

COL216

Computer Architecture

Introduction
6th Jan, 2022

What this course is about?

PROGRAMS

- Expressions
- Types
- Conditions
- Loops
- Functions
- Classes
- Threads

Computer Architecture

CIRCUITS

- Transistors (v, i)
- Gates (1, 0)
- Flip-flops
- Registeers
- Memories
- ALUs
- FSMs

Many programming languages

C

C++

Java

Python

Matlab

Computer
Architecture

CIRCUITS

- Transistors (v, i)
- Gates (1, 0)
- Flip-flops
- Registers
- Memories
- ALUs
- FSMs

Hardware/software interface

software

hardware



} our
focus

Software Abstraction

```
swap (int v[ ], int k);  
{ int temp;  
    temp = v[k];  
    v[k] = v[k+1];  
    v[k+1] = temp;  
}
```

C

swap:

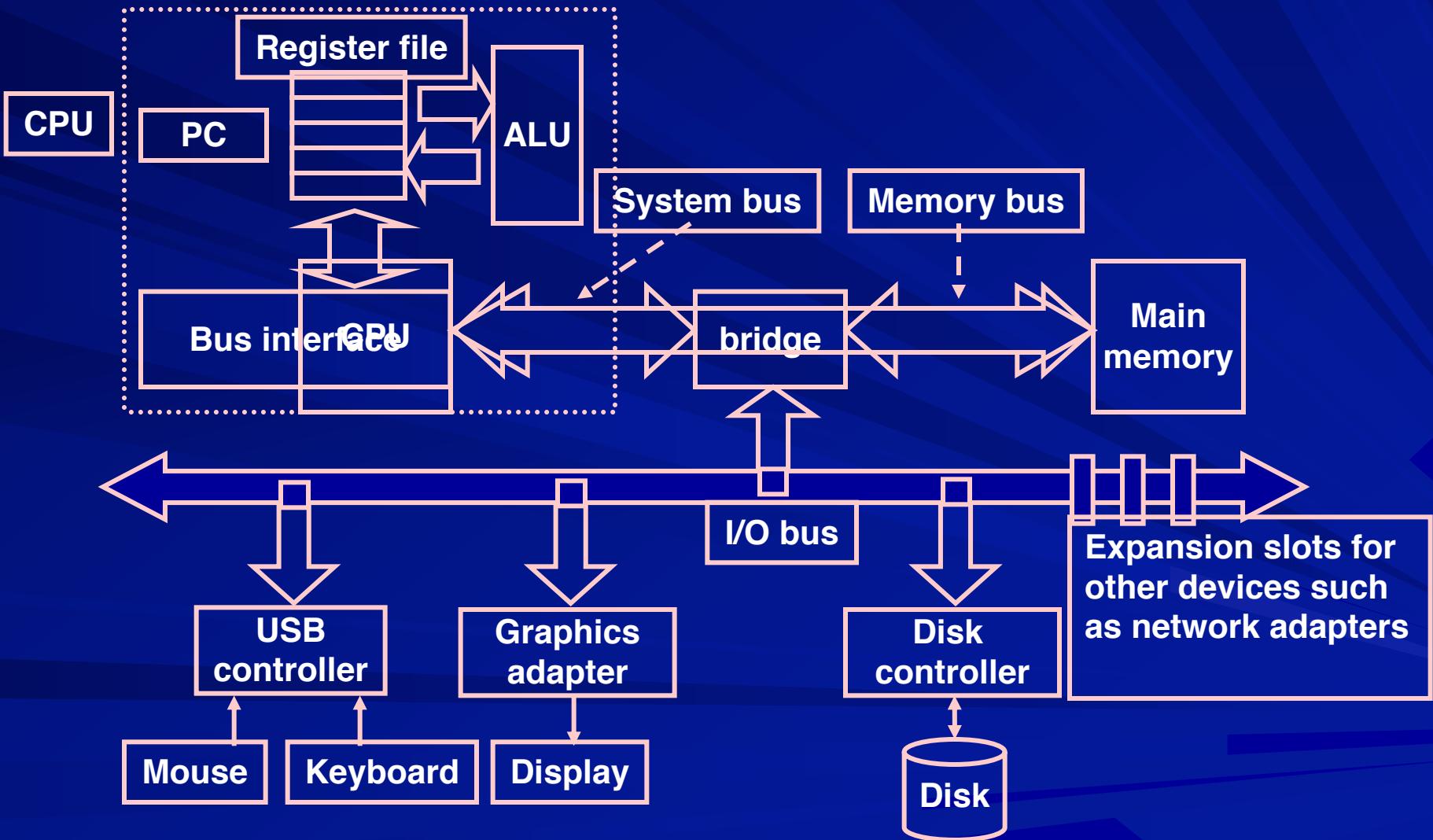
```
mul r3, r5, r4  
add r2, r3, r2  
ldr r6, [r2, #0]  
ldr r7, [r2, #4]  
str r7, [r2, #0]  
str r6, [r2, #4]  
mov pc, lr
```

assembly

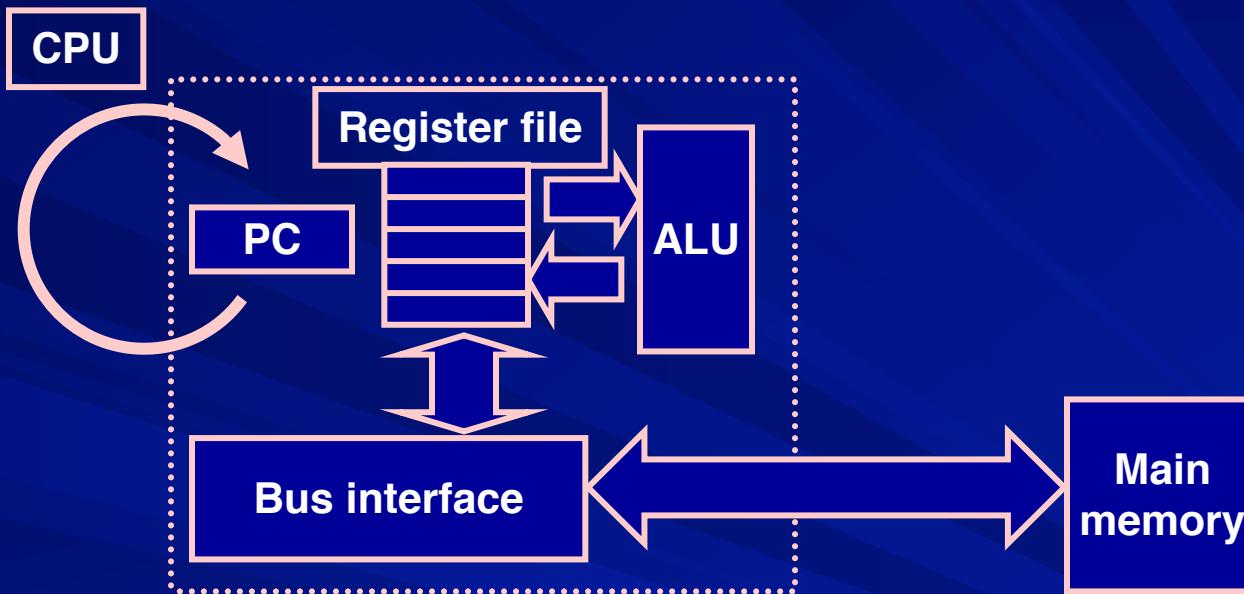
machine
code

00001000:	E0030495
00001004:	E0832002
00001008:	E5926000
0000100C:	E5927004
00001010:	E5827000
00001014:	E5826004
00001018:	E1A0F00E

Hardware abstraction



Instruction execution



- Instructions for arithmetic
- Instructions to move data
- Instructions for decision making

How instructions are encoded?

Can you use a ‘bare’ computer?

- What can a ‘bare computer’ or just the hardware do?
- A ‘bare’ computer is like an unfurnished building
- A computer usually comes with ‘hardware’ and ‘system software’

Role of System Software

- Compiler
 - different compilers for different programming languages
- Libraries
- Linker
- Loader
- Operating system

Why define machine instructions?

- Design hardware that understands high level language program
- Synthesize the program of interest into hardware

Machines (and languages)

- Different types of computers
 - Desktops, laptops, servers
 - Tablets, smart phones
 - Data centres, super computers

■ Multiple manufacturers

AMD

H-P

Intel

IBM

Motorola

NVIDIA

NXP

Qualcomm

Samsung . . .

What we will study

ARM (Advanced RISC Machine)

- Most popular 32 bit ISA (Instruction set architecture)
- Used extensively in embedded systems and mobile / hand held computing devices
- Easy to understand

Arithmetic instructions

- There are 2 operands and 1 result
- The order is fixed (result destination first)

Example:

C code: $a = b + c$

ARM code: add a, b, c

actually: add r1, r2, r3

registers associated with variables by compiler

ARM arithmetic

- Simplicity favors efficient implementation
- Operands must be registers, only 16 registers provided (smaller is faster)
- Expressions need to be broken

C code

$a = b + c + d;$

$e = f - (a + b);$

ARM code

add r7, r2, r3

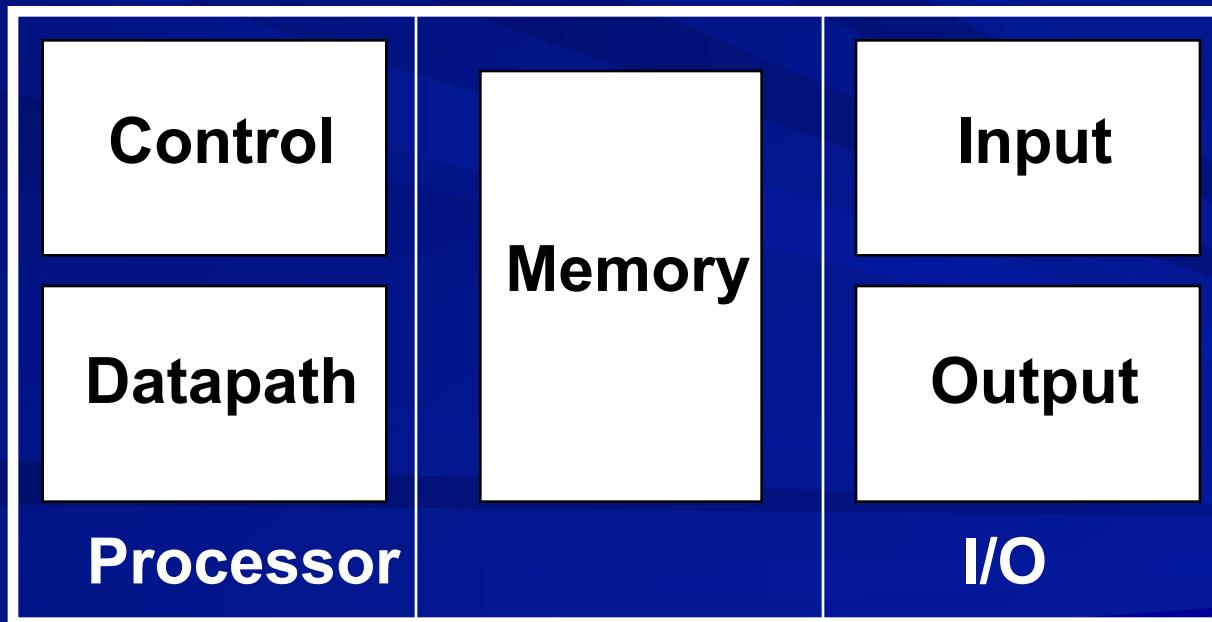
add r1, r7, r4

add r7, r1, r2

sub r5, r6, r7

Registers vs. Memory

- Scalars mapped to registers
- Structures, arrays etc in memory



Memory Organization

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- "Byte addressing" means that the index points to a byte of memory.

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
4	8 bits of data
5	8 bits of data
6	8 bits of data

...

Words and Bytes

- 2^{32} bytes : byte addresses from 0 to $2^{32}-1$
- 2^{30} words : byte addresses 0, 4, 8, ... $2^{32}-4$

Big endian byte order

0	1	2	3
4	5	6	7

Little endian byte order

3	2	1	0
7	6	5	4

Non-aligned word

3	2	1	0
7	6	5	4

Instructions to access memory

Load / store instructions

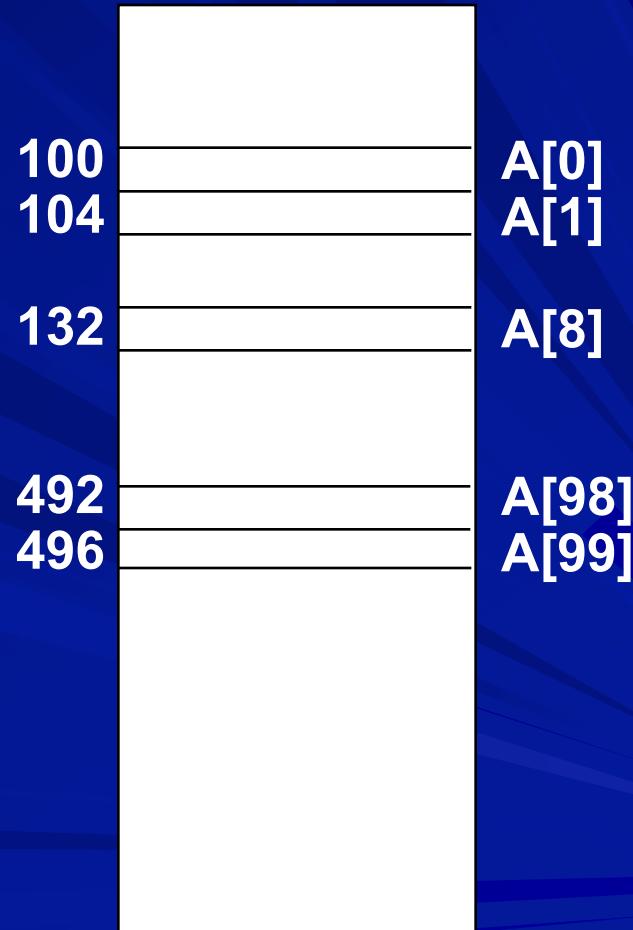
C : $v = A[8];$

ARM :

ldr r1, [r2, #32] {r2 has 100}

or

ldr r1, [r2, #100] {r2 has 32}
{not possible if address of A[0]
is large}



Load / Store example

C code: $A[8] = A[8] + h;$

ARM code:

```
ldr    r1, [r2, #32]
add    r1, r1, r3
str    r1, [r2, #32]
```

A simple example

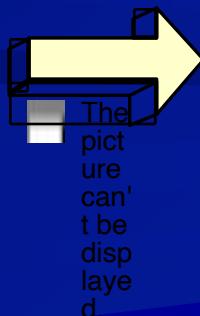
```
swap (int v[ ], int k);  
{ int temp;  
    temp = v[k]  
    v[k] = v[k+1];  
    v[k+1] = temp;  
}
```

What does this code do?

swap:
@ r2 has addr of v[0]
@ r5 has k, r4 has #4

mul r3, r5, r4
add r2, r3, r2
ldr r6, [r2, #0]
ldr r7, [r2, #4]
str r7, [r2, #0]
str r6, [r2, #4]

return



Accessing bytes and half words

- ldr, str load/store word

- ldrb, strb load/store byte

- ldrh, strh load/store half word

Control

- Decision making instructions - alter the control flow
- Compare instruction : cmp
- Conditional branch instructions : beq, bne

■ Example:

if (i == j)

 h = i + j;

 k = k - i;

 cmp r1, r2

 bne L

 add r3, r1, r2

L: sub r4, r4, r1

Control

- Unconditional branch instructions:

b label

- Example:

if (i == j)

 h = i + j;

else

 h = i - j;

k = k - i;

 cmp r1, r2

 bne Lab1

 add r3, r1, r2

 b Lab2

Lab1: sub r3, r1, r2

Lab2: sub r4, r4, r1

Other conditional branch instructions

- blt: branch if less-than
- ble: branch if less-than-or-equal
- bgt: branch if greater-than
- bge: branch if greater-than-or-equal

Writing a simple loop for $\sum_i A[i]$

```
s = 0;  
i = 0;  
L: s = s+A[i];  
    i++;  
    if (i<n) goto L;
```

Writing a simple loop for $\sum_i A[i]$

s = 0;	mov r1, #0
i = 0;	mov r2, #0
L: s = s+A[i];	L: mul r3, r2, r7 @ r7 = 4 add r3, r3, r4 @ r4=&A[0] ldr r5, [r3, #0] add r1, r1, r5
i++;	add r2, r2, #1
if (i<n) goto L;	cmp r2, r6 @ r6 = n blt L

Improving code: pointer vs. index

s = 0;	mov r1, #0
i = 0;	mov r2, #0
p = &A[0];	mov r3, r4
L: s = s + *p;	L: ldr r5, [r3, #0]
	add r1, r1, r5
p++;	add r3, r3, #4
i++;	add r2, r2, #1
if (i<n) goto L;	cmp r2, r6 @ r6 = n
	blt L

Improving code further

s = 0;

i = 0;

p = &A[0];

L: s = s + *p;

p++;

i++;

if (i<n) goto L;

s = 0;

p = &A[0];

q = p+100;

L: s = s + *p;

p++;

if (p<q) goto L;

Improving code further

s = 0;	mov r1, #0
p = &A[0];	mov r3, r4
q = p+100;	add r6, r3, #400
L: s = s + *p;	L: ldr r5, [r3, #0]
	add r1, r1, r5
p++;	add r3, r3, #4
if (p<q) goto L;	cmp r3, r6 @ r6 = q
	blt L

Complete Assembly Program

```
.equ SWI_Exit. 0x11
.text
mov r1, #0
ldr r3, =AA
add r6, r3, #400
L: ldr r5, [r3, #0]
add r1, r1, r5
add r3, r3, #4
cmp r3, r6      @ r6 = q
blt L
swi SWI_Exit
.data
AA: .space 400
.end
```

How to execute this program?

ARMSim#

- Simulator for ARM7TDMI architecture
- Developed by University of Victoria, Canada
- Includes an assembler
- Allows step-by-step execution and breakpoints
- Displays contents of registers and memory
- Supports input-output
- A plug-in simulates a particular ARM board

Before we close,

Book

Primary reference:

- John L. Hennesy & David A. Patterson,
"Computer Organization & Design : The
Hardware / Software Interface", Morgan
Kaufmann Publishers.

Chapters to be covered

1. Computer Abstractions and Technology
2. Instructions: Language of Computer
3. Arithmetic for Computers
4. The Processor
5. Large and Fast: Exploiting Memory Hierarchy
6. Storage and Other I/O Topics
7. Multicores, Multiprocessors and Clusters

Chapters to be covered

1. Computer Abstractions and Technology
2. Instructions: Language of Computer
3. Arithmetic for Computers
4. The Processor
5. Large and Fast: Exploiting Memory Hierarchy
6. Storage and Other I/O Topics
7. ~~Multicores, Multiprocessors and Clusters~~

Evaluation plan

■ Tests + quizzes	70
■ Laboratory work	30

Passing requirement:

tests + quizzes $\geq 30\%$ AND lab $\geq 40\%$

Attendance Policy

- 100%, subject to timely and satisfactory explanation/evidence for any absence
- Required for seeking alternative arrangements for missed tests/quizzes/lab sessions, E grade, I grade or for any other special requests

Thank you

COL216

Computer Architecture

Assembly Language
Programming
10th Jan, 2022

Machine Language

- Primitive compared to HLLs
- Language understood by Machine
- Easily interpreted by hardware
- Programmer's view of hardware

Instruction set design goals

- Maximize performance
- Minimize cost, energy consumption
- Reduce design time

ARM instructions so far

- add, sub
- mov
- cmp
- mul
- ldr, str
- ldrb, strb, ldrh, strh
- b, beq, bne, blt, ble, bgt, bge

Procedural Abstraction



What is required?

- Control flow (call and return)
- Data flow (parameter passing)
- Local and global storage allocation
- Take care of nesting
- Take care of recursion

Control flow - call

```
...  
X: func ( );  
...  
...  
Y: func ( );  
...
```

```
...  
X: bl func  
...  
...  
Y: bl func  
...
```

Control flow - return

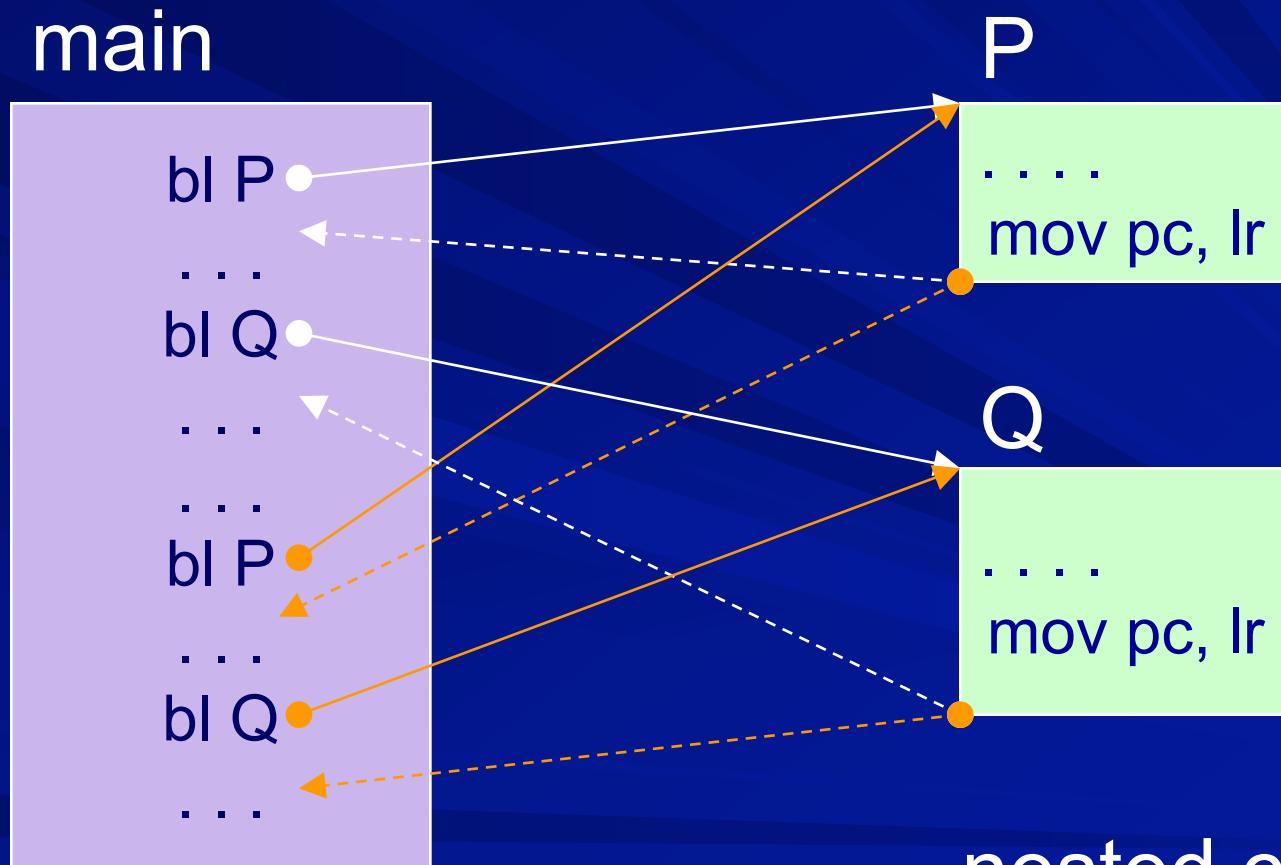
```
void func ( ) {  
    ...  
    ...  
    return;  
}
```

```
func:  
    ...  
    ...  
    mov pc, lr
```

Registers with special role

- r15 pc (program counter)
- r14 lr (link register)
- r13 sp (stack pointer)

Call and Return



nested calls?
recursive calls?

Passing parameters

through registers

caller :

....

....

move parameters
into registers

bl callee

take result from register

....

....

callee :

....

....

access parameters
in registers

....

....

....

mov pc, lr

Passing parameters

through stack

caller :

....

....

push parameters
into stack

bl callee

pop results from stack

....

....

callee :

....

....

access parameters
in stack

....

....

....

mov pc, lr

Conventions

- First 4 parameters through r0, r1, r2, r3
- Result in r0
- Beyond this, use stack
- Callee can destroy r0,r1,r2,r3,r12
- It should preserve other registers, except pc
- Caller should preserve r0,r1,r2,r3,r12

Saving and restoring registers

caller :

.....

.....

save registers

move parameters
into registers

bl callee

take result from register
restore registers

.....

.....

callee :

save registers

.....

.....

access parameters
in registers

.....

.....

.....

restore registers

mov pc, lr

Example with procedure/function

- GCD of n numbers

$G = A[0]$

for ($i = 1; i < n; i++$)

$G = \text{gcd}(G, A[i])$

- Compare with sum of n numbers

$S = A[0]$

for ($i = 1; i < n; i++$)

$S = S + A[i]$

Array sum program

```
.equ SWI_Exit 0x11
.text
mov r1, #0
ldr r3, =AA
add r6, r3, #400
L: ldr r5, [r3, #0]
add r1, r1, r5
add r3, r3, #4
cmp r3, r6      @ r6 = q
blt L
swi SWI_Exit
.data
AA: .space 400
.end
```

Sum => GCD

```
.equ SWI_Exit. 0x11
.text
    mov r1, #0
    ldr r3, =AA
    add r6, r3, #400
L:   ldr r5, [r3, #0]
    add r1, r1, r5
    add r3, r3, #4
    cmp r3, r6      @ r6 = q
    blt L
    swi SWI_Exit
.data
AA: .space 400
.end
```

The diagram illustrates the assembly code flow. It shows a sequence of instructions starting with `mov r1, #0`, followed by `ldr r3, =AA` and `add r6, r3, #400`. These three instructions are highlighted in a rounded rectangle. An arrow points from this group to a second rounded rectangle containing `ldr r3, =AA`, `add r6, r3, #400`, `ldr r1, [r3, #0]`, and `add r3, r3, #4`. Below this, the label `L:` is followed by `ldr r5, [r3, #0]`, `add r1, r1, r5` (which is also highlighted in a rounded rectangle), `add r3, r3, #4`, `cmp r3, r6 @ r6 = q`, `blt L`, `swi SWI_Exit`, and `.data`. Finally, the memory location `AA:` is defined as `.space 400` and the program ends with `.end`.

labeled block 1:
mov r1, #0
ldr r3, =AA
add r6, r3, #400

labeled block 2:
ldr r3, =AA
add r6, r3, #400
ldr r1, [r3, #0]
add r3, r3, #4

labeled block 3:
r1 = gcd (r1, r5)

Array GCD program

```
ldr r3, =AA  
add r6, r3, #400  
ldr r1, [r3, #0]  
add r3, r3, #4  
L: ldr r5, [r3, #0]  
    r1 = gcd (r1, r5) → r0 = gcd (r0, r1)  
    add r3, r3, #4  
    cmp r3, r6      @ r6 = q  
    blt L
```

follow conventions

Array GCD program

```
ldr r3, =AA  
add r6, r3, #400  
ldr r0, [r3, #0]  
add r3, r3, #4  
L: ldr r1, [r3, #0]  
r0 = gcd (r0, r1)  
add r3, r3, #4  
cmp r3, r6      @ r6 = q  
blt L
```

follow conventions

r3 => r4

Array GCD program

```
ldr r4, =AA
add r6, r4, #400
ldr r0, [r4, #0]
add r4, r4, #4
L: ldr r1, [r4, #0]
    r0 = gcd (r0, r1)
    add r4, r4, #4
    cmp r4, r6      @ r6 = q
    blt L
```

Array GCD program with “bl”

```
ldr r4, =AA
add r6, r4, #400
ldr r0, [r4, #0]
add r4, r4, #4
L: ldr r1, [r4, #0]
    bl gcd
    add r4, r4, #4
    cmp r4, r6      @ r6 = q
    blt L
```

GCD function

```
gcd:    cmp r0, r1
        beq ret
        blt sub10
sub01: sub r0, r0, r1
        b    gcd
sub10: sub r1, r1, r0
        b    gcd
ret:   mov pc, lr
```

DP (data processing) instructions

- | | |
|--------------|--------------------------|
| ■ Arithmetic | operation dest, op1, op2 |
| ■ Logical | operation dest, op1, op2 |
| ■ Move | operation dest, src |
| ■ Compare | operation op1, op2 |
| ■ Test | operation op1, op2 |

DP instructions: Arithmetic

- add
- sub
- rsb
- adc
- sbc
- rsc



reverse subtract: $op2 - op1$

add / sub / rsb with carry

DP instructions: Logical

- and bit by bit logical AND
- orr bit by bit logical OR
- eor bit by bit logical XOR
- bic bit clear: op1 and not op2

DP instructions: Move

- | | |
|-------|-----------------|
| ■ mov | dest <= src |
| ■ mvn | dest <= not src |

DP instructions: Compare

- cmp $op1 - op2$
- cmn $op1 - (-op2)$

DP instructions: Test

- **tst** op1 and op2
- **teq** op1 eor op2

DT (data transfer) instructions

- ldr / str load / store word
- ldrb / strb load / store byte
- ldrh / strh load / store half word
- ldrsb / ldrsh load signed byte / half word
(sign extension to fill the register)
- ldm / stm load / store multiple
(any subset of registers can be specified)

Comparison in ARM

- Signed comparison:

equal	beq
not equal	bne
greater or equal	bge
less than	blt
greater than	bgt
less or equal	ble

- Unsigned comparison

equal	beq
not equal	bne
higher or same	bhs
lower	blo
higher	bhi
lower or same	bls

Status flags

These are part of Program Status Register

- N Negative
- Z Zero
- C Carry
- V Overflow

Condition codes and flags

0	eq	$Z = 1$
1	ne	$Z = 0$
2	hs / cs (C set)	$C = 1$
3	lo / cc (C clear)	$C = 0$
4	mi (minus)	$N = 1$
5	pl (plus)	$N = 0$
6	vs (V set)	$V = 1$
7	vc (V clear)	$V = 0$

8	hi	$C = 1$ and $Z = 0$
9	ls	$C = 0$ or $Z = 1$
10	ge	$N = V$
11	lt	$N \neq V$
12	gt	$N = V$ and $Z = 0$
13	le	$N \neq V$ or $Z = 1$
14	al	flags ignored

Assembler directives

.text

.data

.end

.space

.word

.byte

.ascii

.asciz

.equ

Input Output in ARM

SWI instruction

- Instruction to invoke some service provided by system software

Use of SWI in ARMSim

- input/output from/to stdin/stdout
- input/output from/to files
- opening/closing of files
- Halt execution
- ...

I/O example in ARMsim# 1.91

```
ldr    r0, =message  
swi    0x02      @ write on stdout
```

....

message: .asciz “Welcome\n”

I/O example in ARMsim# 2.01

```
ldr    r1, =param
mov    r4, #1          @ file #1 is stdout
str    r4, [r1]
ldr    r4, =message
str    r4, [r1, #4]
mov    r4, #8          @ number of bytes
str    r4,[r1,#8]
mov    r0, #5          @ code for write
swi    0x123456
...

```

param: .word 0, 0, 0

message: .ascii "Welcome\n"

Software Interrupts

- Similar terms –
 - System calls, Traps, Exceptions
- Are there hardware interrupts?
- Hardware interrupts are caused by events/conditions detected by hardware
 - intentional : e.g., I/O event
 - unintentional : e.g., hardware fault, power outage, arithmetic overflow
- Software interrupts are caused by specific instructions

Response to interrupts

- Response mechanism in both cases (h/w and s/w interrupts) is same
- Execution of some code
 - interrupt handler or interrupt service routine (ISR)

ISR vs normal subroutine

- Processors have two or more modes
 - normal mode / user mode
 - privileged mode / kernel mode / supervisor mode
- Application program executes in user mode
- To do certain privileged tasks, execution of some kernel code is required
- ISR executes in privileged mode, provides controlled access to kernel functions

Thank you

COL216

Computer Architecture

ARM Assembly Programming
continued

13th Jan, 2022

Question about ARM instructions

- Range of constants in various instructions?
- Accessing bits/bytes in a register?
- Other instructions between cmp and branch?
- Working with stack?
- Machine language (encoding of assembly language)?

Instruction format

- An instruction has many parts
- These are called fields
- Arrangement of fields is called instruction format

Format for DP instructions

$Rd \leq Rn + \text{operand2}$



add r1, r2, r3



add r1, r2, #3



Format for DP instructions

$Rd \leq Rn + \text{operand2}$

	F	I	opc	Rn	Rd	operand2
4	2	1	4	1	4	12

add r1, r2, r3

	F	0	opc	Rn	Rd	shift	Rm
						8	4

add r1, r2, #3

	F	1	opc	Rn	Rd	rot	Imm
						4	8

F = 00 for DP instructions

Shift operations on registers

Shift type:

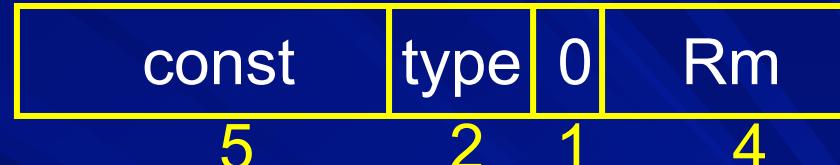
- LSL logical shift left
- LSR logical shift right
- ASR arithmetic shift right
- ROR rotate right

Shift amount

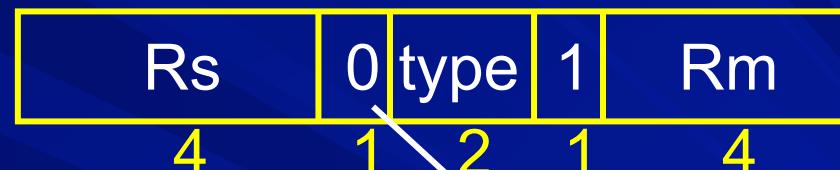
- 5 bit unsigned constant # 0 .. # 31
- 4 bit register number r0 .. r15

Operand2 : Register with shift

Rm, LSL #4



Rm, LSL Rs



Shift type:

- 00 LSL
- 01 LSR
- 10 ASR
- 11 ROR

logical shift left

logical shift right

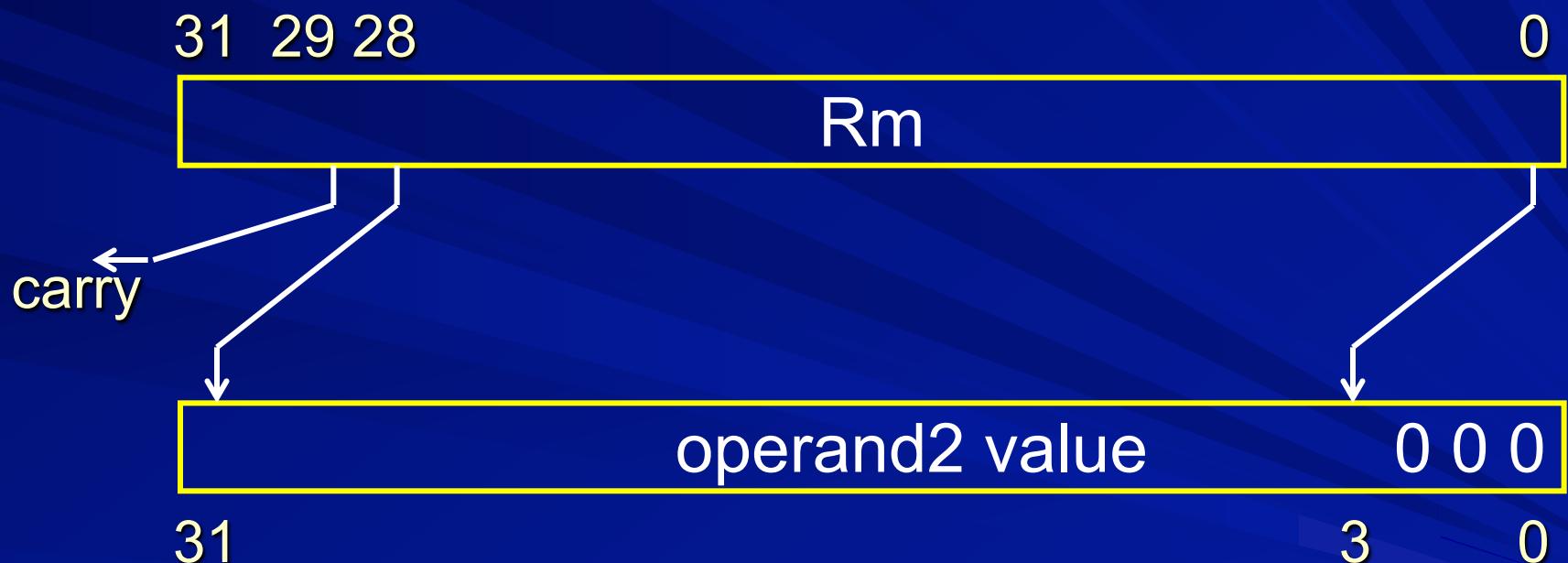
arithmetic shift right

rotate right

1 for multiply and
other instructions

Shift types: Logical Shift Left

■ LSL # 3



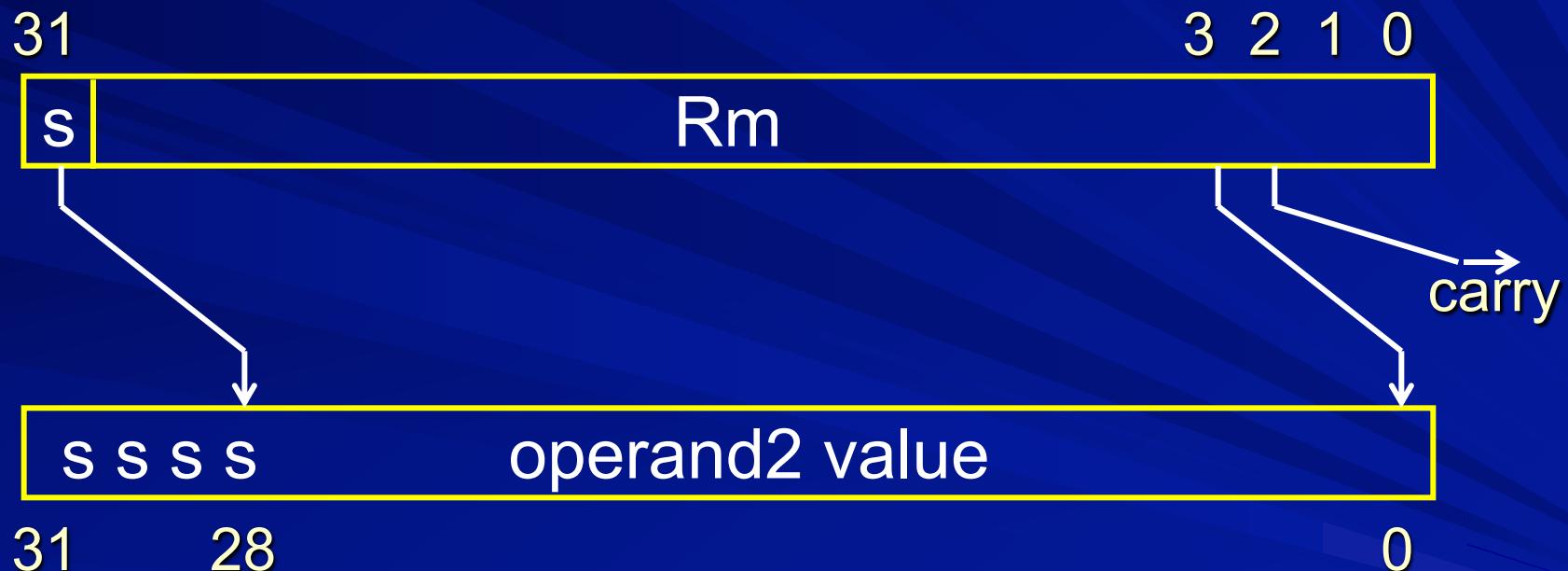
Shift types: Logical Shift Right

■ LSR # 3



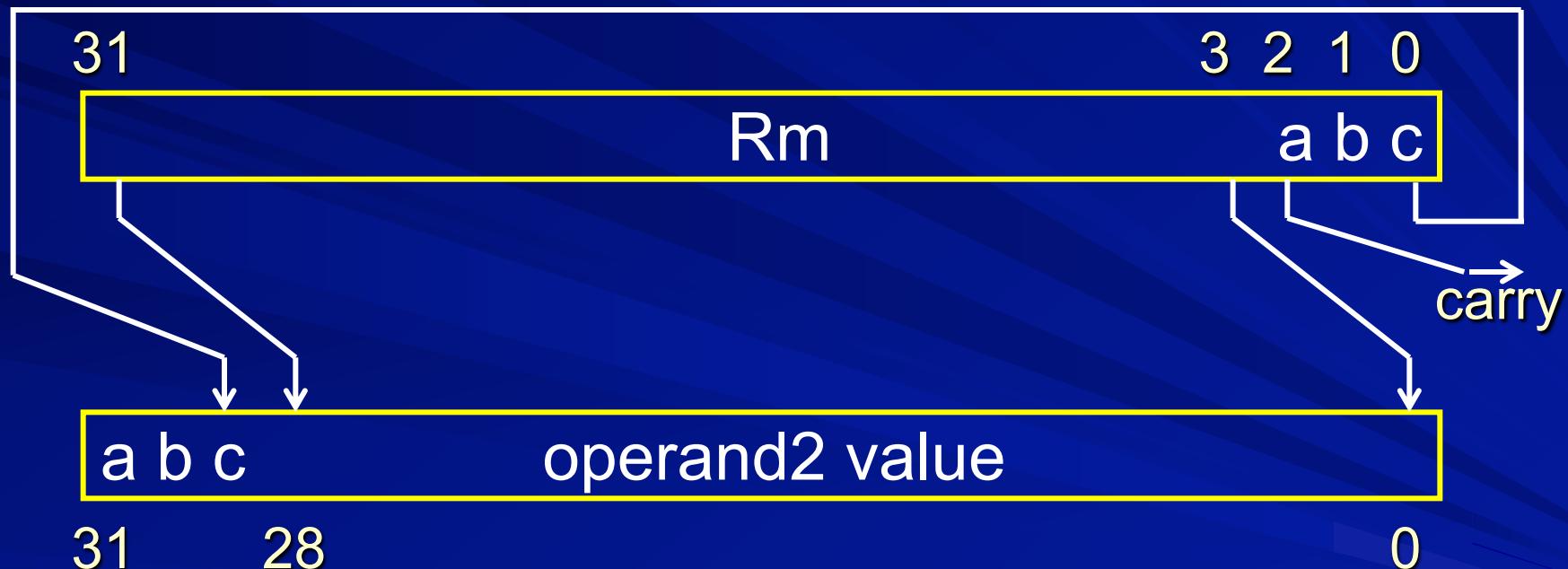
Shift types: Arithmetic Shift Right

■ ASR # 3



Shift types: Rotate Right

■ ROR # 3



Rotate operation on constant

rot	Imm
4	8

- rot is a 4 bit unsigned constant (0 to 15), specifying how much to rotate
- Imm is a 8 bit constant (0 to 255) which is zero extended to 32 bits and rotated right by $2 \times \text{rot}$ bits (that is, by 0 to 30 bits)

e.g., $\text{operand2} = \#400 \Rightarrow \text{Imm} = 100, \text{rot} = 15$

$\text{operand2} = \#800 \Rightarrow \text{Imm} = 50, \text{rot} = ?$

Format for “mul”

mul r1, r2, r3

$r1 \leq r2 * r3$

	F	I	opc	Rn	Rd	Rs	1001	Rm
--	---	---	-----	----	----	----	------	----

	00	0	0000	0000	0001	0011	1001	0010
--	----	---	------	------	------	------	------	------

	0	0	0	0	1	3	9	2
4	2	1	4	1	4	4	4	4

Multiply accumulate

mla r1, r2, r3, r4

$r1 \leq r2 * r3 + r4$

	F	I	opc	Rn	Rd	Rs	1001	Rm
--	---	---	-----	----	----	----	------	----

	00	0	0001	0100	0001	0011	1001	0010
--	----	---	------	------	------	------	------	------

	0	0	1		4	1	3	9	2
4	2	1	4	1	4	4	4	4	4

Multiplying by a constant

mul r1, r2, # 10



mov r3, # 10

mul r1, r2, r3

Multiplying by a power of 2

mul r1, r2, # 16



mov r1, r2, LSL # 4

LSL = Logical Shift Left

Format for DT instructions

$Rd \leq \text{Memory } [Rn + \text{offset}]$

	F	opc	Rn	Rd	offset
4	2	6	4	4	12

$F = 01$

12 bit offset field in DT instructions

- 12 bit unsigned constant

or

- 4 bit register number, 8 bit shift specification
(same as constant shift spec of DP instructions)

Example of DT instruction

ldr r4, [r5, #32]

	F	opc	Rn	Rd	operand2
--	---	-----	----	----	----------

	01	011001	0101	0100	000000100000
--	----	--------	------	------	--------------

	1	25	5	4	32
4	2	6	4	4	12

Indexing an array

mul r4, r5, #4

add r2, r4, r2

ldr r6, [r2, #0]



add r2, r2, r5, LSL #2

ldr r6, [r2]



ldr r6, [r2, r5, LSL #2]

Instruction similar to ldr

- str (opcode = 24)

rd is the source, not destination

memory address is the destination, not source

Opcode field in DT instructions

- 6 opcode bits specify I, P, U, B, W, L
 - I (immediate): constant or register with shift
 - P (pre/post) pre or post indexing
 - U (up/down) whether to add or subtract offset
 - B (byte) byte or word transfer
 - W (write back) whether to write back address into base register (Rn) or not
 - L (load/store) load from memory or store into memory

DT instruction examples

ldr r4, [r5, - r6]

str r4, [r5, r6, LSL # 2]

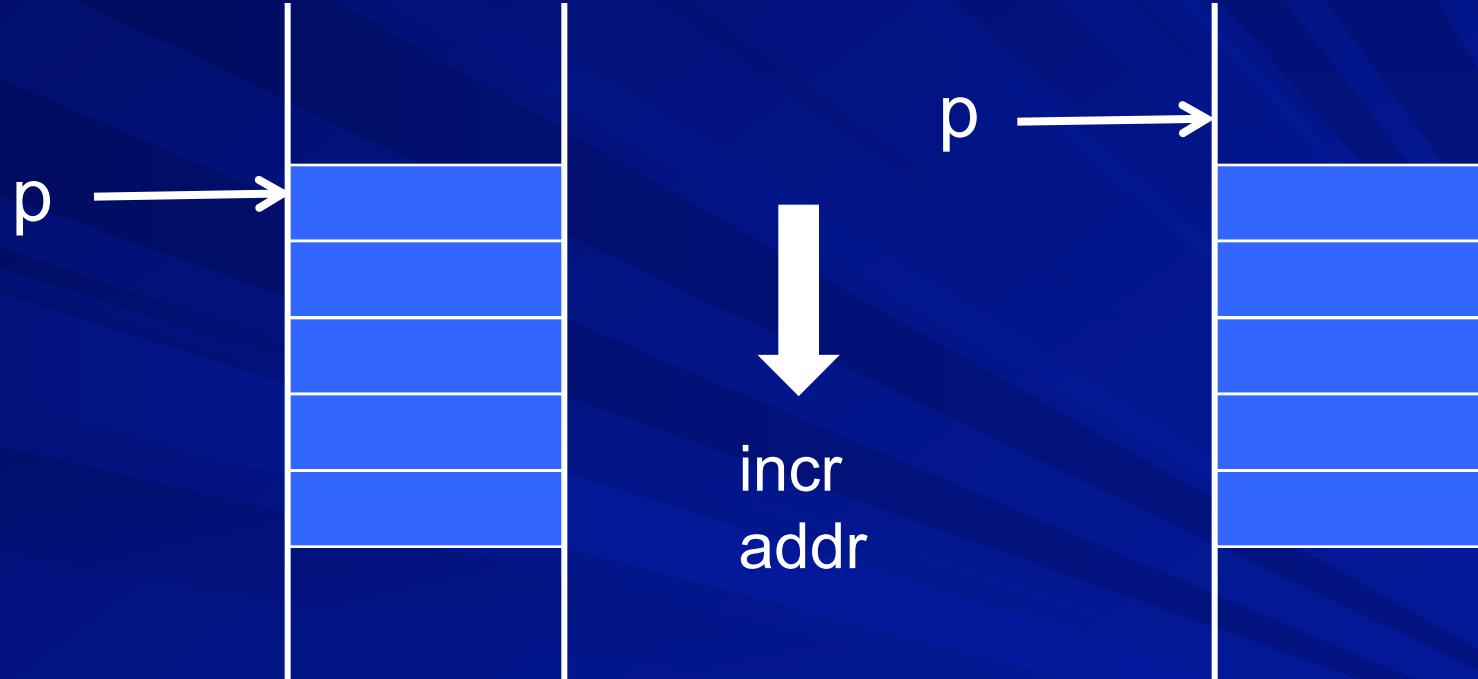
ldrb r4, [r5, # 32] !

strb r4, [r5, # -32]

ldr r4, [r5], r6

str r4, [r5], r6, LSL # 2

Using auto-increment/decrement



get/pop: post increment
put/push: pre decrement

pre increment
post decrement

Thank you

COL216

Computer Architecture

ARM Assembly Programming

continued

17th Jan, 2022

Instruction Formats in ARM

Formats: DP, DT, branch, swi

Constants:

- Range of values?
- Signed or unsigned?

Addresses:

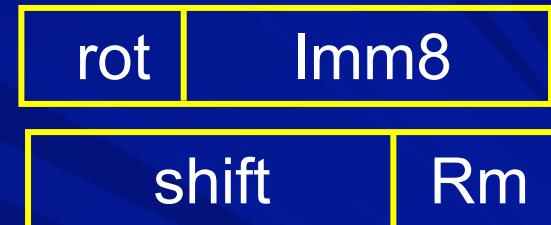
- Absolute or relative?
- Byte or word?

