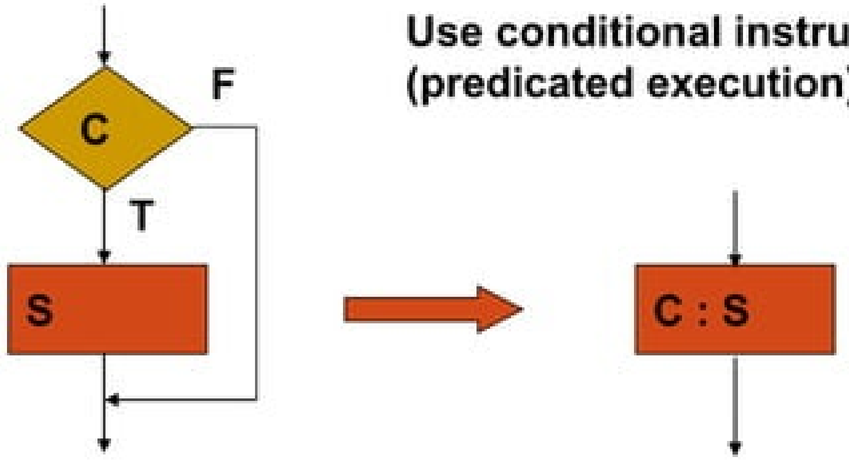
Time = CPI * T / S

Improving Branch Performance

- Branch Elimination
  - Replace branch with other instructions
- Branch Speed Up
  - Reduce time for computing CC and TIF
- Branch Prediction
  - Guess the outcome and proceed, undo if necessary
- Branch Target Capture

Branch Elimination

Use conditional instructions (predicated execution)



```
OP1
BC  CC = Z, * + 2
ADD  R3, R2, R1
OP2
```

→

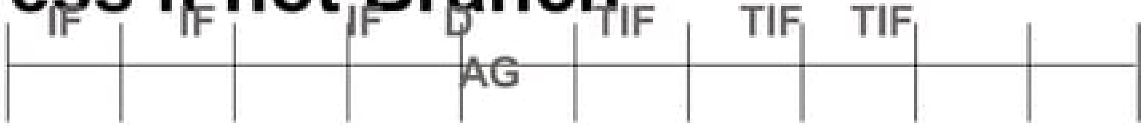
```
OP1
ADD  R3, R2, R1, NZ
OP2
```

Branch Speed Up :

Early target address generation

- Assume each instruction is Branch
- Generate target address while decoding
- If target in same page omit translation
- After decoding discard target address if not Branch

BC



Branch Prediction

- Treat conditional branches as unconditional branches / NOP
- Undo if necessary

Strategies:

- Fixed *(always guess inline)*
- Static *(guess on the basis of instruction type)*
- Dynamic *(guess based on recent history)*

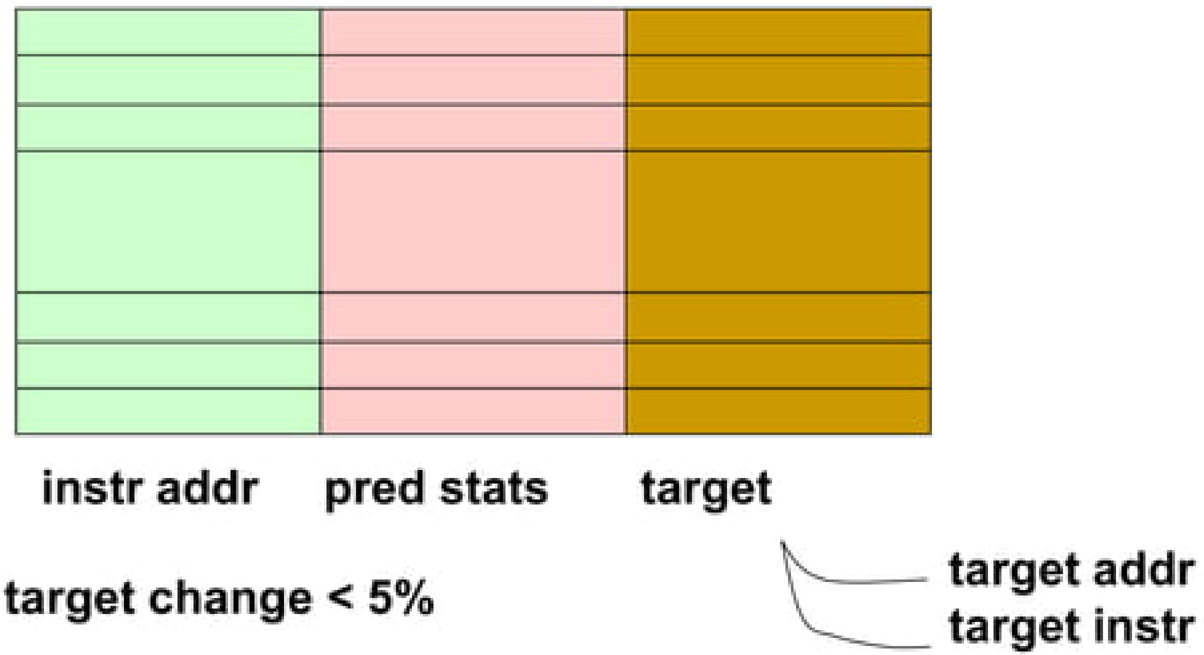
Static Branch Prediction

Instr	%	Guess	Branch	Correct
uncond	14.5	always	100%	14.5%
cond	58	never	54%	27%
loop	9.8	always	91%	9%
call/ret	17.7	always	100%	17.7%

Total 68.2%

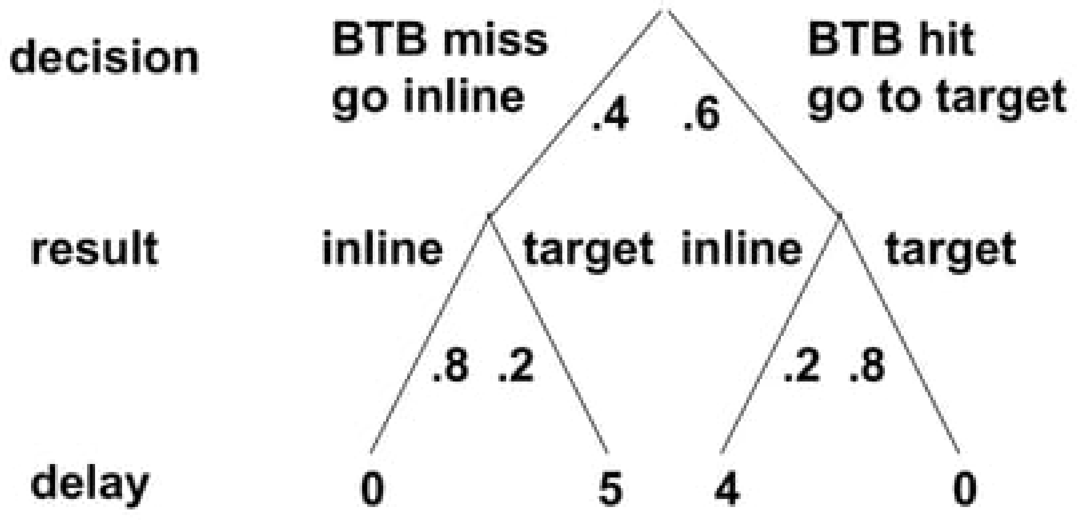
Branch Target Capture

- Branch Target Buffer (BTB)
- Target Instruction Buffer (TIB)