DP instructions



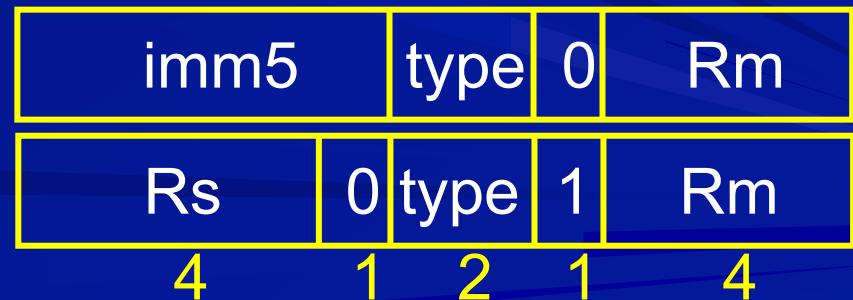
12 bit operand2

- 8 bit unsigned, 4 bit rotate
- 4 bit reg no., 8 bit shift spec



Shift amount

- 5 bit unsigned const
- 4 bit reg no. ($Rs \neq 15$)



Multiply instructions



With or without accumulation

Result 32 or 64 bits

Signed or unsigned multiplication

R15 can't be used

$Rd \neq Rm$

DT instructions



6 opcode bits specify I, P, U, B, W, L

12 bit offset field in DT instructions

- 12 bit unsigned constant

or

- 4 bit register number, 8 bit shift specification
(same as constant shift spec of DP instructions)

Branch instructions



L = 0 b, beq, bne

L = 1 bl, bleq, blne

24 bit immediate field in branch instructions

- signed offset
- w.r.t. PC, (address of current instruction + 8)
- multiplied by 4 and sign extended (word offset, not byte)

Condition codes and flags

0	eq	$Z = 1$
1	ne	$Z = 0$
2	hs / cs (C set)	$C = 1$
3	lo / cc (C clear)	$C = 0$
4	mi (minus)	$N = 1$
5	pl (plus)	$N = 0$
6	vs (V set)	$V = 1$
7	vc (V clear)	$V = 0$

8	hi	$C = 1$ and $Z = 0$
9	ls	$C = 0$ or $Z = 1$
10	ge	$N = V$
11	lt	$N \neq V$
12	gt	$N = V$ and $Z = 0$
13	le	$N \neq V$ or $Z = 1$
14	al	flags ignored

Condition code – general use

- Use condition codes for conditional /predicated execution of any instruction, not just branch

Example 1:

if (i == j)	cmp r1,r2
	bne L
h = i + j;	add r3,r1,r2
k = k - i;	L: sub r4,r4,r1

Condition code – general use

- Use condition codes for conditional /predicated execution of any instruction, not just branch

Example 1:

if (i == j)	cmp r1,r2	cmp r1,r2
	bne L	
h = i + j;	add r3,r1,r2	addeq r3,r1,r2
k = k - i;	L: sub r4,r4,r1	sub r4,r4,r1

Condition code – general use

Example 2:

if ($i == j$) cmp r1,r2

 bne L1

$h = i + j;$ add r3,r1,r2

else b L2

$h = i - j;$ L1: sub r3,r1,r2

$k = k - i;$ L2: sub r4,r4,r1

Condition code – general use

Example 2:

if ($i == j$)

 cmp r1,r2

 cmp r1,r2

 bne L1

$h = i + j;$

 add r3,r1,r2

 addeq r3,r1,r2

else

 b L2

$h = i - j;$

L1: sub r3,r1,r2

subne r3,r1,r2

$k = k - i;$

L2: sub r4,r4,r1

sub r4,r4,r1

Which instructions modify flags?

- Compare, Test (obviously!)
- Additionally (if you want)
 - all other DP instructions (add => adds)
 - mul and mla instructions (mul => muls)
 - Flags are affected if S = 1
- But not
 - DT instructions
 - b and bl instructions

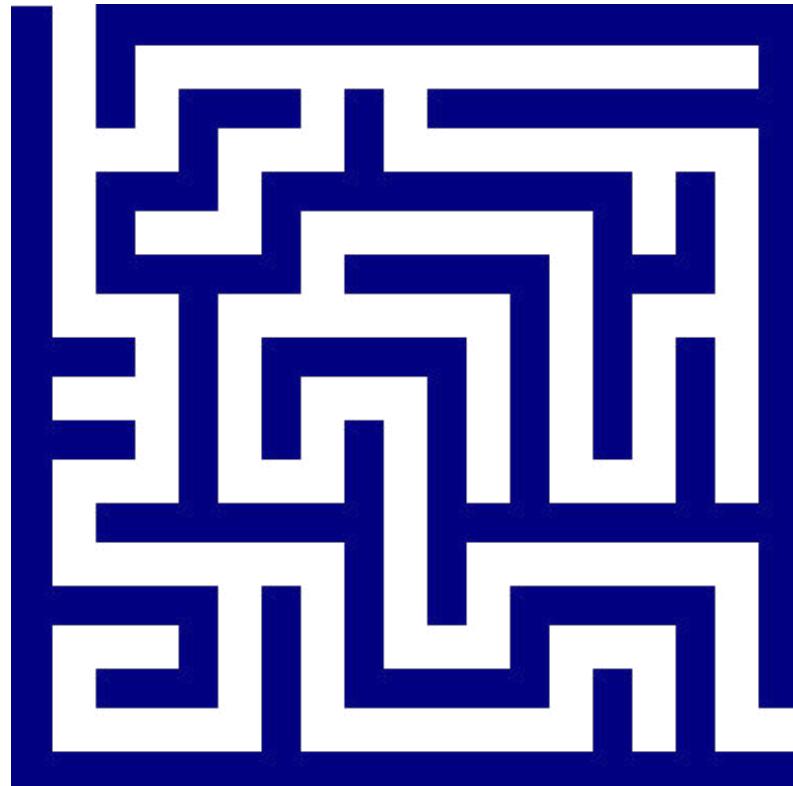
SWI instruction



Software Interrupt
used for invoking OS function

Recursive function example

- Solving a MAZE



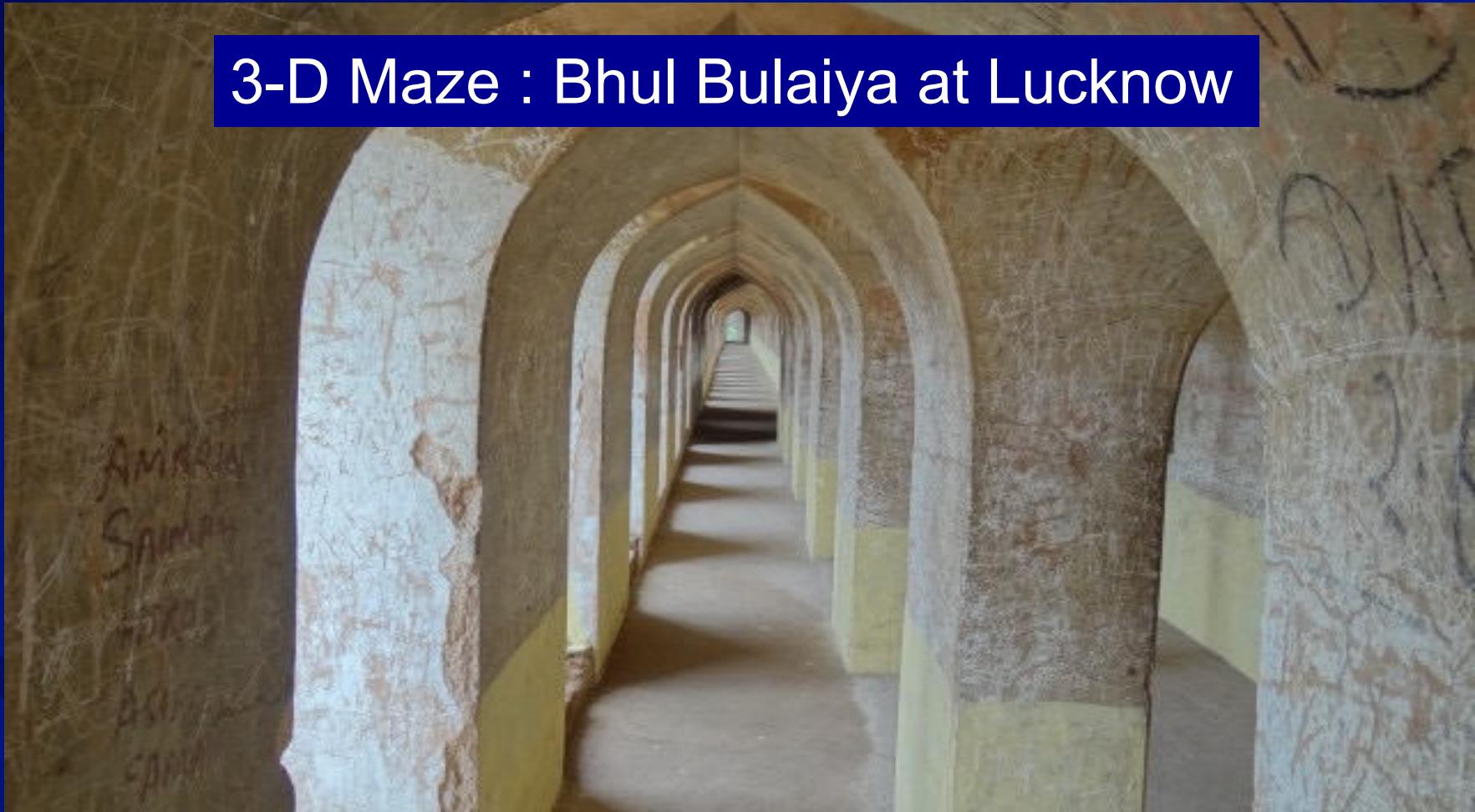
Real Life Maze

Hedge Maze at St Louis, USA



Real Life Maze

3-D Maze : Bhul Bulaiya at Lucknow



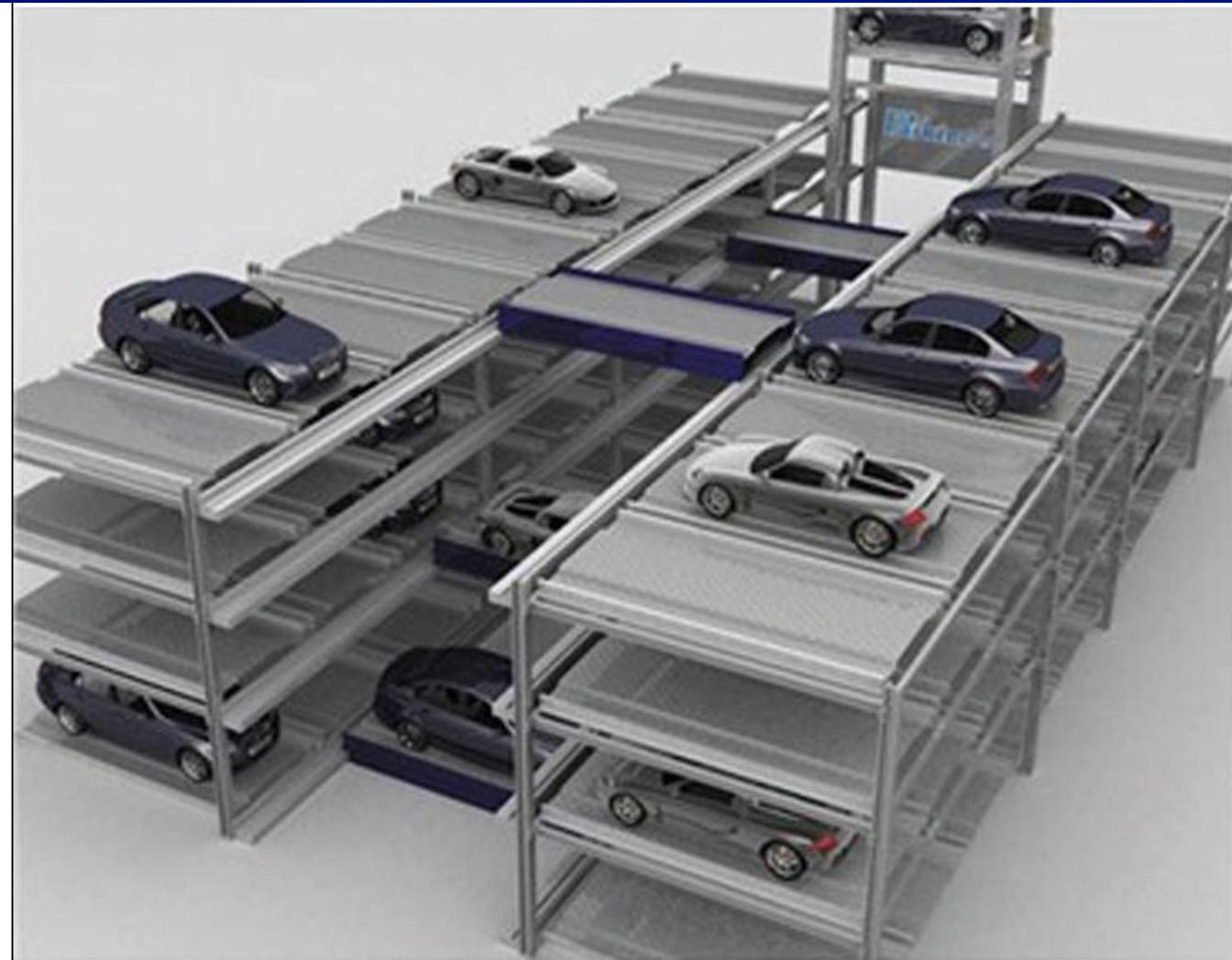
Real Life Maze



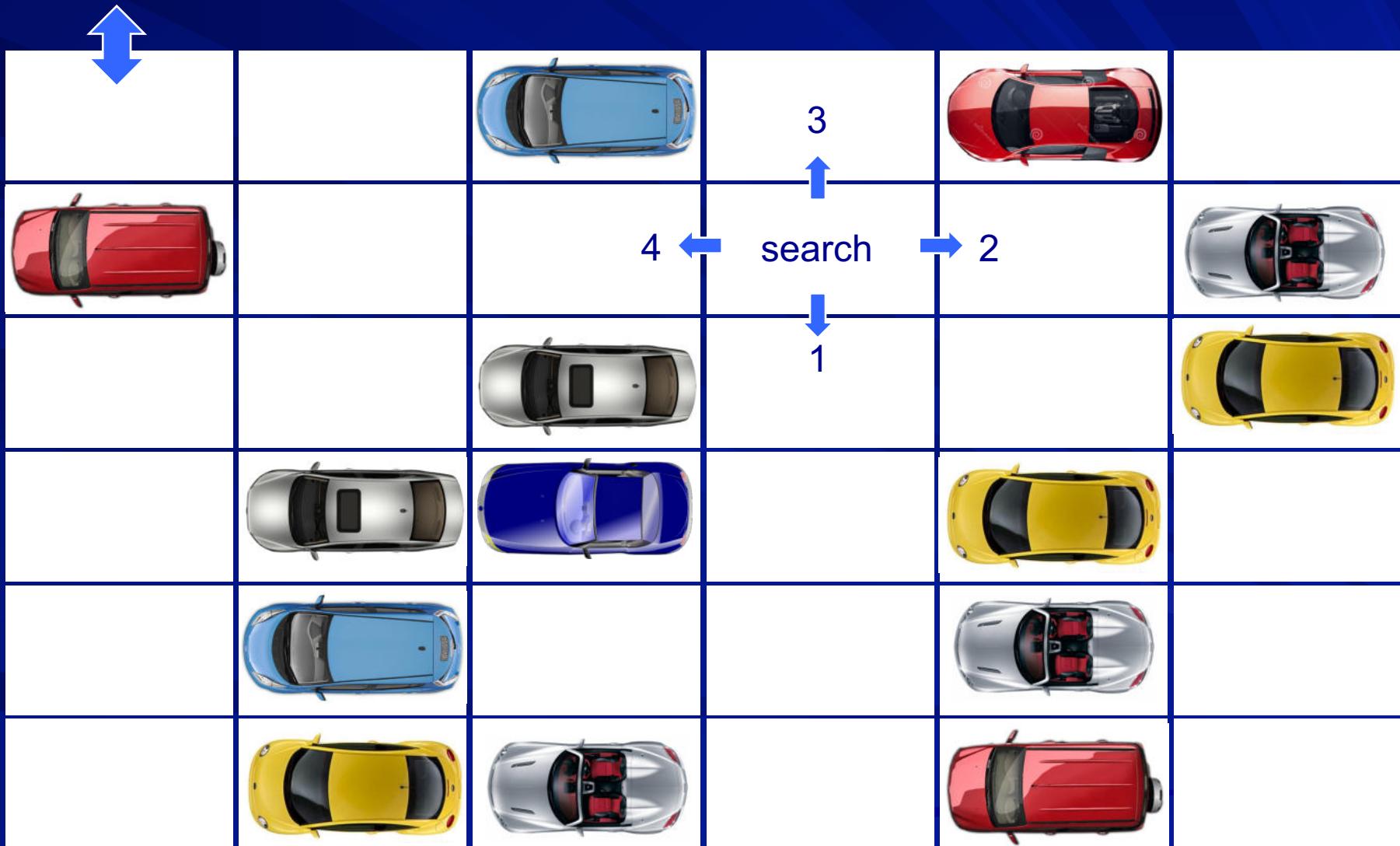
Maze to be actually solved by computer



Maze to be actually solved by computer



6x6 Parking floor



6x6 Parking floor

B	B	B	B	B	B	B	B
B	S	O	X	O	X	O	B
B	X	O	O	O	O	X	B
B	O	O	X	O	O	X	B
B	O	X	F	O	X	O	B
B	O	X	O	O	X	O	B
B	O	X	X	O	X	O	B
B	B	B	B	B	B	B	B

Recursive search

```
int search (int i, j) {  
    if (Visited[i, j] == 1) return (0);  
    Visited[i, j] = 1;  
    if (Maze[i, j] == 'B' || Maze[i, j] == 'X') return (0);  
    if (Maze[i, j] == 'F') {print (i); print (j); return (1)};  
    if (search (i+1, j) == 1) {print ('N'); return (1)};  
    if (search (i, j+1) == 1) {print ('W'); return (1)};  
    if (search (i-1, j) == 1) {print ('S'); return (1)};  
    if (search (i, j-1) == 1) {print ('E'); return (1)};  
    return (0)  
};
```

Steps

```
int search (int i, j){  
    v = V[i, j];  
    if (v == 1)  
        return (0);  
    V[i, j] = 1;  
    q = M[i, j];  
    if (q == 'B')  
        return (0);  
    if (q == 'X')  
        return (0);  
    if (q == 'F') {  
        print (i);  
        print (j);  
        return (1);  
    }
```

```
    p = search (i+1, j);  
    if (p == 1) {  
        print ('N');  
        return (1);  
    }  
    p = search (i, j+1);  
    if (p == 1) {  
        print ('W');  
        return (1);  
    }  
    p = search (i-1, j);  
    ....  
    p = search (i, j-1);  
    ....  
    ....  
    return(0);  
}
```

Assembly program - 1

```
int search (int i, j){
```

```
    v = V[i, j];
```

```
    if (v == 1)
```

```
        return (0);
```

```
    V[i, j] = 1;
```

```
search: str lr, [sp,#-4]!
```

```
    str r0, [sp,#-4]!
```

```
    str r1, [sp,#-4]!
```

```
    add r2, r1, r0, LSL #3
```

```
    ldr r3, =V
```

```
    ldrb r4, [r3, r2]
```

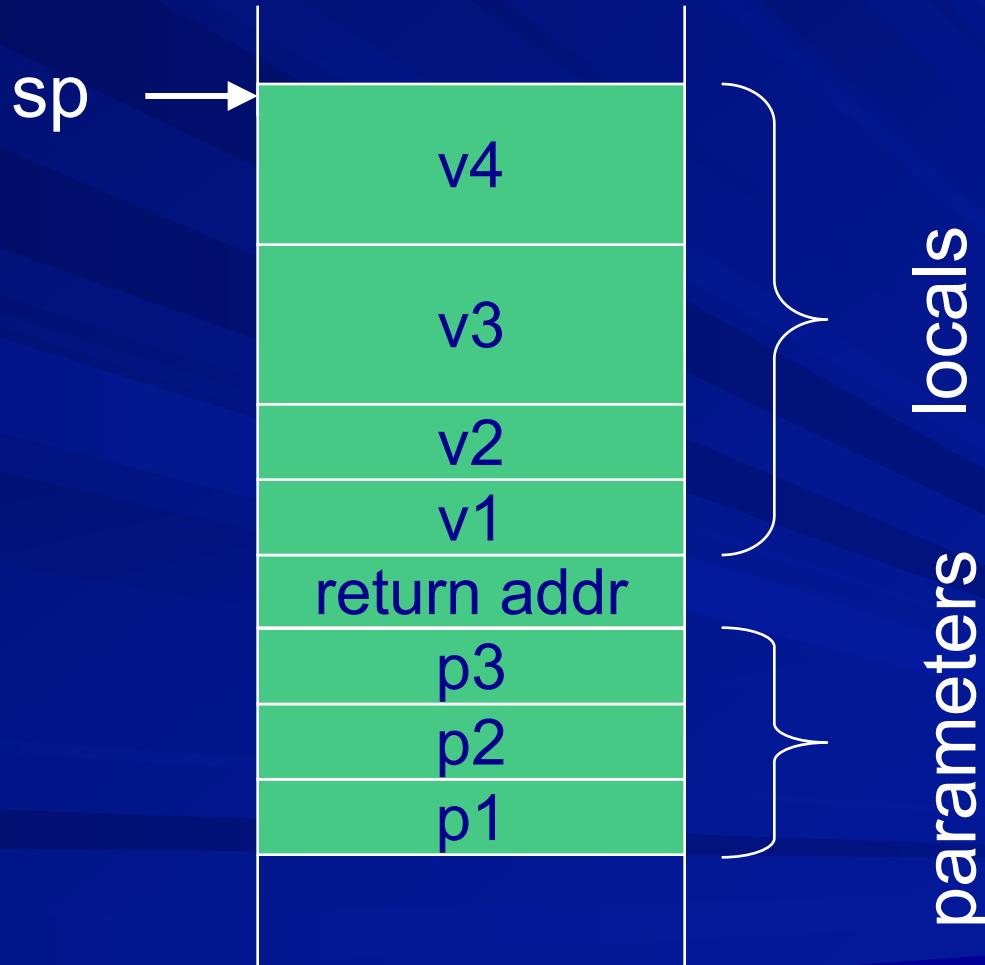
```
    cmp r4, #1
```

```
    beq Ret0
```

```
    mov r4, #1
```

```
    strb r4, [r3, r2]
```

Activation record / stack frame



Assembly program - 2

q = M[i, j];

ldr r3, =M

if (q == 'B')
 return (0);

ldrb r4, [r3, r2]
cmp r4, #'B
beq Ret0

if (q == 'X')
 return (0);

cmp r4, #'X
beq Ret0

if (q == 'F') {
 print (i);

cmp r4, #'F
bne L1
add r0, r0, #'0
print r0

 print (j);

add r0, r1, #'0
print r0

}

b Ret1

Assembly program - 3

p = search(i+1,j);

if(p == 1){

print('N');

return(1)};

L1: ldr r0, [sp,#4]
ldr r1, [sp]
add r0, r0, #1
bl search
cmp r0, #1
bne L2
mov r0, #'N
print r0
b Ret1

L2:

Assembly program - 4

return (0);

Ret0: mov r0, #0
ldr lr, [sp, #8]
add sp, sp, #12
mov pc, lr

return (1);

Ret1: mov r0, #1
ldr lr, [sp, #8]
add sp, sp, #12
mov pc, lr

The data

.data

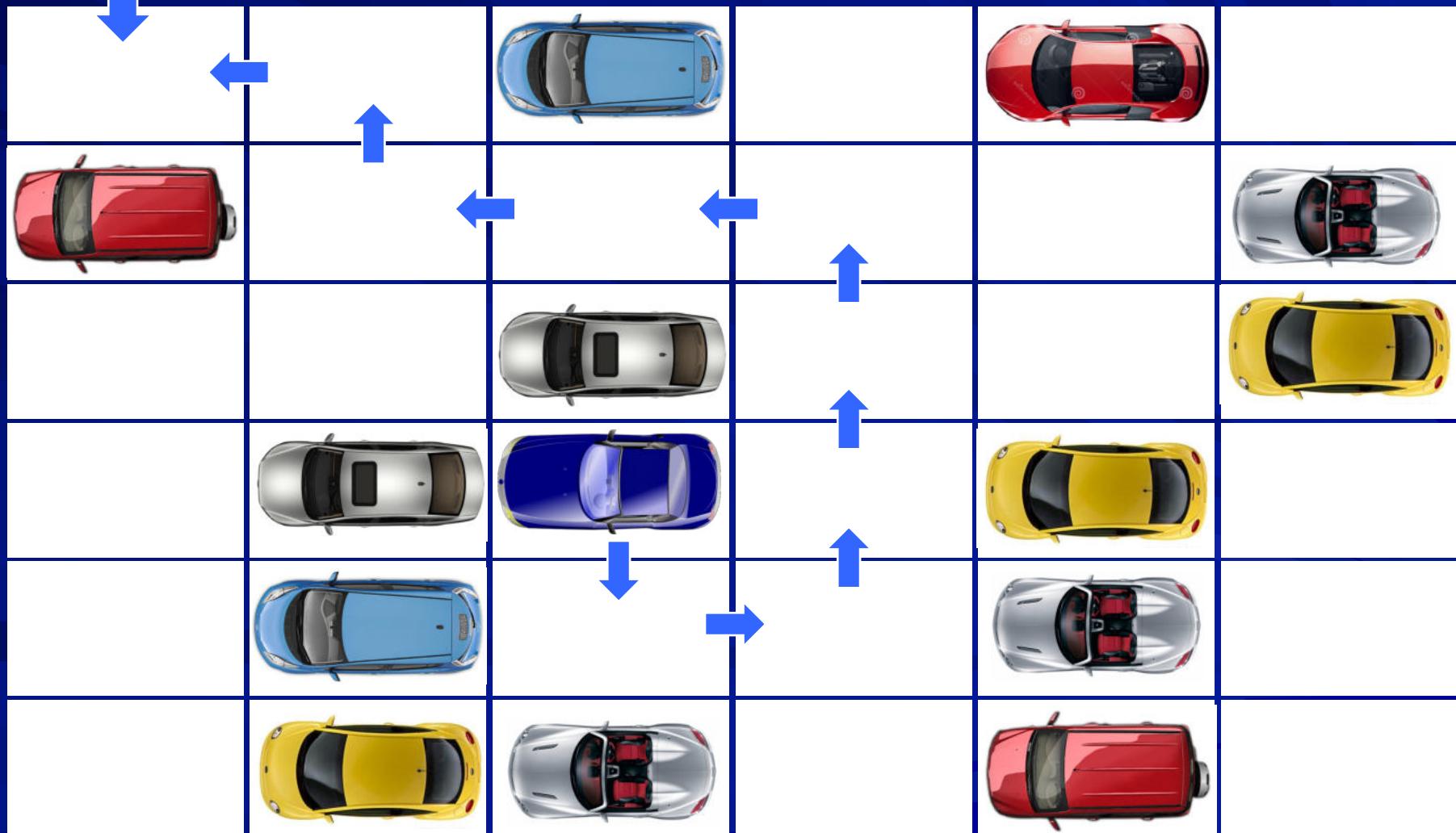
M: .ascii "BBBBBBBBSOXOXOBBXOOO"

V: .word 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0

.end

Result

4 3 S E N N N W W W N W



What happens in this case?



Thank you

COL216

Computer Architecture

More assembly programming

20th Jan, 2022

How to develop assembly program

- High level program



- Low level program in C



- Assembly program



- Machine language program

Low Level Program in C

- Split complex expressions into expressions involving one operation only
- Re-write for loops as simpler loops
 - while () { }
 - do { } while () (preferable)
- Re-write switch – case statements as multi-way branches (may use tables)
- Access arrays using pointers

Complex expression examples

```
x[ i ] = x[ i ] * z + c ;
```

```
t1 = x[ i ] ;  
t2 = t1 * z ;  
t2 = t2 + c ;  
x[ i ] = t2 ;
```

Complex expression examples

```
x[ j ] += y[ j - i ] * z[ i ] + c ;
```

```
q = j - i ;  
t1 = y[ q ] ;  
t2 = z[ i ] ;  
t3 = t1 * t2 ;  
t1 = x[ j ] ;  
t1 += t3 ;  
t1 += c ;  
x[ j ] = t1 ;
```

Loop examples

```
for (i = 0 ; i < max ; i++) {  
    statements  
}
```

```
i = 0 ;  
while (i < max) {  
    statements  
    i++ ;  
}
```

Loop examples

```
for (i = 0 ; i < max ; i++) {  
    statements  
}
```

```
i = 0 ;  
do {  
    statements  
    i++ ;  
} while (i < max) ;
```

Arrays and pointers

```
void make_zero (int * x) {  
  
    for (i = 0 ; i < max ; i++) {  
        x[ i ] = 0 ;  
    } ;  
};  
....  
int A[ max ];  
  
make_zero (A) ;
```

```
void make_zero (int * x) {  
    int * r ;  
    r = x + max ;  
    do {  
        * x++ = 0 ;  
    } while (x < r) ;  
};  
....  
int A[ max ] ;  
int * p ;  
p = & A[ 0 ] ;  
make_zero (p) ;
```

Register allocation

■ Conventions

- r0-r3 used for passing parameters
- callee can destroy r0-r3, r12, preserves the rest

■ Local allocation

- scope of registers allocation is within a function

■ Global allocation

- scope of register allocation is across all functions

Assembly examples : switch

```
switch (k) {  
    case 0:  
        stmt_seq_0 ;  
        break ;  
    case 1:  
        stmt_seq_1 ;  
        break ;  
    case 2:  
        stmt_seq_2 ;  
        break ;  
    case 3:  
        stmt_seq_3 ;  
    };  
stmt_seq_4
```

multi-way branch
back: seq4
....
L0: seq0
 b back
L1: seq1
 b back
L2: seq2
 b back
L3: seq3
 b back

Table of addresses

.text

.....

L0:

L1:

L2:

L3:

.....
.data

Table: .word L0, L1, L2, L3

Table



L0

L1

L2

L3

Multi-way branch using table

ldr r4, =k

ldr r5, [r4]

ldr r2, =Table

@ address of table

ldr pc, [r2, r5, LSL #2]

Input / Output

I/O in ARMSim 1.91

Opcode	Description and Action	Inputs	Outputs	EQU
swi 0x00	Display Character on Stdout	r0: the character		SWI_PrChr
swi 0x02	Display String on Stdout	r0: address of a null terminated ASCII string	(see also 0x69 below)	
swi 0x11	Halt Execution			SWI_Exit
swi 0x12	Allocate Block of Memory on Heap	r0: block size in bytes	r0:address of block	SWI_MeAlloc
swi 0x13	Deallocate All Heap Blocks			SWI_DAlloc
swi 0x66	Open File (mode values in r1 are: 0 for input, 1 for output, 2 for appending)	r0: file name, i.e. address of a null terminated ASCII string containing the name r1: mode	r0:file handle If the file does not open, a result of -1 is returned	SWI_Open
swi 0x68	Close File	r0: file handle		SWI_Close
swi 0x69	Write String to a File or to Stdout	r0: file handle or Stdout r1: address of a null terminated ASCII string		SWI_PrStr
swi 0x6a	Read String from a File	r0: file handle r1: destination address r2: max bytes to store	r0: number of bytes stored	SWI_RdStr
swi 0x6b	Write Integer to a File	r0: file handle r1: integer		SWI_PrInt
swi 0x6c	Read Integer from a File	r0: file handle	r0: the integer	SWI_RdInt
swi 0x6d	Get the current time (ticks)		r0: the number of ticks (milliseconds)	SWI_Timer

I/O in ARMSim 2.1

R0	R1 ^a	Description	Operands in Memory (at address provided by R1)
0x01	M	Open a File	Filename address; filename length; file mode
0x02	M	Close a File	File handle
0x05	M	Write to File	File handle; buffer address; number of bytes to write
0x06	M	Read from File	File handle; buffer address; number of bytes to read
0x09	M	Is a TTY?	File handle
0x0A	M	File Seek	File handle; offset from file start
0x0C	M	File Length	File handle
0x0D	M	Temp File Name	Buffer address; unique integer; buffer length
0x0E	M	Remove File	Filename address; filename length
0x0F	M	Rename a File	Filename 1 address; length 1; Filename 2 address; length 2
0x10	—	Execution Time	
0x11	—	Absolute Time	
0x13	—	Get Error Num	
0x16	A	Get Heap Info	
0x18	Code	Exit Program	

UsefulFunctions.s

- prints -- print a string^[1] to stdout
- fprints – print a string^[1] to file
- fgets -- read one line from file / stdin
- strlen -- compute length of a string^[1]
- atoi -- convert from ASCII to binary
- itoa -- convert from binary to ASCII

[1] null-terminated string

Print integer in radix 10

00000000	00000001	00000000	00001010
----------	----------	----------	----------

should print as 65546

convert to

00000110	00000101	00000101	00000100	00000110
----------	----------	----------	----------	----------

and then to

00110110	00110101	00110101	00110100	00110110
----------	----------	----------	----------	----------

Can you print in radix 2?

Bits and Bytes

Loading address in register

How does the following work ?

```
ldr r4, = A
```

...

```
A: .space 100
```

Loading address in register

ldr r4, = A	1040	ldr r4, [r15, # 0x30]
....
....	1078	1100
....
A: .space 100	1100	A[0]
	1104	A[1]

Pseudo Instructions : logical

xnor r1, r2, r3

eor r1, r2, r3

mvn r1, r1

nand r1, r2, r3

and r1, r2, r3

mvn r1, r1

nor r1, r2, r3

orr r1, r2, r3

mvn r1, r1

Pseudo Instructions : compare and set

slt r1, r2, r3

cmp r2, r3
movlt r1, #1
movge r1, #0

sge r1, r2, r3

cmp r2, r3
movge r1, #1
movlt r1, #0

sgeu r1, r2, r3

cmp r2, r3
mov r1, #0
adc r1, r1, #0

Pseudo Instruction : sign check

sign r1, r2

mov r1, r2, ASR #31

(result is 0 or -1)

mov r1, r2, ASR #31
orr r1, r1, #1

(result is +1 or -1)

Pseudo Instructions : absolute

abs r1, r2

cmp r2, #0

movge r1, r2

rsblt r1, r2, #0

mov r3, r2, ASR #31

eor r1, r2, r3

sub r1, r1, r3

mov r3, r2, ASR #31

add r1, r2, r3

eor r1, r1, r3

Pseudo Instruction : swap

swap r1, r2

eor r1, r1, r2

eor r2, r1, r2

eor r1, r1, r2

sub r1, r1, r2

add r2, r1, r2

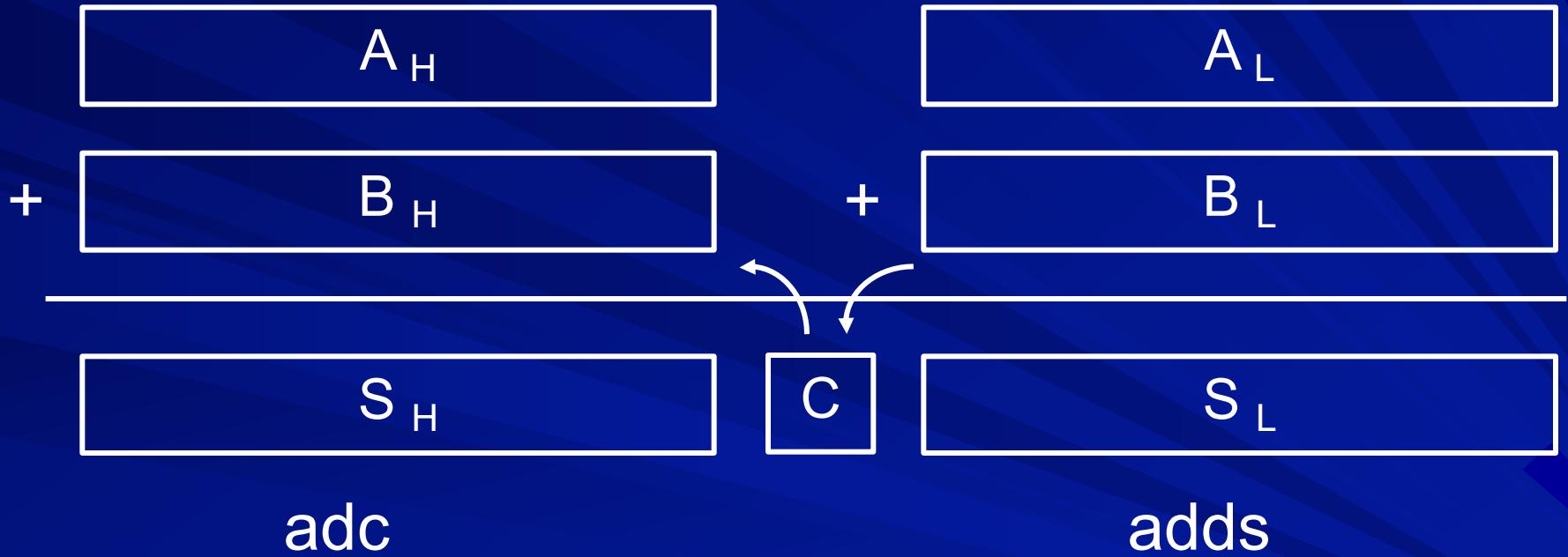
sub r1, r2, r1

add r1, r1, r2

sub r2, r1, r2

sub r1, r1, r2

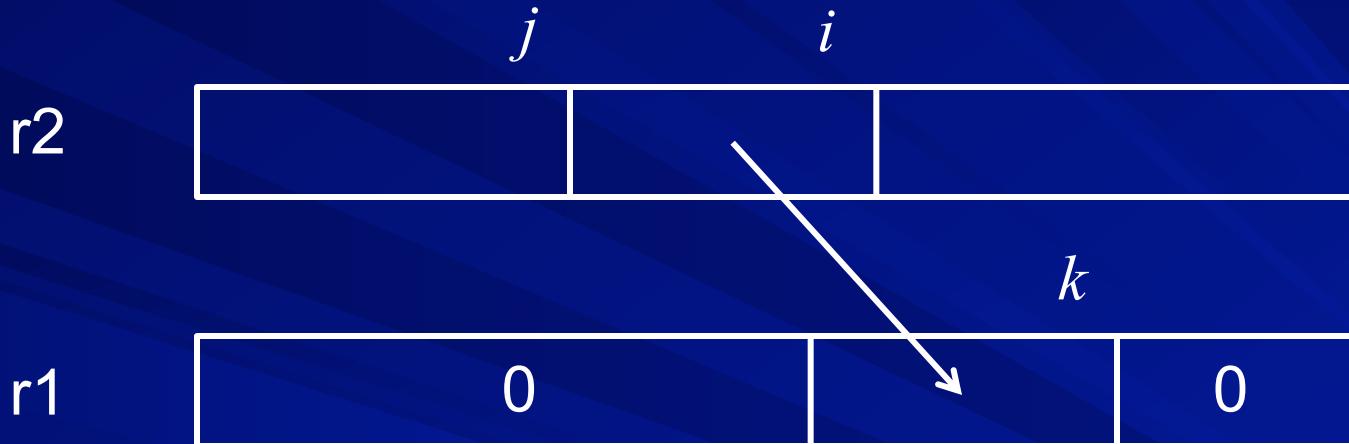
Double word addition



adc
adds r0, r2, r4
adc r1, r3, r5

adds
@ A is in r3, r2
@ B is in r5, r4
@ S is in r1, r0

Field Extraction

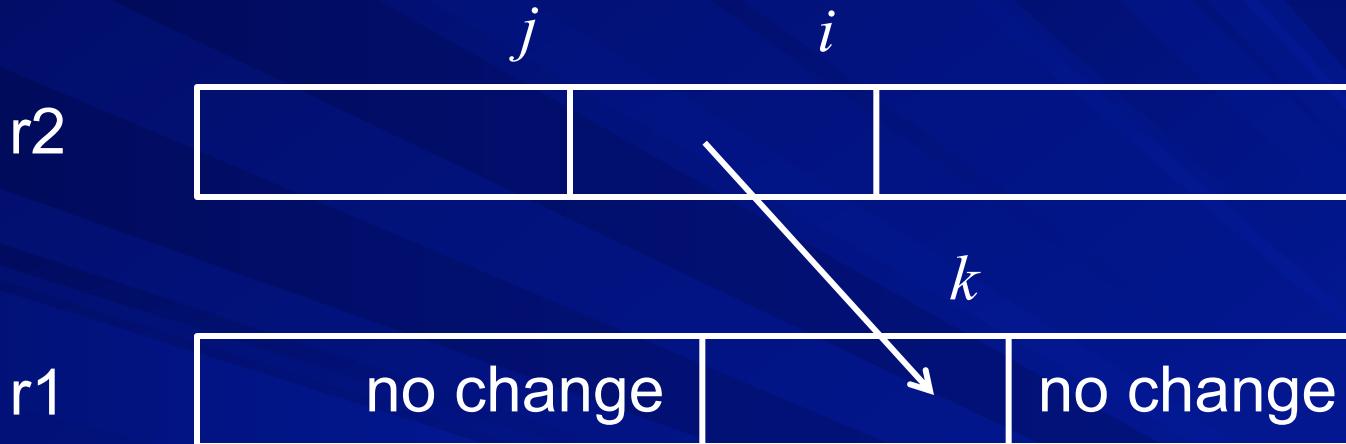


mov r1, r2, LSL # 32 - j

mov r1, r1, LSR # 32 - j + i

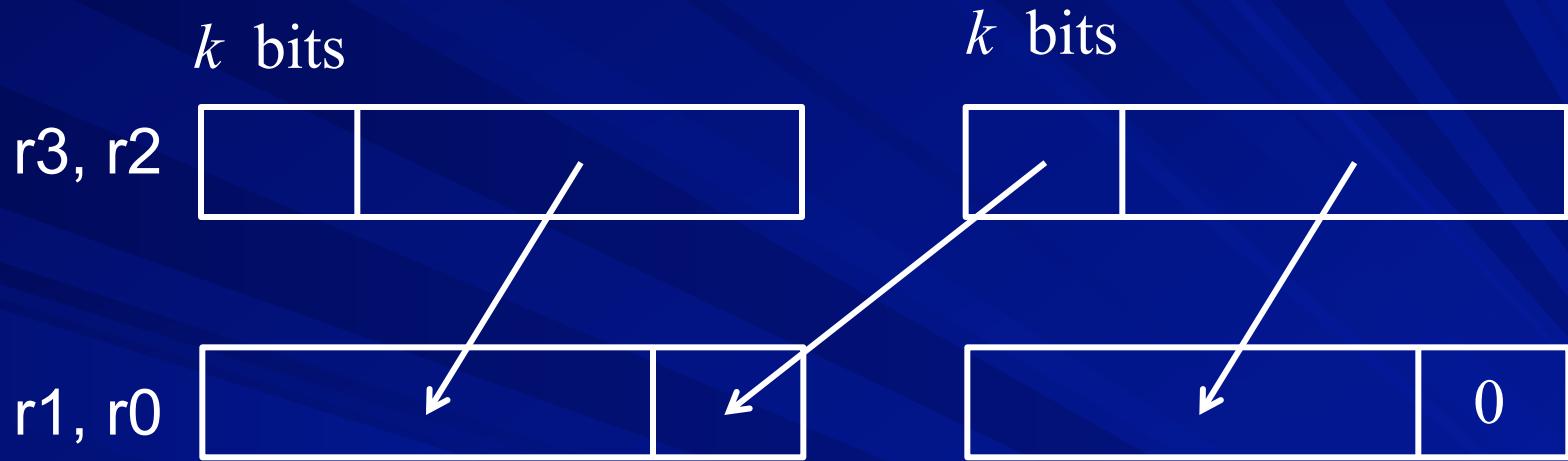
mov r1, r1, LSL # k

Field Extraction



```
mov r3, r2, LSL # 32 - j  
mov r3, r3, LSR # 32 - j + i  
mov r3, r3, LSL # k  
mov r1, r1, ROR # k  
mov r1, r1, LSR # j - i  
mov r1, r1, ROR # 32 - j + i - k  
orr r1, r1, r3
```

Double word shift



```
mov  r1, r3, LSL # k  
mov  r0, r2, LSL # k  
orr  r1, r1, r2, LSR # 32 - k
```

Looking at 1's in a number

- Check if there is a single 1
 - Is the number a power of 2 ?
- If number is 2^k , find k
- Count how many 1's are there
- Check if there are odd no. of 1's or even no. of 1's

Computing ($v \& (v - 1)$)

- What is special about this expression ?
- Something interesting happens to the rightmost 1 in v .

$b_{31}b_{30}\dots b_{k+2}b_{k+1}100\dots 00$

v

& $b_{31}b_{30}\dots b_{k+2}b_{k+1}011\dots 11$

$v - 1$

$b_{31}b_{30}\dots b_{k+2}b_{k+1}000\dots 00$

If v is a power of 2

$$v = 2^k$$

$$\Rightarrow b_{31} = b_{30} \dots = b_{k+2} = b_{k+1} = 0$$

$$\Rightarrow (v \& (v - 1)) = 0$$

$$v = 0$$

$$\Rightarrow (v \& (v - 1)) = 0$$

therefore, $v \neq 0$ and $(v \& (v - 1)) = 0$

$$\Rightarrow v = 2^k \text{ for some } k$$

Check if power of 2

movs r0, r1	@ v is in r1
beq out	@ v = 0
sub r2, r1, #1	@ r2 = v - 1
ands r2, r1, r2	@ r2 = v & (v - 1)
movne r0, #0	@ not a power of 2
moveq r0, #1	@ power of 2
out:	@ result in r0

Counting 1's

c = 0

while (v > 0) do {

v &= v - 1 => clears least significant 1

c++

}

Counting 1's

```
@ c: r0,  v: r1,  t:r2
c = 0                      mov  r0, #0
while (v > 0) do {        loop: movs r2, r1
                           beq  out
                           sub   r2, r2, #1
                           v &= t          and   r1, r1, r2
                           c++            add   r0, r0, #1
                           }              b     loop
                                         out:
```

Find k where $v = 2^k$
 v has a single 1. Find its position.
Perform binary search

bbbbbbbbbbbbbbbb**bbbbbbbbbbbbbbbb**

bbbbbbb**bbbbbbb**bbbbbbb**bbbbbbb**

bbb**bbb**bbb**bbb**bbb**bbb**bbb**bbb**bbb**bbb**

bbbbbb**bbb**bbb**bbb**bbb**bbb**bbb**bbb**bbb**bbb**

bbbbbb**bbb**bbb**bbb**bbb**bbb**bbb**bbb**bbb**bbb**

Find k where v = 2^k

```
c = 0 ;  
if (v & 0x0000FFFF = 0)  
    {c += 16; v >>= 16;};  
if (v & 0x0000000FF = 0)  
    {c += 8; v >>= 8;};  
if (v & 0x00000000F = 0)  
    {c += 4; v >>= 4;};  
if (v & 0x000000003 = 0)  
    {c += 2; v >>= 2;};  
if (v & 0x000000001 = 0)  
    {c += 1; v >>= 1;};
```

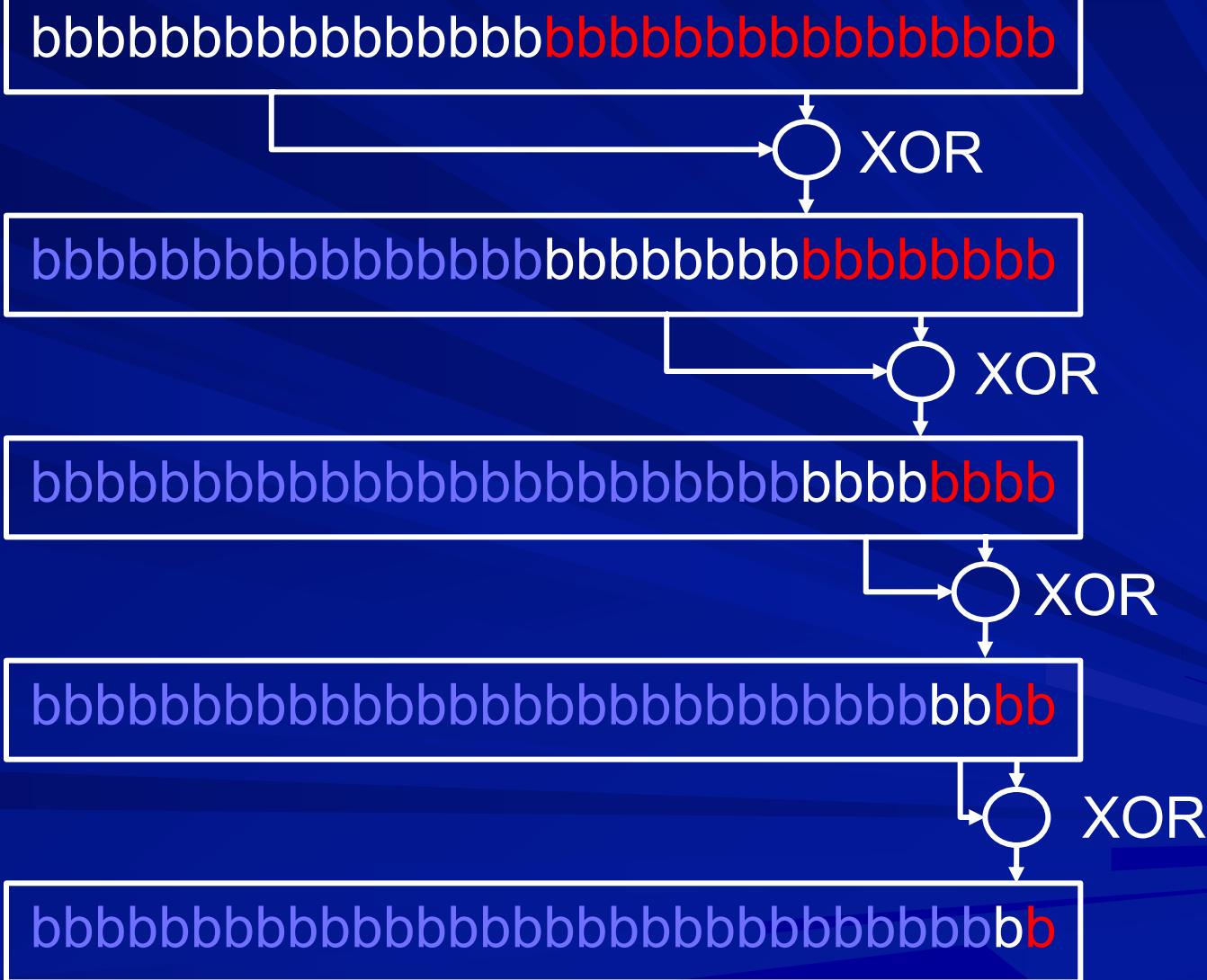
Find k where $v = 2^k$

```
c = 0 ;  
if (v & 0x0000FFFF = 0)  
  {c += 16; v >>= 16;};  
if (v & 0x000000FF = 0)  
  {c += 8; v >>= 8;};  
if (v & 0x0000000F = 0)  
  {c += 4; v >>= 4;};  
if (v & 0x00000003 = 0)  
  {c += 2; v >>= 2;};  
if (v & 0x00000001 = 0)  
  {c += 1; v >>= 1;};
```

ands r2, r1, 0xFF
addeq r0, r0, #8
moveq r1, r1, LSR #8

Computing Parity

4 of the
5 steps
are shown



Computing Parity

$v = v \text{ xor } (v >> 16)$	eor r0, r0, r0, LSR #16
$v = v \text{ xor } (v >> 8)$	eor r0, r0, r0, LSR #8
$v = v \text{ xor } (v >> 4)$	eor r0, r0, r0, LSR #4
$v = v \text{ xor } (v >> 2)$	eor r0, r0, r0, LSR #2
$v = v \text{ xor } (v >> 1)$	eor r0, r0, r0, LSR #1
$v = v \& 1$	and r0, r0, #1

Bit reversal

- interchange two 16 bit fields
- interchange four 8 bit fields
- interchange eight 4 bit fields
- interchange sixteen 2 bit fields
- interchange thirty two 1 bit fields

Bit reversal

mask16 = 0x0000FFFF

bbbbbbbbbbbbbbbbbbbb
 $\text{b} \text{b} \text{b} \text{b} \text{b} \text{b} \text{b} \text{b}$ $\text{b} \text{b} \text{b} \text{b} \text{b} \text{b} \text{b} \text{b}$

mask8 = 0x00FF00FF

bbbbbbbb
 $\text{b} \text{b} \text{b} \text{b}$ $\text{b} \text{b} \text{b} \text{b}$ $\text{b} \text{b} \text{b} \text{b}$ $\text{b} \text{b} \text{b} \text{b}$

mask4 = 0x0F0F0F0F

bbbb
 $\text{b} \text{b} \text{b} \text{b}$ $\text{b} \text{b} \text{b} \text{b}$ $\text{b} \text{b} \text{b} \text{b}$ $\text{b} \text{b} \text{b} \text{b}$

mask2 = 0x33333333

bb
 $\text{b} \text{b}$ $\text{b} \text{b} \text{b} \text{b}$ $\text{b} \text{b} \text{b} \text{b}$ $\text{b} \text{b} \text{b} \text{b}$ $\text{b} \text{b} \text{b} \text{b}$

mask1 = 0x55555555

bbb
 $\text{b} \text{b} \text{b}$ $\text{b} \text{b} \text{b} \text{b}$ $\text{b} \text{b} \text{b} \text{b}$ $\text{b} \text{b} \text{b} \text{b}$ $\text{b} \text{b} \text{b} \text{b}$

Bit reversal

$R = v \& \text{mask16}; L = (v >> 16) \& \text{mask16}$

$v = R << 16 | L$

$R = v \& \text{mask8}; L = (v >> 8) \& \text{mask8}$

$v = R << 8 | L$

$R = v \& \text{mask4}; L = (v >> 4) \& \text{mask4}$

$v = R << 4 | L$

$R = v \& \text{mask2}; L = (v >> 2) \& \text{mask2}$

$v = R << 2 | L$

$R = v \& \text{mask1}; L = (v >> 1) \& \text{mask1}$

$v = R << 1 | L$

Bit reversal (one step)

R = v & mask8

L = (v >> 8) & mask8

v = R << 8 | L

```
@ v : r0, R : r1, L : r0,  
@ masks : r4,r5,r6,r7,r8  
and r1, r0, r5  
and r0, r5, r0, LSR #8  
orr r0, r0, r1, LSL #8
```

Reference

- <http://graphics.stanford.edu/~seander/bithacks.html>

Thank you

COL216

Computer Architecture

Architecture Space
24th Jan, 2022

What is “Computer Architecture”?