prob of target change < 5%

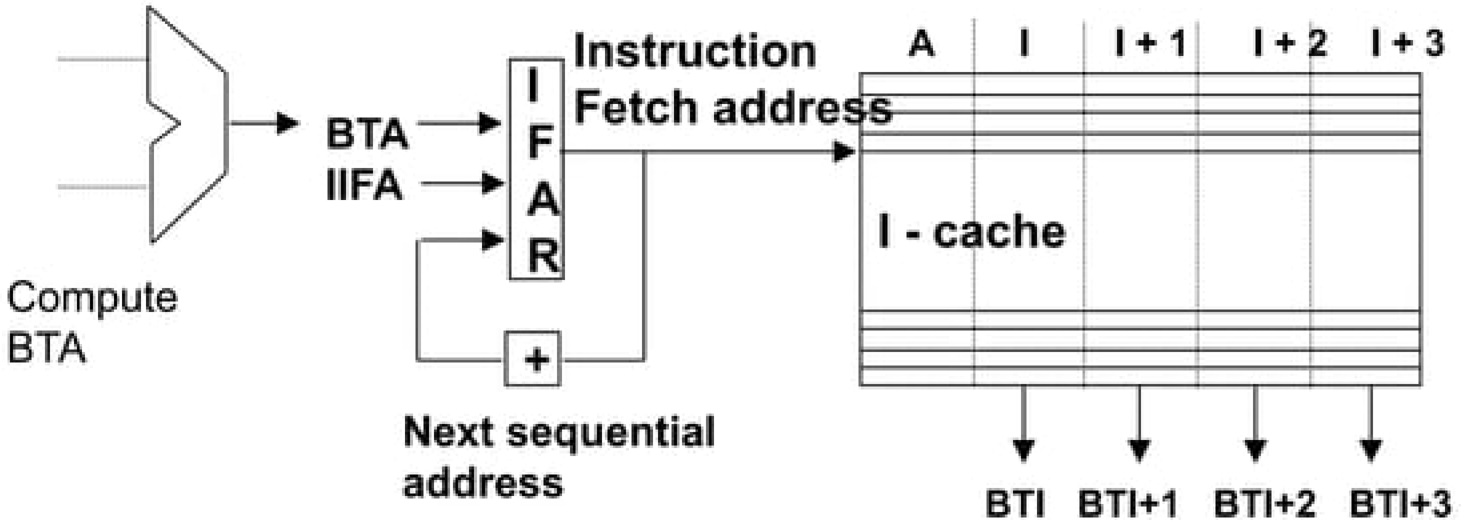
BTB Performance



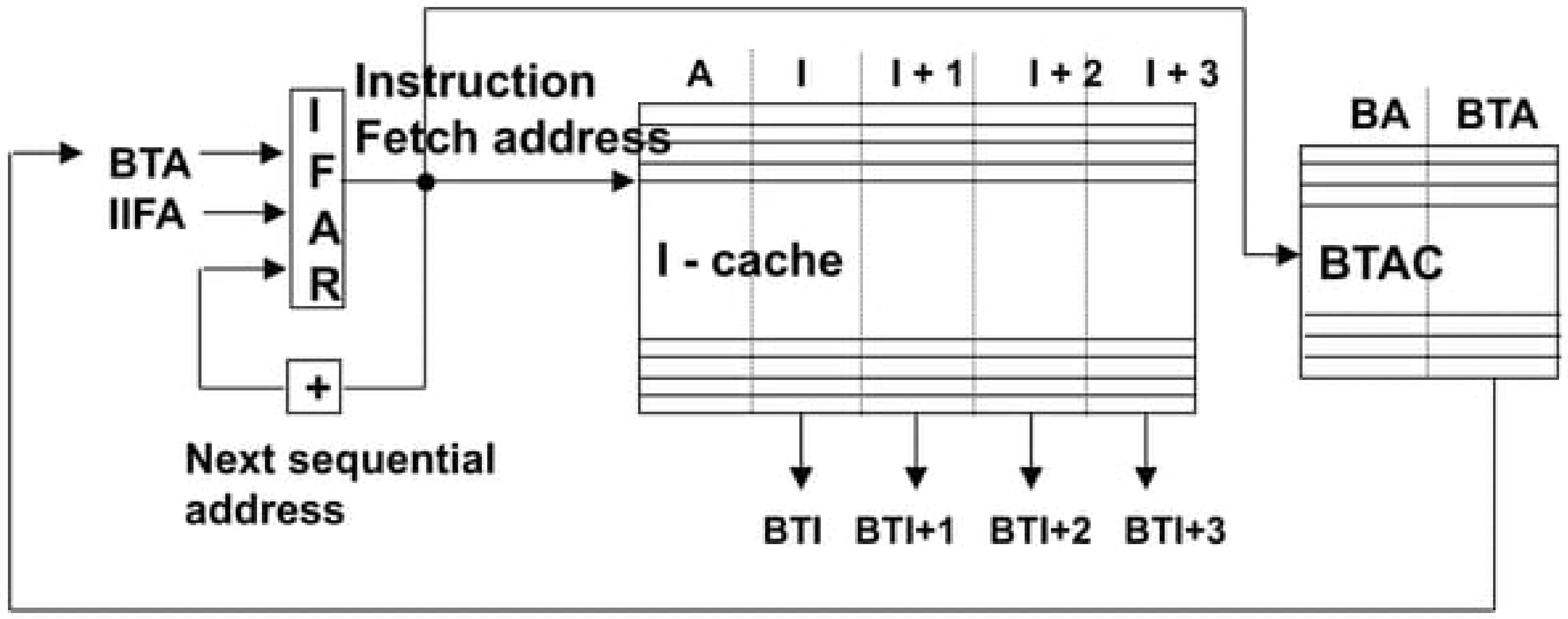
.4*.8*0 + .4*.2*5 + .6*.2*4 + .6*.8*0 = 0.88
(Eff.Delay)

Compute/fetch scheme

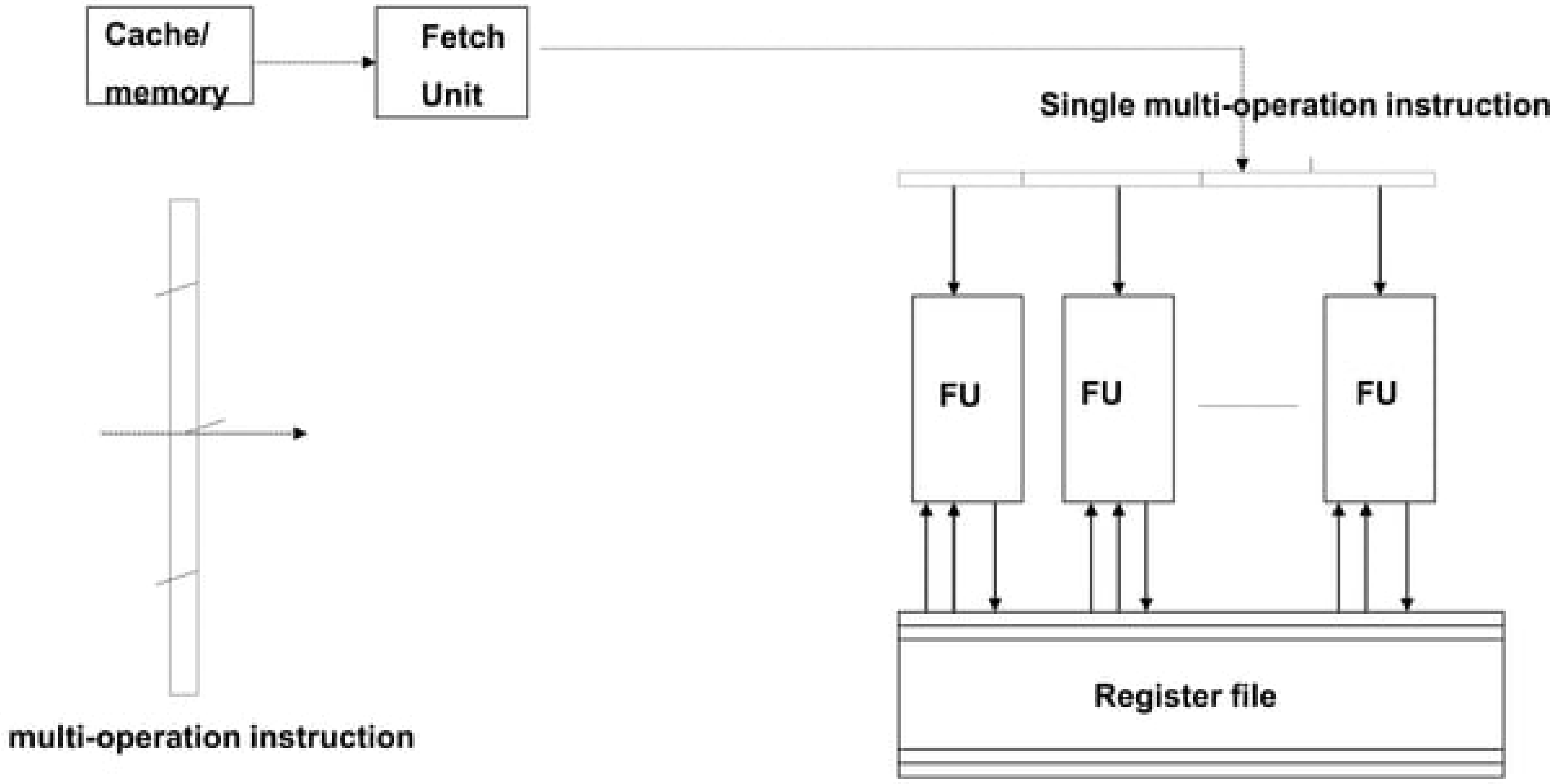
(no dynamic branch prediction)