Any relation with architecture (of buildings) ?

Is there an analogy ?

architecture (function, appearance)

vs

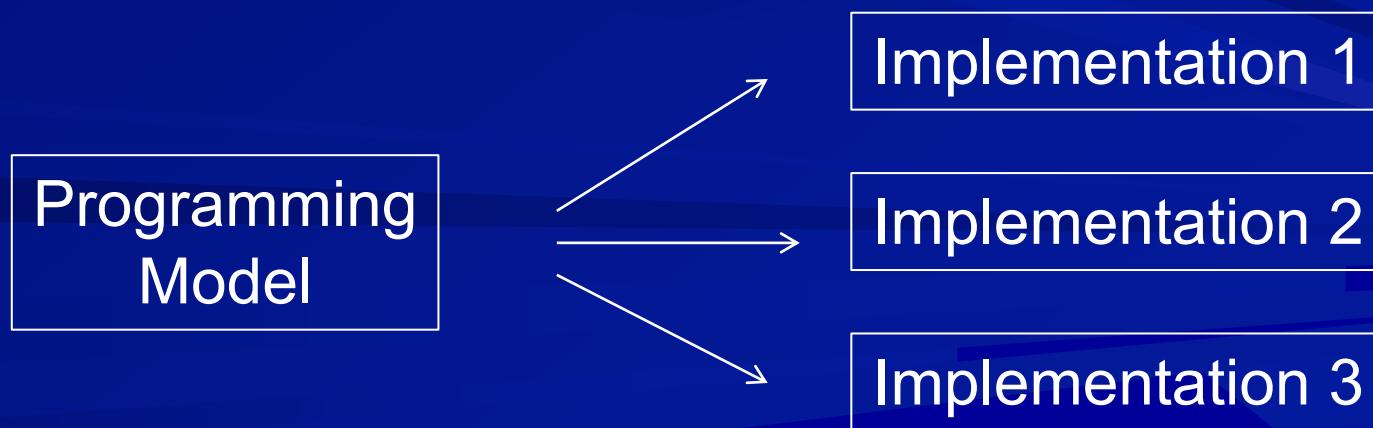
engineering (structure, material)

of

buildings | computers

Architecture of a Computer

- Conceptual design and fundamental operational view of a computer system
- Programming model of a computer, but not a particular implementation

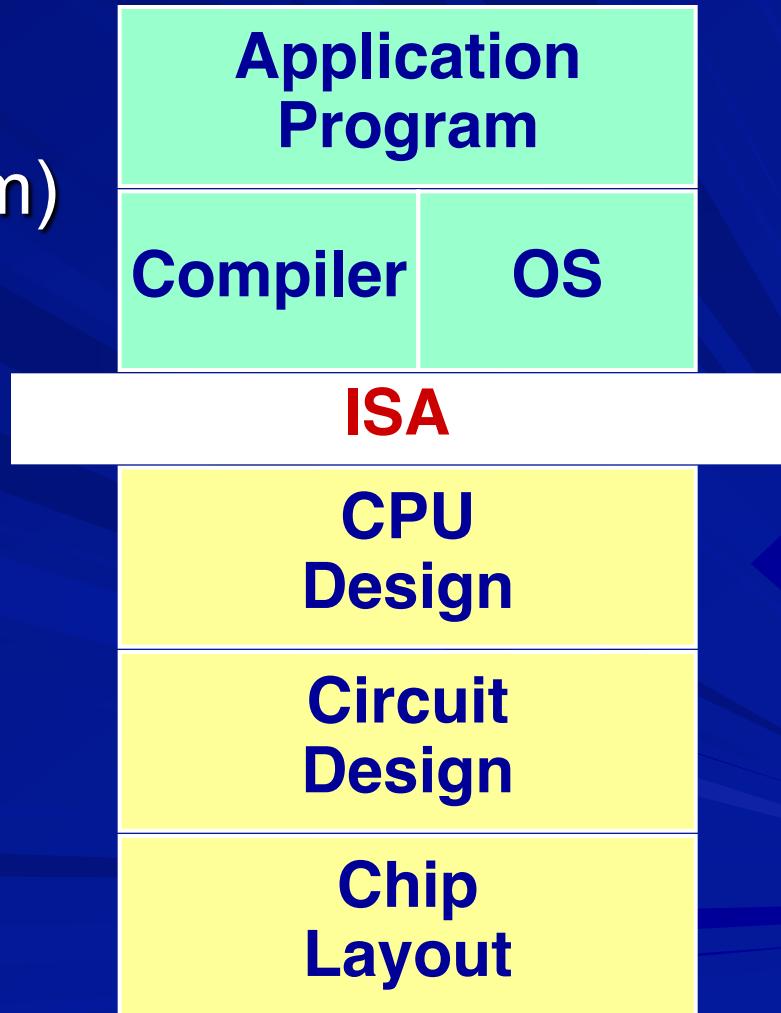


Architecture levels

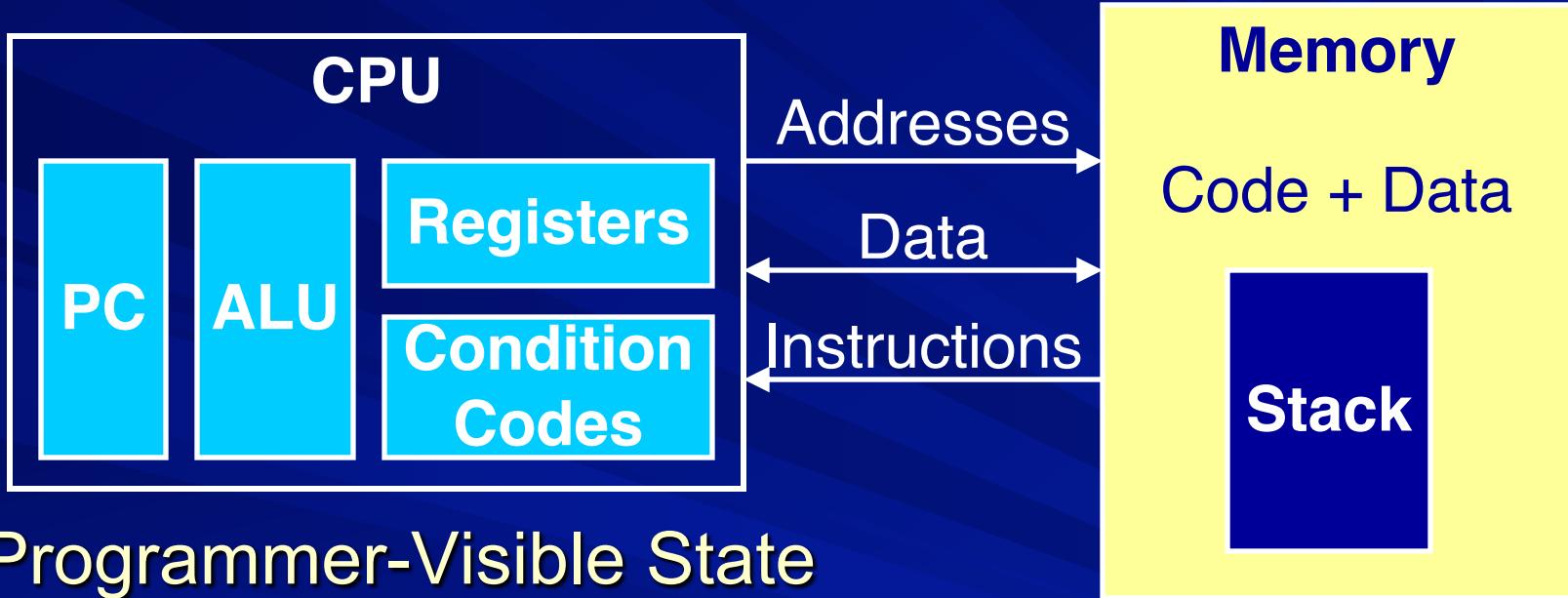
- Instruction set architecture (ISA)
 - Lowest level visible to a programmer
- Micro architecture
 - Fills the gap between instructions and logic modules
- System architecture
 - How processors, memories, buses, etc are put together

Instruction Set Architecture

- Machine 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 built – how to make it run fast



The Abstract Machine



■ Programmer-Visible State

- PC Program Counter
- Register File
 - heavily used data
- Condition Codes

- Memory
 - Byte array
 - Code + data
 - stack

Computers in past

Univac, early 1950s



Main frame
computer

PDP 1, early 1960s

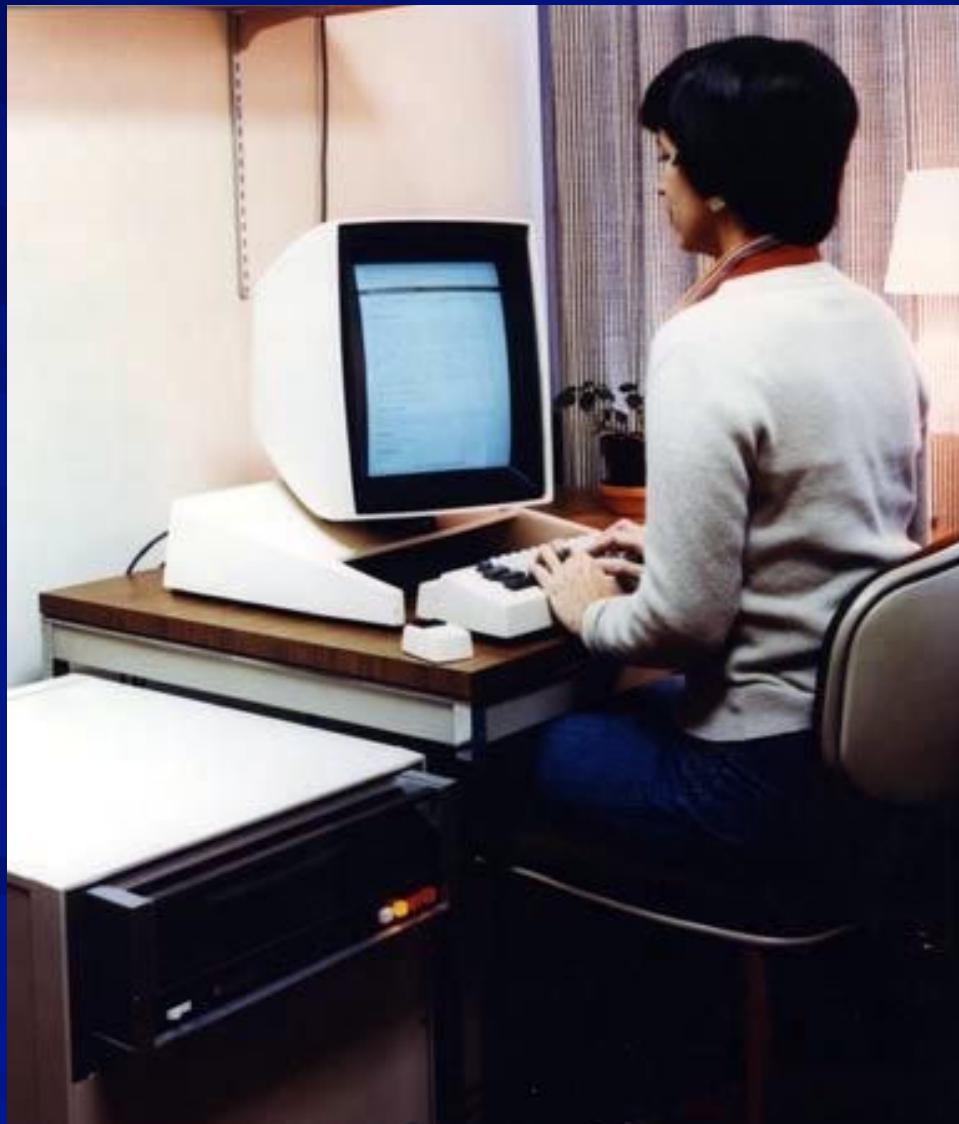


Mini
computer

IBM 360, mid 1960s



Xerox PARC Alto, mid 1970s



Personal
computer

IBM PC, early 1980s



Apple Macintosh, mid 1980s



Apple PowerBook, early 1990s



Sony Vaio Mid 1990s



Earth simulator, 2002



Super
computer

Computers today

- Laptops
- Desktops
- Servers
- Data centres
- Super computers
- Phones, tablets, watches?
- Embedded computers?
- IoT ?

Embedded computers

- Processor as an intelligent electronic component **rather than** Processor as a computing engine
- Real time operation
- Requires hardware-software co-design
- Highly customized

Difference between processors?

- What is the difference between processors used in desk-tops, lap-tops, mobile phones, washing machines etc.?
 - Performance / speed
 - Power consumption
 - Cost
 - General purpose / special purpose

Studying computer architecture

- How computers work, basic principles
- How to analyze their performance
- How computers are designed and built to meet functional, performance and cost goals
- Advanced concepts related to modern processors (caches, pipelines, etc.)

Why should we study this?

This knowledge will be useful if you need to

- design/build a new computer
 - rare opportunity
- design/build a new version of a computer
- improve software performance
- purchase a computer
- provide a solution with an embedded computer

Is it enough to study ARM ISA?

- YES

It is possible to learn the basic principles through this example architecture

- NO

It is useful to have some idea about the other prominent architectures also

Basic Principles

- How a program is represented and executed in hardware
 - Performing computations
 - Organizing data
 - static and dynamic structures
 - Organizing control flow
 - conditionals, loops, functions, recursion

What constitutes ISA?

Main features:

- Set of basic/primitive operations
- Storage structure – registers/memory
- How addresses are specified
- How instructions are encoded

Set of operations

Basic/primitive operations only vs more powerful operations

- e.g. “ $K++$ and branch to L if $K>N$, where K is in memory” or “copy a block of data in memory”
- Goal is to reduce number of instructions executed
- Danger is a slower cycle time and/or a higher CPI

Location of operands – R/M

- R-R both operands in registers
- R-M one operand in register and one in memory
- M-M both operands in memory
- R+M Combines R-R, R-M and M-M

How many operand fields?

- 3 address machine
- 2 address machine
- 1 address machine
- 0 address machine

$r1 = r2 + r3$

$r1 = r1 + r2$

$Acc = Acc + x$

Acc is implicit

add values on
top of stack

Register organizations

- Register-less machine
- Accumulator based machine
- A few special purpose registers
- Several general purpose registers
- Large number of registers / register windows

RISC vs. CISC

Reduced (vs. Complex) Instruction Set Computer

- Uniformity of instructions
- Simple set of operations and addressing modes
- Register based architecture with 3 address instructions
- Virtually all new instruction sets since 1982 have been RISC

RISC and CISC

- Concept introduced by Patterson & Ditzel in 1980
- Followed by development of MIPS at Stanford and Berkeley RISC at UCB
- Virtually all new instruction sets since 1982 have been RISC
- Patterson & Hennessy were given Touring Award in 2018 for their “pioneering work on computer chip design”

“The case for the Reduced Instruction Set Computer”

by Patterson & Ditzel, ACM Sigarch, 1980

- “.. *the next generation of VLSI computers may be more efficiently implemented as RISC's than CISC's.*”
- Reasons for increased complexity
- Usage and consequences of CISC instructions
- RISC and VLSI technology

Why processors became complex?

- Trend to provide complex instruction to do common tasks
- Adding instructions facilitated by microprogrammed control approach
- New generations created by adding new instructions, rather than fresh designs
- Code density was considered important
- Attempt to support HLLs

Consequences of CISC approach

- Only a few instructions used by compilers
- Sequence of simpler instructions may be faster than complex instructions
- Advantage of higher code density diminished with advances in memory technology
- Benefit for HLL features questionable
- Lengthened design times, increased design errors

RISC and VLSI technology

- Easier to fit in a chip
- Short design time suitable for rapidly changing technology
- Efficient implementation
- Area saved usable for cache, pipelining etc

Thank you

COL216

Computer Architecture

Architecture Space – contd.

27th Jan, 2022

RISC and CISC

- Concept introduced by Patterson & Ditzel in 1980
- Followed by development of MIPS at Stanford and Berkeley RISC at UCB
- Virtually all new instruction sets since 1982 have been RISC
- Patterson & Hennessy were given Touring Award in 2018 for their “pioneering work on computer chip design”

“The case for the Reduced Instruction Set Computer”

by Patterson & Ditzel, ACM Sigarch, 1980

- “.. *the next generation of VLSI computers may be more efficiently implemented as RISC's than CISC's.*”
- Reasons for increased complexity
- Usage and consequences of CISC instructions
- RISC and VLSI technology

Why processors became complex?

- Factors that motivated design of complex processors
- Factors that facilitated design of complex processors

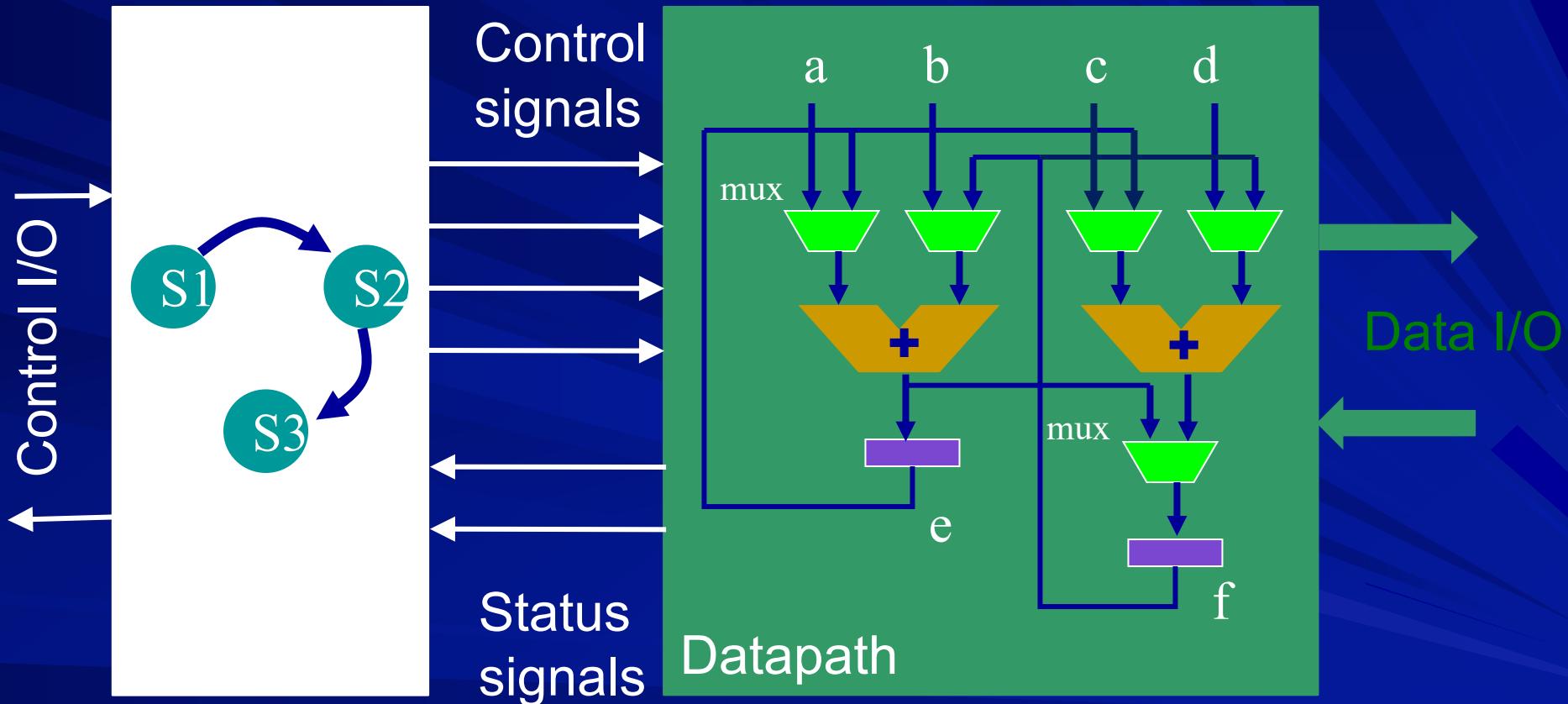
Motivations for complex processors

- To provide complex instruction to do common tasks, typically required by high level language programs
- Since memory was expensive and slower than processor, executing fewer instructions to do a given task was better
- New generations created by adding new instructions, rather than fresh designs

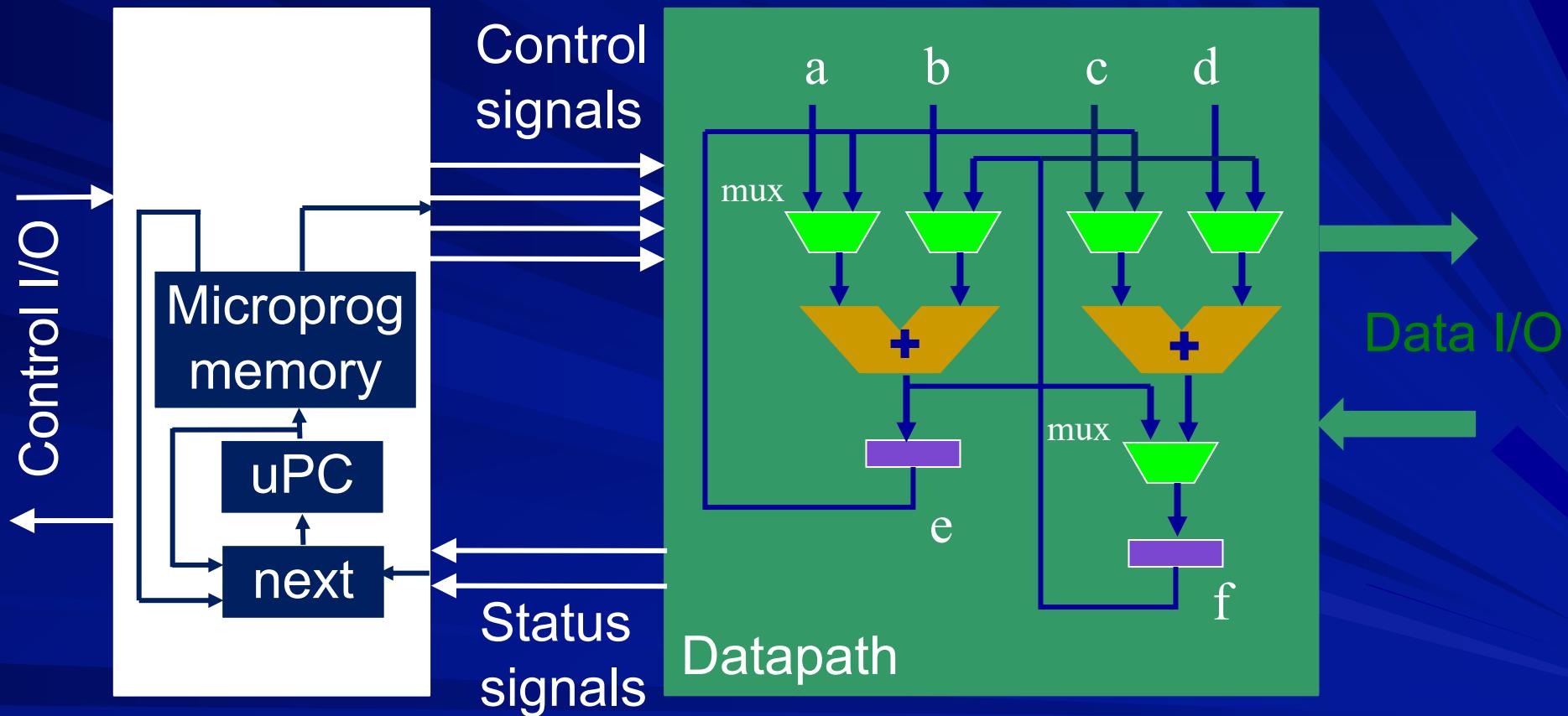
What facilitated complex designs?

- Processors were typically implemented using microprogrammed control
- Adding instructions was made easy by adding microcode (control store size 256x56 in PD11, 5120x96 in VAX 11/780)
- Simple processors can be easily implemented using FSM based control

FSM based control



Microprogrammed control



Consequences of CISC approach

- Only a few instructions used by compilers
- Sequence of simpler instructions may be faster than complex instructions
- Advantage of higher code density diminished with advances in memory technology
- Benefit for HLL features questionable
- Lengthened design times, increased design errors

RISC and VLSI technology

- Easier to fit in a chip
- Short design time suitable for rapidly changing technology
- Efficient implementation
- Area saved usable for cache, pipelining etc

RISC characteristics

- Uniform instruction size, limited formats
- Simple set of operations and addressing modes
- Register based architecture (R-R)
- 3 address instructions for arithmetic/logic operations
- Separate load – store operations

RISC examples

- MIPS at Stanford and Berkeley RISC
 - MIPS used by NEC, Nintendo, Silicon Graphics, Sony
- SUN's SPARC
 - Sun, Texas Instr, Toshiba, Fujitsu, Cypress, Tatung
- PowerPC: developed by IBM+Motorola+Apple in 1993
 - Based on POWER architecture of IBM (System/6000)
 - Used in Macintosh, embedded systems
- HP's PA-RISC
- DEC's Alpha
- ARM, Hitachi SuperH, Mitsubishi M32R (embedded)
- CDC 6600 (1960's)

CISC examples

- Main frames
- VAX-11/780 from Digital Equipment Corporation (DEC) - 1977
 - upward compatible with mini computer PDP-11
- Motorola 680x0
 - Apple Macintosh and other desk top computers
- Intel 80x86
 - Personal computers, servers

MIPS, PowerPC and SPARC

- MIPS - only 3 very simple formats
 - No flags, slt instruction
 - lui instruction
 - Opcode implies addressing mode, eg add / addi
- PowerPC
 - condition register, predication
 - link register for subroutines and loops
 - count register - counter update and branch
- SPARC: **S**calable **P**rocessor **ARChitecture**
 - register windows, condition flags

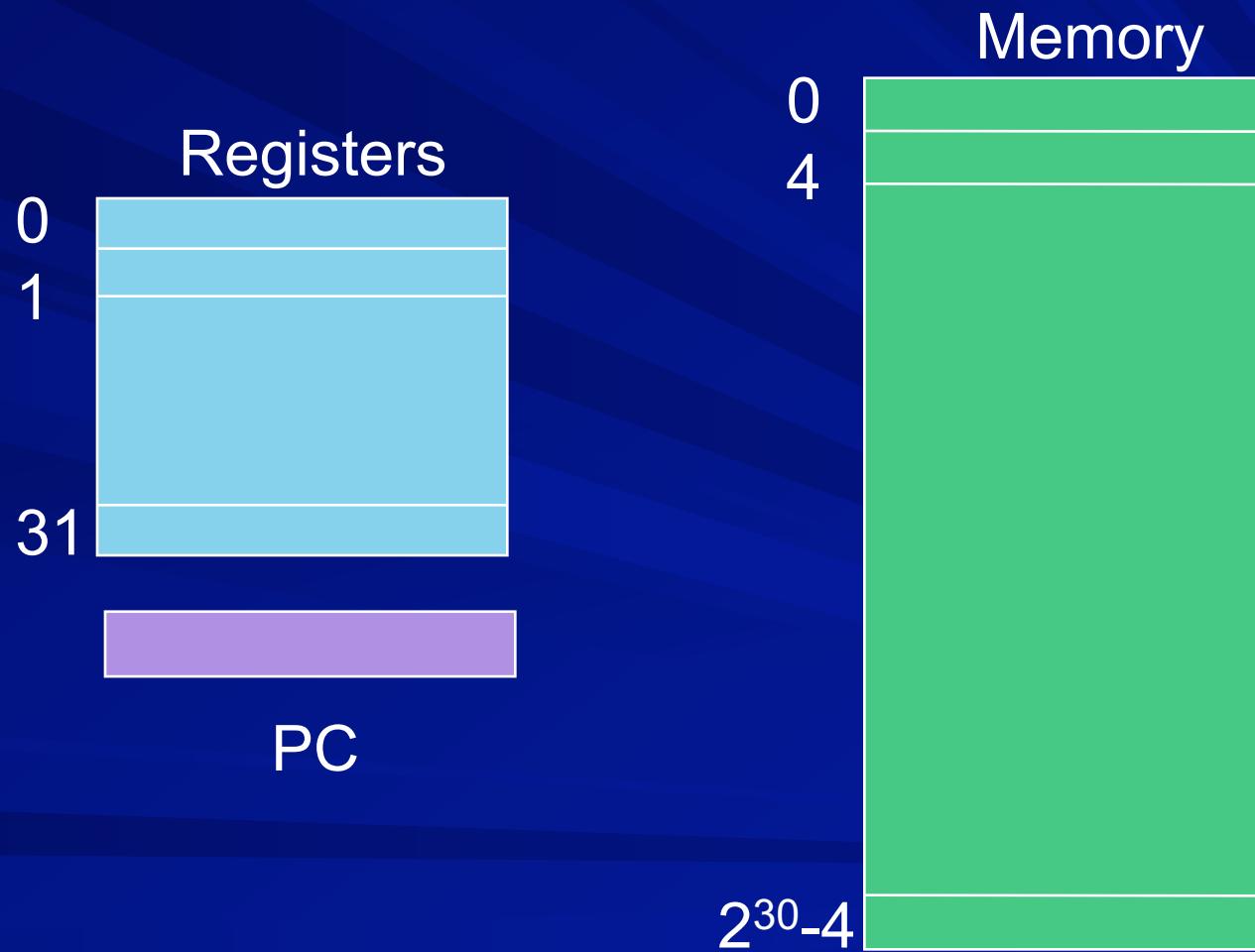
Advanced RISC Machine

- Available as a core that can be embedded in a chip.
- Supports implementations across a wide spectrum of performance points.
- Billions of devices have shipped, dominant architecture across many market segments.
- The architectural simplicity has led to very small implementations allowing devices with very low power consumption.

Modes, Extensions

- Thumb mode
 - 16 bit instructions
 - Slower but saves power
- Jezelle extension
 - Java byte code execution
- NEON
 - SIMD extension
- VFP
 - Floating point

MIPS ISA - storage



MIPS ISA – addressing modes`

Purpose

- Operand sources
- Result destinations
- Jump targets

Addressing modes

- Immediate
- Register
- Base/index
- PC relative
- (pseudo) Direct
- Register indirect

MIPS ISA features - encoding

- addi, lui, beq, bne, lw, sw I - format

op	rs	rt	16 bit number
----	----	----	---------------

- j, jal J - format

op	26 bit number
----	---------------

- add, slt, jr R - format

op	rs	rt	rd	shamt	funct
----	----	----	----	-------	-------

PowerPC Architecture

- Developed by IBM, Motorola and Apple in 1993
- Based on POWER architecture of IBM (System/6000)
- 32 bit as well as 64 bit architectures
- Used in Macintosh, embedded systems

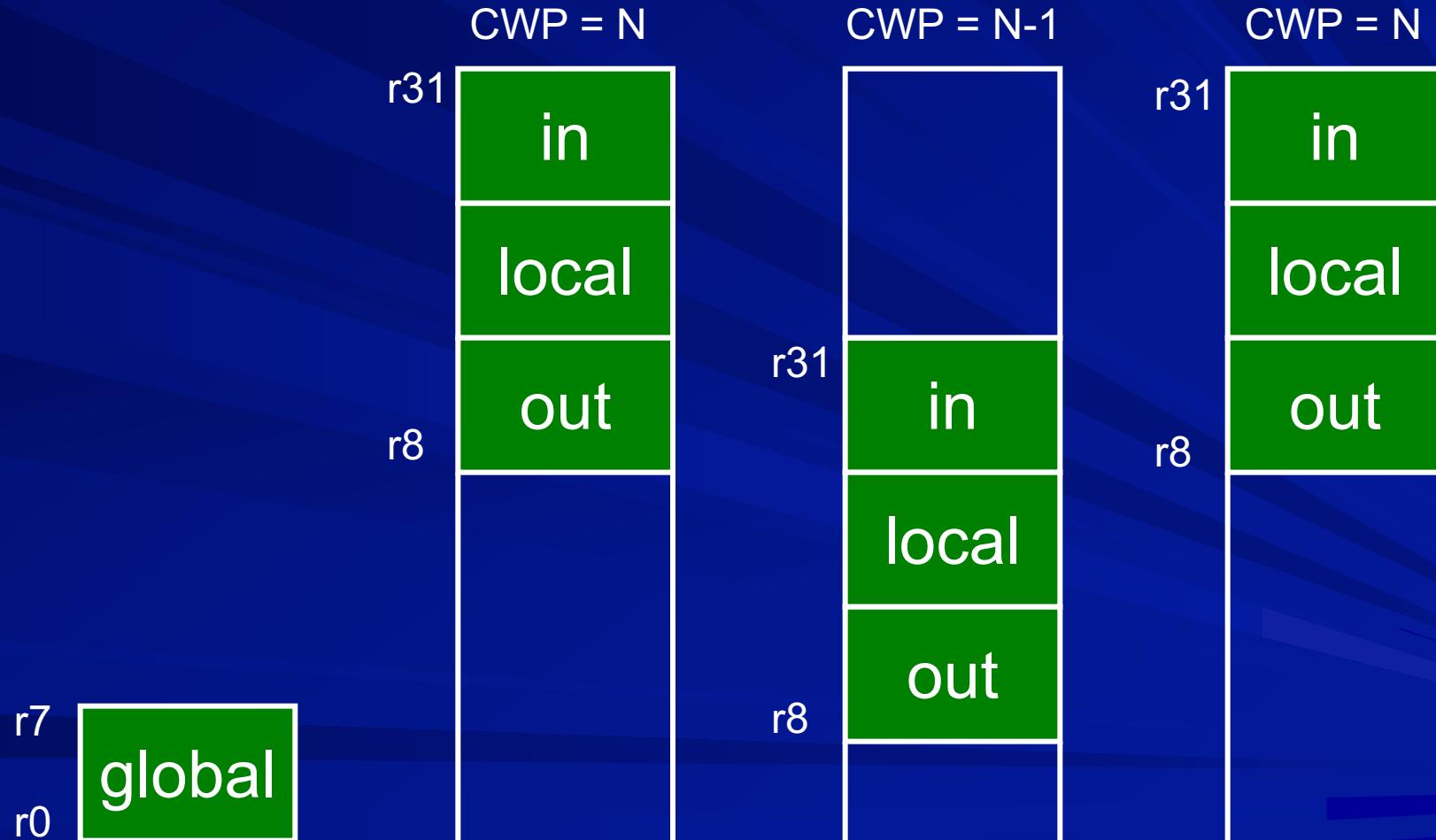
PowerPC Registers

- 32 general purpose registers
- Condition register: set by arithmetic/logical instructions, tested by conditional branches
- Link register: stores return addresses for procedure calls and loops
- Count register: iteration count for a loop

SPARC Architecture

- Scalable Processor ARChitecture
- Scalable: the design allows for forward compatibility of programs
- Has been ‘scaled’ to 64 bit processors
- Implemented by Sun, Texas Instruments, Toshiba, Fujitsu, Cypress, Tatung etc.

SPARC Registers



Data Processing Instructions

	31	25	20	15	10	4	0
Alpha		Op ⁶	Rs1 ⁵	Rs2 ⁵	Opx ¹¹	Rd ⁵	
MIPS		Op ⁶	Rs1 ⁵	Rs2 ⁵	Rd ⁵	Const ⁵	Opx ⁶
PowerPC		Op ⁶	Rd ⁵	Rs1 ⁵	Rs2 ⁵		Opx ¹¹
PA-RISC		Op ⁶	Rs1 ⁵	Rs2 ⁵	Opx ¹¹	Rd ⁵	
SPARC	Op ²	Rd ⁵	Opx ⁶	Rs1 ⁵	0	Opx ⁸	Rs2 ⁵

31 29 24 18 13 12 4 0

	31	27	19	15	11	3	0
ARM		Opx ⁴	Opx ⁴	Rs1 ⁴	Rd ⁴	Opx ⁸	Rs2 ⁴

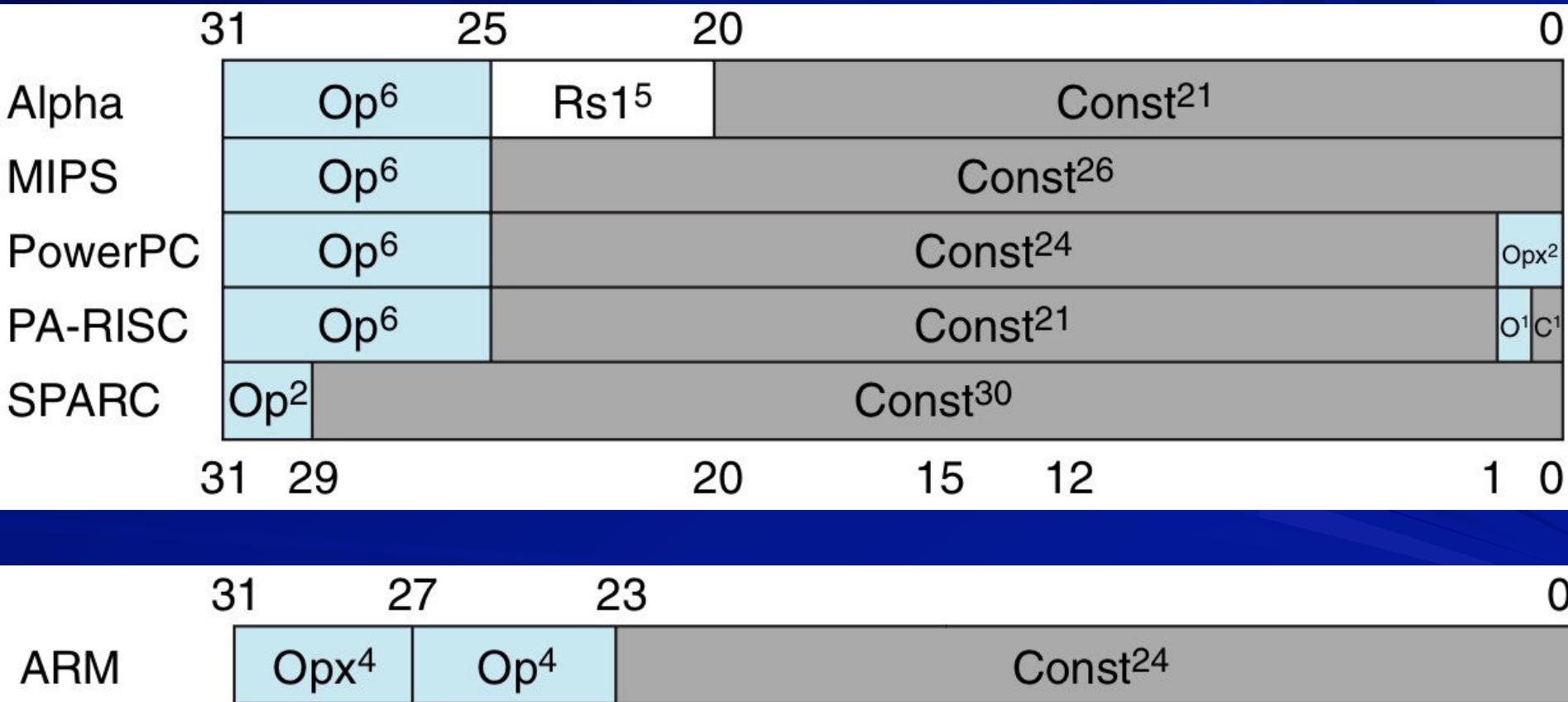
Immediate, Load/store

	31	25	20	15	0
Alpha		Op ⁶	Rd ⁵	Rs1 ⁵	Const ¹⁶
MIPS		Op ⁶	Rs1 ⁵	Rd ⁵	Const ¹⁶
PowerPC		Op ⁶	Rd ⁵	Rs1 ⁵	Const ¹⁶
PA-RISC		Op ⁶	Rs2 ⁵	Rd ⁵	Const ¹⁶
SPARC	Op ²	Rd ⁵	Opx ⁶	Rs1 ⁵	1 Const ¹³
	31	29	24	18	13 12 0
ARM	31	27	19	15	11 0
	Opx ⁴	Op ³	Rs1 ⁴	Rd ⁴	Const ¹²

Conditional Branch

	31	25	20	15	0
Alpha	Op ⁶	Rs1 ⁵		Const ²¹	
MIPS	Op ⁶	Rs1 ⁵	Opx ⁵ /Rs2 ⁵	Const ¹⁶	
PowerPC	Op ⁶	Opx ⁶	Rs1 ⁵	Const ¹⁴	Opx ²
PA-RISC	Op ⁶	Rs2 ⁵	Rs1 ⁵	Opx ³	Const ¹¹
SPARC	Op ²	Opx ¹¹		Const ¹⁹	OC
	31	29	18	12	1 0
ARM	Opx ⁴	Op ⁴		Const ²⁴	
	31	27	23	0	

Unconditional jump / call



VAX Architecture

- Objective: minimize code size, make assembly language easy
instructions from 1 to 54 bytes long!
- 32-bit words and addresses
- virtual memory
- 16 GPRs (r15 = PC, r14 = SP), CCs
- extremely orthogonal and memory-memory
- decode as byte stream - variable in length
- opcode: operation, #operands, operand types

VAX data types and addr modes

■ Data types

- 8, 16, 32, 64, 128
- char string - 8 bits/char
- decimal - 4 bits/digit

■ Addressing modes

- literal 6 bits. immediates 8, 16, 32 bits
- register, register deferred
- 8, 16, 32 bit displacements [deferred]
- indexed (scaled)
- autoincrement, autodecrement [deferred]

VAX Operations

- data transfer including string move
- arith/logical (2 and 3 operands)
- control (branch, jump, etc)
- function calls save state
- bit manipulation
- floating point - add, sub, mul, div, polyf
- crc (cyclic redundancy check),
- insque (insert in Q)

VAX instruction format example

Instruction length: 1 to 54 bytes

addl3 R1, 737(R2), #456

byte 1: addl3

byte 2: mode, R1

byte 3: mode, R2

byte 4,5: 737

byte 6: mode

byte 7-10: 456

VAX instruction with 6 operands

addp6 op1, op2, op3, op4, op5, op6
⇒ add two packed decimal numbers

op1, op2: length and start addr of number1
op3, op4: length and start addr of number2
op5, op6: length and start addr of sum

Intel x86 history

Grown from 4 bit \Rightarrow 8 bit \Rightarrow 16 bit \Rightarrow 32 bit \Rightarrow 64 bit

- 1978: 8086 16 bit architecture
- 1980: 8087 floating point coprocessor
- 1982: 80286 24 bit addr space, more instr
- 1985: 80386 32 bits, new addressing modes
- 1989-1995: 80486, Pentium, -- Pro, performance
- 1997: MMX, Pentium II
- 1999: Pentium III (Streaming SIMD Extension)
- 2001: SSE2, double precision FP
- 2003: AMD64
- 2004: SSE3, 2006: SSE4, 2007: SSE5(AMD) . . .

Intel x86 features

■ Complexity:

- Instructions (~900) from 1 to 17 bytes long
- one operand must act as both a src and dst
- one operand can come from memory
- complex addressing modes, e.g., “base + scaled index with 8 - 32 bit displacement”
- Permitted instruction - address mode combinations irregular (lots of special cases, hard to learn!)
- Effect by each instruction on condition codes is somewhat complex, irregular

Other registers in Intel x86

- 16 bit segment registers:
CS, SS, DS, ES, FS, GS
- Flags (condition codes) 32 bit
- Instruction pointer (PC) 32 bit

Intel x86 : peculiar feature

Up to 3 “prefixes” for an instruction

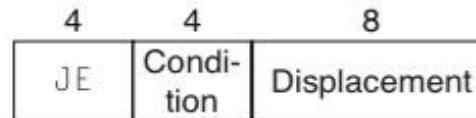
- override default data size
- override default segment register
- lock bus for a semaphore
- repeat the following instruction
- override default address size

Instruction format

	Prefixes	Opcode	ModR/M	SIB	Displacement	Immediate			
Bytes	0-3	1-3	0-1	0-1	0-4	0-4			
ModR/M			<table border="1"><tr><td>Mod</td><td>Reg</td><td>R/M</td></tr></table>	Mod	Reg	R/M			
Mod	Reg	R/M							
	Bits	2	3	3					
SIB			<table border="1"><tr><td>Scale</td><td>Index</td><td>Base</td></tr></table>	Scale	Index	Base			
Scale	Index	Base							
	Bits	2	3	3					

Some Instruction Formats

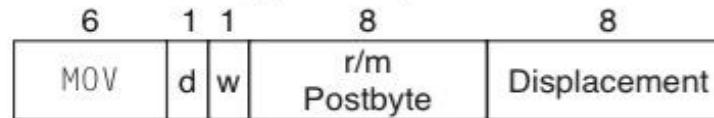
a. JE EIP + displacement



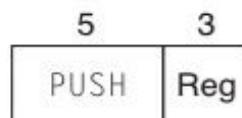
b. CALL



c. MOV EBX, [EDI + 45]



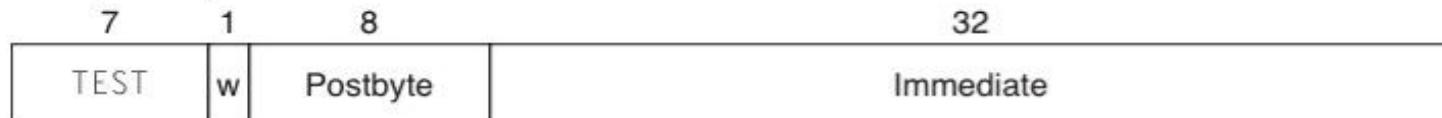
d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



x86 instruction listings - Wikipedia

Thank you

COL216

Computer Architecture

Designing a processor:
Datapath building blocks

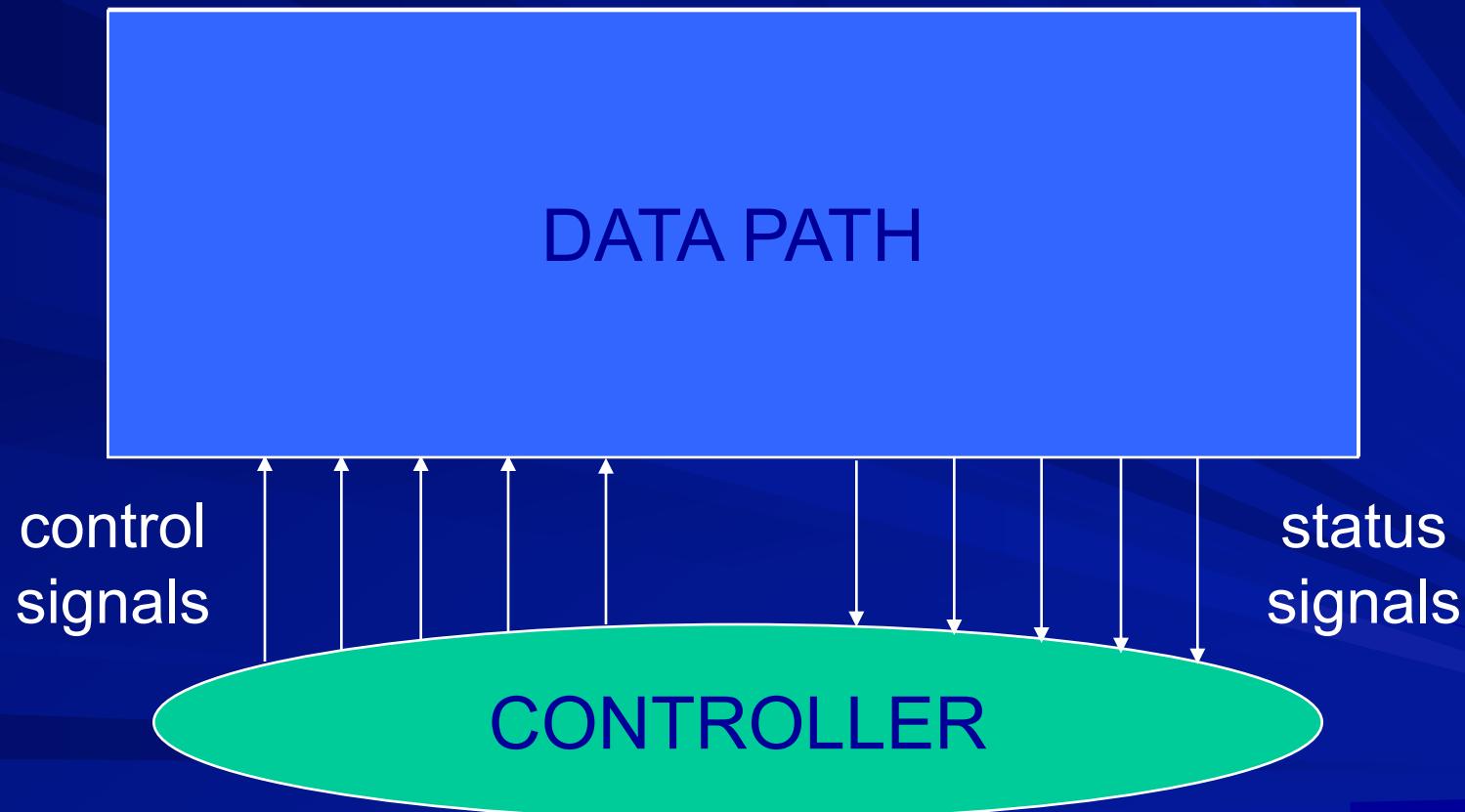
31st January, 2022

Overview

- Given an ISA, how to design a Micro architecture
- There are numerous possibilities
- Different combinations of performance – cost – power consumption are possible
- CAD tools help in analyzing the trade offs between these parameters
- CAD tools are required for physical implementation

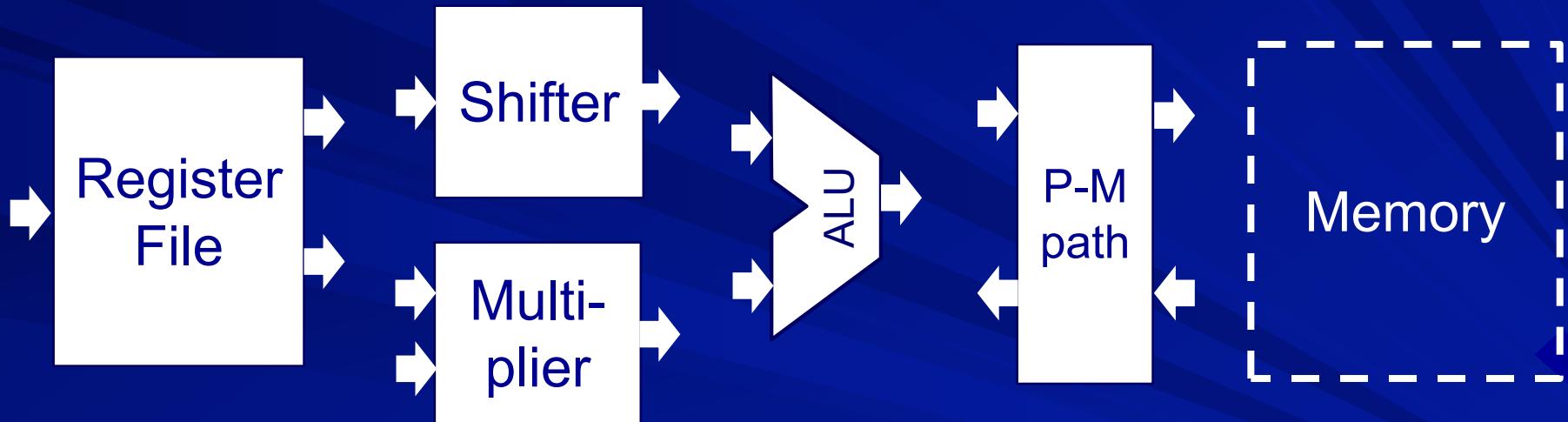
CPU datapath + controller

Focus of this lecture: Major building blocks for datapath



Datapath major blocks

Some additional blocks are required to glue these together



Later we will see how instruction fetching, instruction decode, operand access and state updation are done

ARM instruction subset

adc	add	rsb	rsc
sbc	sub		
and	bic	eor	orr
cmn	cmp	teq	tst
mov	mvn		
mla	mul		
ldr	str		
b	bl		
swi			

Instructions excluded - 1

- cdp : co-processor data processing
- mcr : move from CPU register to co-processor register
- mrc : move from co-processor register to CPU register
- ldc : load co-processor register from memory
- stc : store co-processor register to memory

Instructions excluded - 2

- bx : branch and exchange
 - switch between ARM and Thumb modes
- ldm : load multiple registers
- stm : store multiple registers
- swp : swap register - memory (atomically)

Instruction classes

- Data processing (DP)
 - arithmetic
 - logical
 - test
 - move
- Branch
- Multiply
- Data transfer (DT)

ALU operation for each class

- Data processing (DP)
 - arithmetic
 - logical
 - test
 - move
- Branch
- Multiply
- Data transfer (DT)
 - Perform specified arithmetic or logical or relational operation or no operation
 - Target address
 - Multiply {and add}
 - Memory address

ALU operations for DP instructions

Instr	ins [24-21]	Operation	
and	0 0 0 0	Op1 AND	Op2
eor	0 0 0 1	Op1 EOR	Op2
sub	0 0 1 0	Op1 + NOT Op2 + 1	
rsb	0 0 1 1	NOT Op1 +	Op2 + 1
add	0 1 0 0	Op1 +	Op2
adc	0 1 0 1	Op1 +	Op2 + C
sbc	0 1 1 0	Op1 + NOT Op2 + C	
rsc	0 1 1 1	NOT Op1 +	Op2 + C
tst	1 0 0 0	Op1 AND	Op2
teq	1 0 0 1	Op1 EOR	Op2
cmp	1 0 1 0	Op1 + NOT Op2 + 1	
cnn	1 0 1 1	Op1 +	Op2
orr	1 1 0 0	Op1 OR	Op2
mov	1 1 0 1		Op2
bic	1 1 1 0	Op1 AND NOT Op2	
mvn	1 1 1 1		NOT Op2

ALU operations for other instructions

Instr	ins [23] : U	Operation
ldr	1	Op1 + Op2
ldr	0	Op1 + NOT Op2 + 1
str	1	Op1 + Op2
str	0	Op1 + NOT Op2 + 1

b	Op1 + Op2 + k
bl	Op1 + Op2 + k

mul	Op1 * Op2
mla	Op1 * Op2 + Op3

Instruction variations / suffixes

■ Suffixes

- Predication
- Setting flags
- Word / half word / byte transfer

■ Operand variations

- Shifting / rotation
- Pre / post increment / decrement
- Auto increment / decrement

Instructions with Suffixes

- Arithmetic: <add|sub|rsb|adc|sbc|rsc> {cond} {s}
- Logical: <and | orr | eor | bic> {cond} {s}
- Test: <cmp | cmn | teq | tst> {cond}
- Move: <mov | mvn> {cond} {s}
- Branch: <b | bl> {cond}
- Multiply: <mul | mla> {cond} {s}
- Load/store: <ldr | str> {cond} {b | h | sb | sh }

Shift/rotate in DP instructions

- 12 bit operand2 in DP instructions



– 8 bit unsigned number,



4 bit rotate spec, or

– 4 bit register number,



8 bit shift specification

shift type: LSL, LSR, ASR, ROR

shift amount: 5 bit constant or 4 bit register no.

Shift/rotate in DT instructions

- 12 bit offset field in DT instructions

cond	F	opc	Rn	Rd	operand2
4	2	6	4	4	12

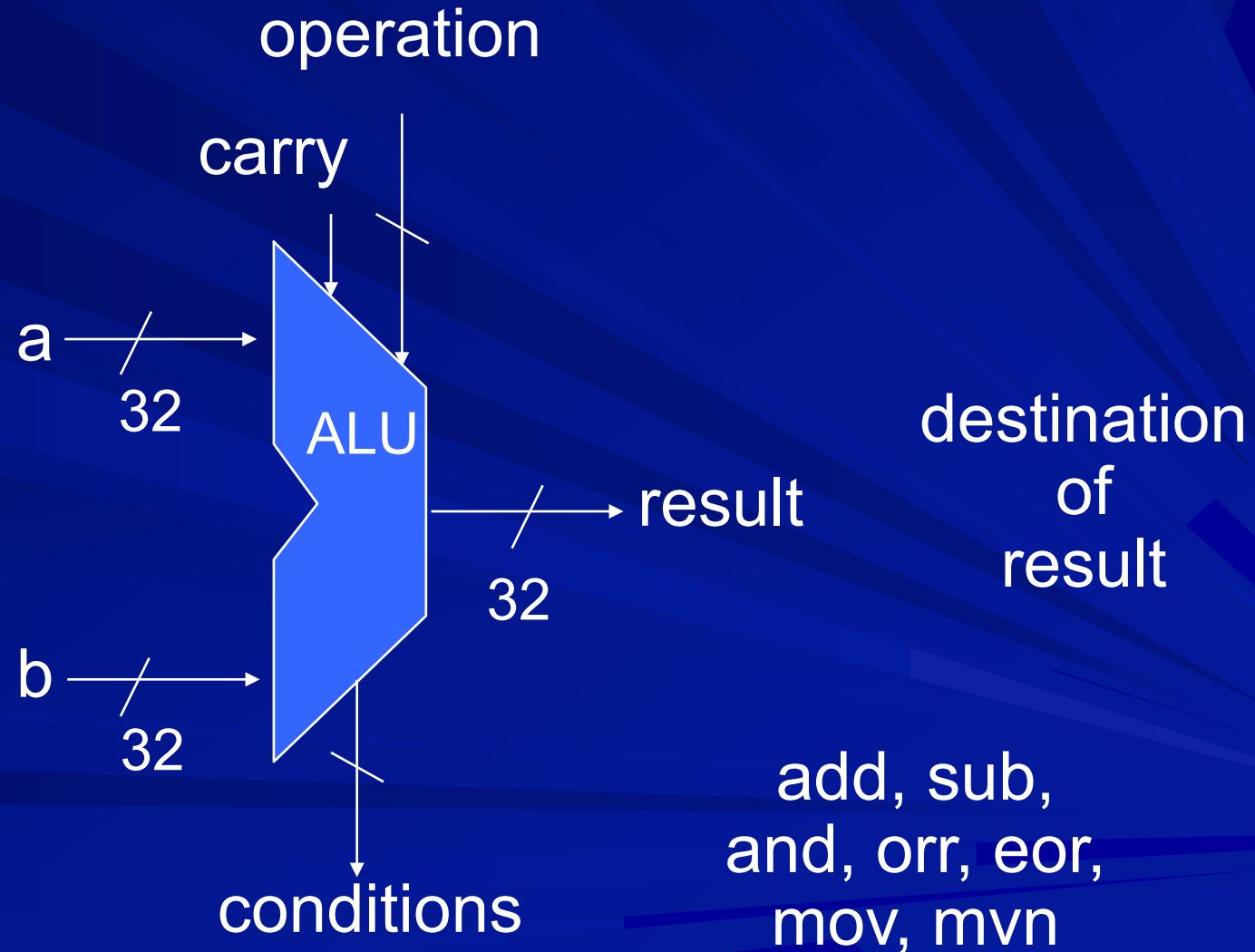
- 12 bit unsigned constant

- or

- 4 bit register number, 8 bit shift specification
(same format as DP instructions)

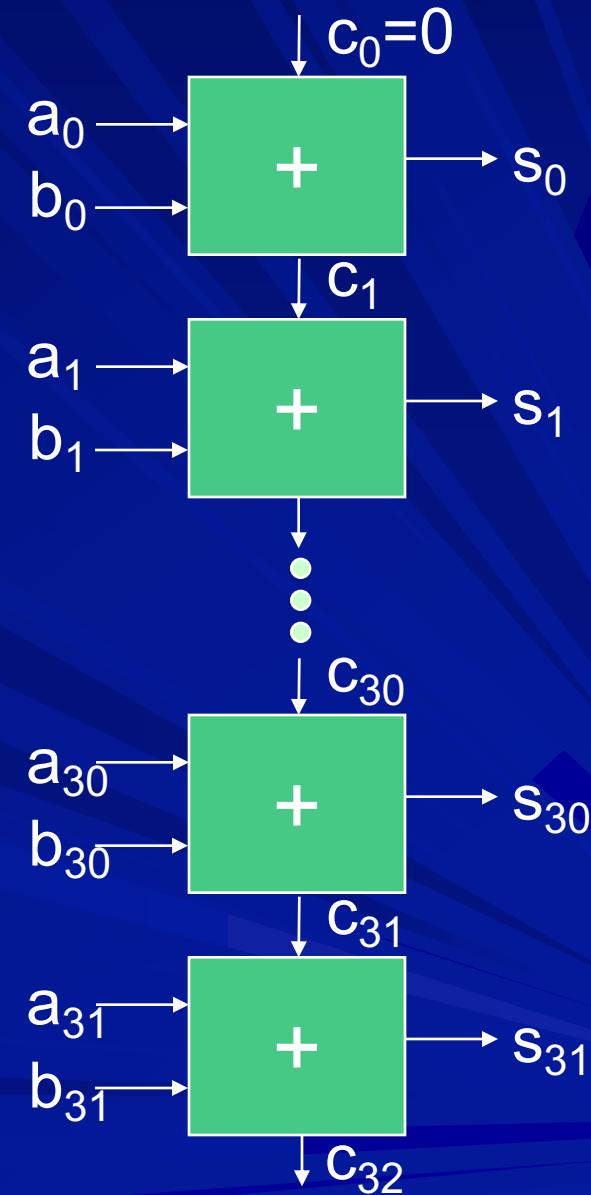
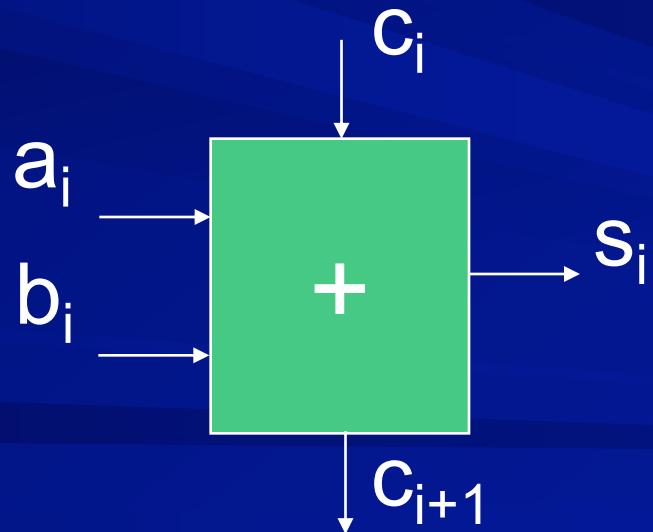
Arithmetic and Logical operations

sources
of
operands



Adder circuit

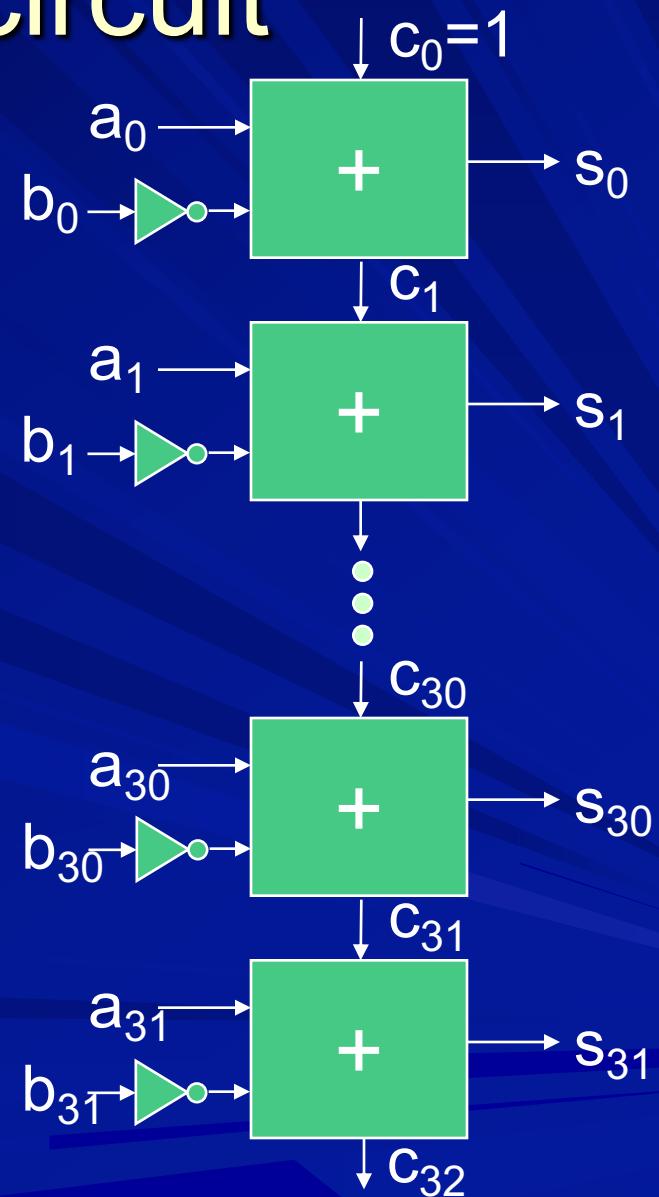
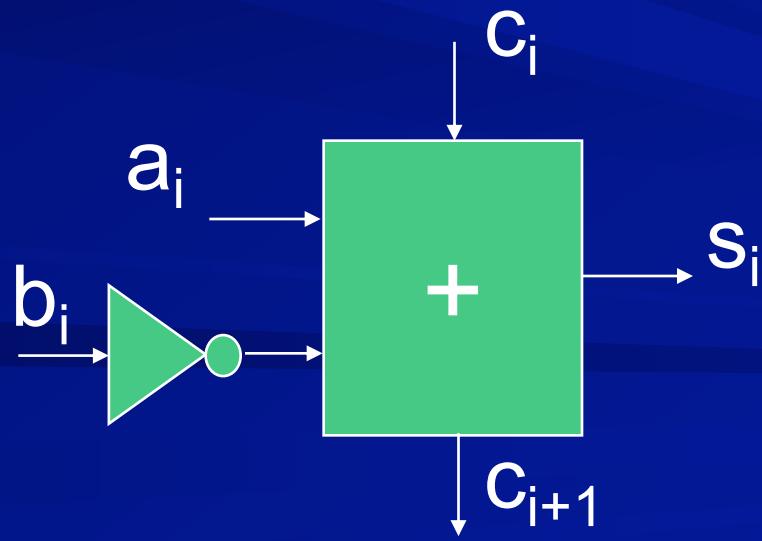
- Perform addition with carry propagation or carry look ahead



Subtraction circuit

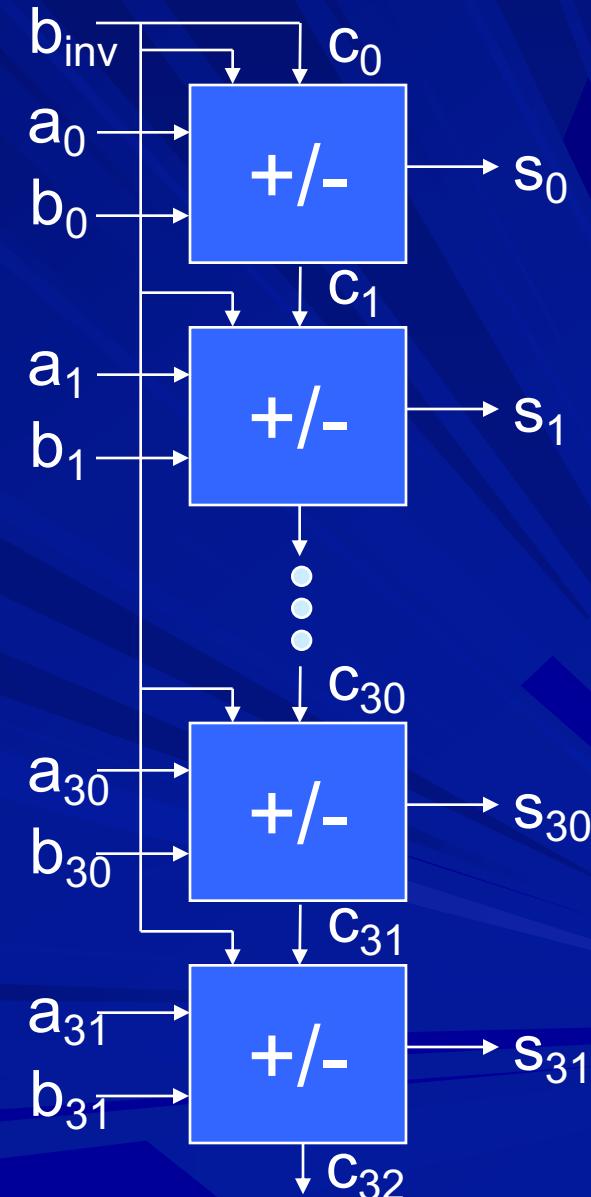
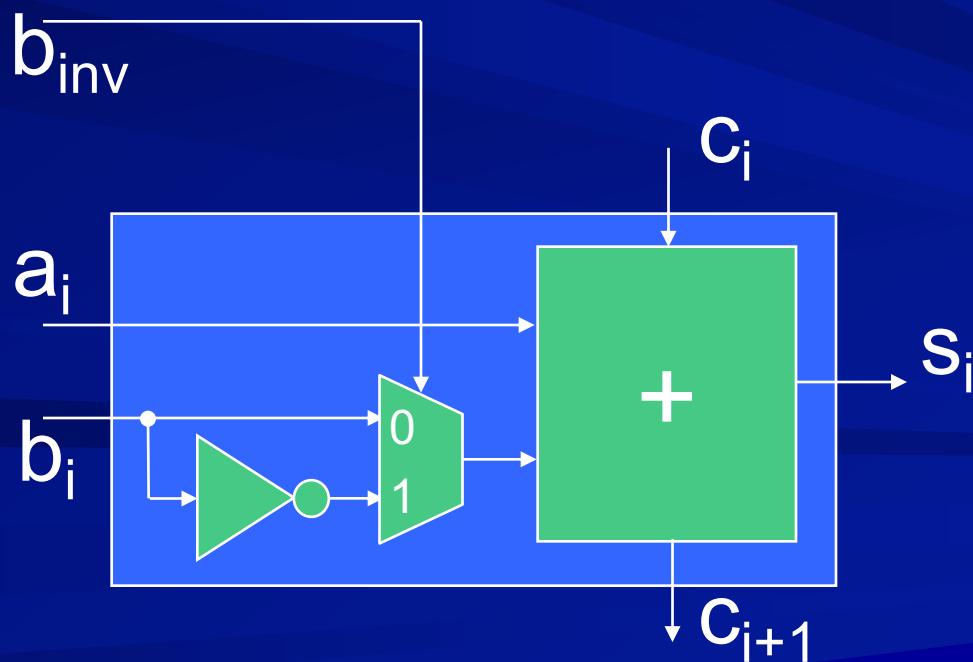
- Use the formula

$$X - Y = X + Y' + 1$$

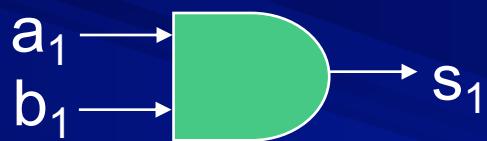
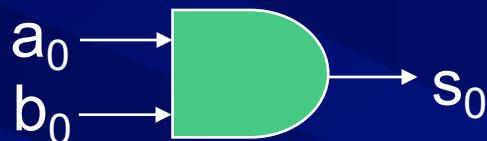


Combining addition and subtraction

- Use a multiplexer circuit



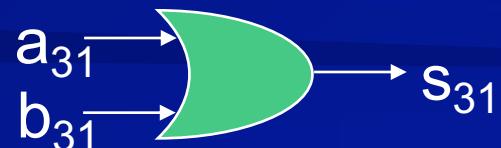
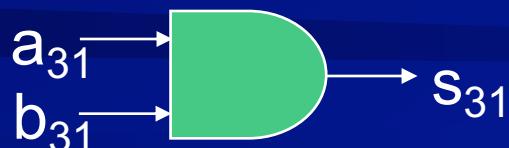
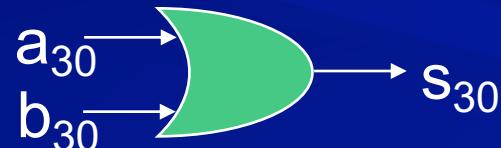
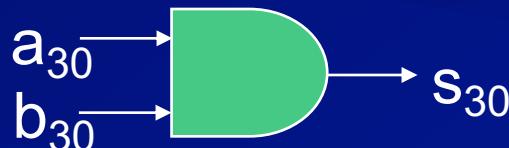
Circuit for and, or, xor



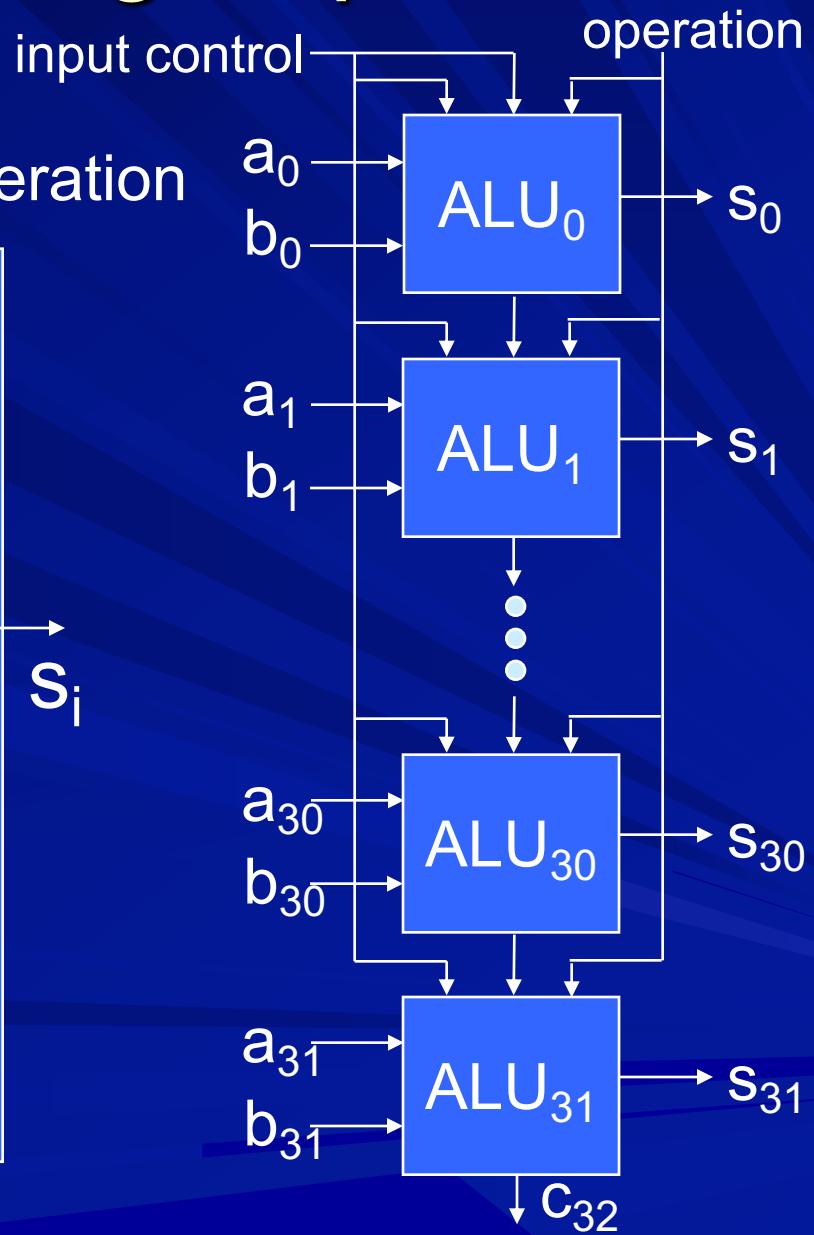
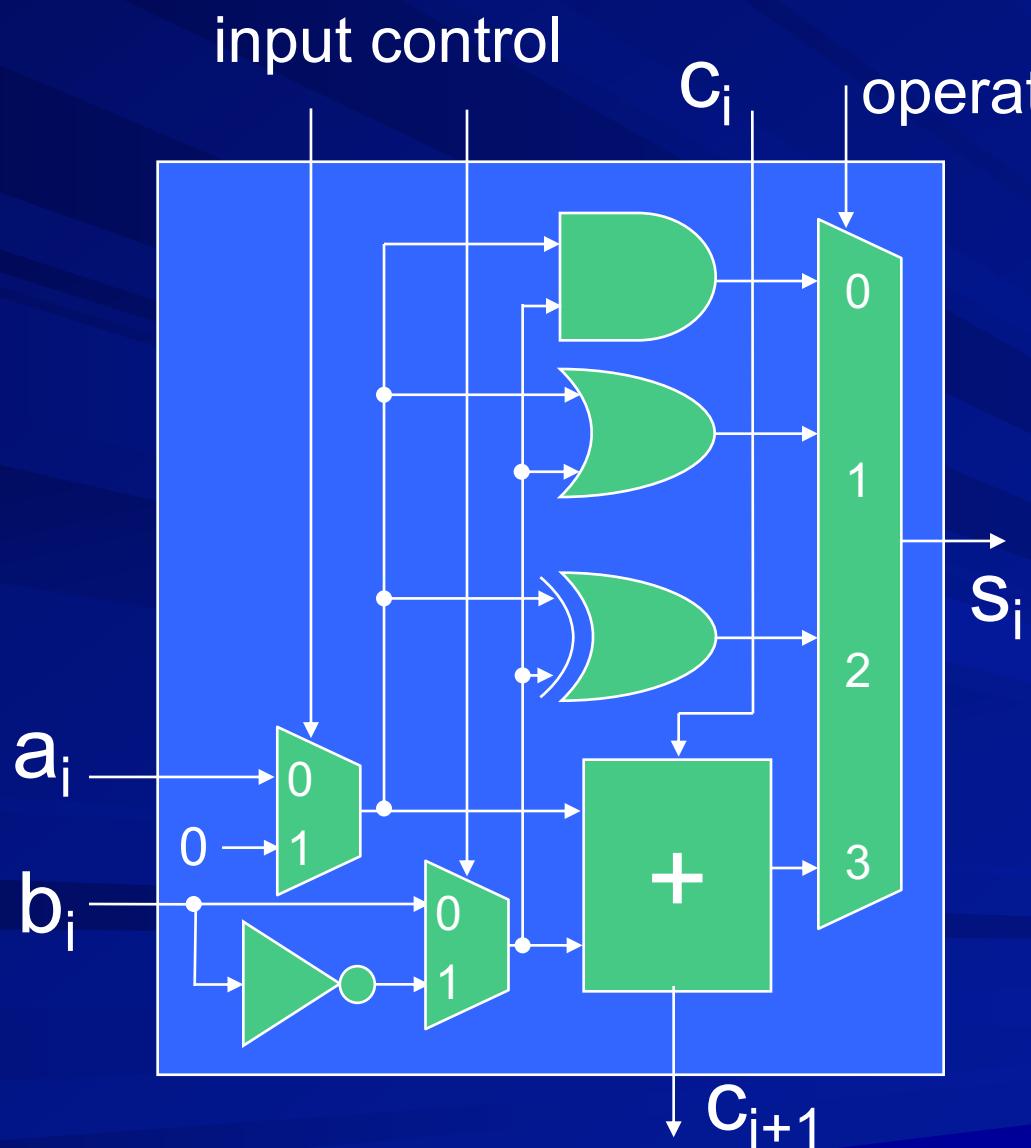
⋮

⋮

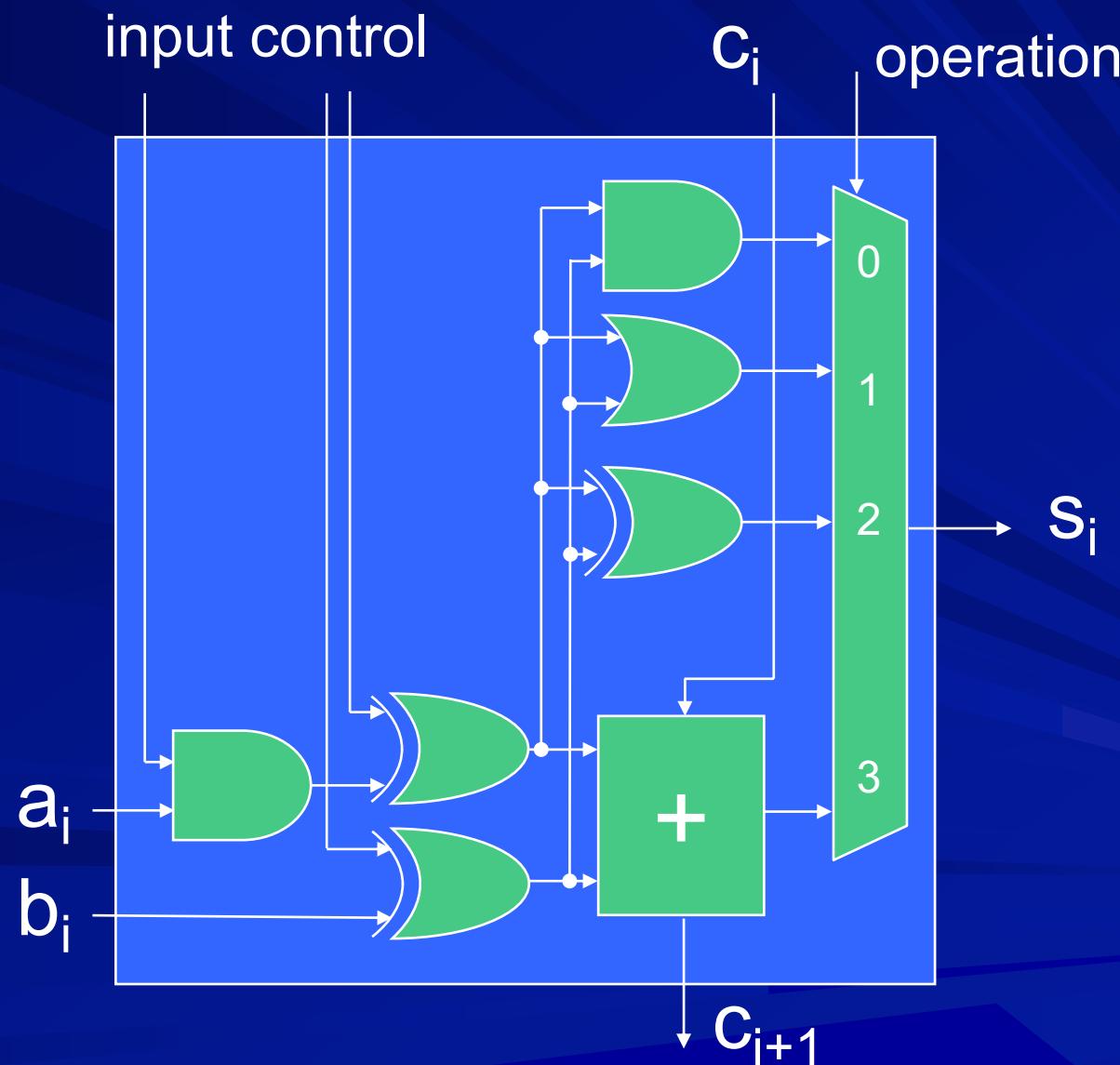
⋮



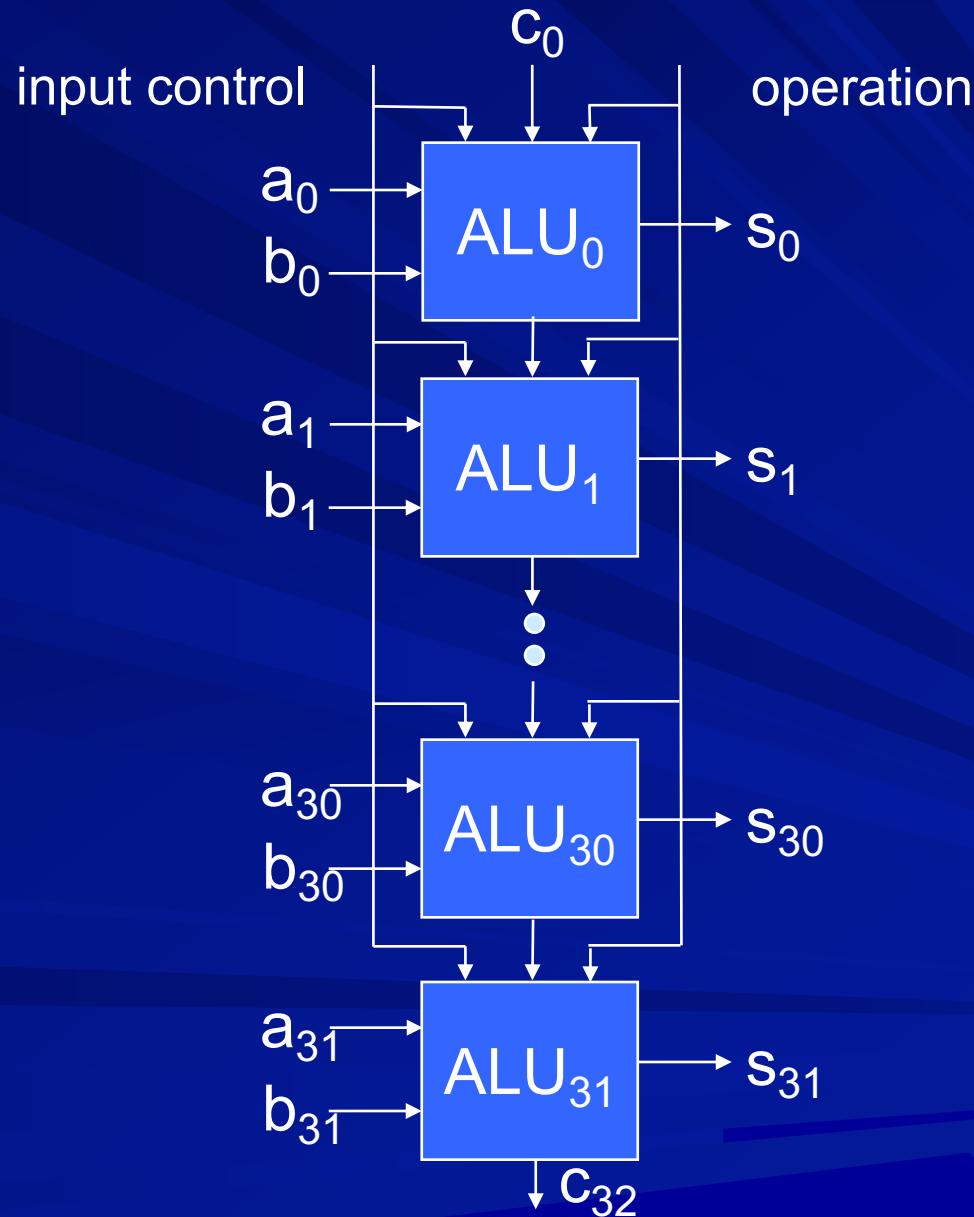
Combining arith & logic operations



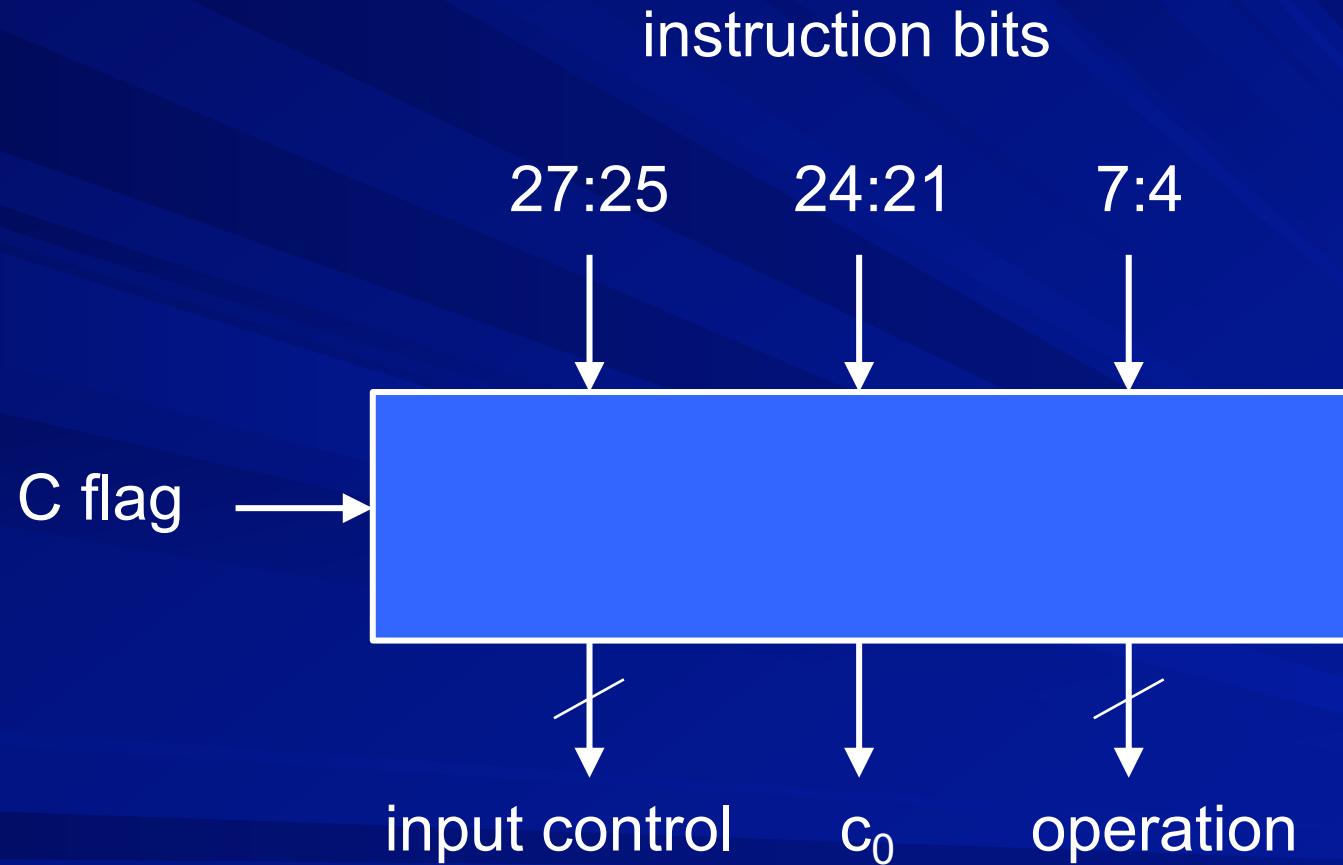
ALU Cell with reverse subtract



ALU – 32 bits

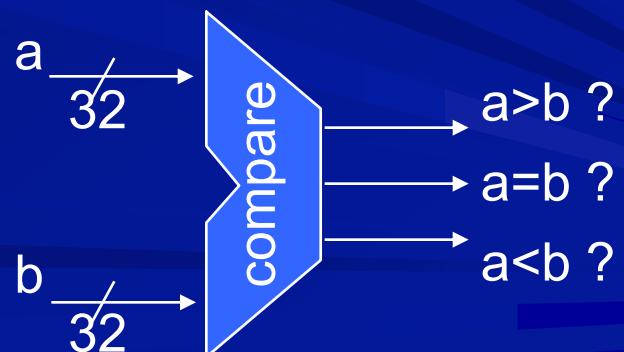
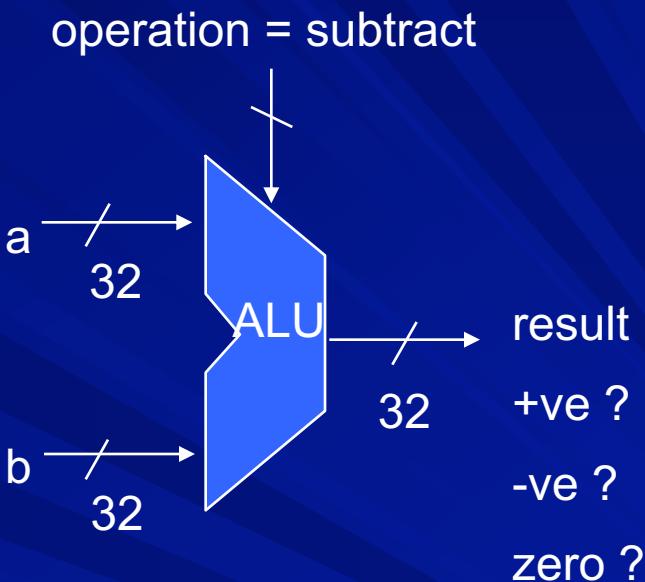


ALU control inputs

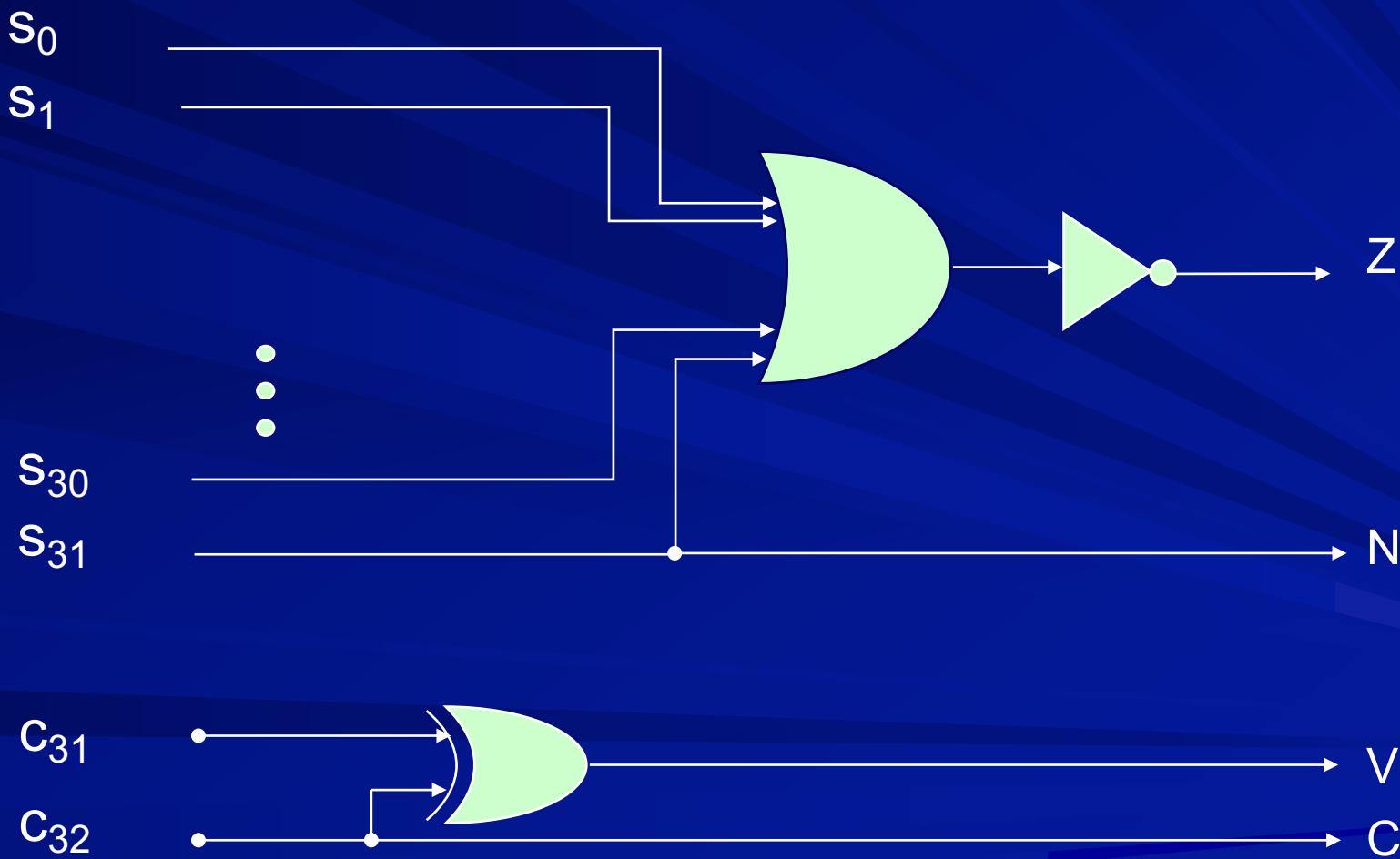


Comparing two integers

- Subtract and check the result
- Compare directly



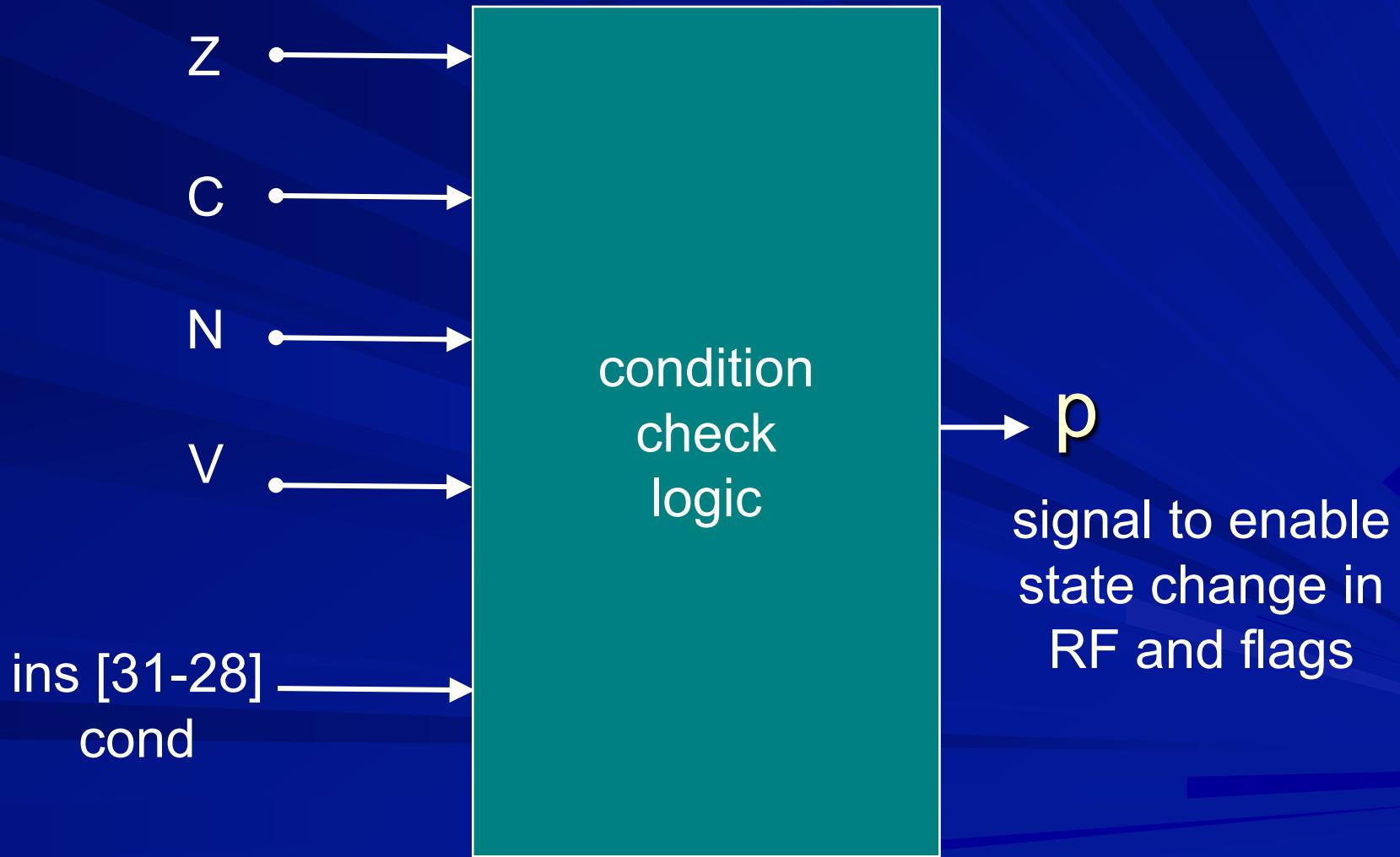
Setting flags



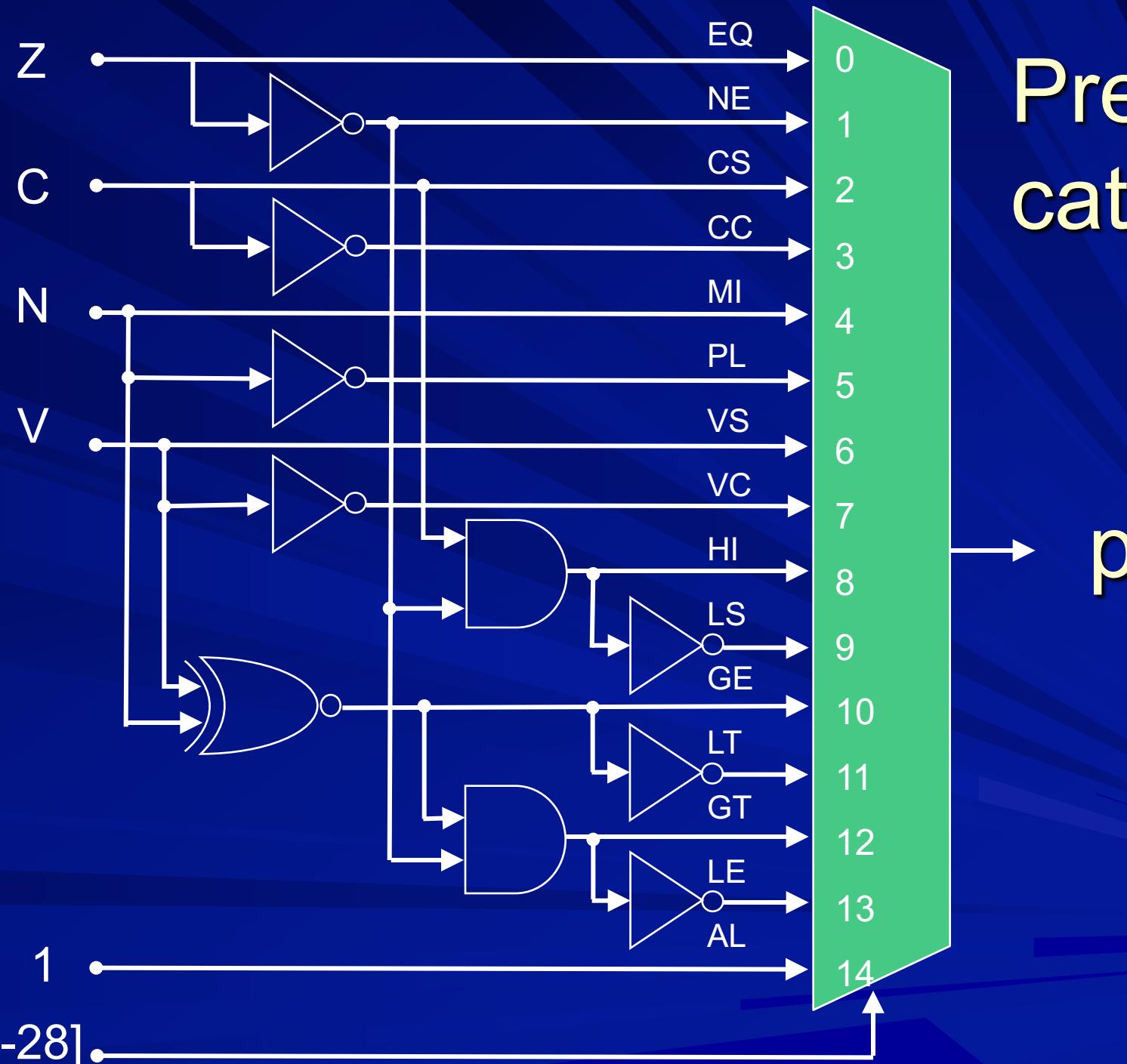
Predication

cond	ins [31-28]	Flags
EQ	0 0 0 0	Z set
NE	0 0 0 1	Z clear
CS HS	0 0 1 0	C set higher or same
CC LO	0 0 1 1	C clear lower
MI	0 1 0 0	N set
PL	0 1 0 1	N clear
VS	0 1 1 0	V set
VC	0 1 1 1	V clear
HI	1 0 0 0	C set and Z clear
LS	1 0 0 1	C clear or Z set
GE	1 0 1 0	N = V
LT	1 0 1 1	N ≠ V
GT	1 1 0 0	Z clear and (N = V)
LE	1 1 0 1	Z set or (N ≠ V)
AL	1 1 1 0	ignored

Predication



Predi- cation



Instructions doable by earlier ALU

- add, sub
- and, eor, orr, bic
- cmp, cmn
- tst, teq
- mov, mvn

Additional DP instructions to be done

- adc, sbc
- rsb, rsc

For reverse subtraction, we can either include multiplexers to switch between Op1 and Op2, or include option to invert Op1.