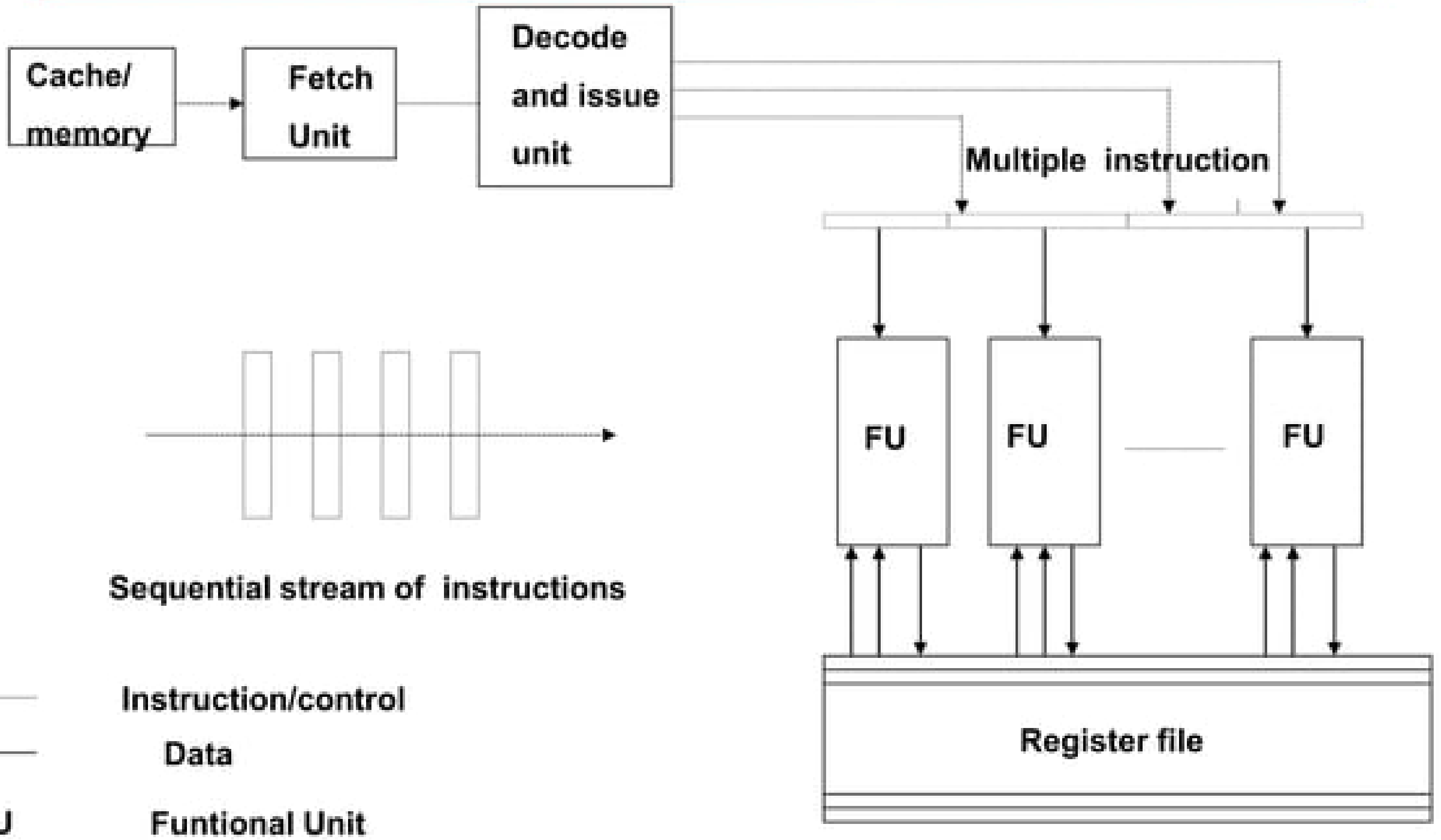
BTAC scheme


BTA
IIFA
I F A R
Instruction Fetch address
+
Next sequential address
A
I
I + 1
I + 2
I + 3
I - cache
BTI
BTI+1
BTI+2
BTI+3
BA
BTA
BTAC