The initial carry can be constant ‘0’, constant ‘1’ or C Flag.

Op2 variants

- Rm

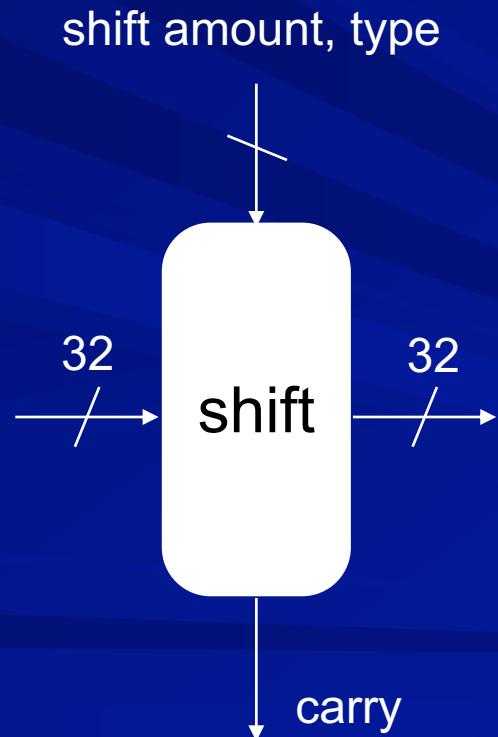
⇒ Rm, <Shift type>, Rs

⇒ Rm, <Shift type>, #constant

- #constant

⇒ #constant, ROR, constant

Shifting and rotation



Shift operations

shift left logical 3 bits

a_{31}	a_{30}	...	a_1	a_0
----------	----------	-----	-------	-------



a_{28}	a_{27}	...	a_1	a_0	0	0	0
----------	----------	-----	-------	-------	---	---	---

shift right logical 3 bits

a_{31}	a_{30}	...	a_1	a_0
----------	----------	-----	-------	-------



0	0	0	a_{31}	a_{30}	...	a_4	a_3
---	---	---	----------	----------	-----	-------	-------

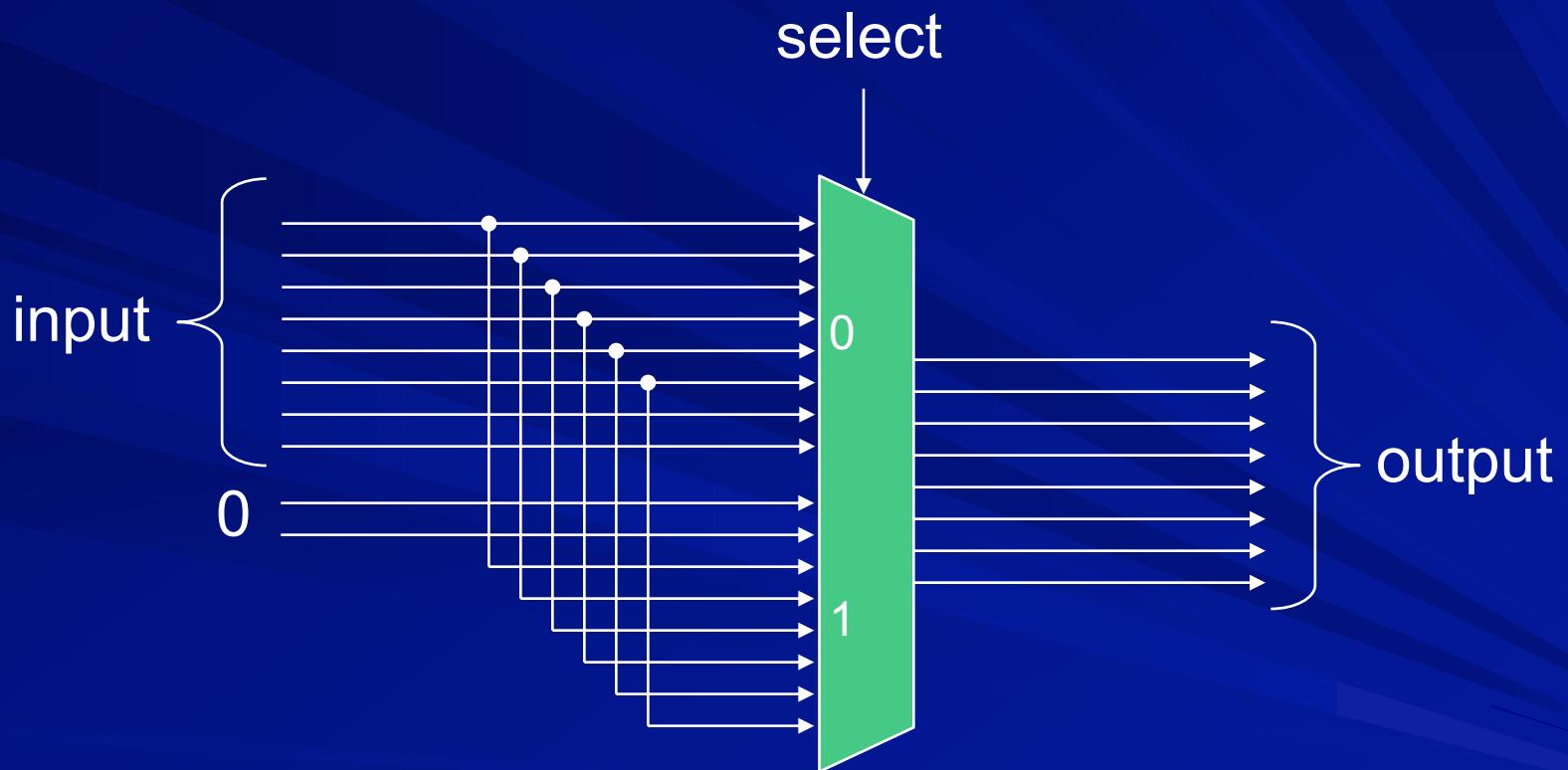
shift right arithmetic 2 bits

a_{31}	a_{30}	...	a_1	a_0
----------	----------	-----	-------	-------

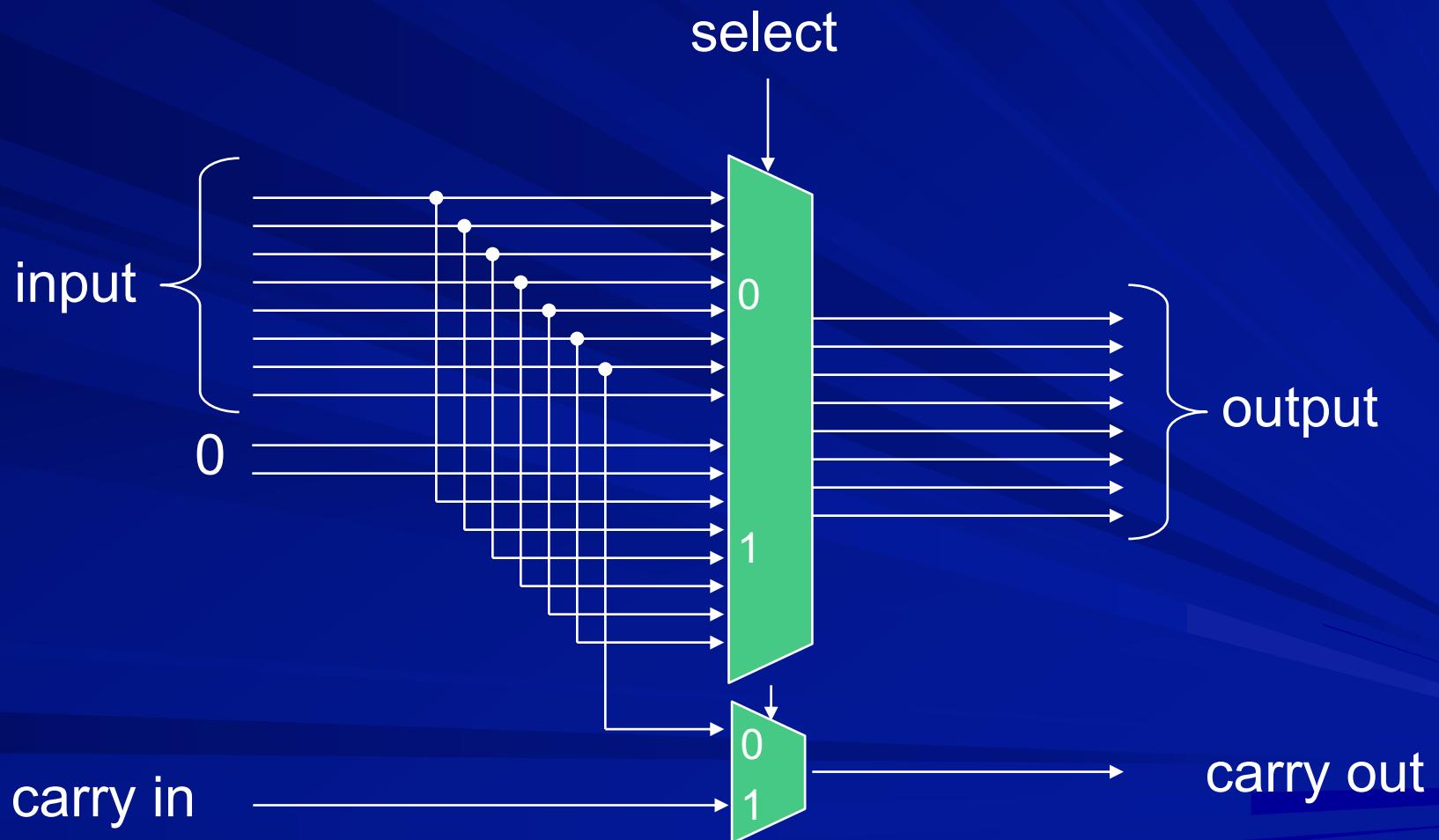


a_{31}	a_{31}	a_{31}	a_{30}	...	a_3	a_2
----------	----------	----------	----------	-----	-------	-------

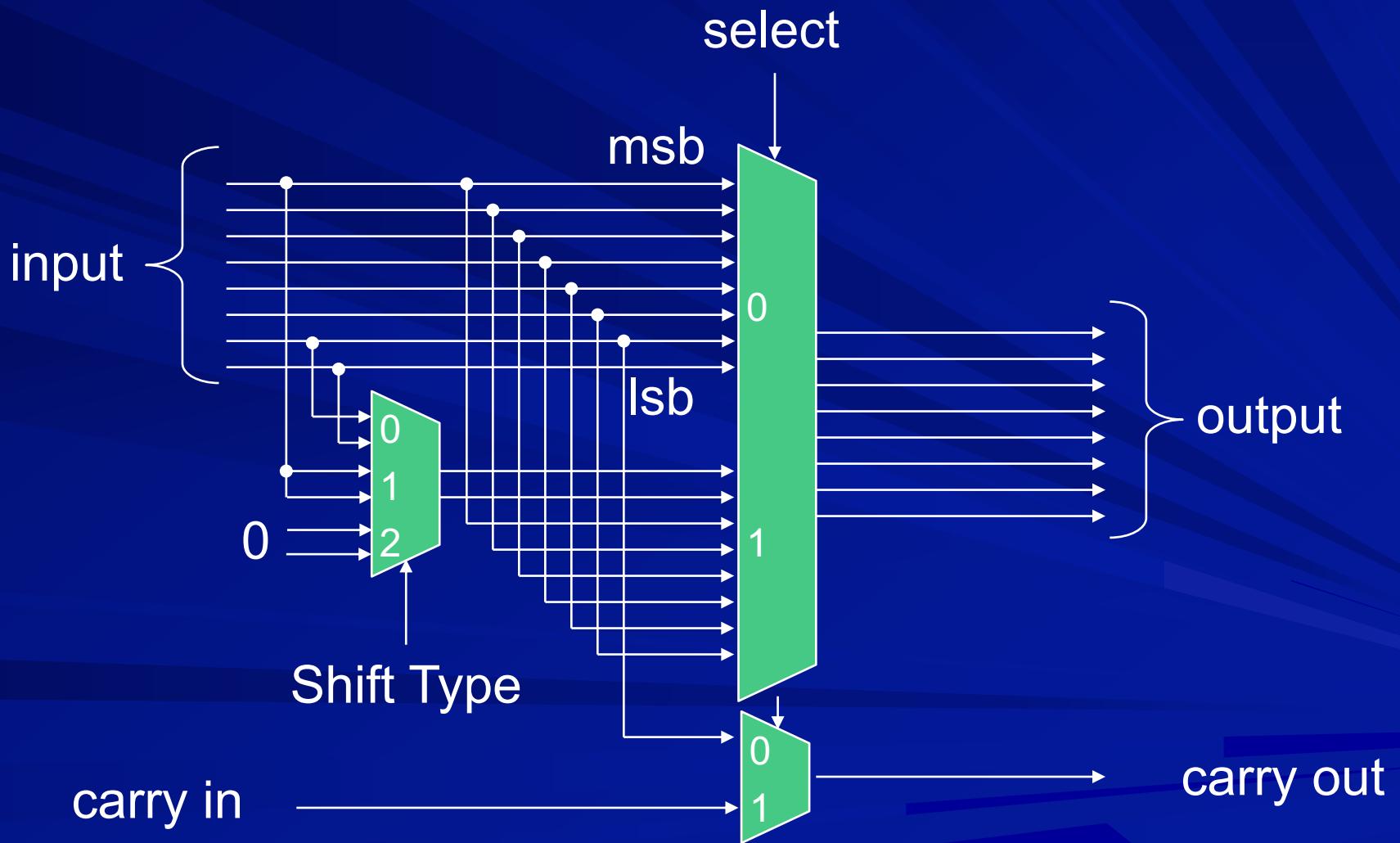
Circuit for shifting by 2 bits



Circuit for shifting by 2 bits



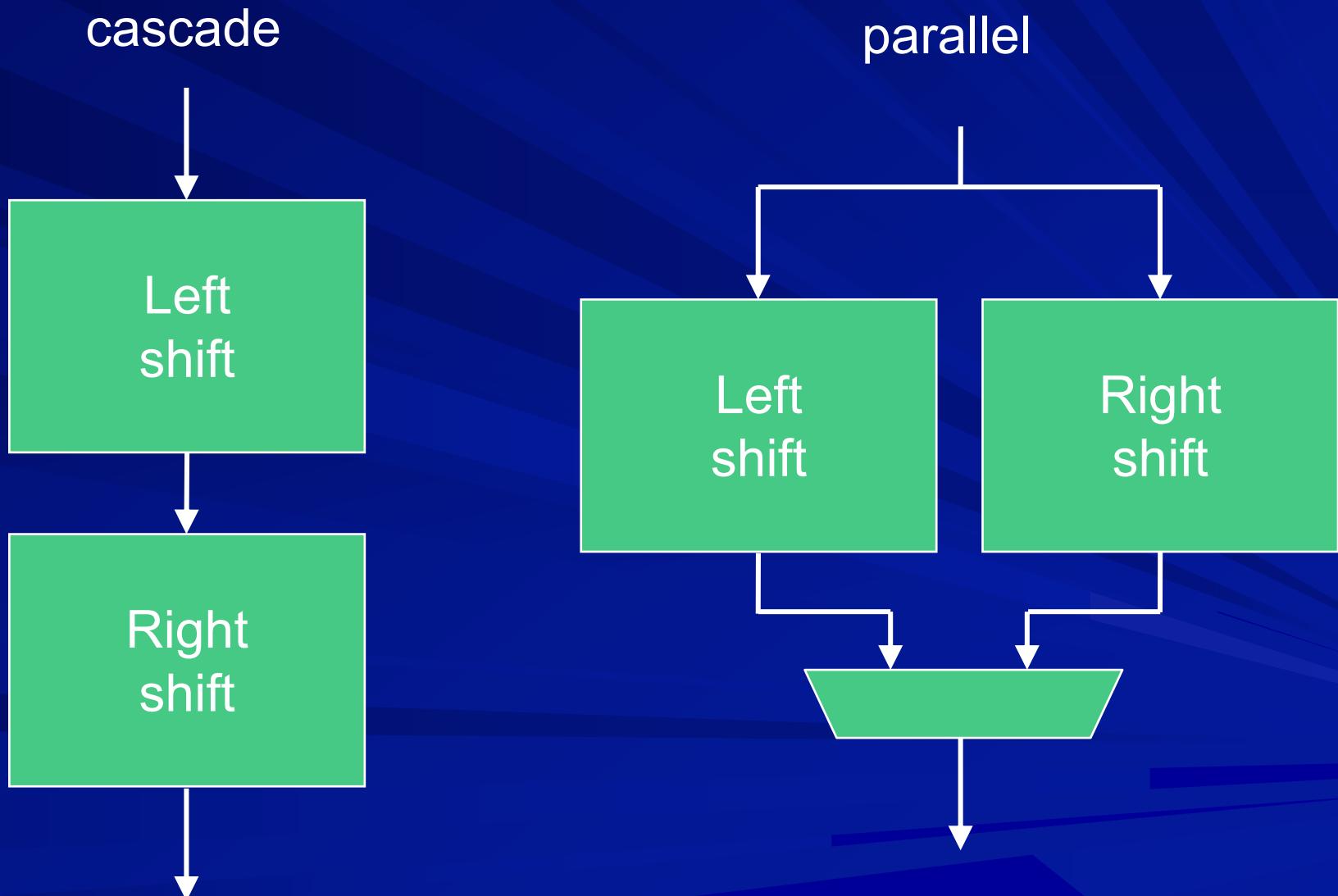
Combining ROR, ASR, LSR



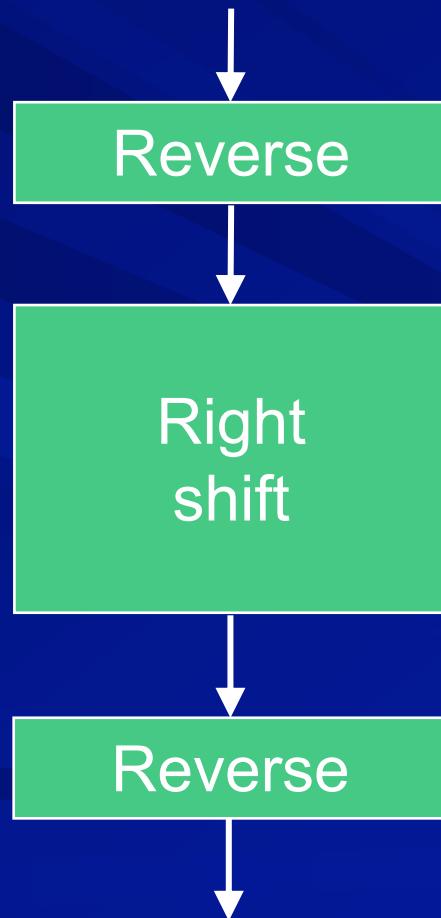
Circuit for shifting by 0 to 31 bits



Combining Left and Right Shift



Combining Left and Right Shift



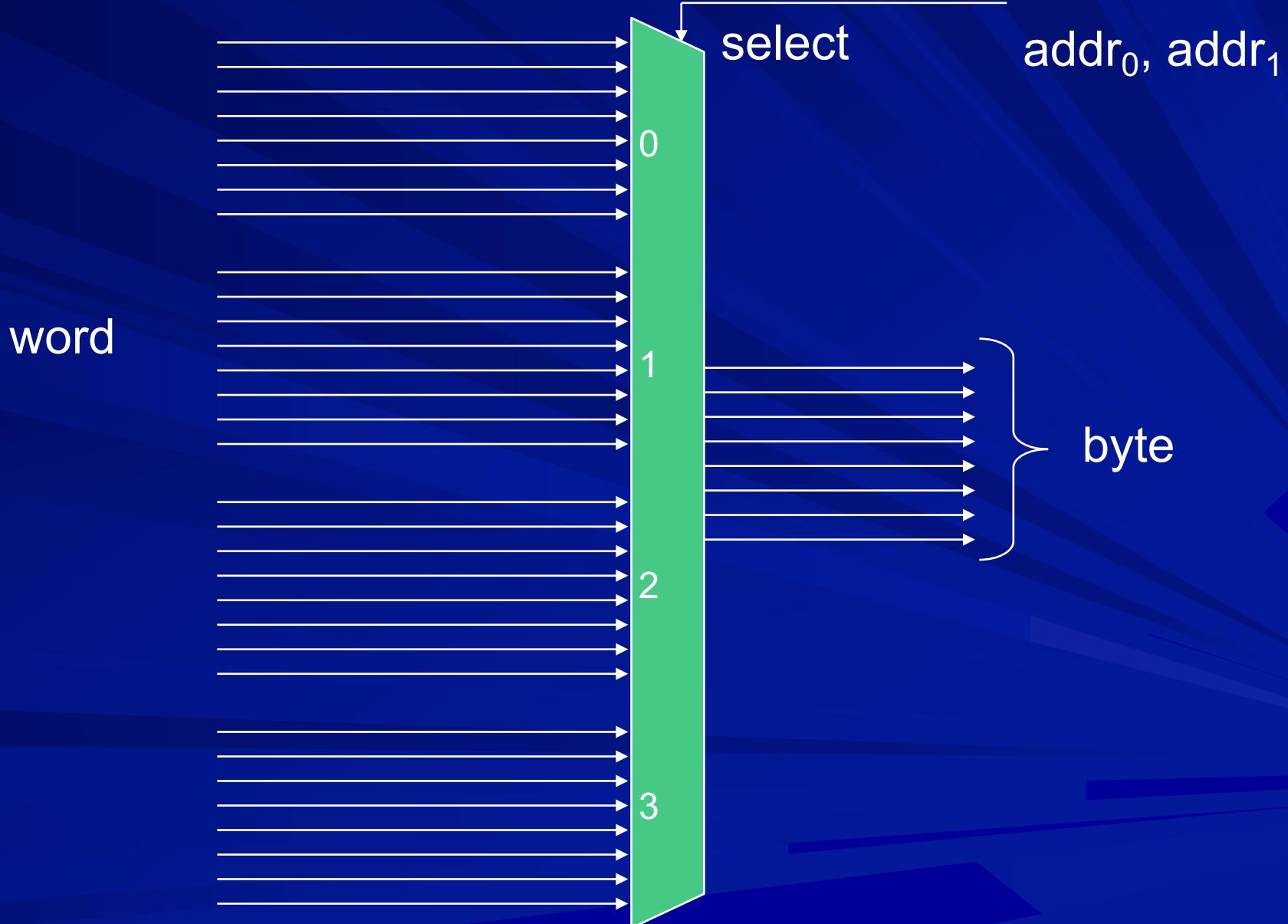
Pre/post increment/decrement

- Same operation for ALU whether pre or post
- Only the order / timings of address computation and memory access may differ

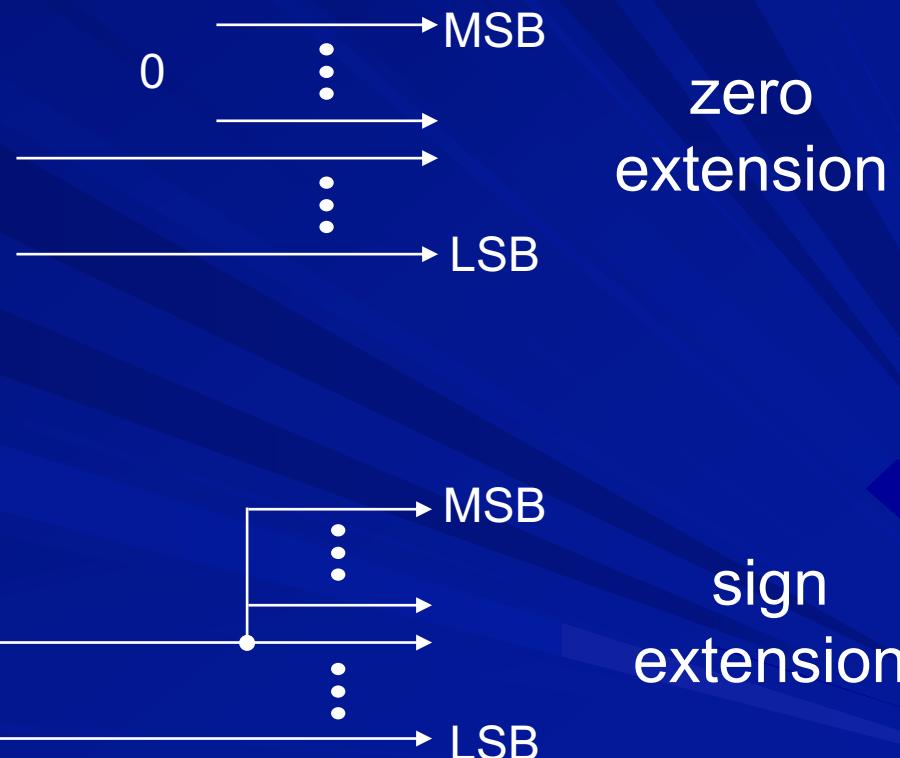
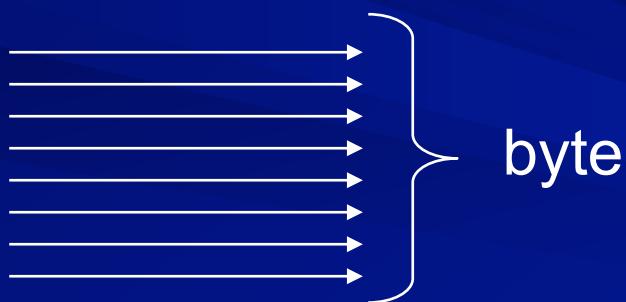
Auto increment/decrement

- Same operation for ALU whether auto or not
- Only the updation of base register may or may not take place

Word / half-word / byte transfer

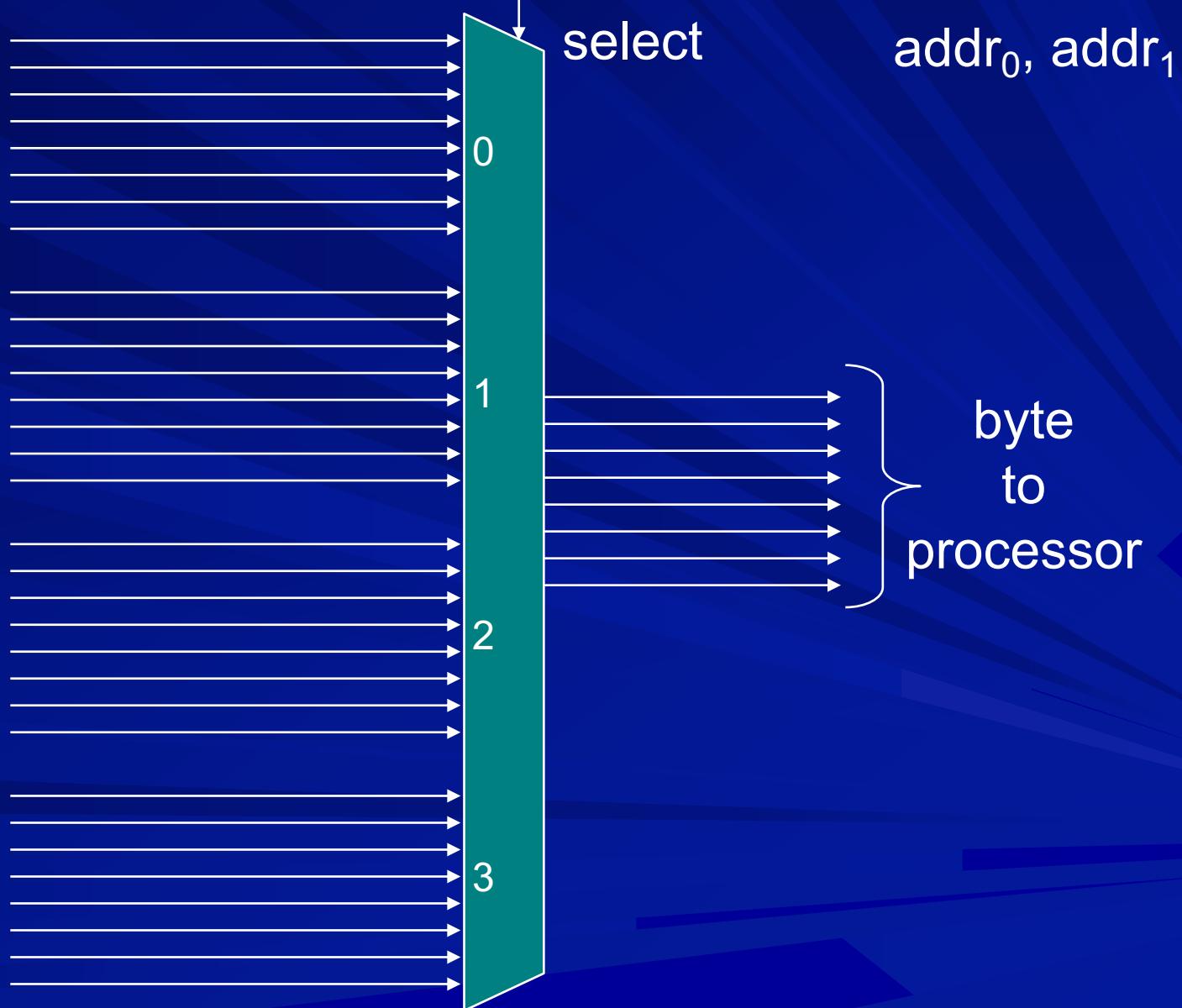


Word / half-word / byte transfer



Selecting byte for ldrb / ldrsb

word
from
memory

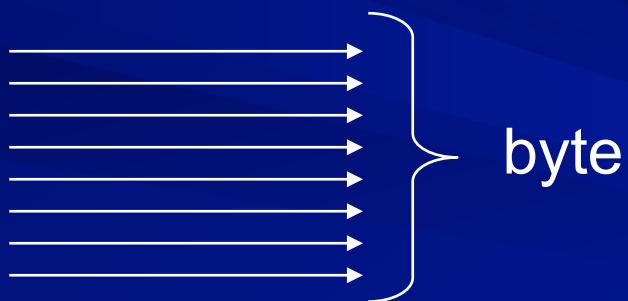


Selecting half word for ldrh / ldrsh

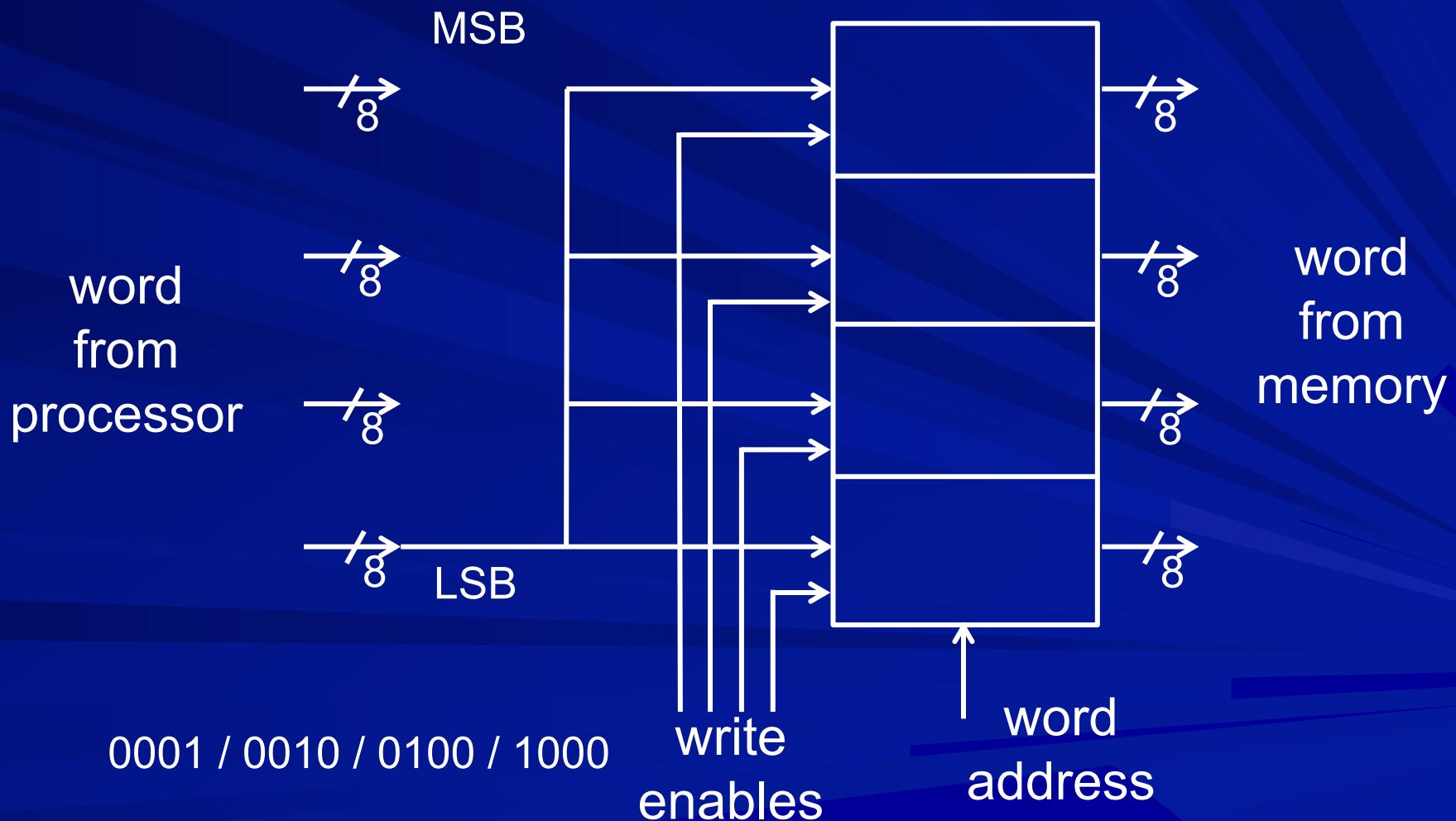
word
from
memory



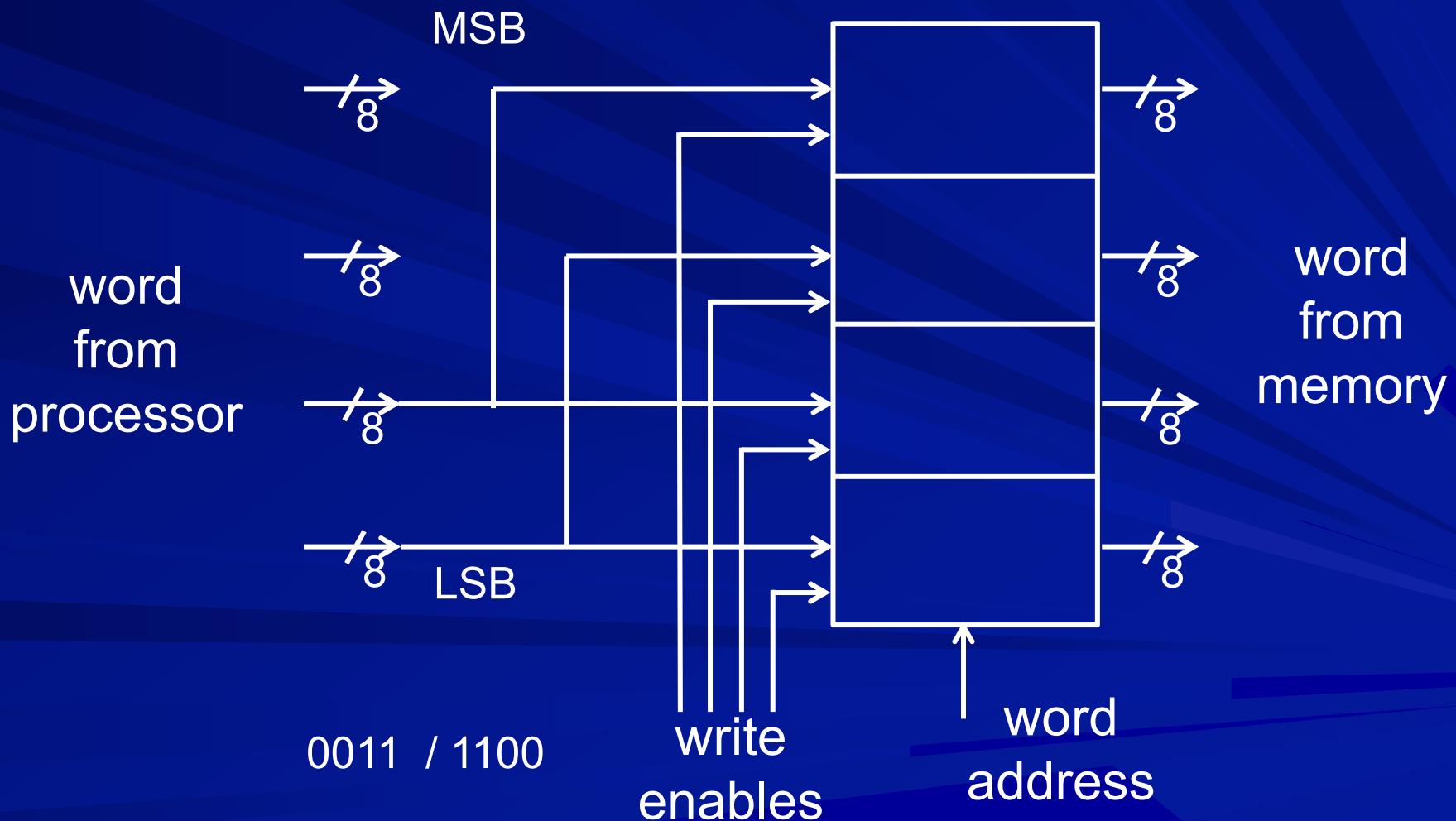
Extending from byte to word



Writing a byte in memory



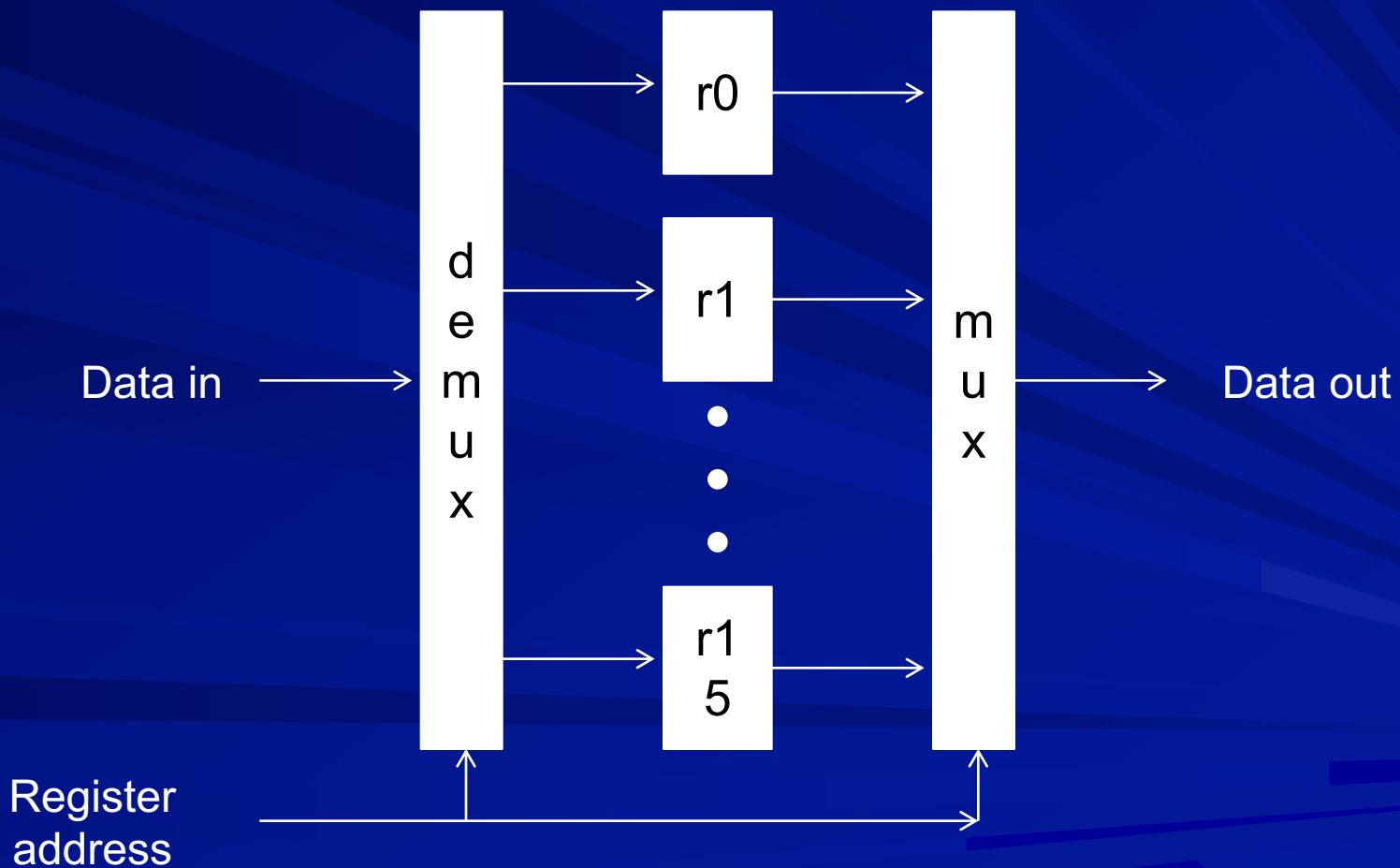
Writing a half word in memory



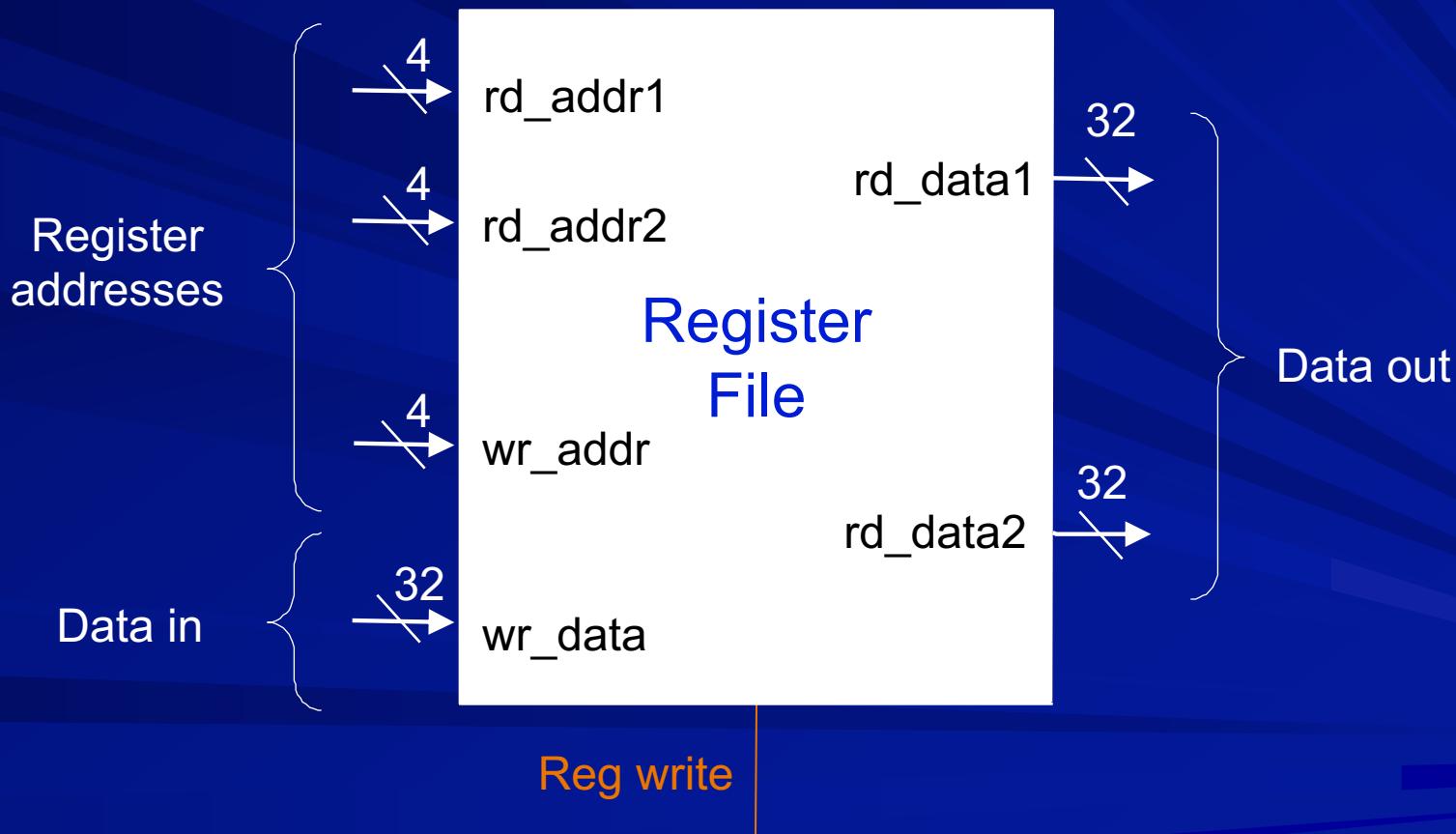
Register file



Register file



Register file (2R + 1W ports)

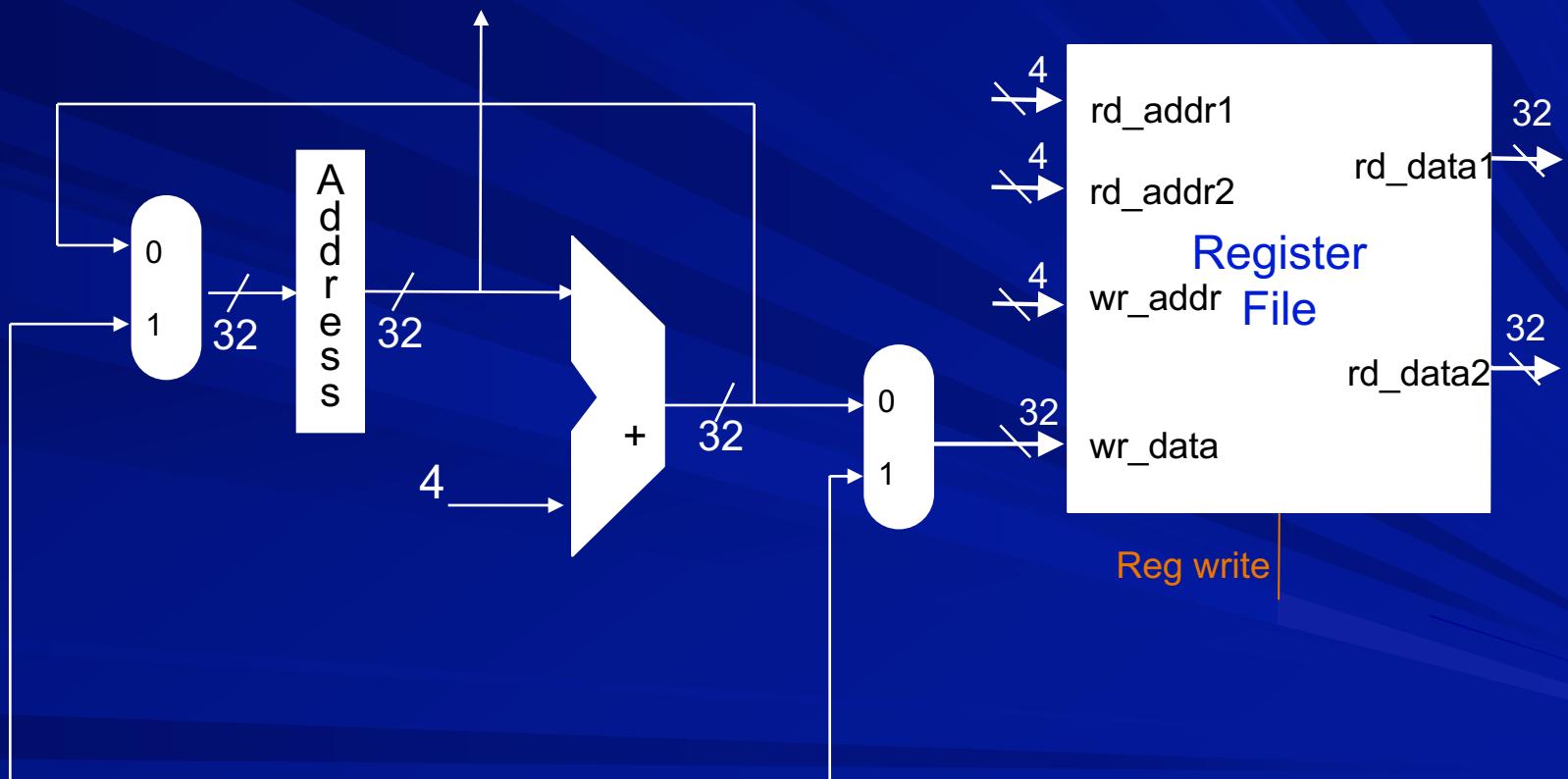


RF ports for ARM

- add r1, r2, r3, LSL r4
requires 3 reads (r2, r3, r4) and 1 write (r1)
- ldr r1, [r2, r3]! and ldr r1, [r2], r3
require 2 reads (r2, r3) and 2 writes (r1, r2)
- str r1, [r2, r3]! and str r1, [r2], r3
require 3 reads (r1, r2, r3) and 1 write (r2)
- Additionally, 1 read + 1 write for r15 (PC) –
all instructions read and write PC

Accessing PC

to instruction memory



from ALU or data memory

Thanks

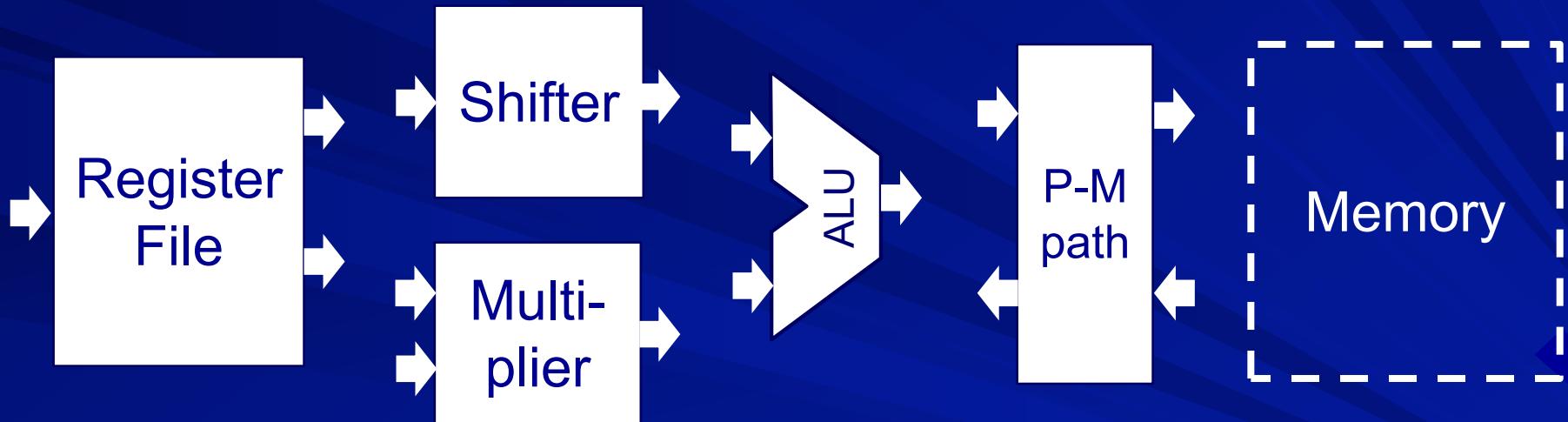
COL216

Computer Architecture

Designing a processor -
A simple approach
3rd February, 2022

Datapath major blocks

Some additional blocks are required to glue these together



Later we will see how instruction fetching, instruction decode, operand access and state updation are done

Multiply instructions

- MUL
- MLA

Multiply

Multiply Accumulate

32 bit result

64 bit result

- SMULL Signed Multiply Long
- SMLAL Signed Multiply Accumulate Long
- UMULL Unsigned Multiply Long
- UMLAL Unsigned Multiply Accumulate Long

4×4 multiplication

$$\begin{array}{r} a_3 \ a_2 \ a_1 \ a_0 \quad \times b_0 \\ a_3 \ a_2 \ a_1 \ a_0 \quad \times b_1 \\ a_3 \ a_2 \ a_1 \ a_0 \quad \times b_2 \\ a_3 \ a_2 \ a_1 \ a_0 \quad \times b_3 \end{array}$$

4x4 unsigned

4x4 signed

0	0	0	a_3	a_2	a_1	a_0	$\times b_0$
0	0	a_3	a_2	a_1	a_0		$\times b_1$
0	a_3	a_2	a_1	a_0		$\times b_2$	
a_3	a_2	a_1	a_0		$\times b_3$		

a_3	a_3	a_3	a_3	a_2	a_1	a_0	$\times b_0$
a_3	a_3	a_3	a_2	a_1	a_0		$\times b_1$
a_3	a_3	a_2	a_1	a_0		$\times b_2$	
a_3	a_2	a_1	a_0			$\times -b_3$	

Multiplying a number by -1

$$X = x_{n-1} \ x_{n-2} \ \dots \ \dots \ \dots \ x_1 \ x_0$$

Let x_{k-1} be the rightmost 1 $(1 \leq k \leq n)$

Then

$$X = x_{n-1} \ x_{n-2} \ \dots \ \dots \ \dots \ x_k \ 1 \ 0 \dots 0 \ 0$$

$$\overline{X} = \overline{x}_{n-1} \ \overline{x}_{n-2} \ \dots \ \dots \ \dots \ \overline{x}_k \ 0 \ 1 \dots 1 \ 1$$

$$-X = \underbrace{\overline{x}_{n-1} \ \overline{x}_{n-2} \ \dots \ \dots \ \dots \ \overline{x}_k}_{\text{left } n-k \text{ bits complemented}} \underbrace{1 \ 0 \dots 0 \ 0}_{\text{right } k \text{ bits unchanged}}$$

left $n - k$ bits
complemented

right k bits
unchanged

4x4 unsigned

4x4 signed

0 0 0	$a_3\ a_2\ a_1\ a_0$	$\times b_0$
0 0	$a_3\ a_2\ a_1\ a_0$	$\times b_1$
0	$a_3\ a_2\ a_1\ a_0$	$\times b_2$
$a_3\ a_2\ a_1\ a_0$		$\times b_3$

$a_3\ a_3\ a_3\ a_3\ a_2\ a_1\ a_0$	$\times b_0$
$a_3\ a_3\ a_3\ a_2\ a_1\ a_0$	$\times b_1$
$a_3\ a_3\ a_2\ a_1\ a_0$	$\times b_2$
$\bar{a}_3\ \bar{a}_2\ a_1\ a_0$	$\times b_3$

4x4 unsigned

4x4 signed

0	0	0	a ₃	a ₂	a ₁	a ₀	x b ₀
0	0	a ₃	a ₂	a ₁	a ₀		x b ₁
0	a ₃	a ₂	a ₁	a ₀			x b ₂
a ₃	a ₂	a ₁	a ₀				x b ₃

a ₃	a ₃	a ₃	a ₃	a ₂	a ₁	a ₀	x b ₀
a ₃	a ₃	a ₃	a ₃	a ₂	a ₁	a ₀	x b ₁
a ₃	a ₃	a ₂	a ₁	a ₀			x b ₂
\bar{a}_3	\bar{a}_2	\bar{a}_1	\bar{a}_0				x b ₃

Same for both

Multiplication in VHDL

Signed multiplication:

```
signal a_s, b_s : signed (31 downto 0);
signal p_s : signed (63 downto 0);
p_s <= a_s * b_s;
```

Unsigned multiplication:

```
signal a_u, b_u : unsigned (31 downto 0);
signal p_u : unsigned (63 downto 0);
p_u <= a_u * b_u;
```

Synthesizing multipliers

Signed multiplication:

```
signal a_s, b_s : signed (31 downto 0);  
signal p_s : signed (63 downto 0);  
p_s <= a_s * b_s;      -- uses 4 DSP48E1  
                        18x18 multiplier
```

Unsigned multiplication:

```
signal a_u, b_u : unsigned (31 downto 0);  
signal p_u : unsigned (63 downto 0);  
p_u <= a_u * b_u;      -- uses 4 DSP48E1  
                        18x18 multiplier
```

Combining results

```
signal op1, op2: std_logic_vector (31 downto 0);
signal result: std_logic_vector (63 downto 0);
signal p_s : signed (63 downto 0);
signal p_u : unsigned (63 downto 0);
p_s <= signed (op1) * signed (op2);
p_u <= unsigned (op1) * unsigned (op2);
result <= std_logic_vector(p_s) when instr = smull
      else std_logic_vector(p_u);
-- uses 7 DSP48E1, but fails during routing
```

Another way

- Use a signed multiplier to do unsigned multiplication
- 0-extend the operands
- Signed and unsigned interpretations of these are same now.

Another way

```
signal op1, op2: std_logic_vector (31 downto 0);
signal result: std_logic_vector (63 downto 0);
signal p_s: signed (65 downto 0);
signal x1, x2: std_logic;
x1 <= op1(31) when instr = smull else '0';
x2 <= op2(31) when instr = smull else '0';
p_s <= signed (x1 & op1) * signed (x2 & op2);
result <= std_logic_vector(p_s (63 downto 0));
-- uses 4 DSP48E1 !
```

Processor Design :

Going from

Instruction Set Architecture
(ISA)

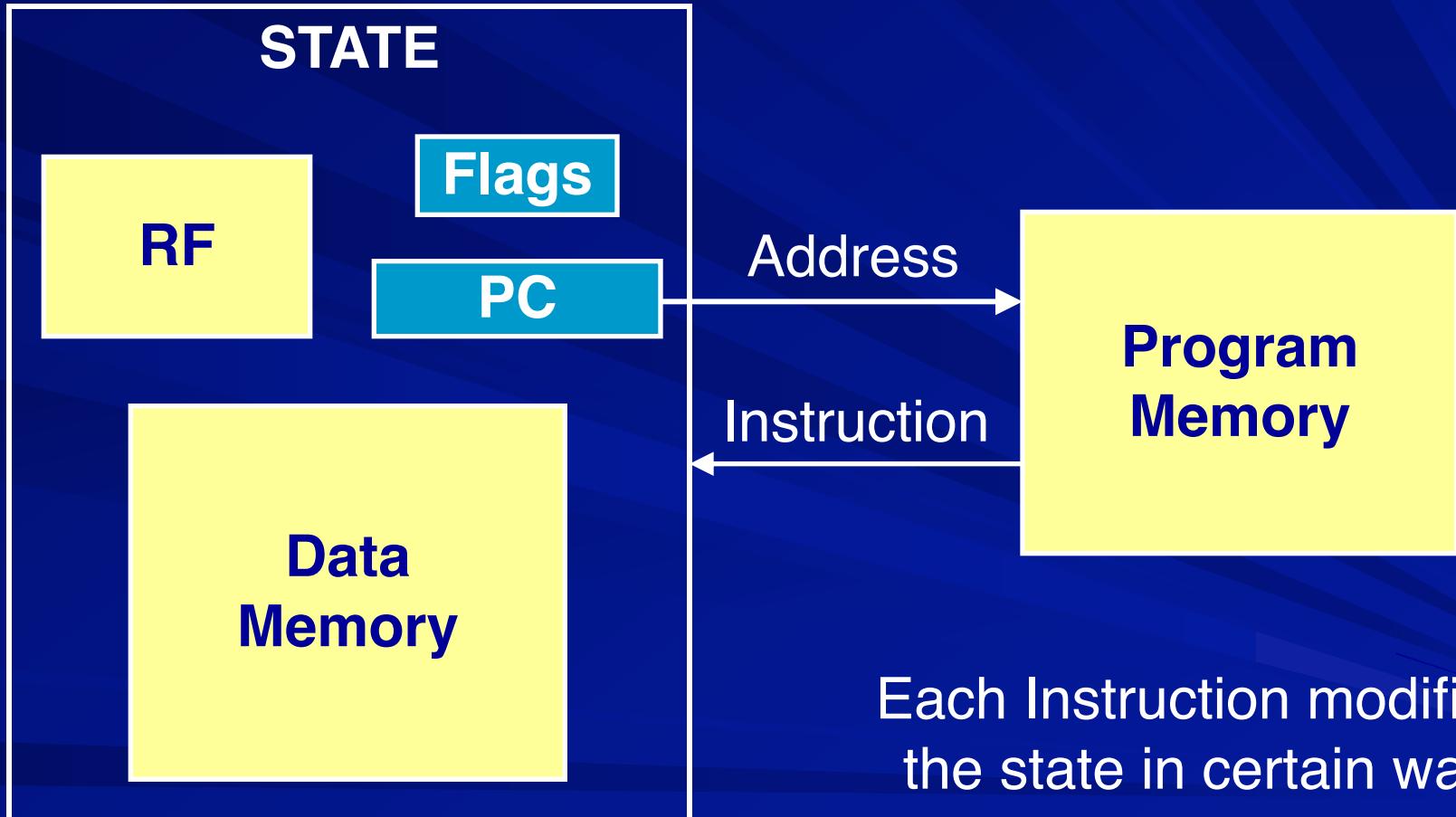
Lowest level
visible to a
programmer

to

Micro Architecture

High level
view of
hardware

The Abstract Machine



How instructions are executed?

- Look at the state (PC, Flags, RF, Memory)
- Based on current state, decide what should the next state be
- Update the state (PC, Flags, RF, Memory)

More details

- 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
- Update PC, registers, Flags, Memory

Some basic questions

- Hardware resources:
 - What building blocks are to be used to execute the instructions?
- Timings:
 - What is the overall approach to be followed for timing the instructions?
- The two are interlinked
- The answer depends upon the design goals

Instruction timings

- Timings within individual instructions
- Timings across instructions

Timings within

- Instruction execution involves many actions
- These actions are timed by clock cycles
- Number of cycles per instruction
 - Each instruction is done in a single cycle
 - Instructions may take multiple cycles
 - same number of cycles for all instructions
 - some may take fewer cycles some may take more

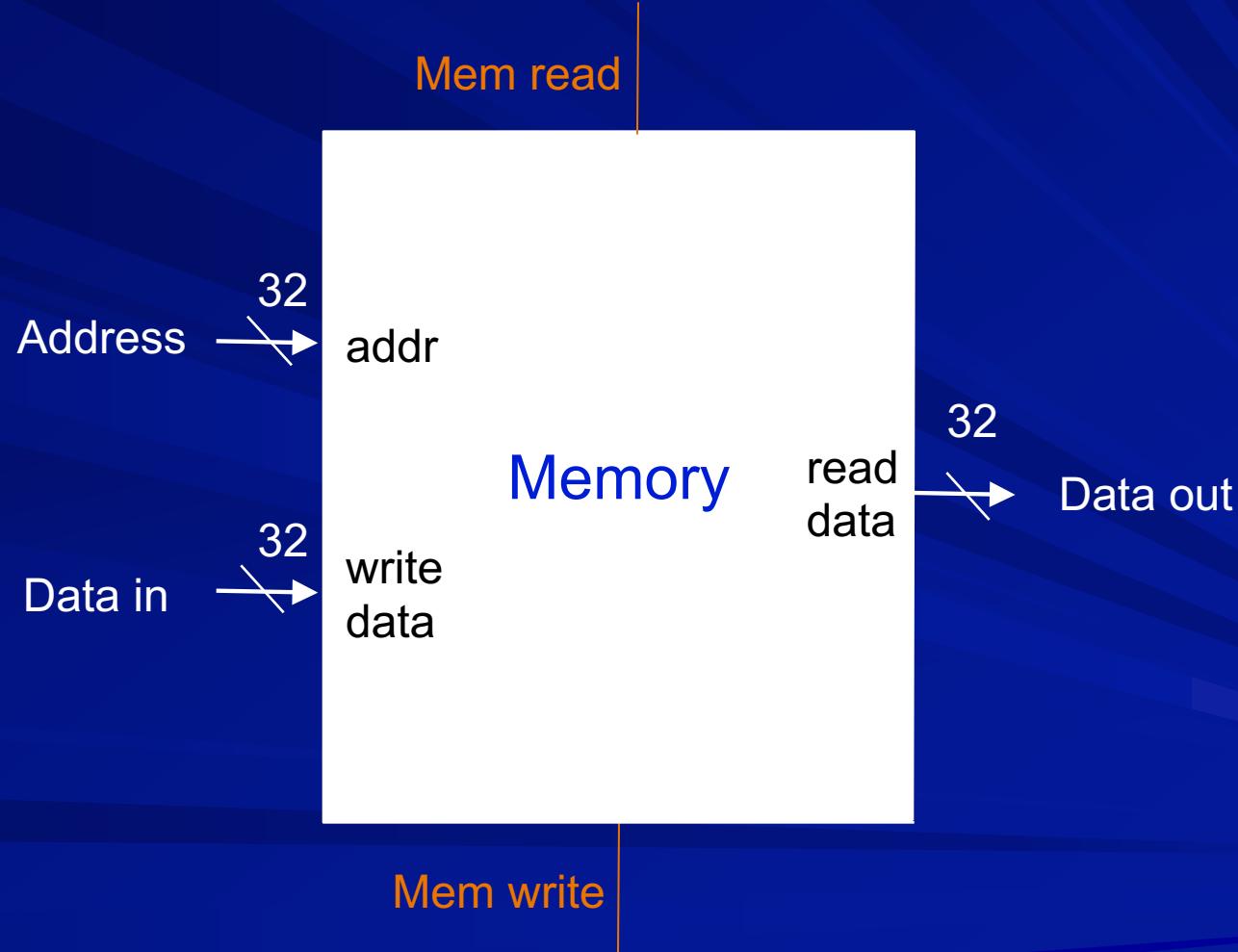
Timings across

- When one instruction finishes, only then another instruction starts or execution of instructions can overlap
- Only one instruction starts at a time or multiple instructions can start concurrently
- Instructions start in strict program order or the order can be changed

Choices for first design

- Simplest design, not necessarily best in terms of speed, cost or power consumption
- Instruction subset: {add, sub, cmp, mov, ldr, str, b, beq, bne}
- Single cycle for every instruction
- No instruction overlap, no concurrency
- Execution in strict program order

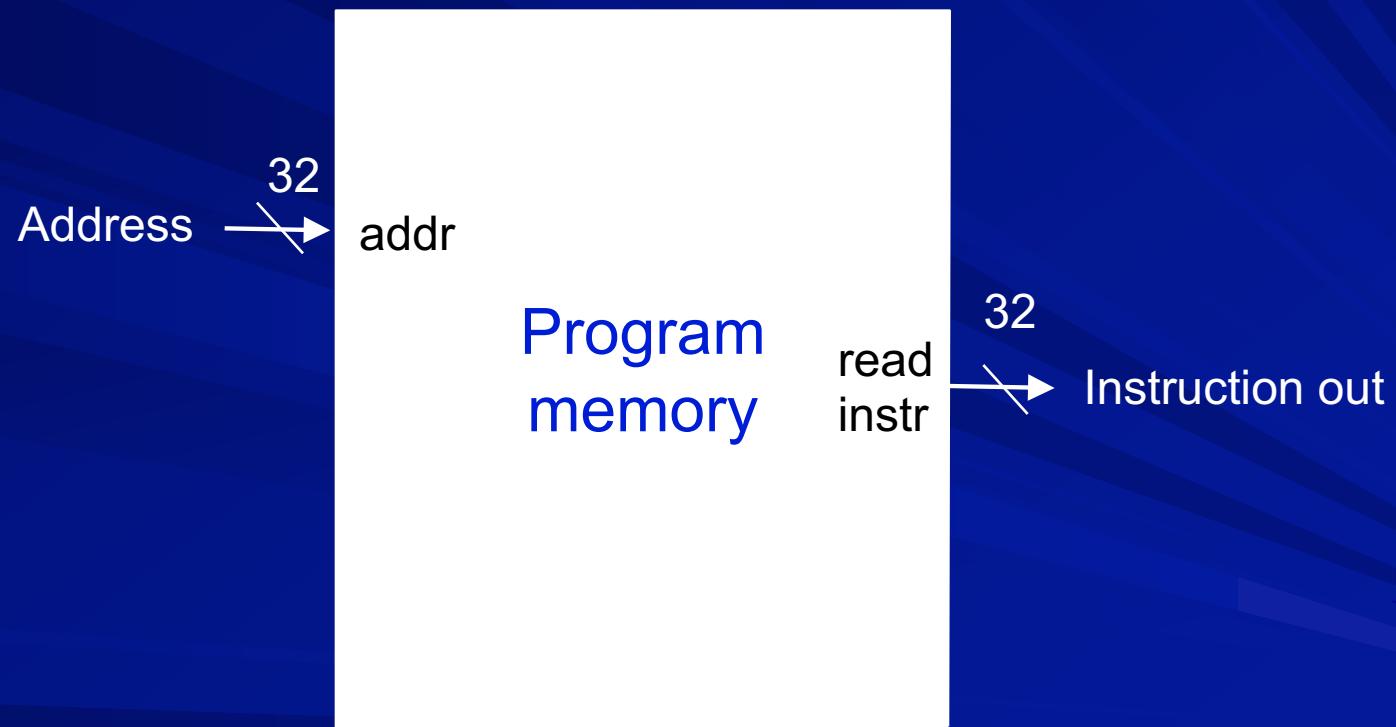
Memory



Program and data memory

- Separation of program and data memory
 - allows
 - accessing instruction and data within same cycle

Program memory

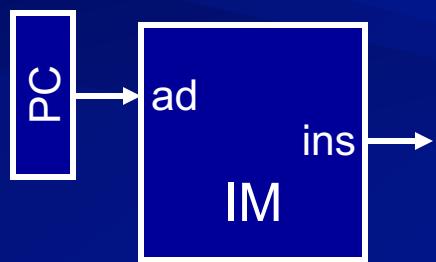


Building datapath

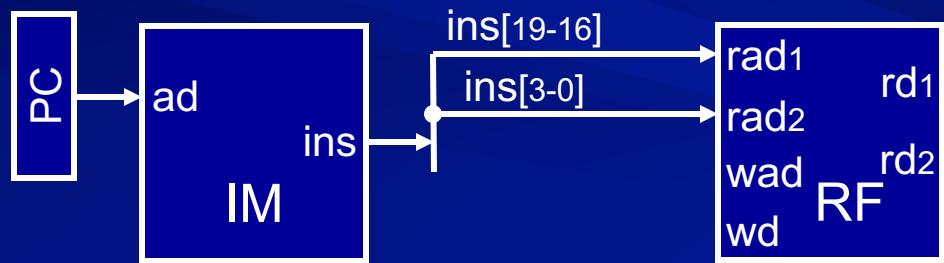
Actions for DP instructions

- fetch instruction
- access the register file
- pass operands to ALU
- pass result to register file
- increment PC

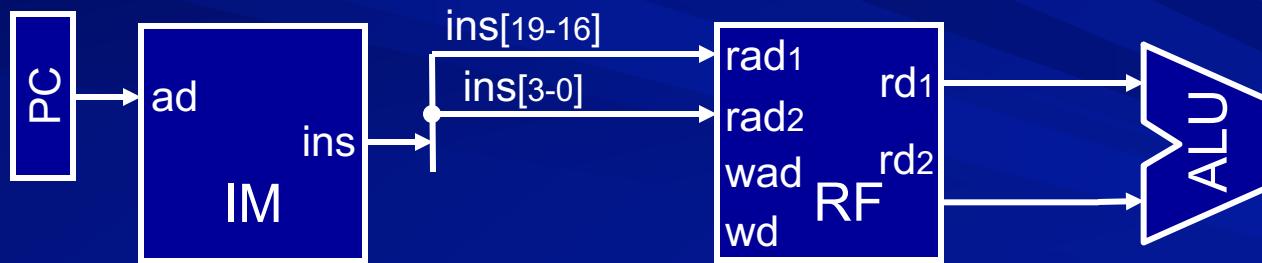
Fetching instruction



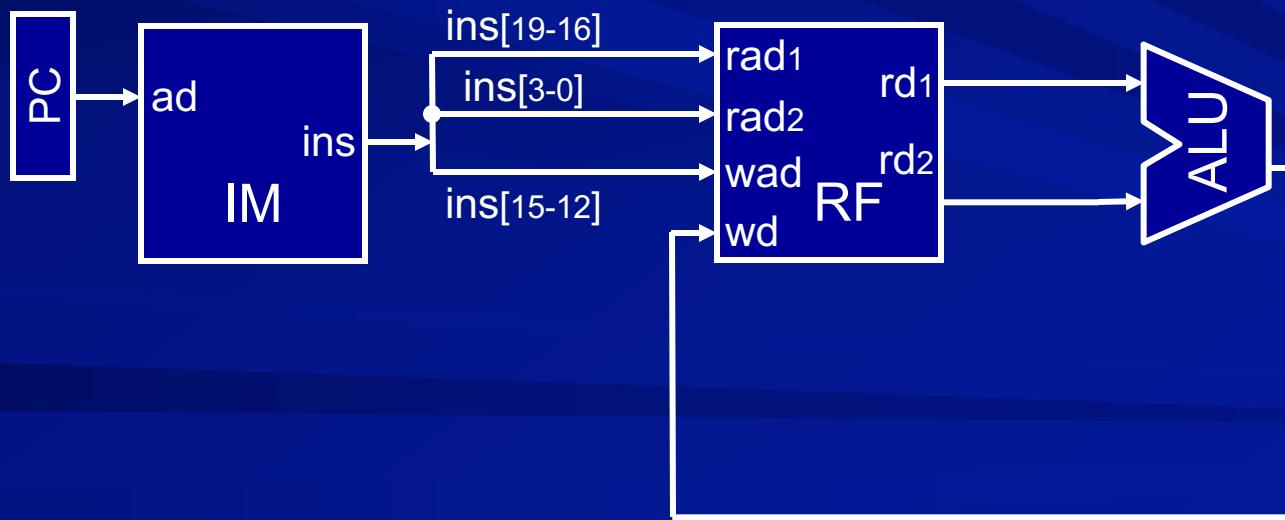
Accessing RF



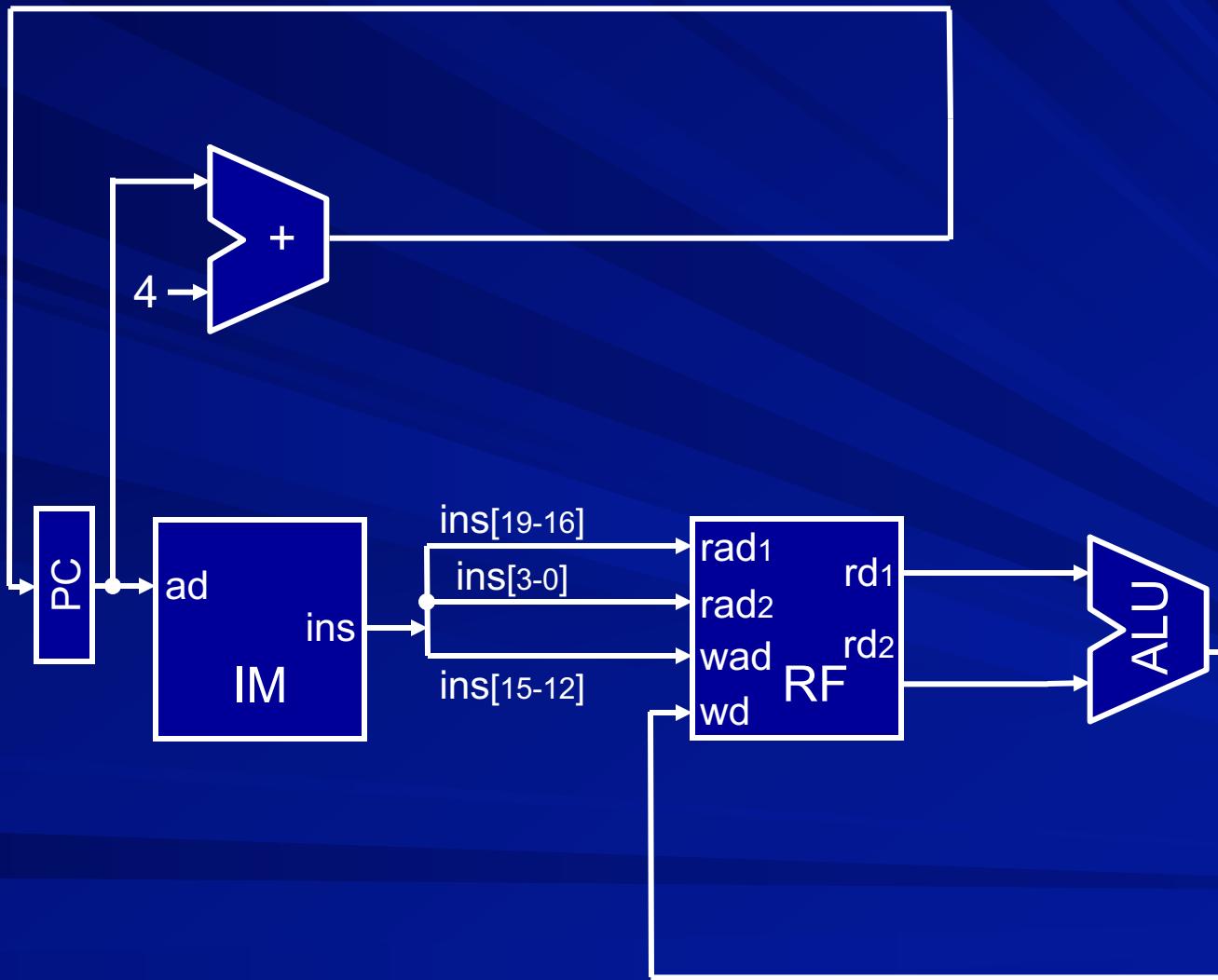
Passing operands to ALU



Passing the result to RF



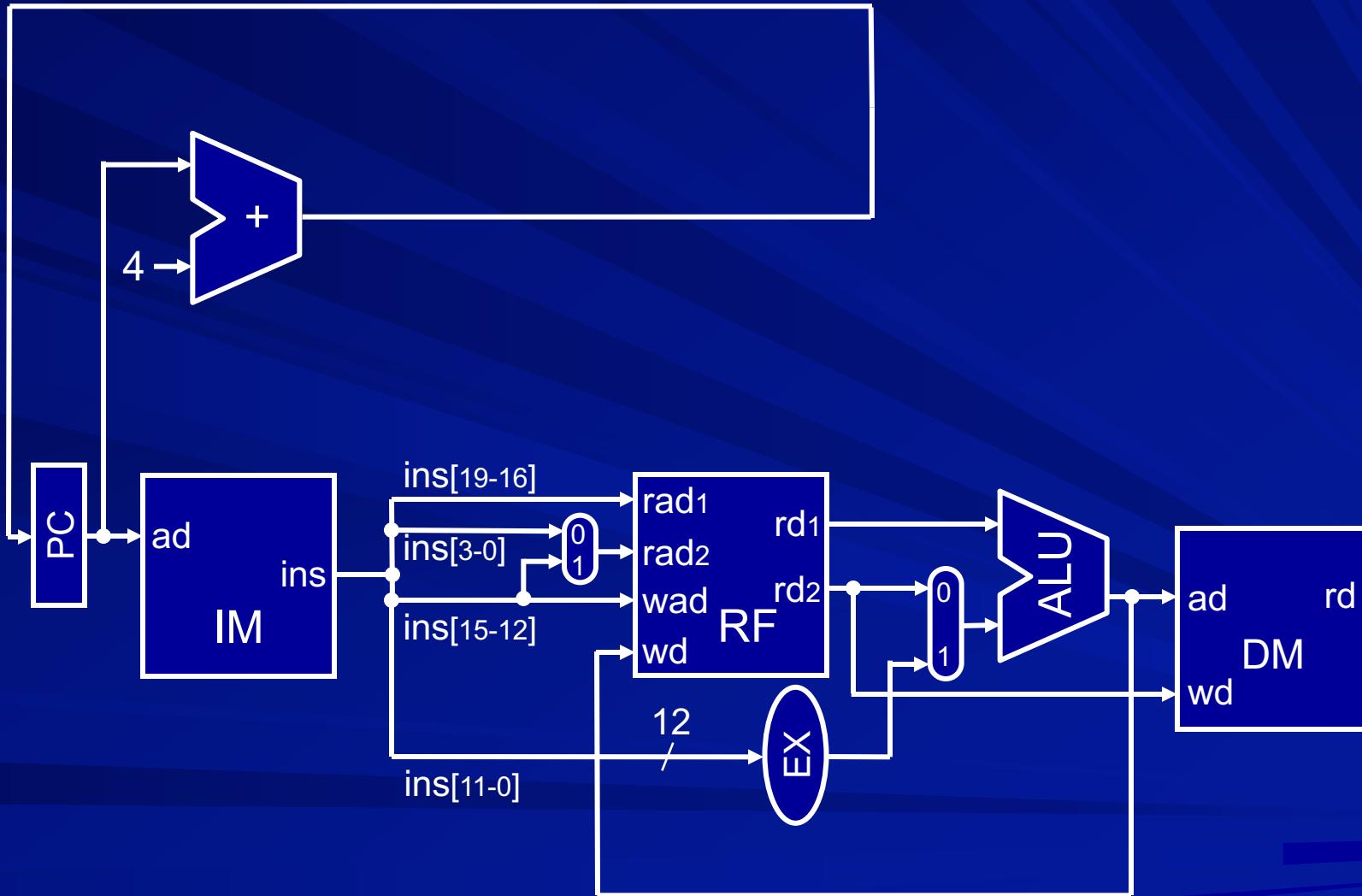
Incrementing PC



Actions for str instructions

- fetch instruction
- access the register file
- compute address in ALU
- write data into memory
- increment PC

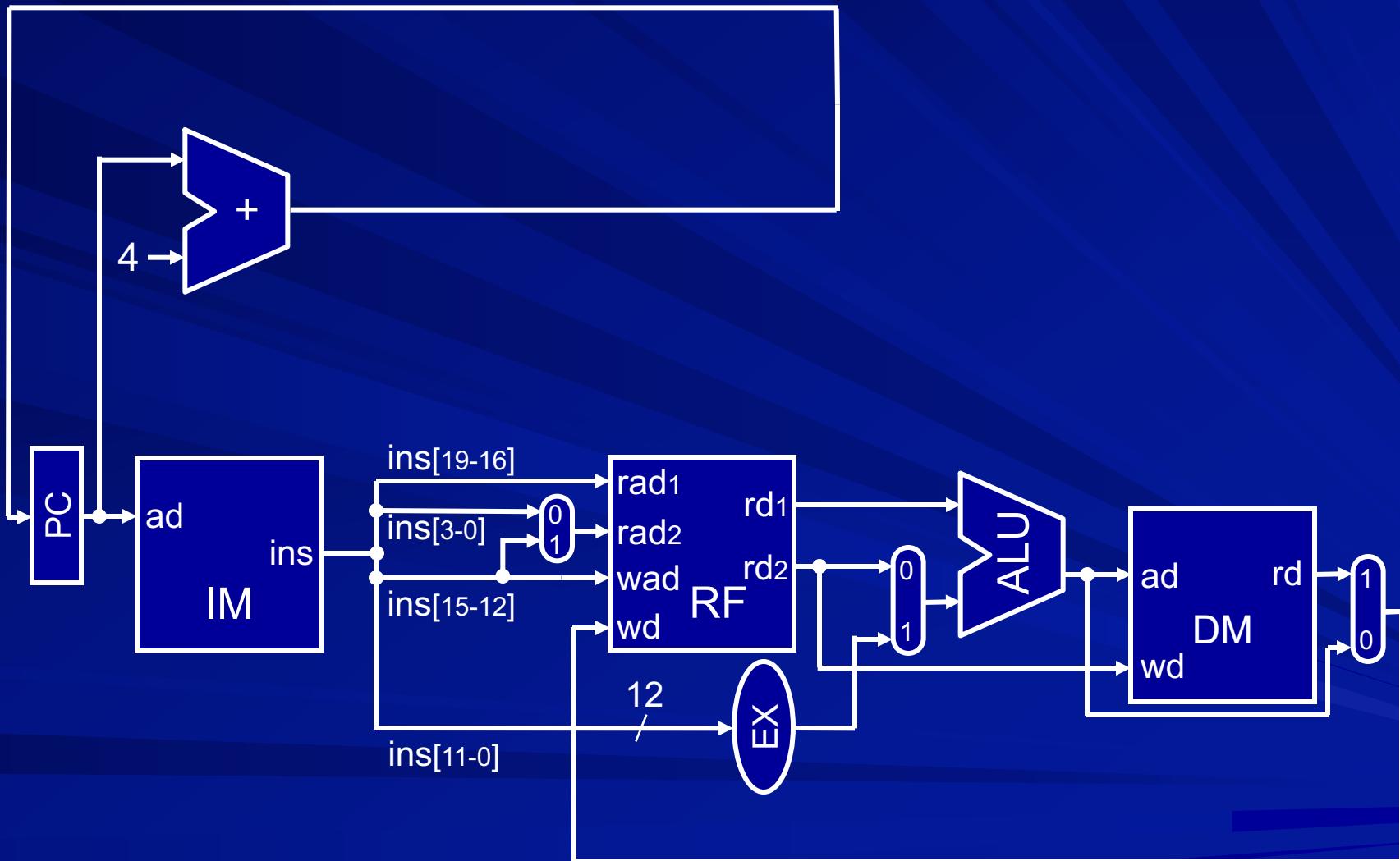
Adding “str” instruction



Actions for ldr instructions

- fetch instruction
- access the register file
- compute address in ALU
- read data from memory
- put data in register file
- increment PC

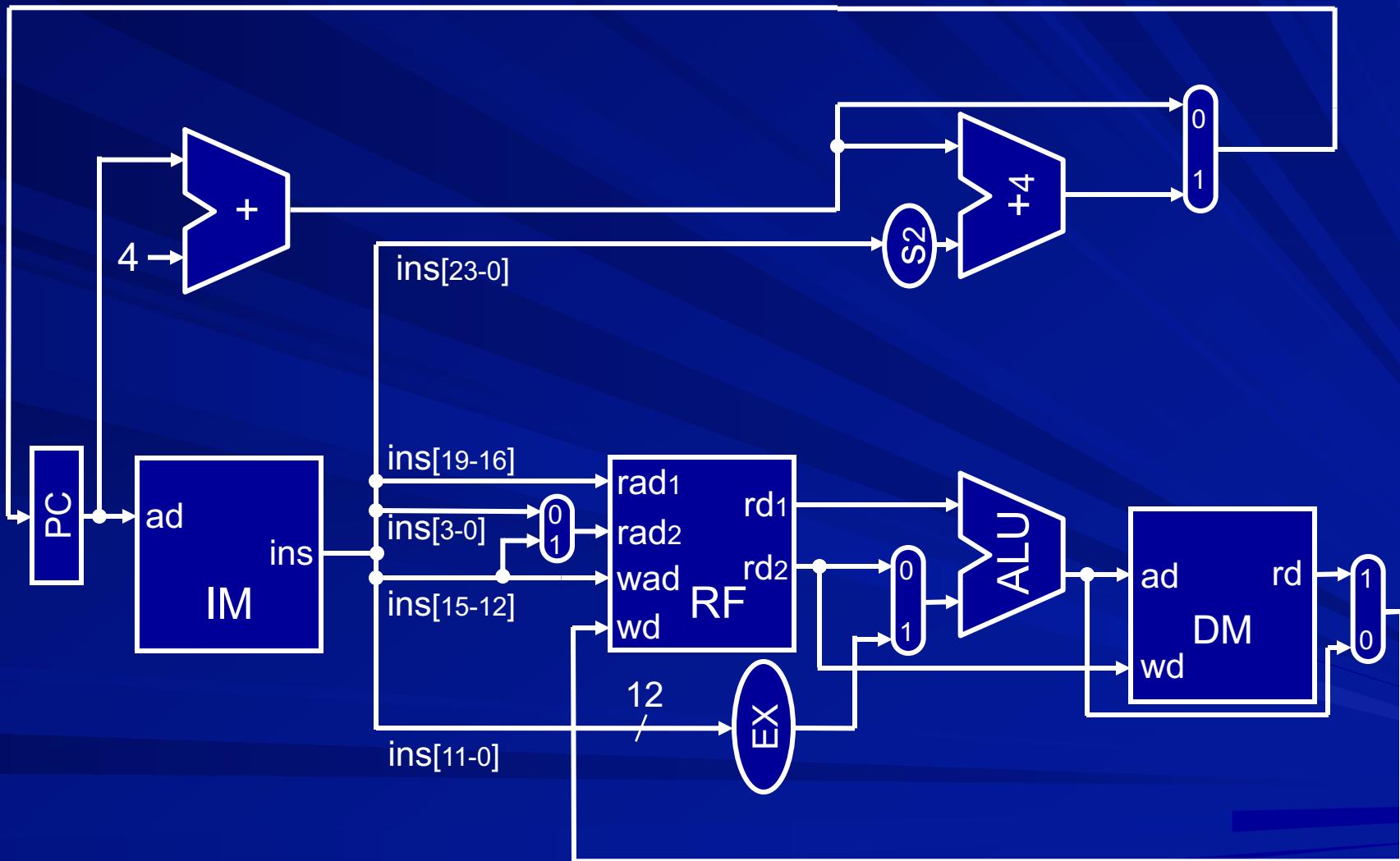
Adding “ldr” instruction



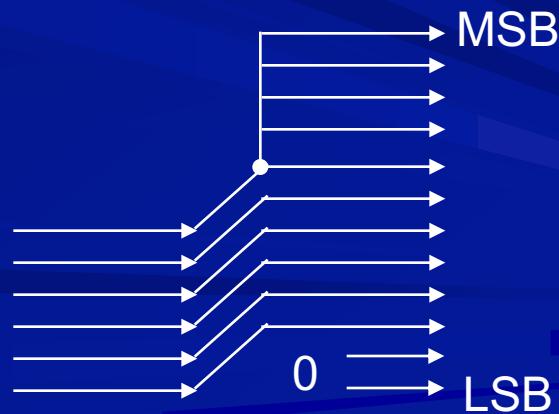
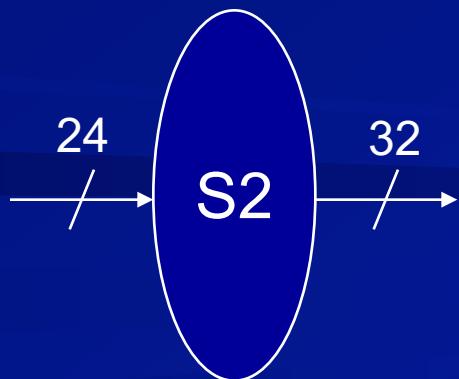
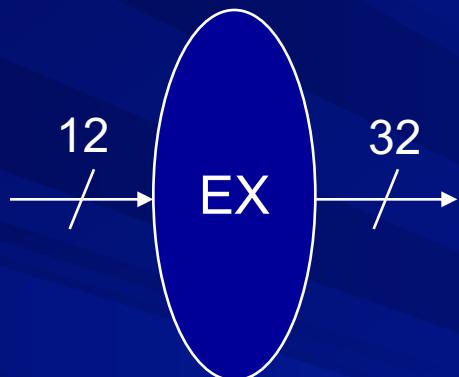
Actions for branch instruction