ILP in VLIW processors


Cache/ memory
Fetch Unit
Single multi-operation instruction
FU
FU
FU
Register file
multi-operation instruction

ILP in Superscalar processors

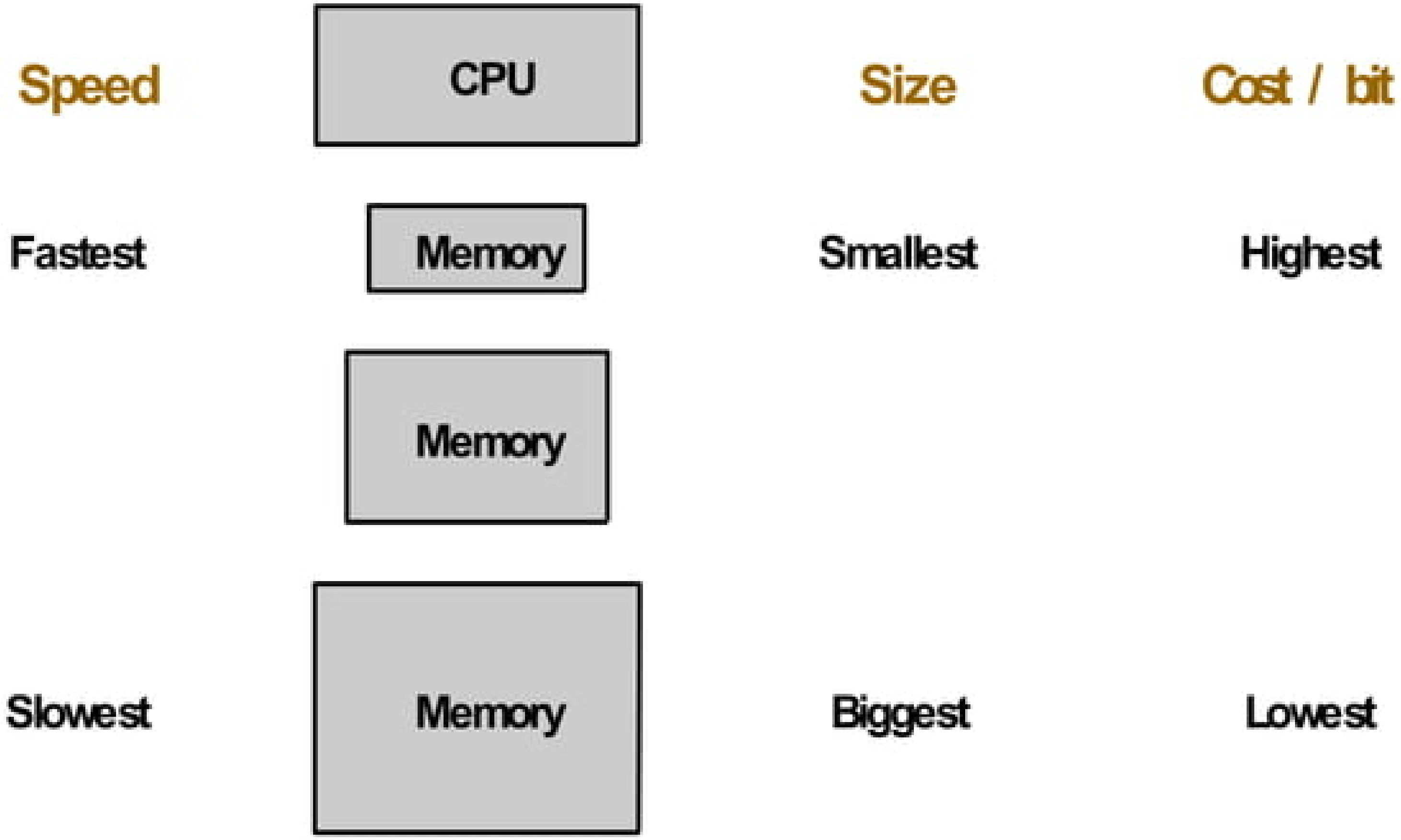


─ (light)	Instruction/control
─	Data
FU	Funtional Unit

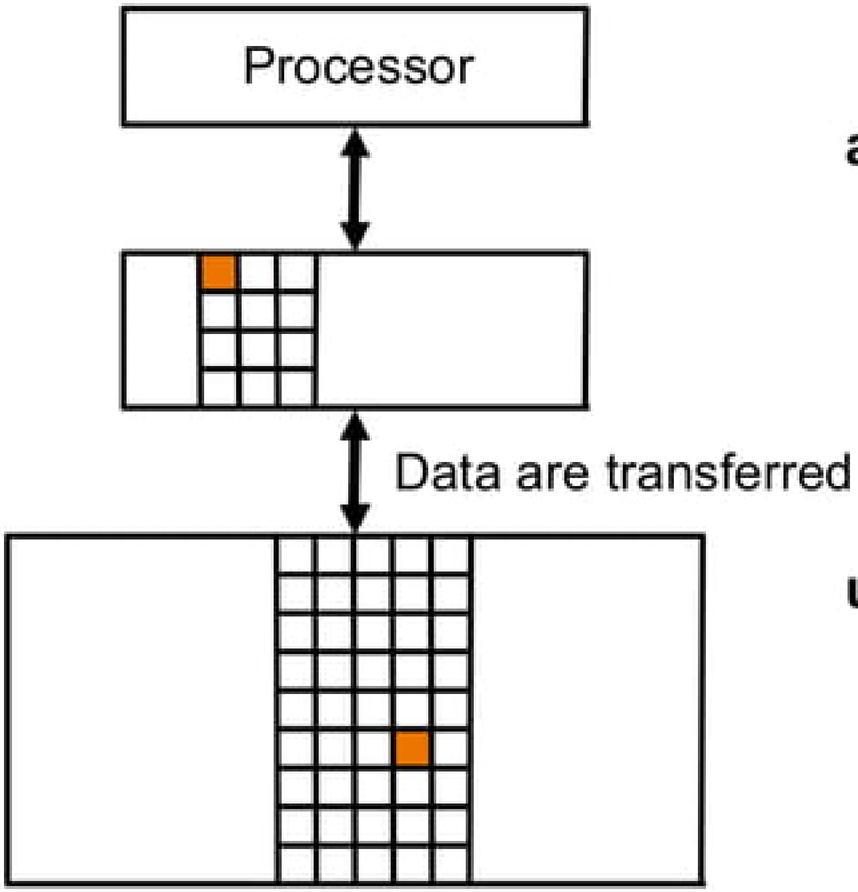
Why Superscalars are popular ?

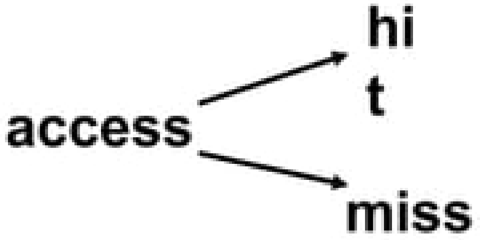
- Binary code compatibility among scalar & superscalar processors of same family
- Same compiler works for all processors (scalars and superscalars) of same family
- Assembly programming of VLIWs is tedious
- Code density in VLIWs is very poor -