- fetch instruction
- compute target address
- transfer address to pc

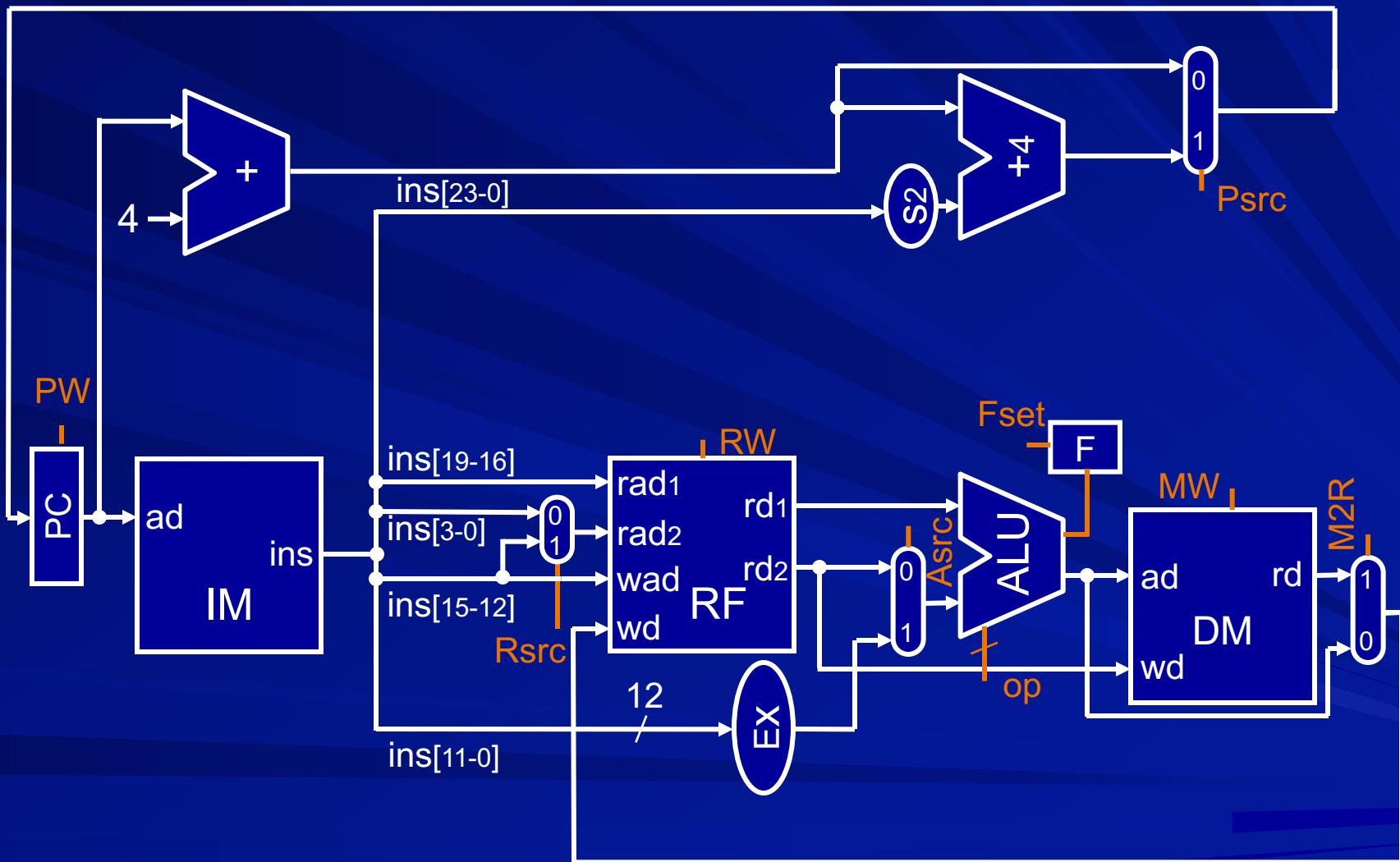
Adding “b” instruction



Extending offsets



Single cycle Datapath



Problems with single cycle design

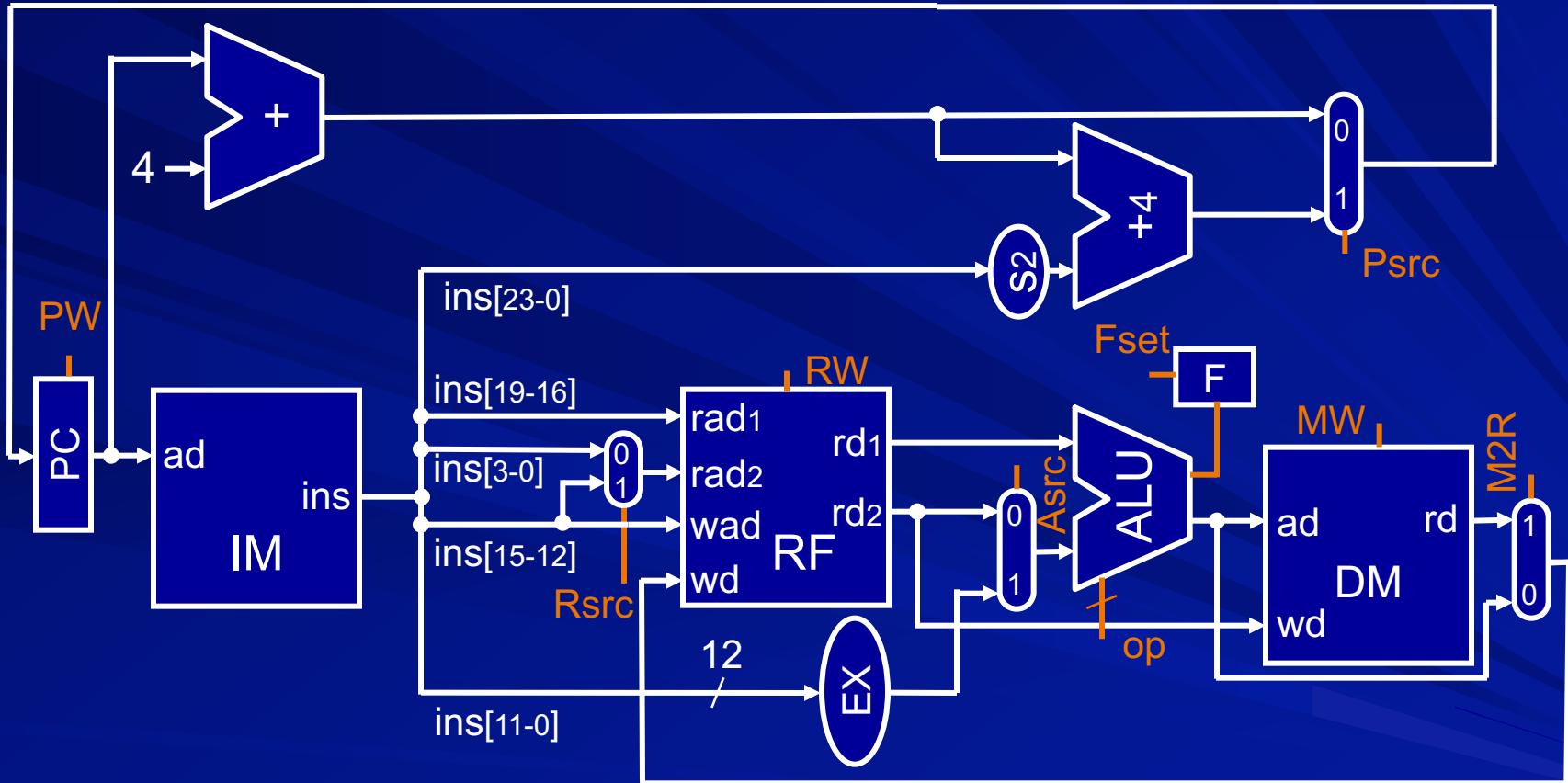
- Slowest instruction pulls down the clock frequency
- Resource utilization is poor
- There are some instructions which are impossible to be implemented in this manner

Analyzing performance

Component delays

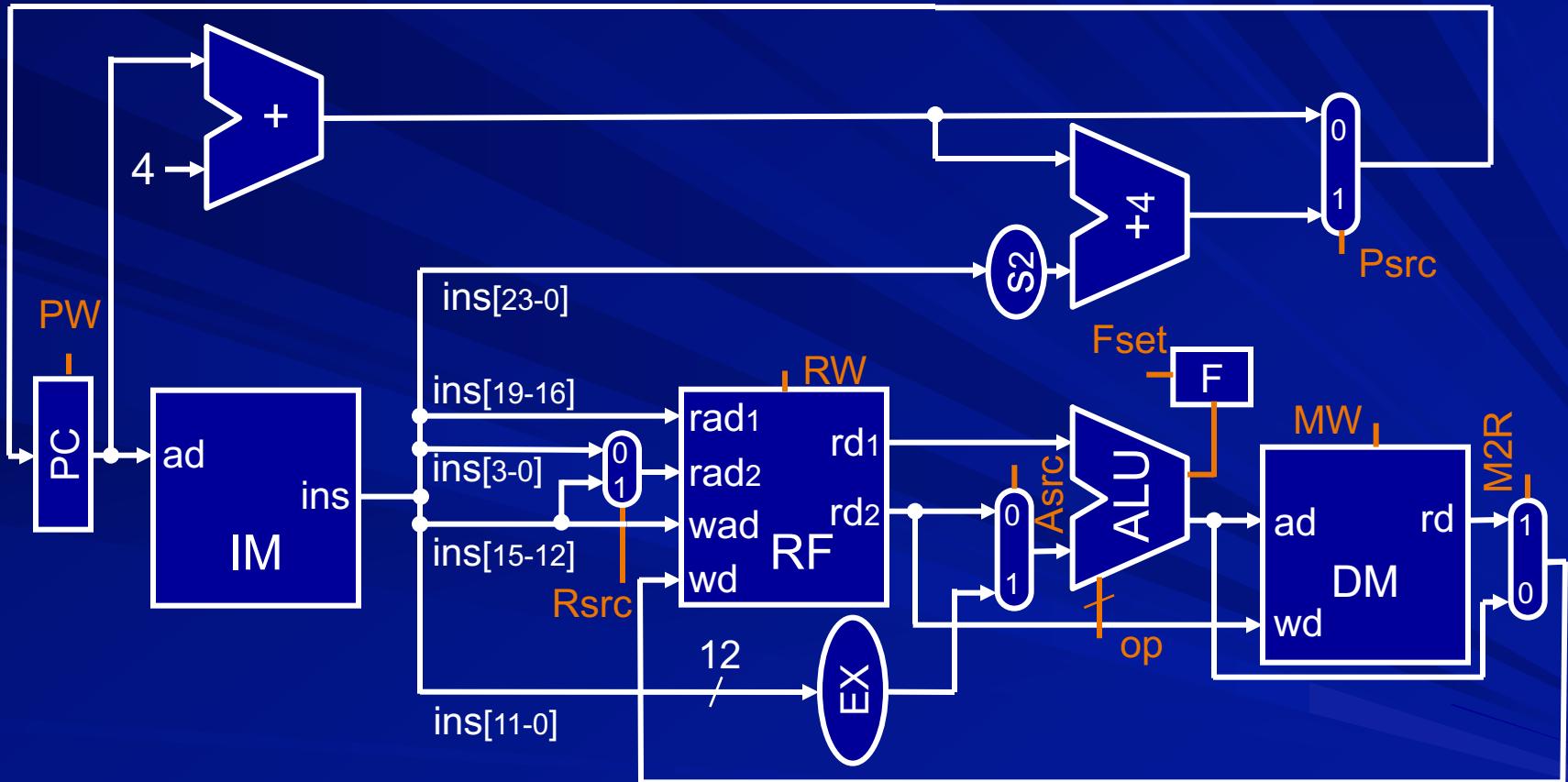
■ Register	0
■ Adder	t_+
■ ALU	t_A
■ Multiplexer	0
■ Register file	t_R
■ Program memory	t_I
■ Data memory	t_M
■ Bit manipulation components	0

Delay for {add, sub, ...}



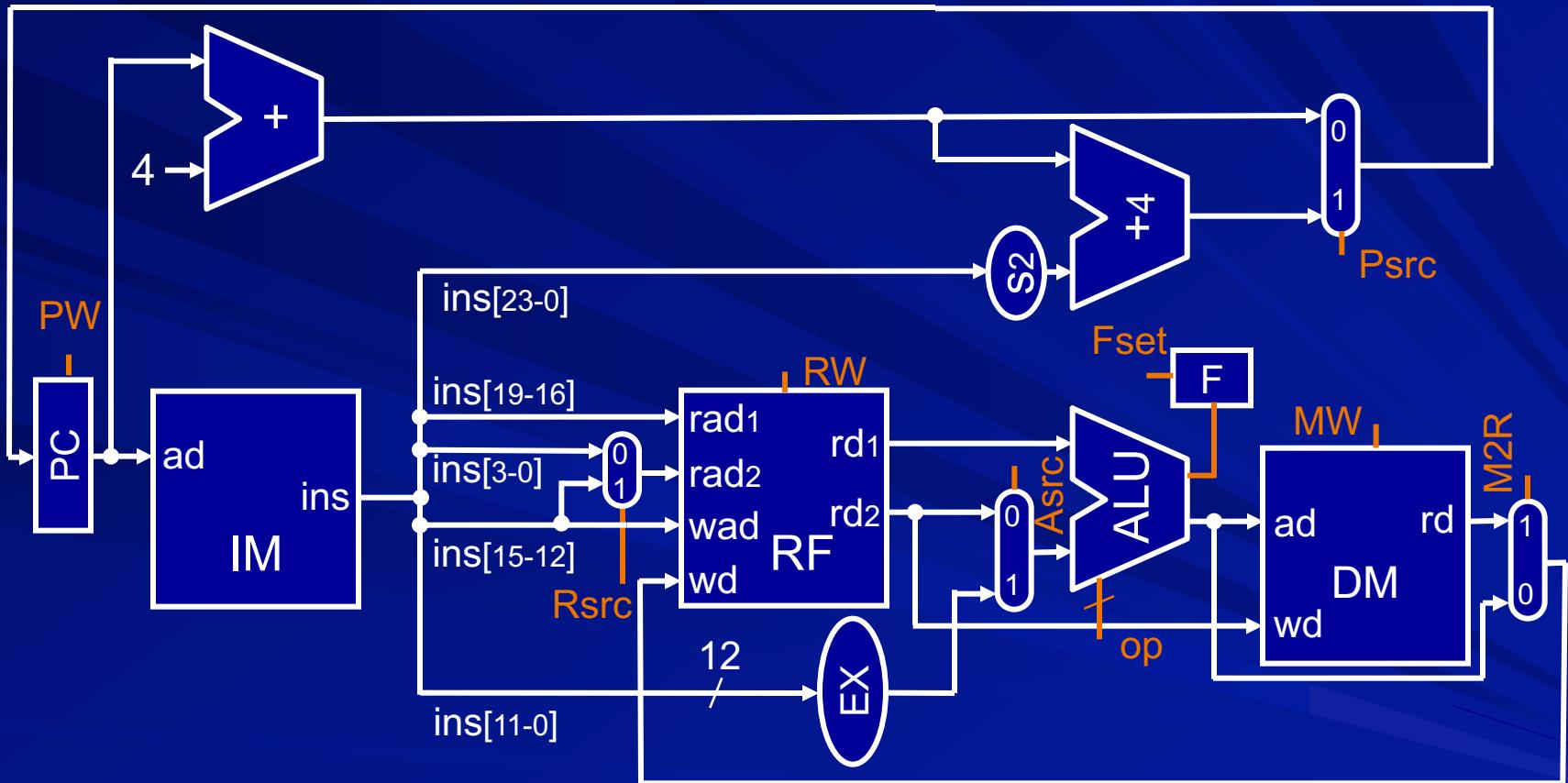
$$\max \left\{ \begin{array}{c} t_+ \\ t_I + t_R + t_A + t_R \end{array} \right\}$$

Delay for {cmp, tst, ...}



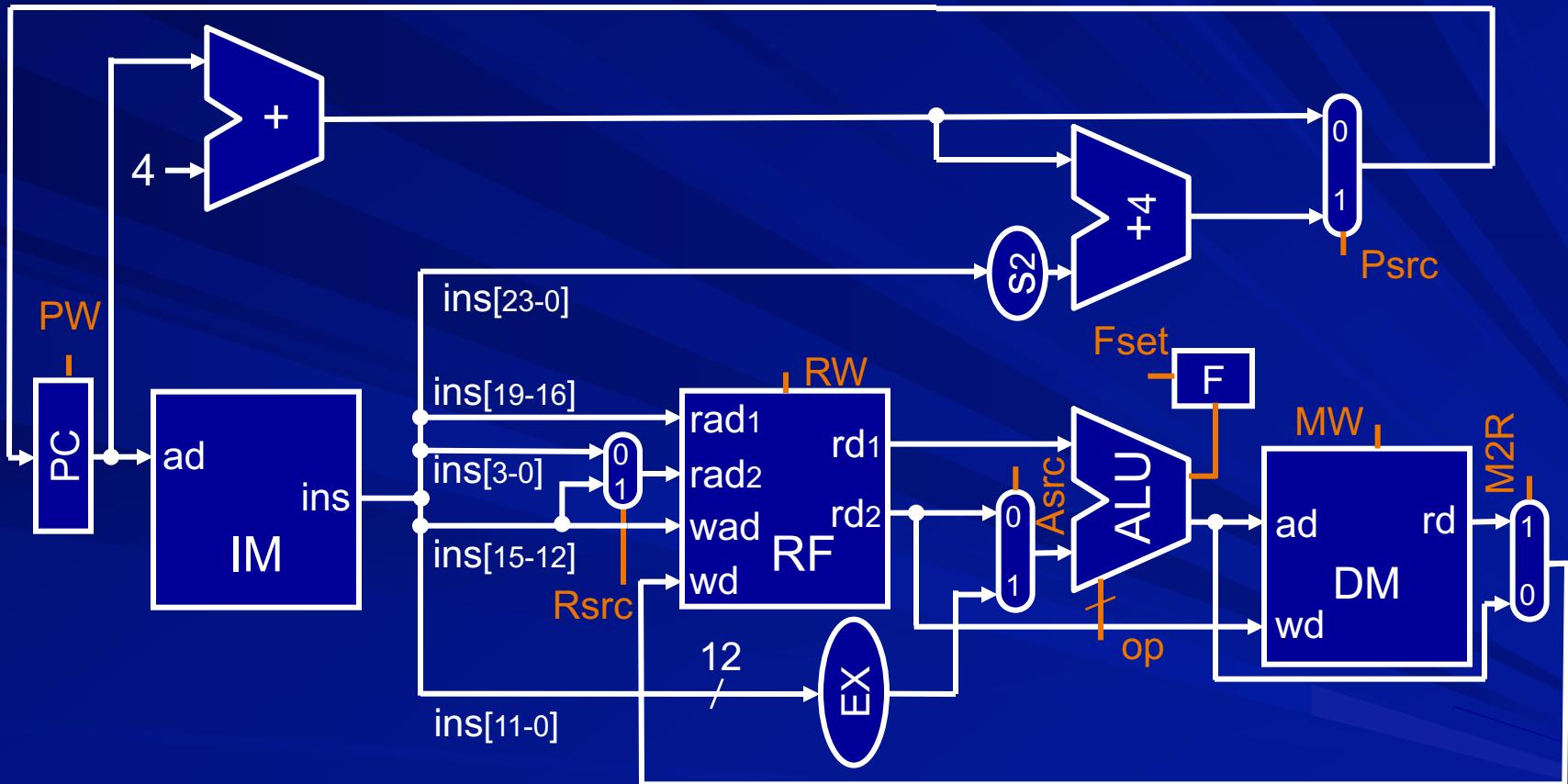
$$\max \left\{ \begin{array}{l} t_+ \\ t_I + t_R + t_A \end{array} \right\}$$

Delay for {str}



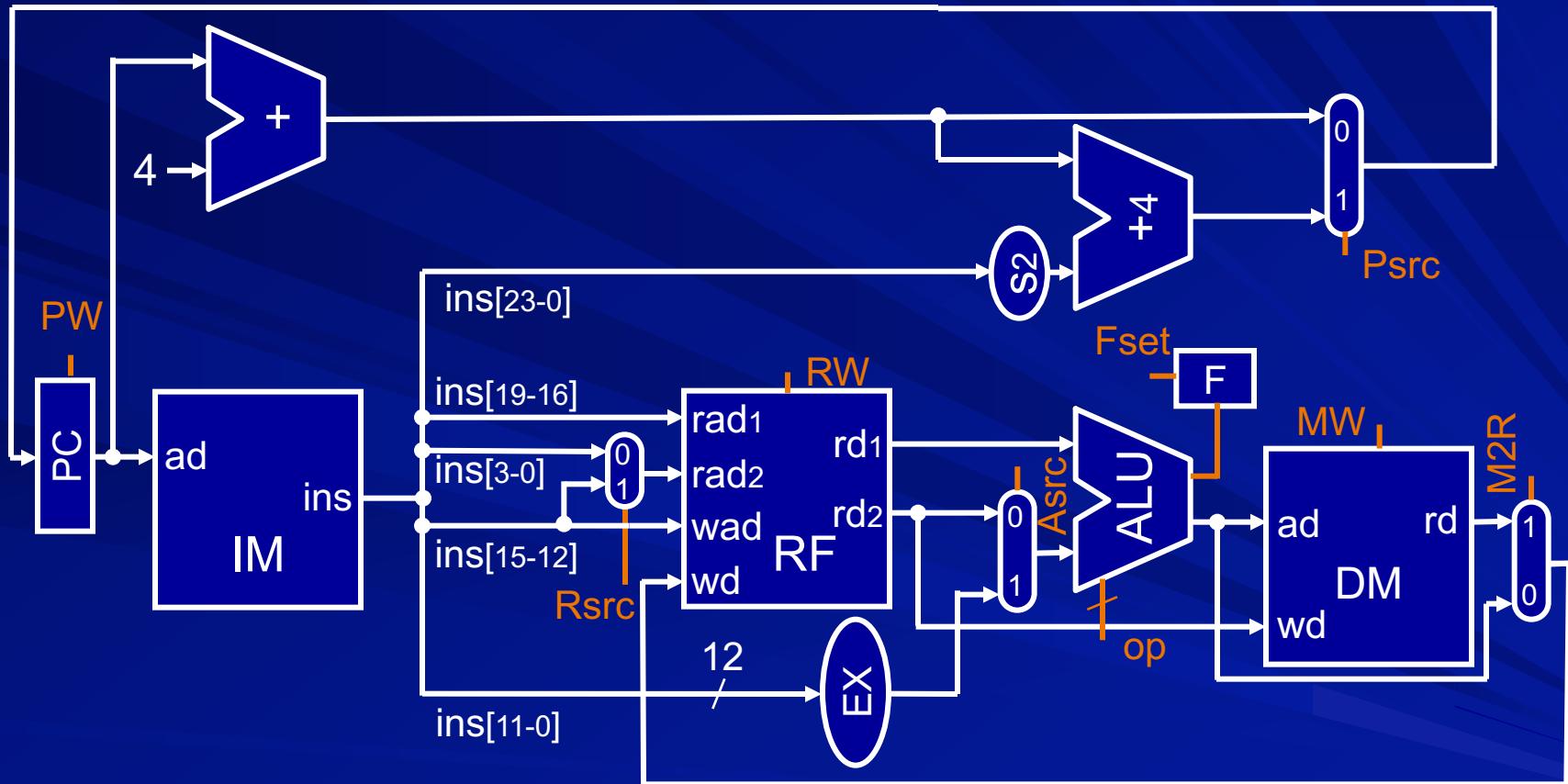
$$\max \left\{ \begin{array}{l} t_+ \\ t_I + t_R + t_A + t_M \end{array} \right\}$$

Delay for {ldr}



$$\max \left\{ t_+, t_I + t_R + t_A + t_M + t_R \right\}$$

Delay for {b}



$$\max \left\{ \begin{array}{l} t_I + t_+ \\ t_+ + t_+ \end{array} \right\}$$

Overall clock period

$$\max \left\{ \begin{array}{ll} t_+, & t_I + t_R + t_A + t_R \\ t_+, & t_I + t_R + t_A \\ t_+, & t_I + t_R + t_A + t_M \\ t_+, & \overbrace{t_I + t_R + t_A + t_M + t_R} \\ t_I + t_+, & \overbrace{t_+ + t_+} ? \end{array} \right.$$

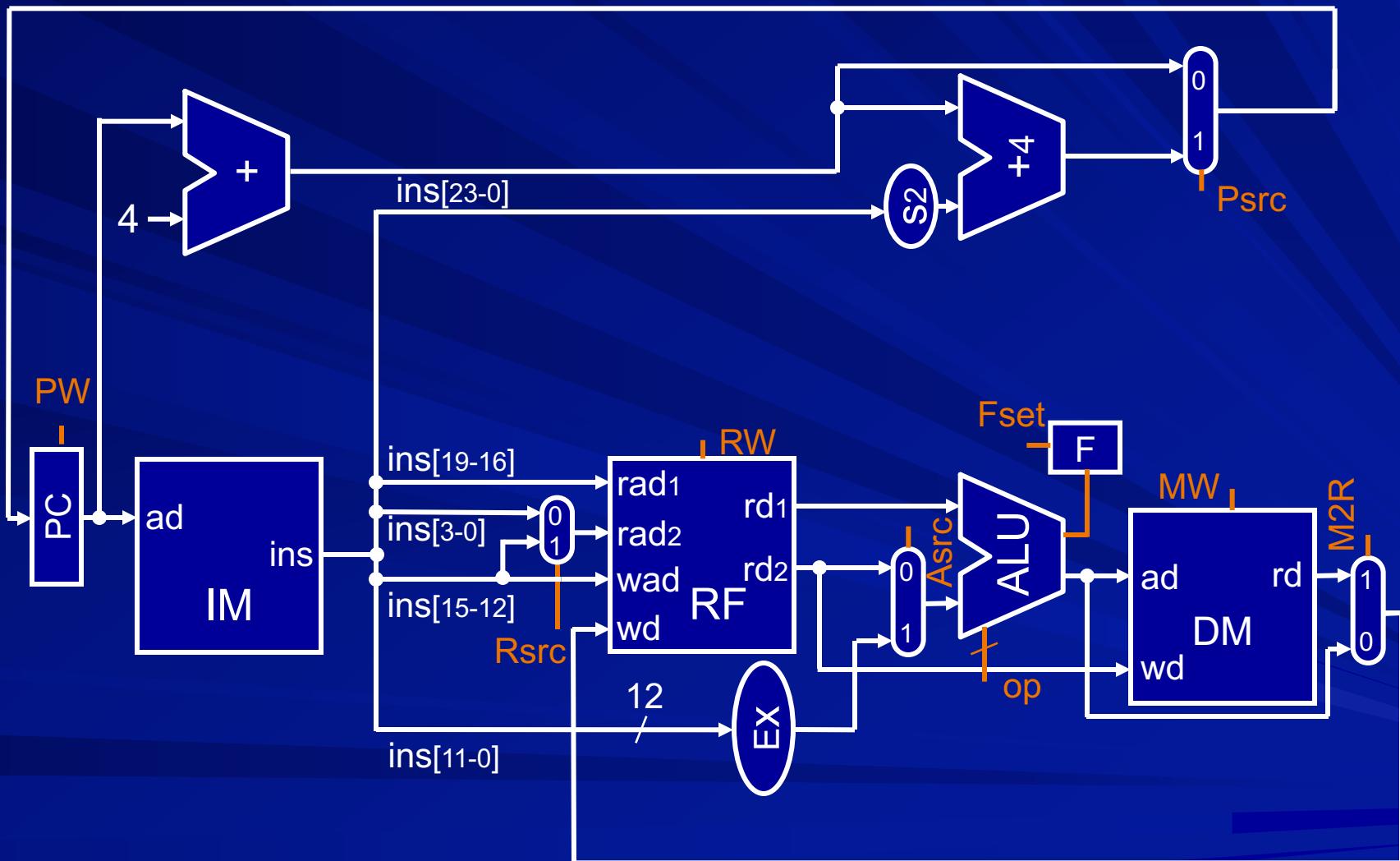
Thanks

COL216

Computer Architecture

Design a processor -
Multi-cycle design approach
5th February, 2022

Single cycle Datapath



Problems with single cycle design

- Slowest instruction pulls down the clock frequency
- Resource utilization is poor
- There are some instructions which are impossible to be implemented in this manner

Analyzing performance

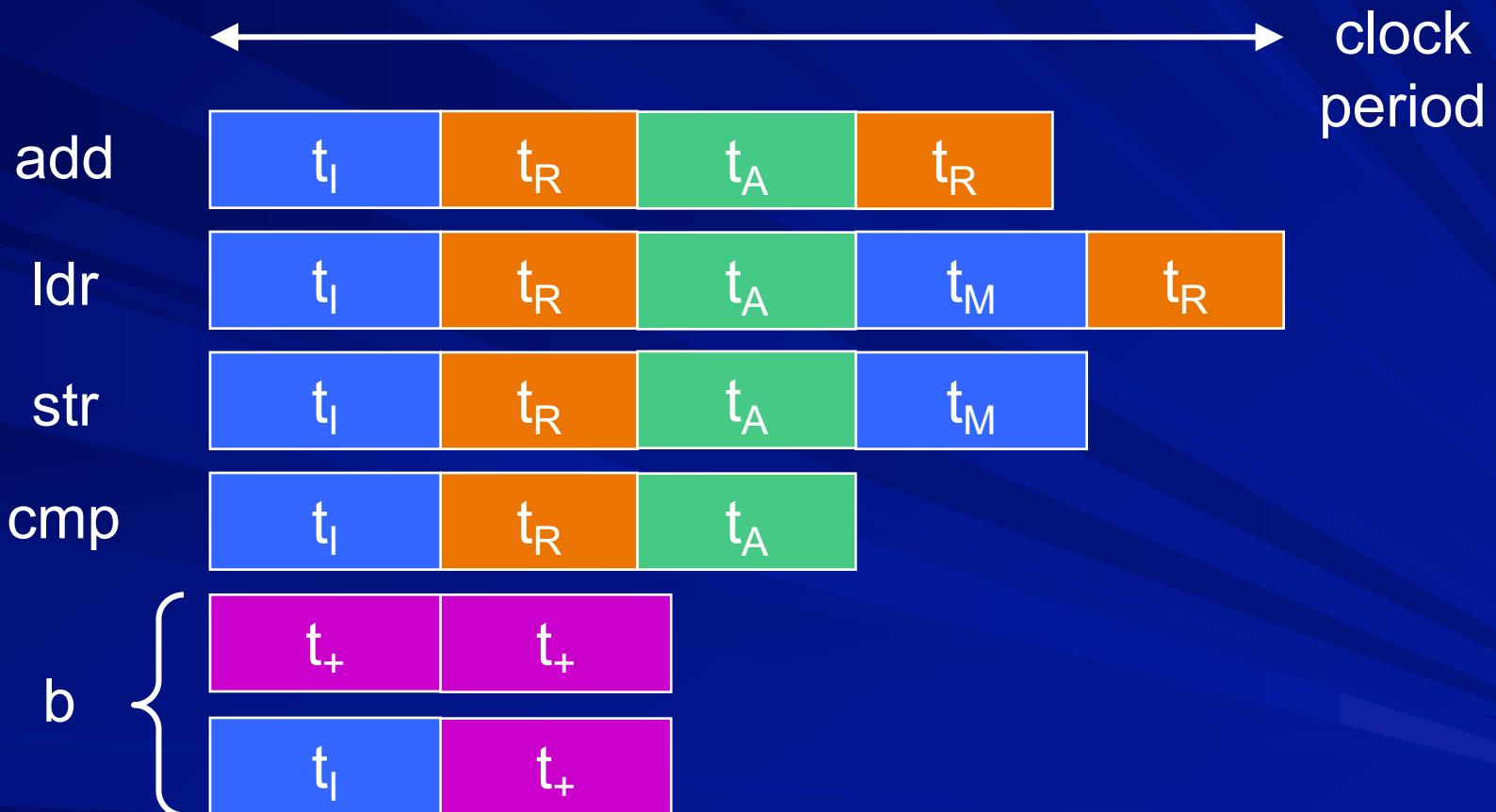
Component delays

■ 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

Overall clock period

$$\max \left\{ \begin{array}{ll} t_+, & t_I + t_R + t_A + t_R \\ t_+, & t_I + t_R + t_A \\ t_+, & t_I + t_R + t_A + t_M \\ t_+, & t_I + t_R + t_A + t_M + t_R \\ t_I + t_+, & t_+ + t_+ \end{array} \right. ?$$

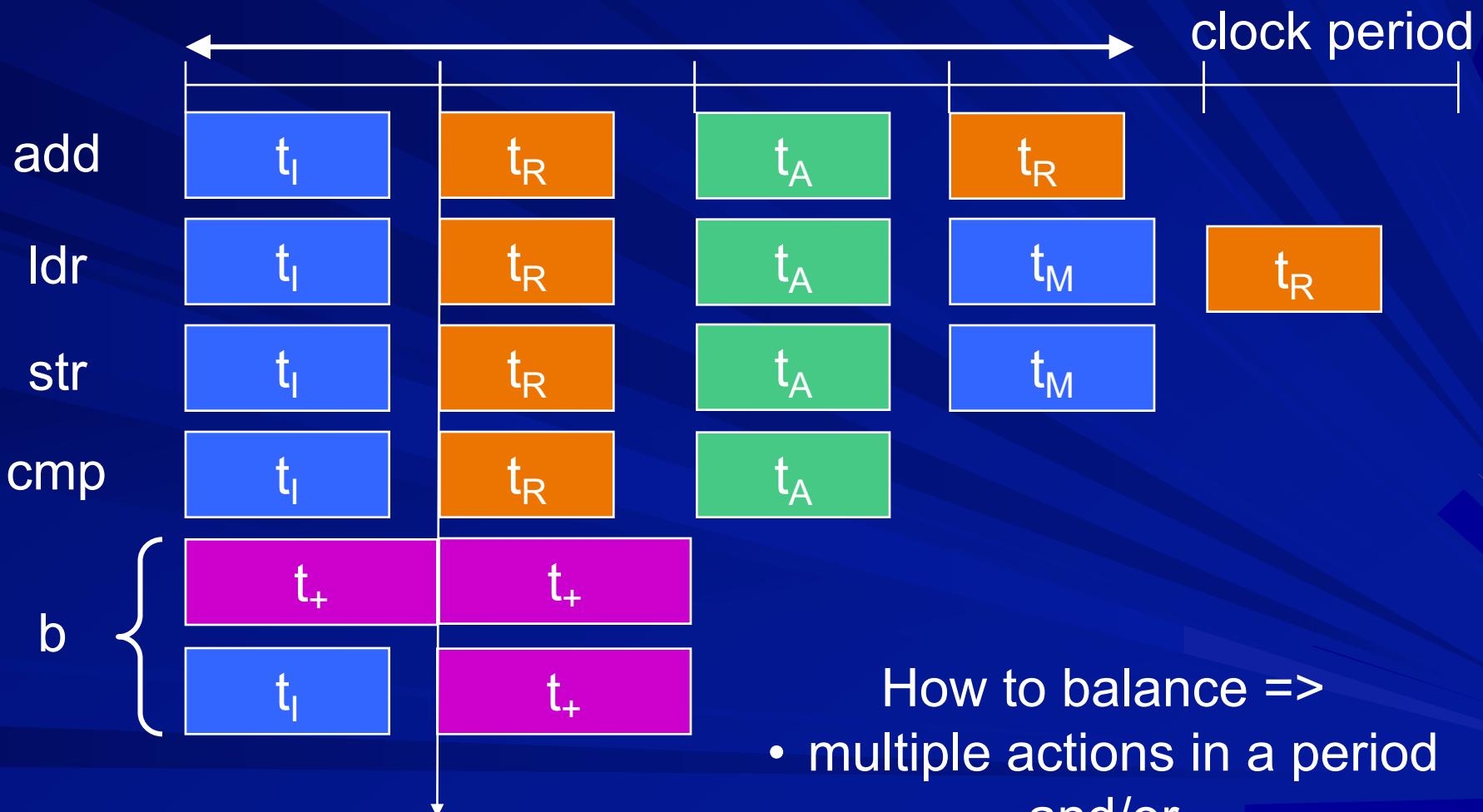
Clock period in single cycle design



Split the cycle into multiple cycles



Unbalanced delays



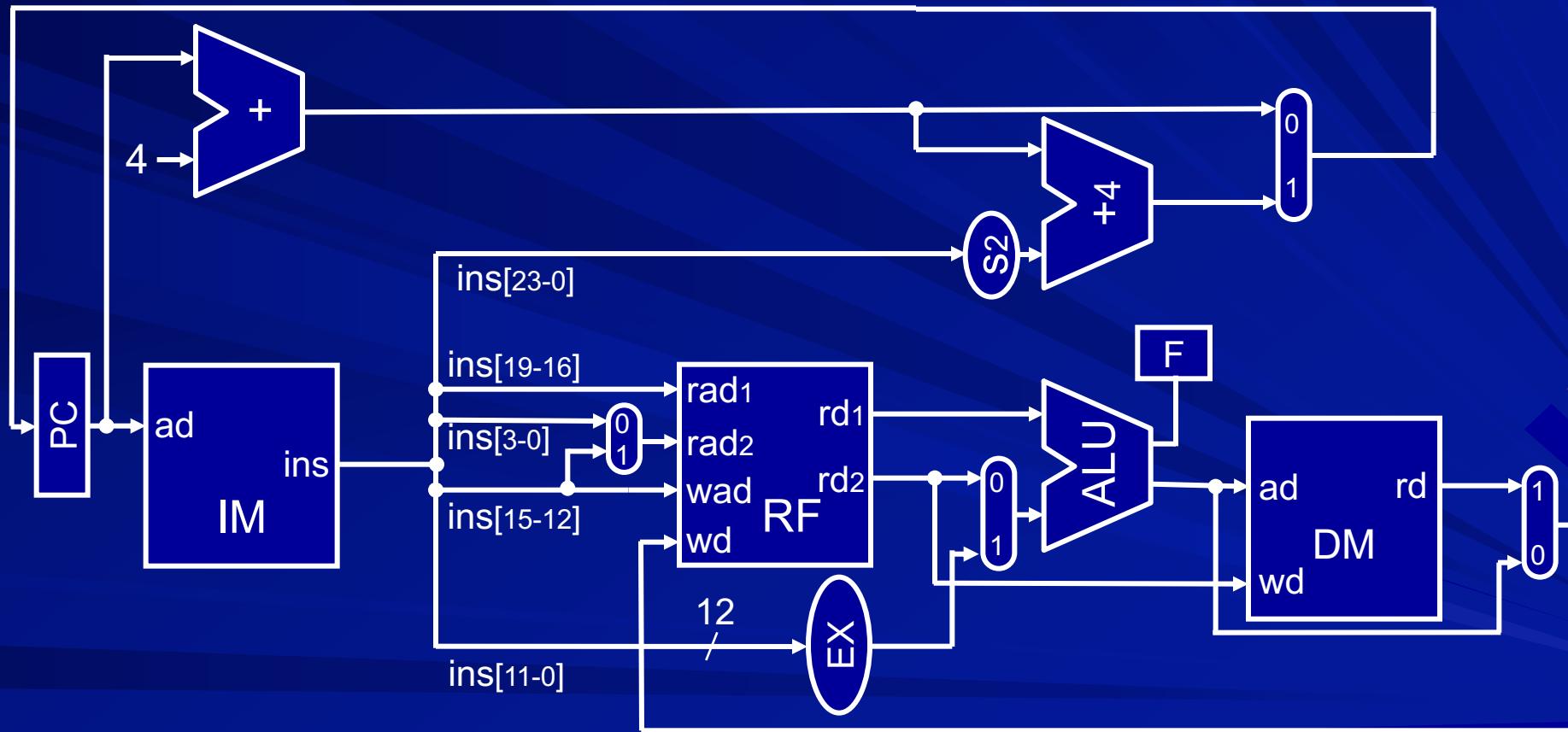
How to balance =>

- multiple actions in a period and/or
- multiple periods for an action

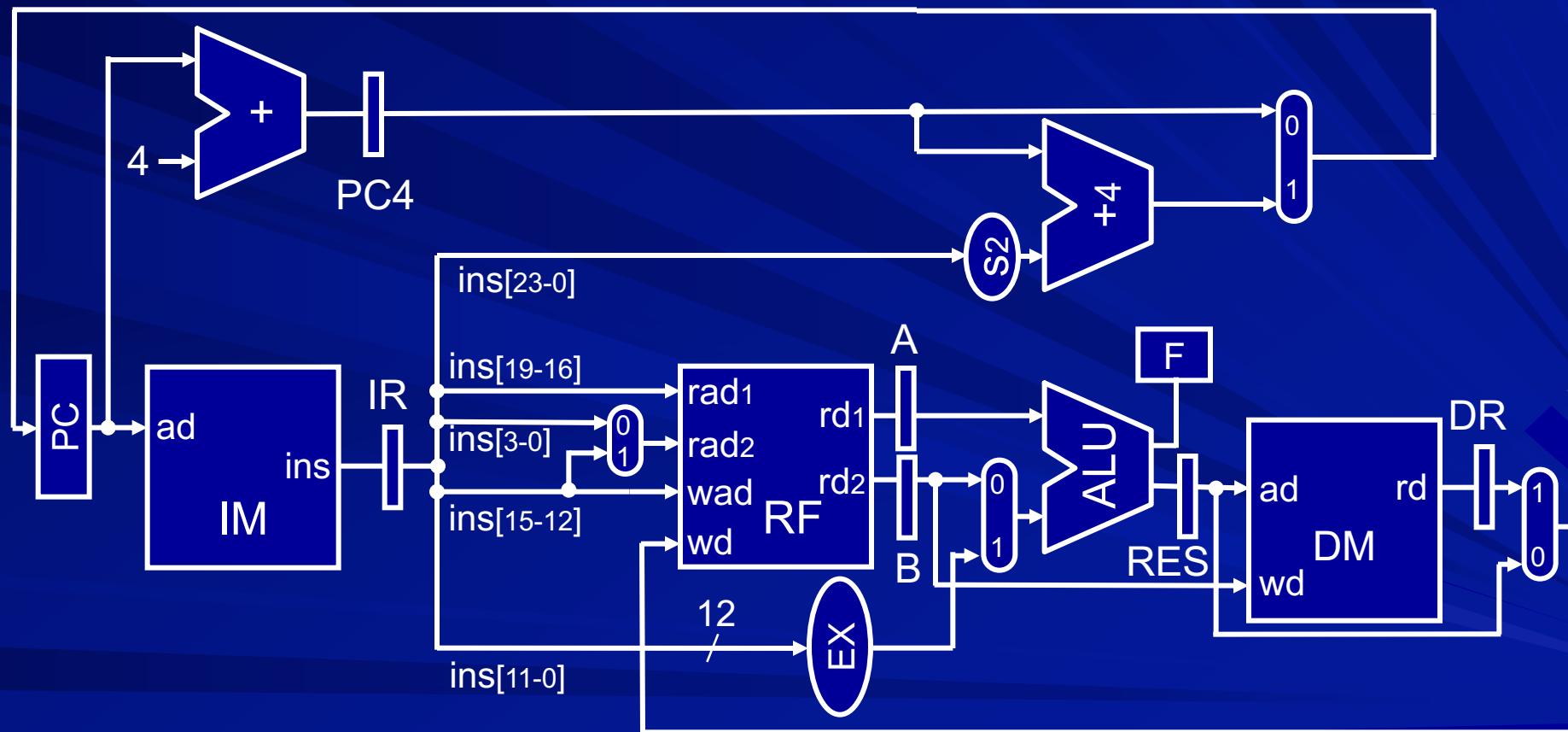
Improving resource utilization

- Can we eliminate two adders?
- How to share (or reuse) a resource (say ALU) in different clock cycles?
- Store results in registers.
- Of course, more multiplexing may be required!
- Resources in this design: RF, ALU, MEM.