Hierarchical structure



Data transfer between levels





unit of transfer = block

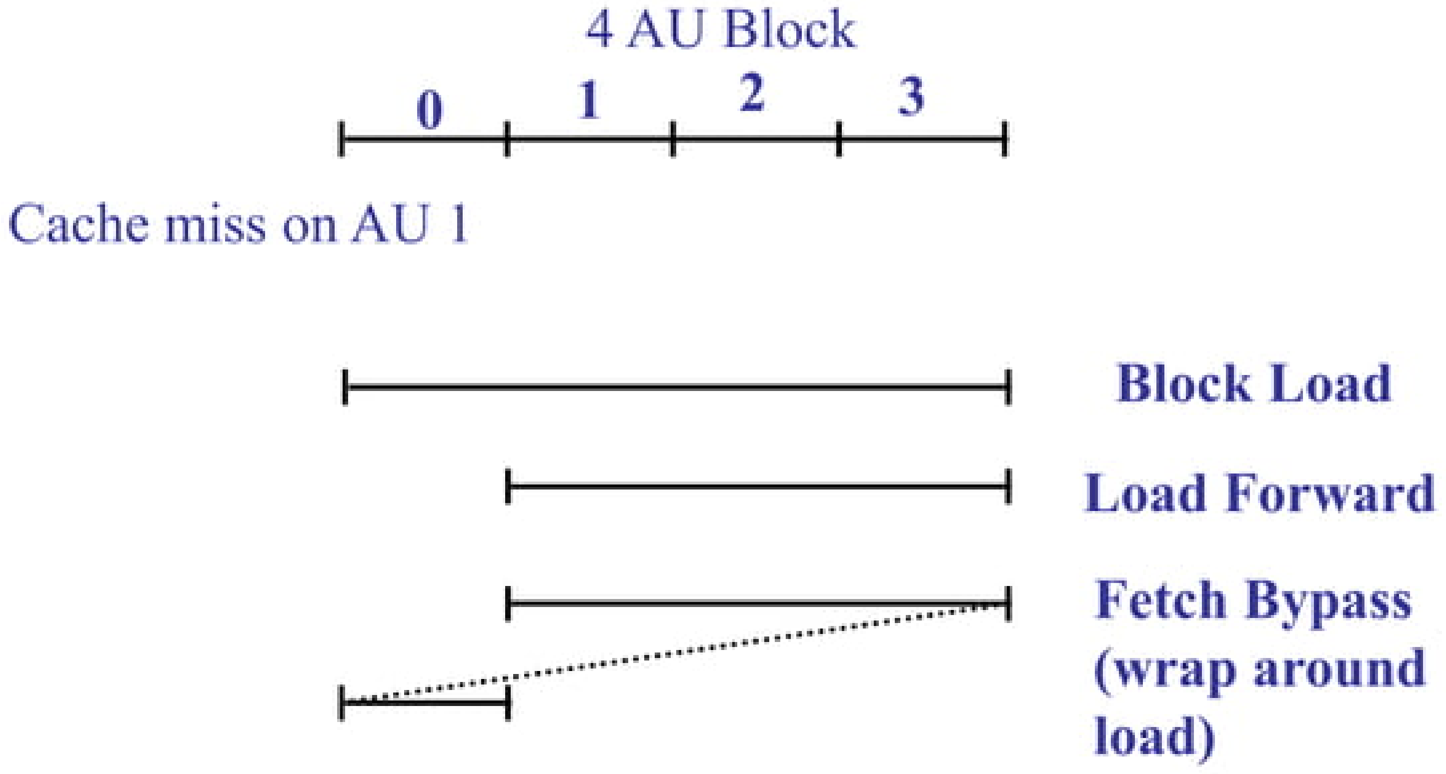
Principle of locality & Cache Policies

- Temporal Locality
  - references repeated in time
- Spatial Locality
  - references repeated in space
  - Special case: Sequential Locality

==========================

- **Read**
  - **Sequential / Concurrent**
  - **Simple / Forward**
- **Load**
  - **Block load / Load forward / Wrap around**
- **Replacement**
  - **LRU / LFU / FIFO / Random**

Load policies



Fetch Policies

- Demand fetching
  - fetch only when required (miss)
- Hardware prefetching
  - automatically prefetch next block
- Software prefetching
  - programmer decides to prefetch

questions:

- how much ahead (prefetch distance)
- how often

Write Policies

- Write Hit
  - Write Back
  - Write Through
- Write Miss
  - Write Back
  - Write Through
    - With Write Allocate
    - With No Write Allocate

Cache Types

Instruction | Data | Unified | Split

Split vs. Unified:

- Split allows specializing each part
- Unified allows best use of the capacity

On-chip | Off-chip

- on-chip : fast but small
- off-chip : large but slow

References

1. Patterson, D A.; Hennessy, J L. *Computer Organization and Design: The Hardware/software Interface*. Morgan Kaufman, 2000
2. Sima, T, FOUNTAIN, P KACSUK, *Advanced Computer Architectures: A Design Space Approach*, Pearson Education, 1998
3. Flynn M J, *Computer Architecture: Pipelined and Parallel Processor Design,* Narosa publishing India, 1999
4. John L. Hennessy, David A. Patterson, Computer architecture: a quantitative approach, 2nd Ed, Morgan Kauffman, 2001

Thanks