Single Cycle Datapath

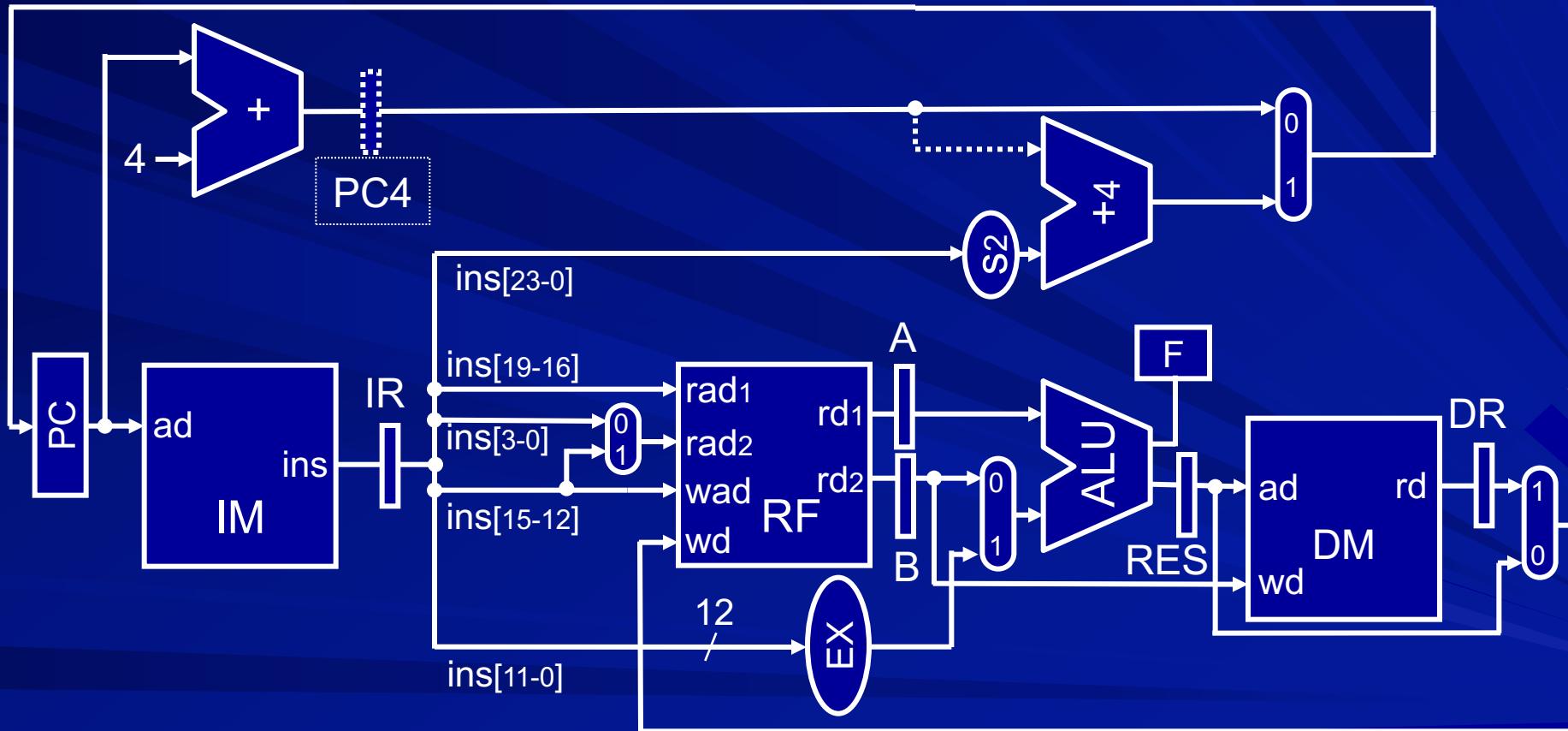


Introducing registers

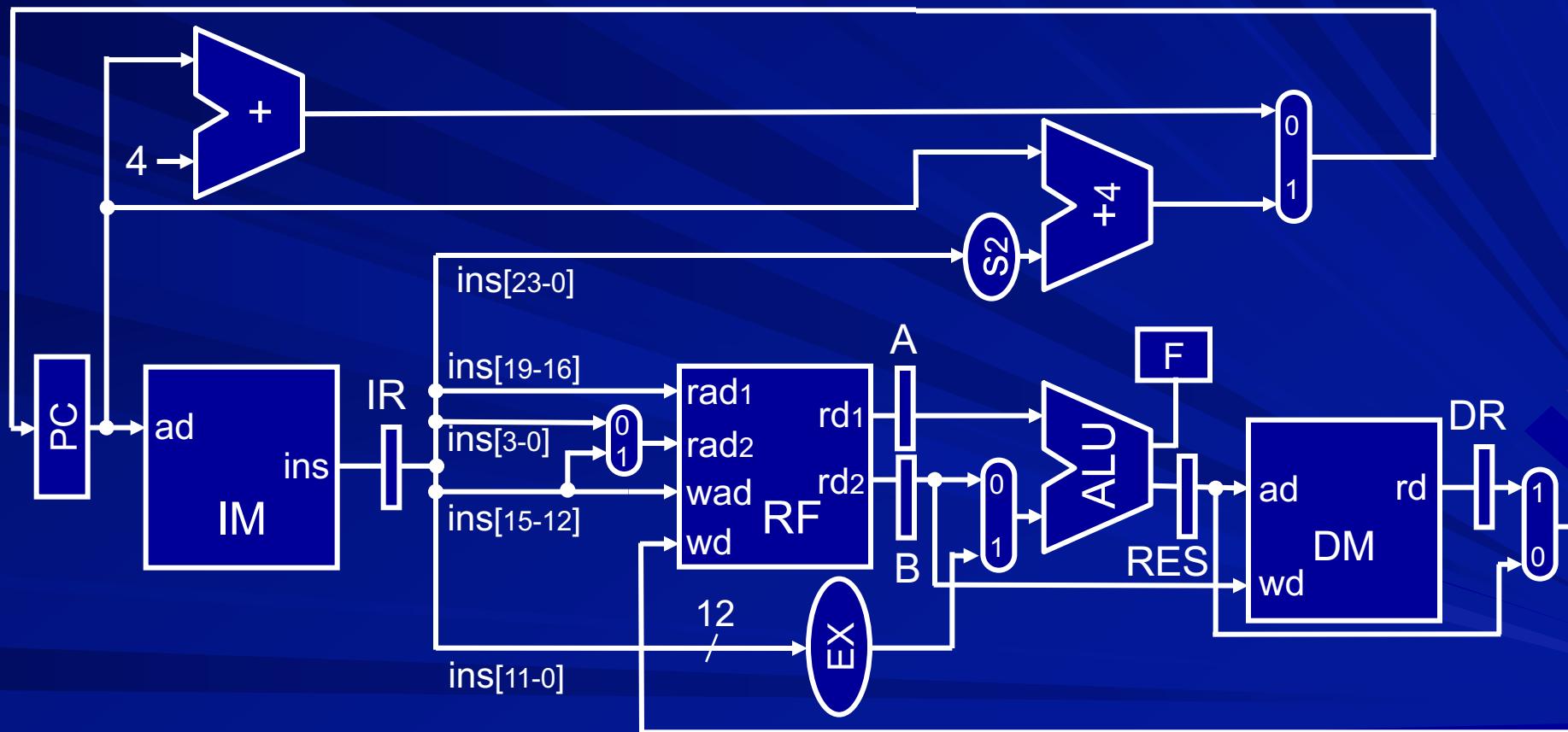


PC4 can be eliminated

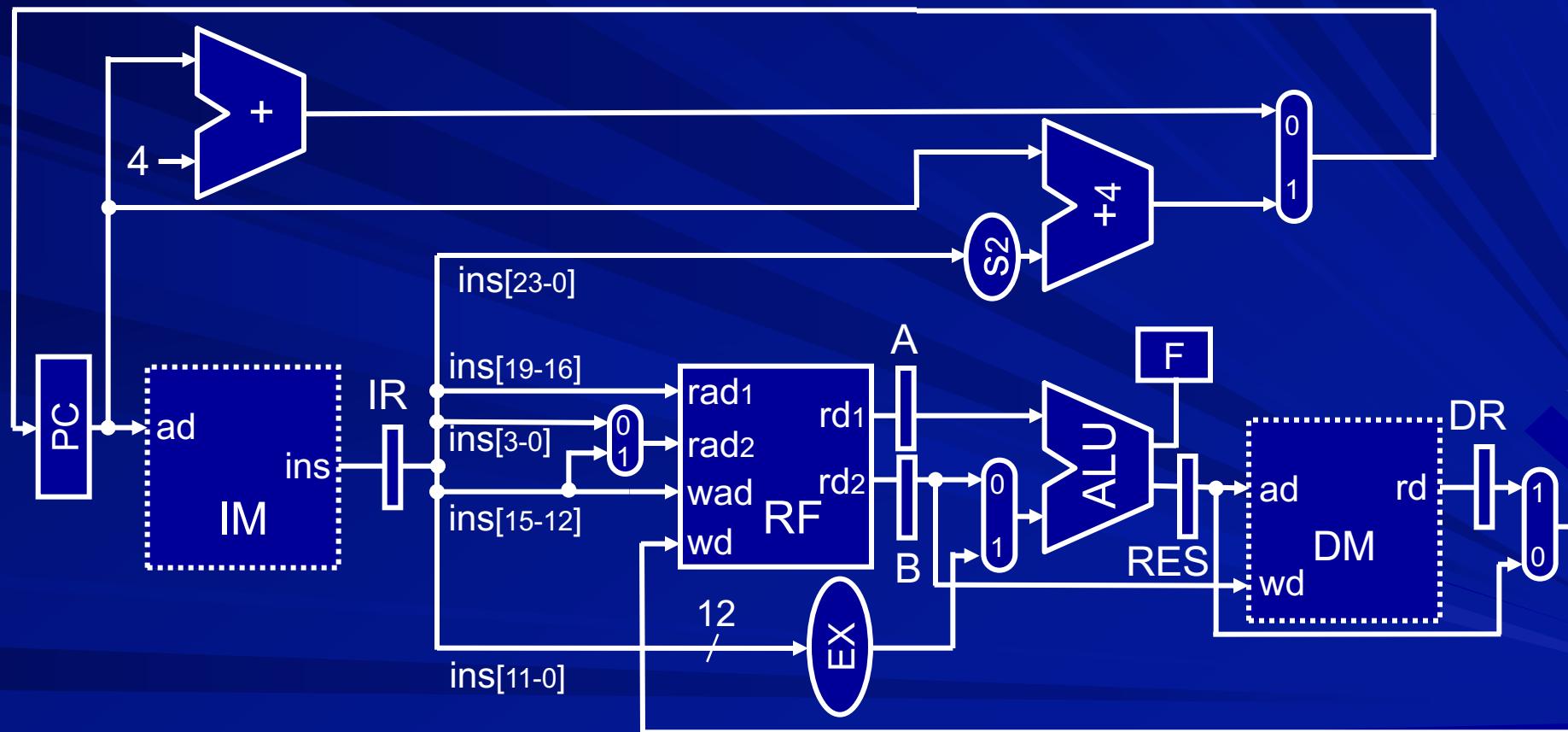
Store PC + 4 in PC itself



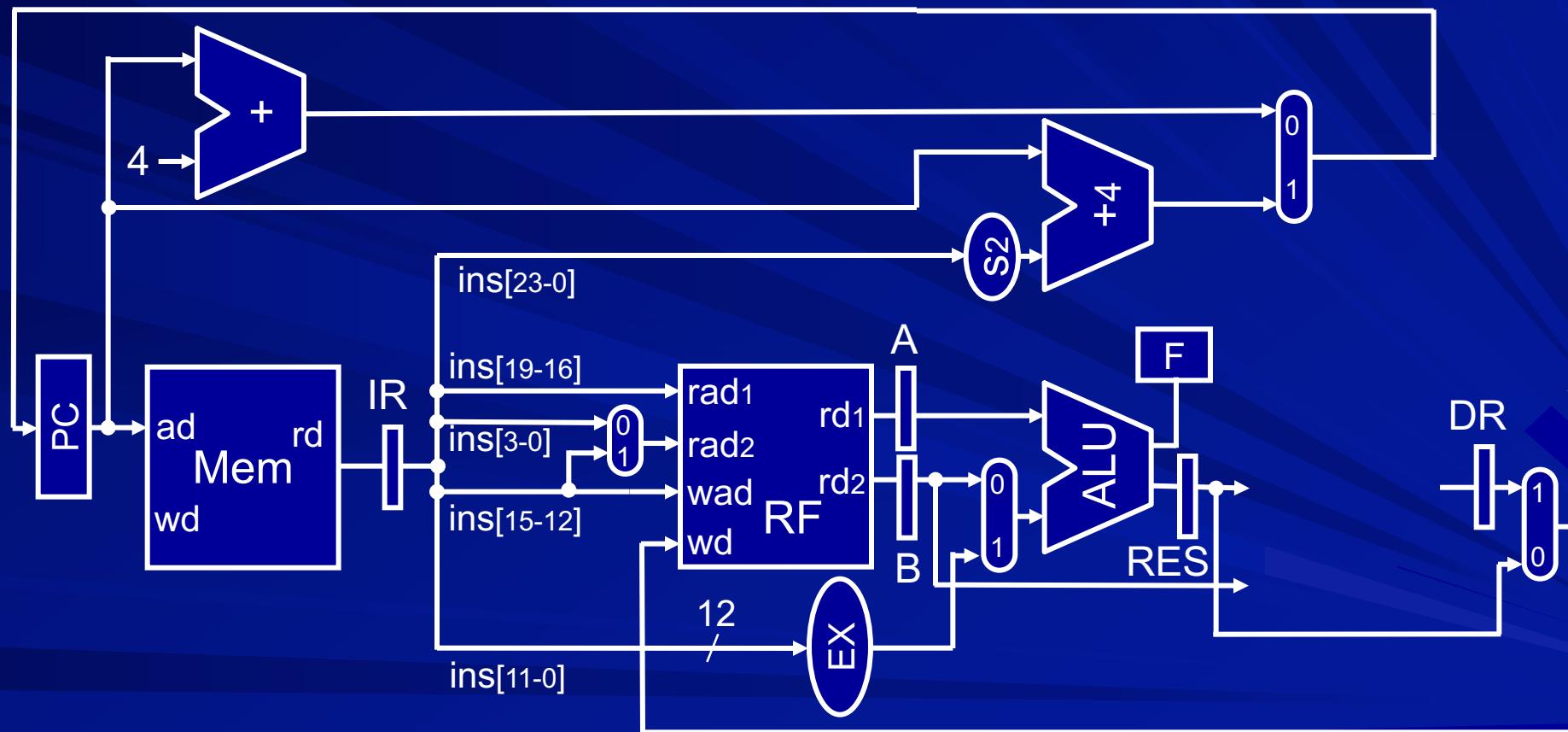
PC4 can be eliminated



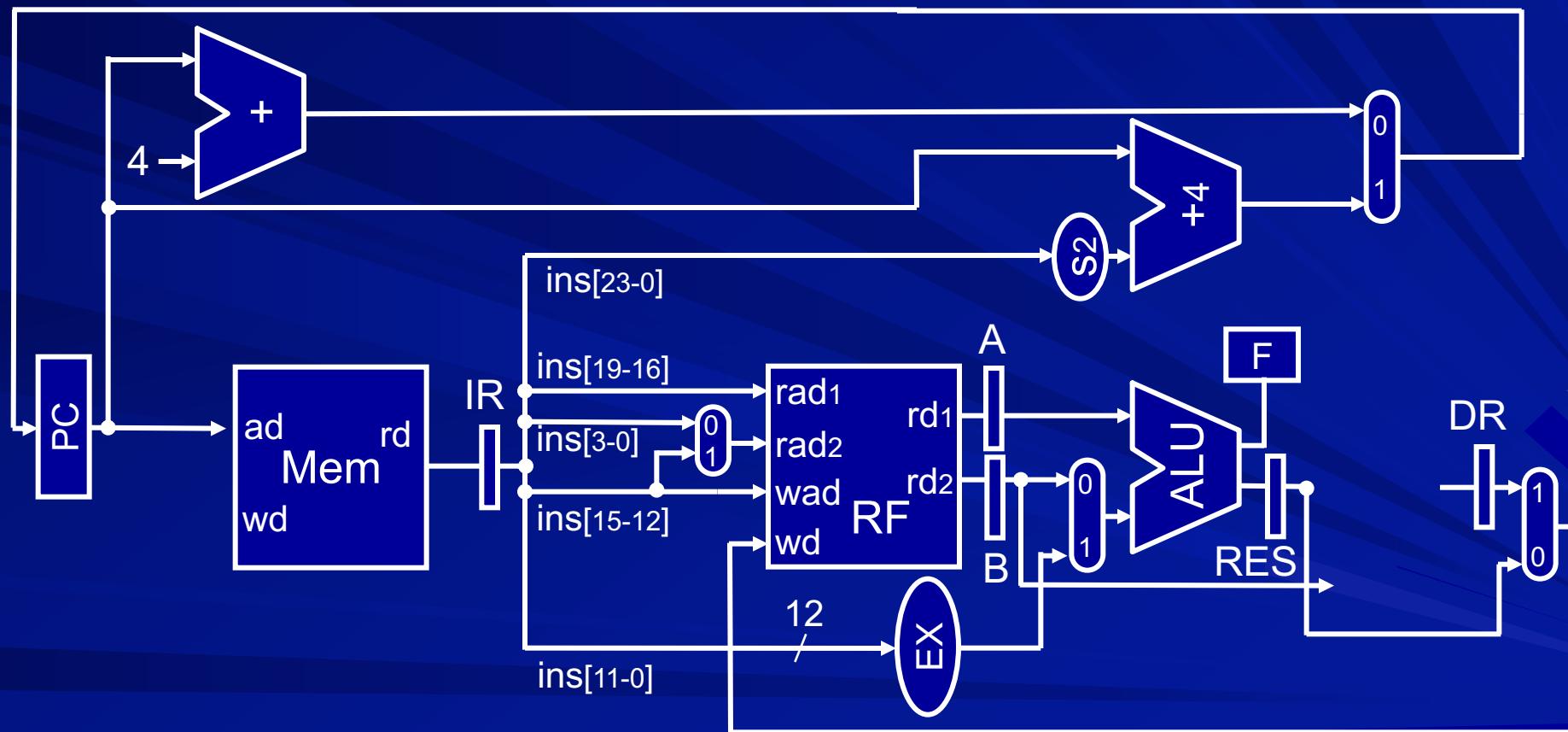
Merge IM and DM



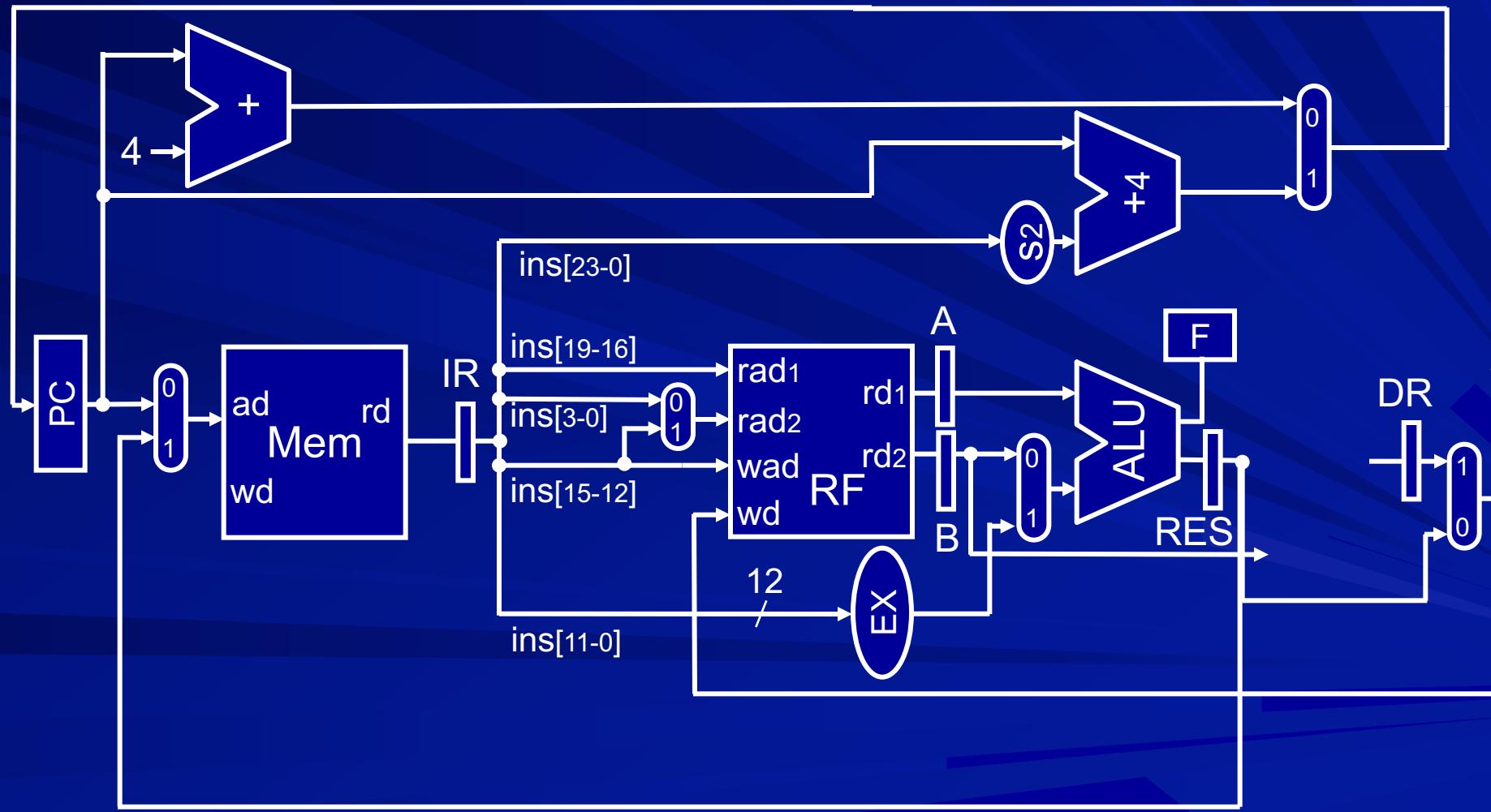
Merge IM and DM



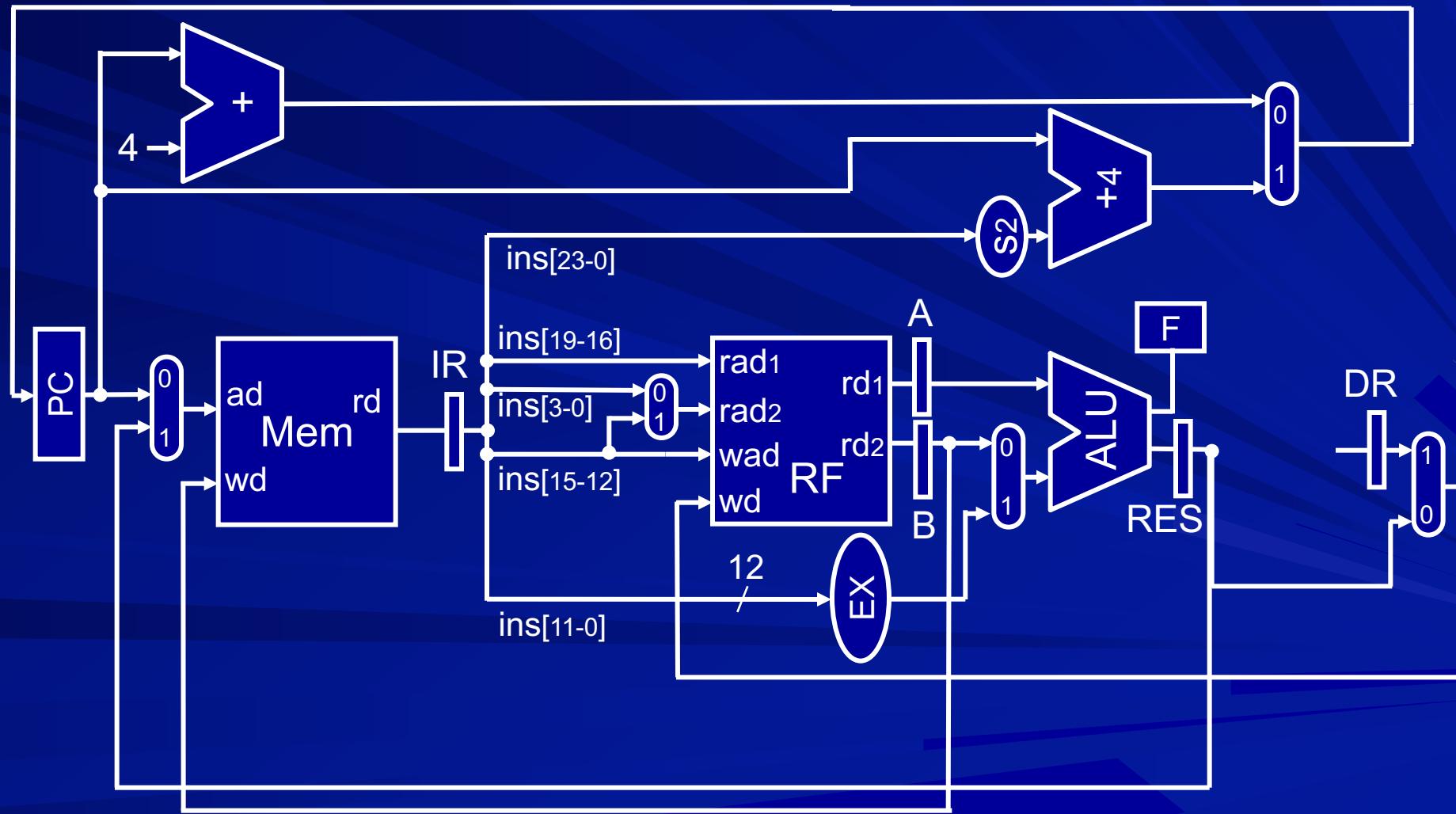
Merge IM and DM



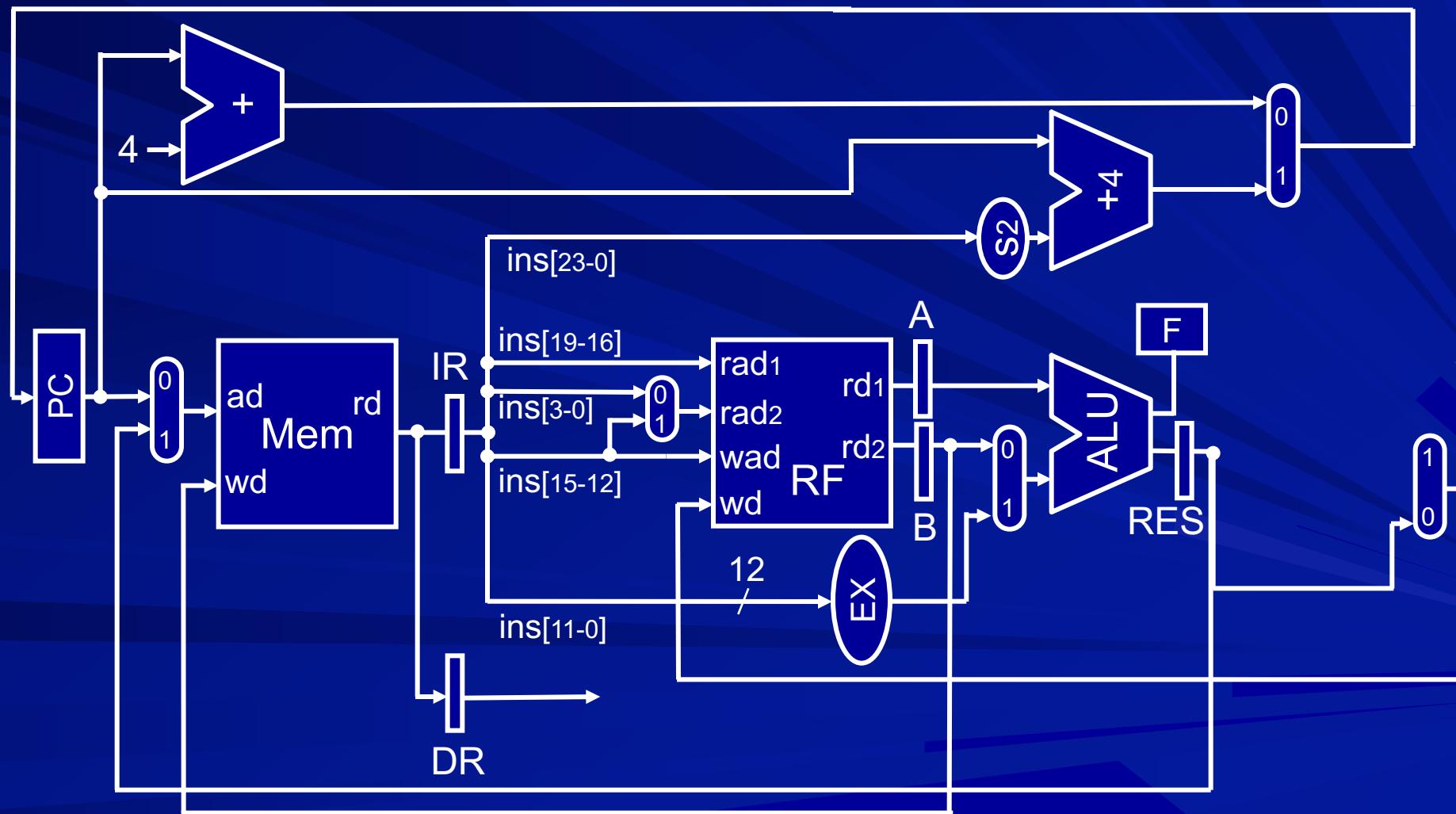
Merge IM and DM



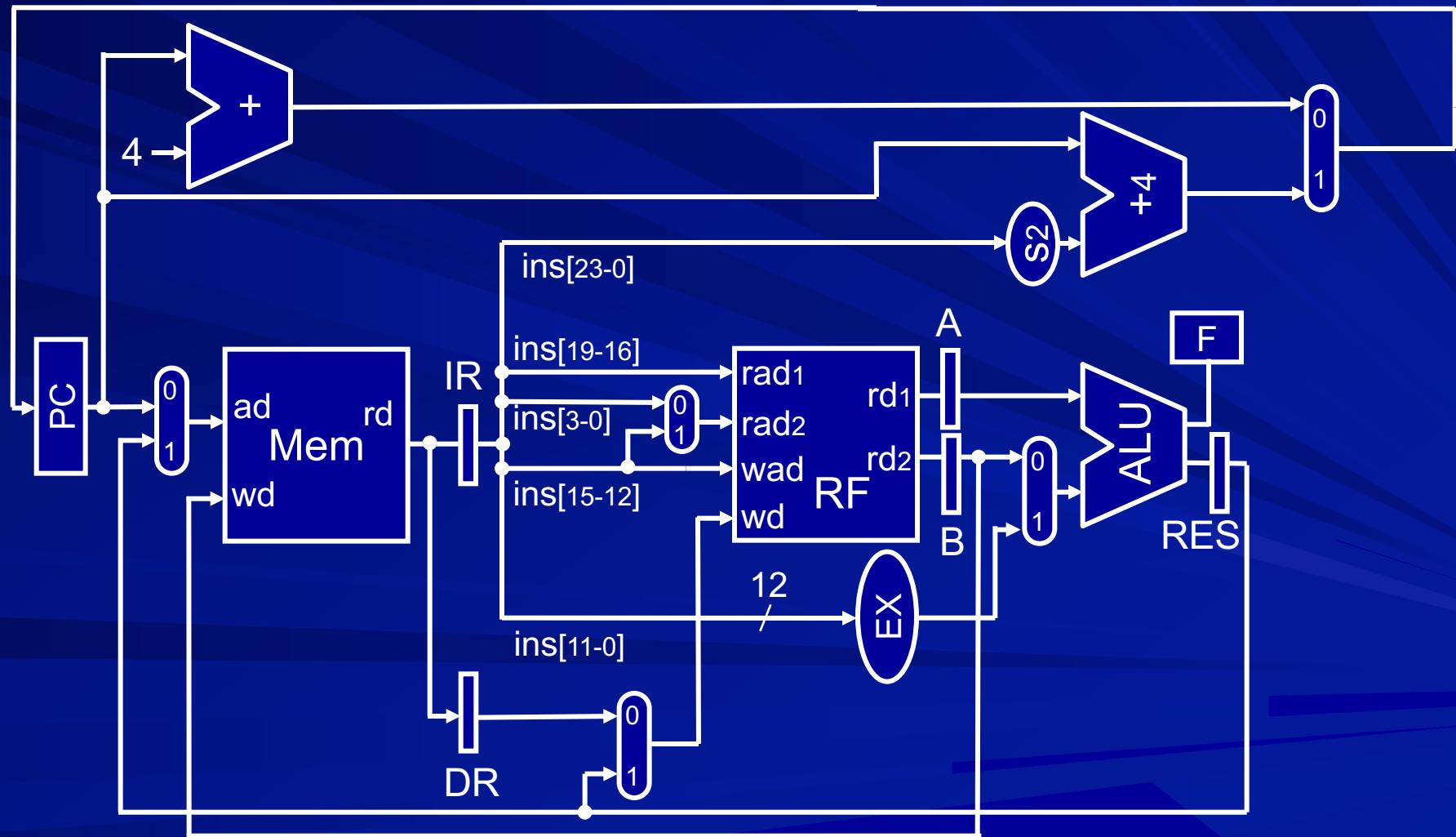
Merge IM and DM



Merge IM and DM

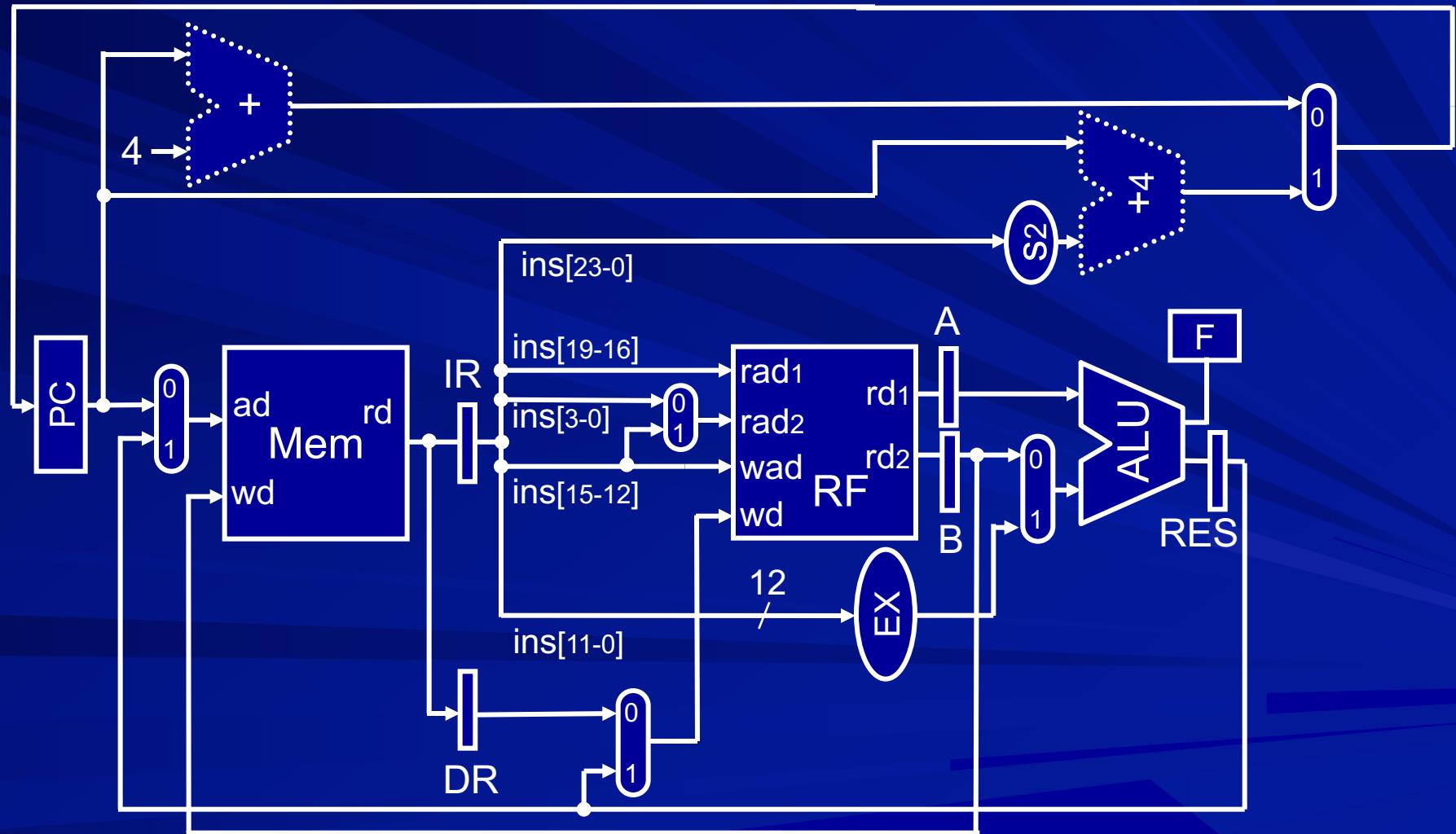


Merge IM and DM

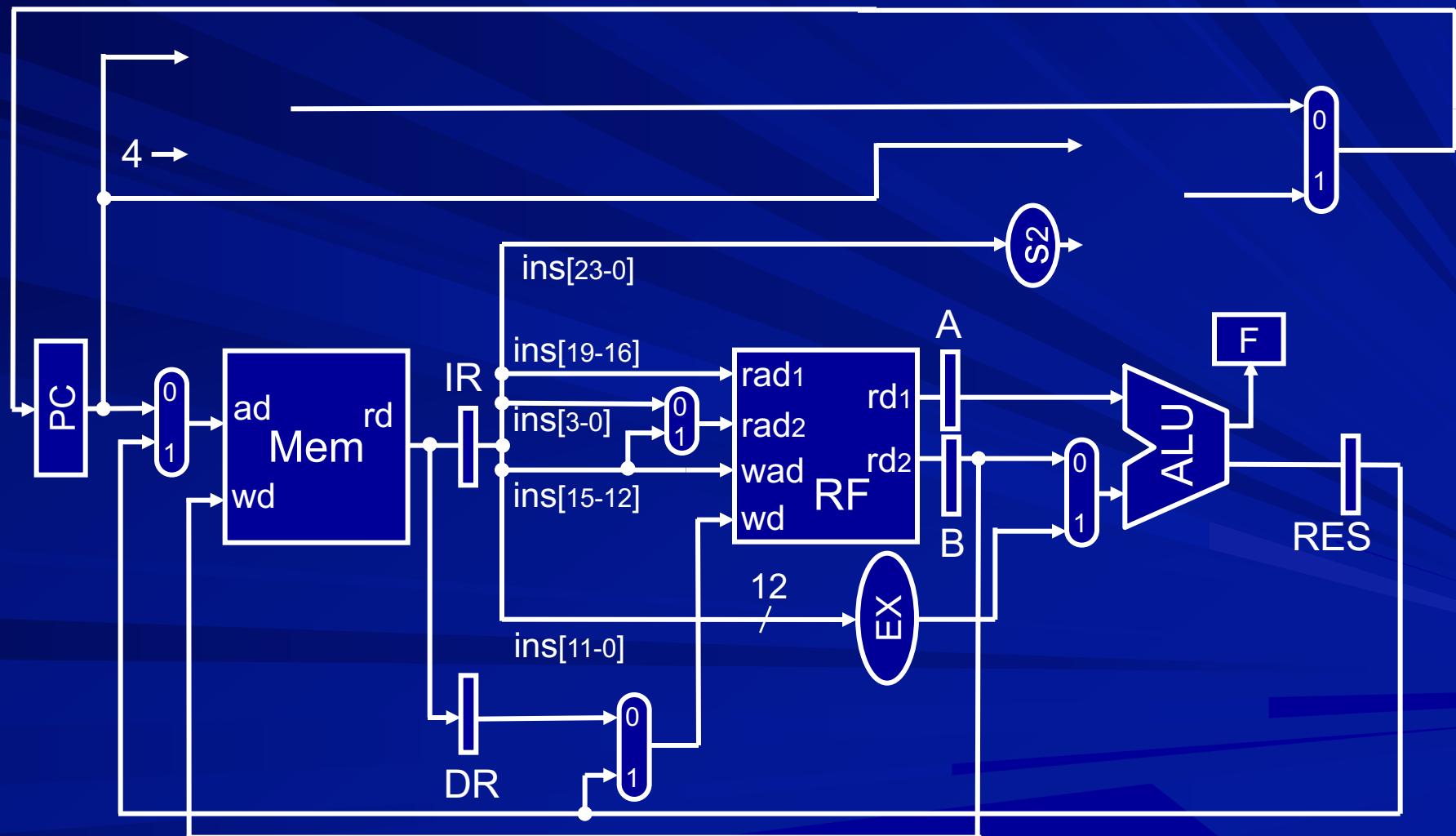


Eliminate adders

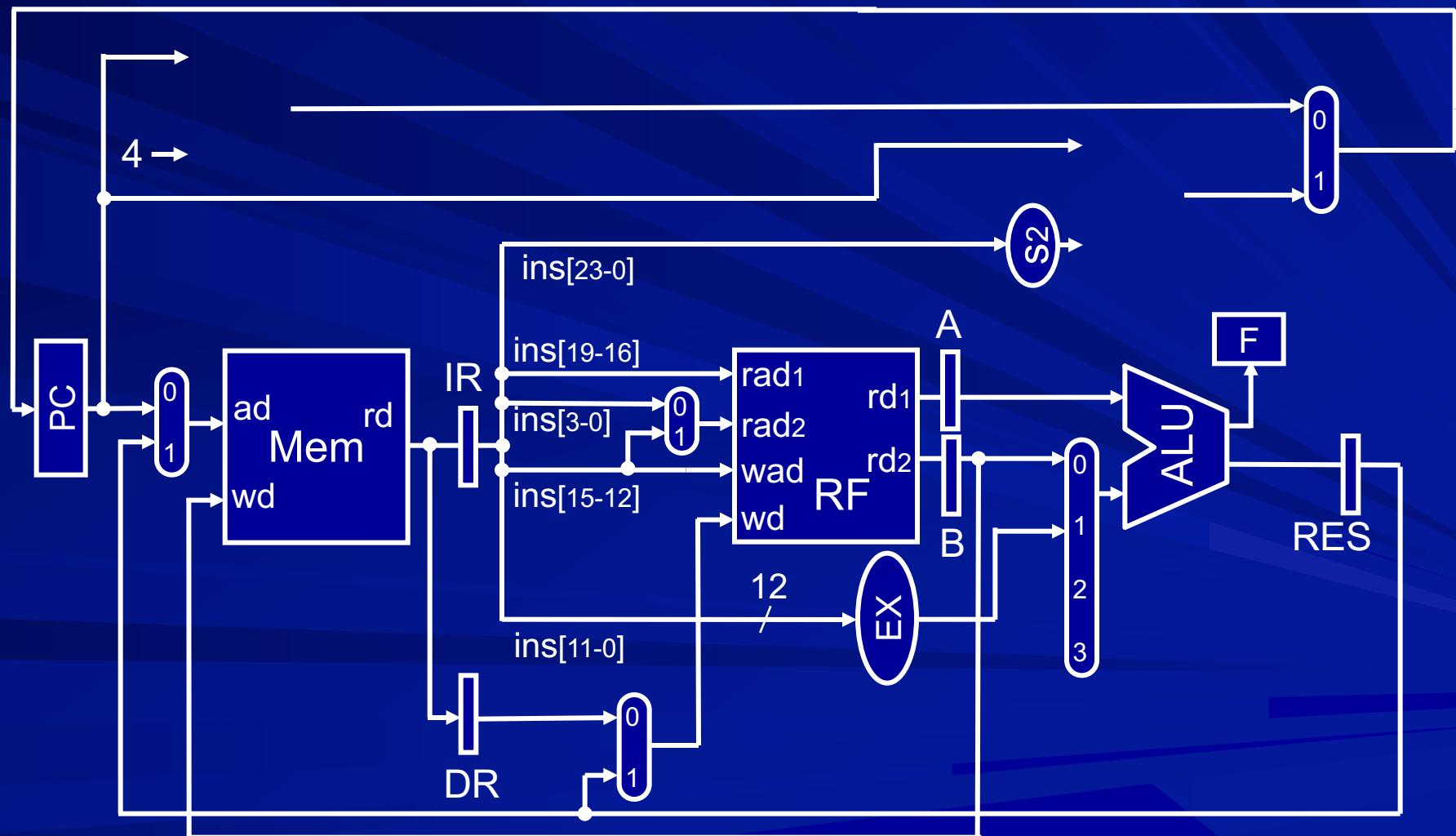
Use ALU



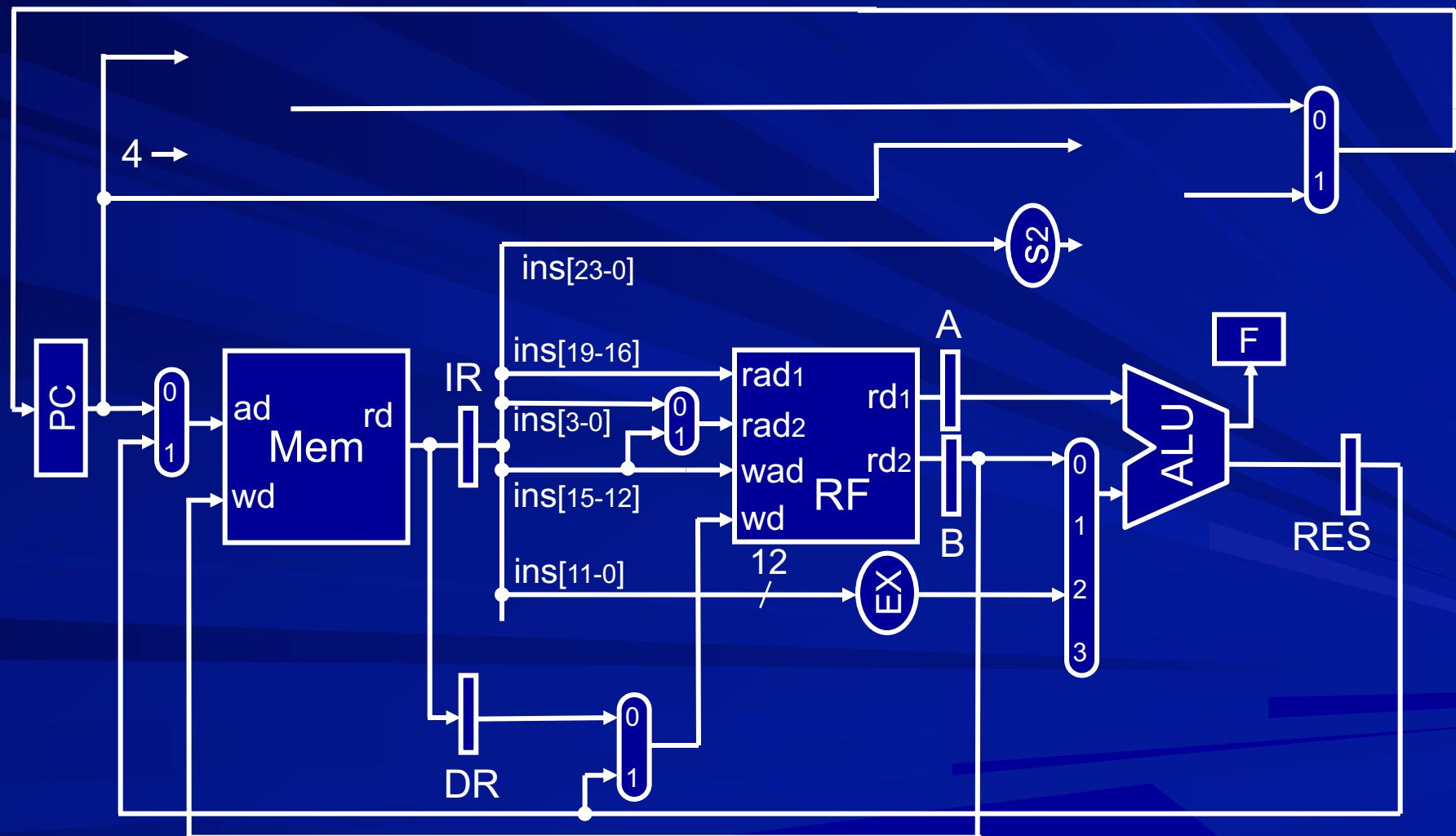
Eliminate adders



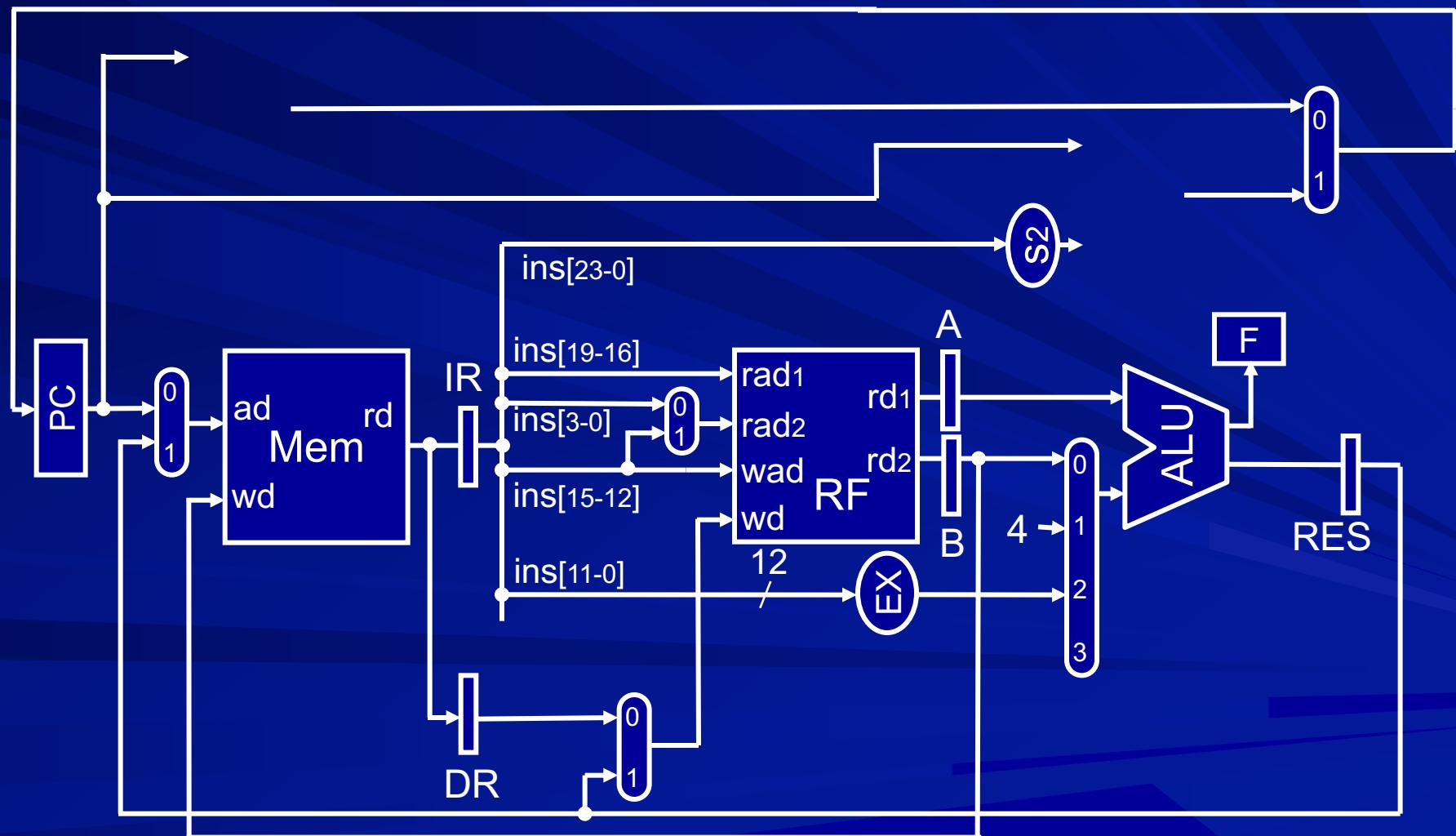
Eliminate adders



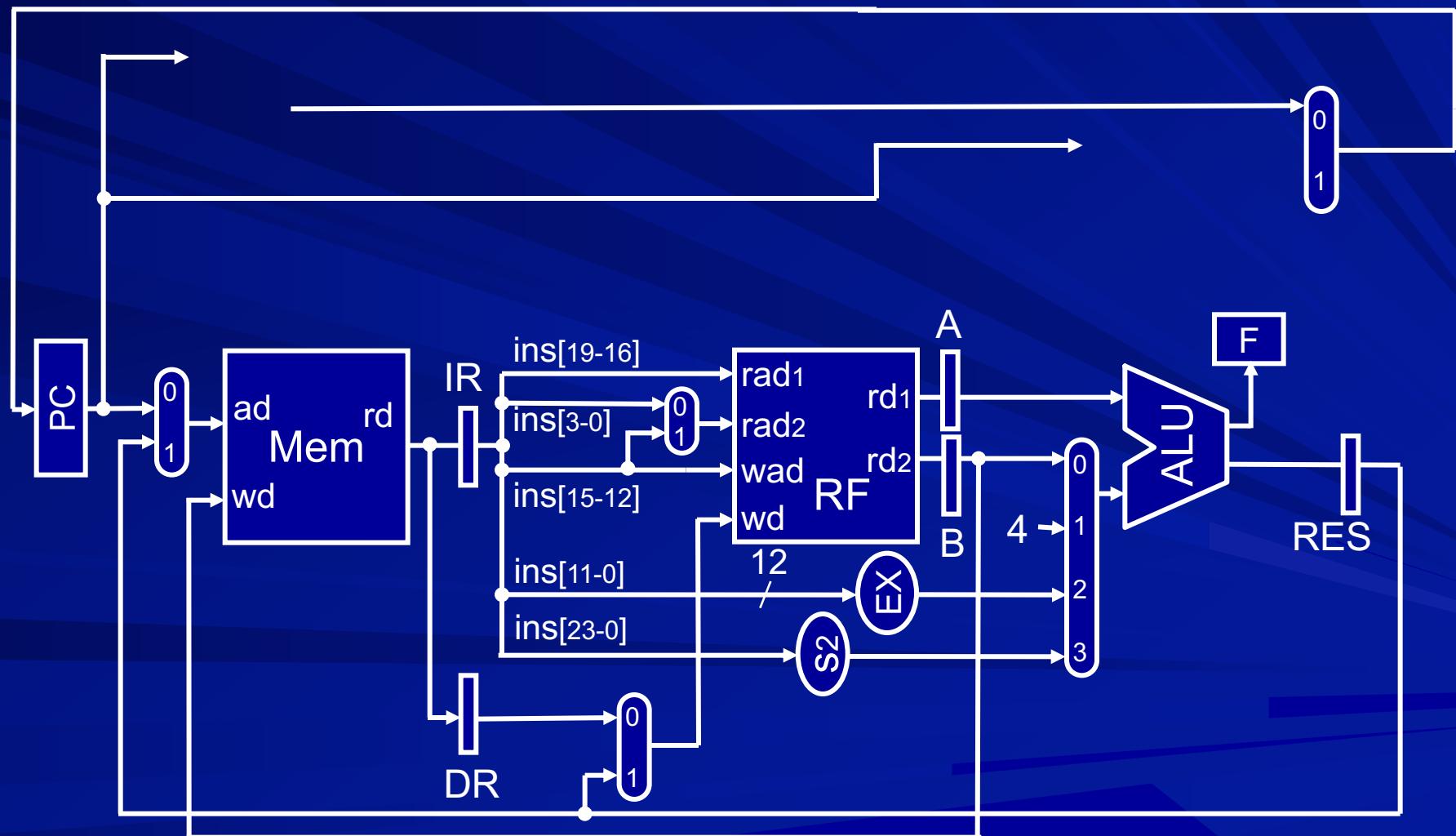
Eliminate adders



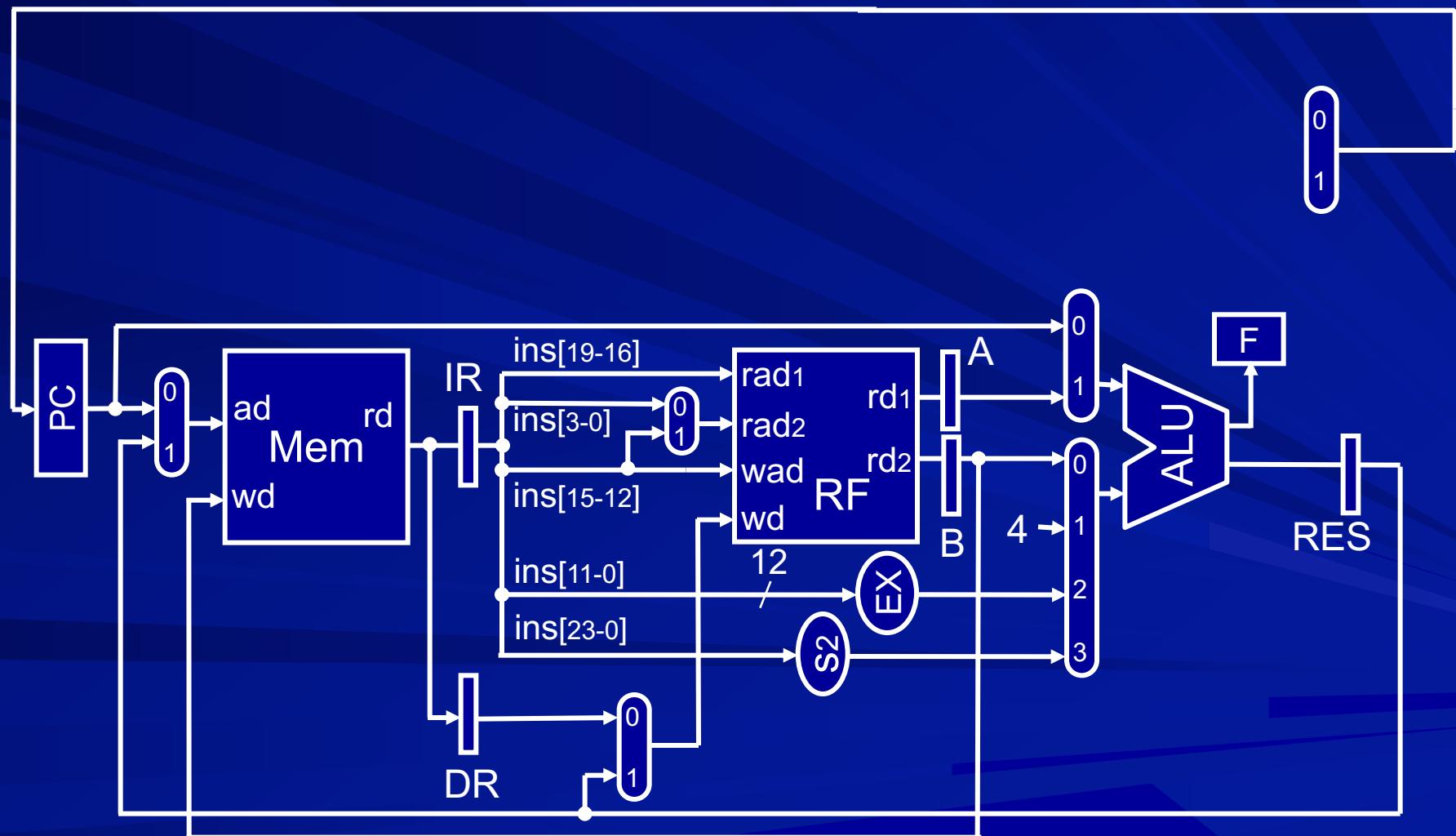
Eliminate adders



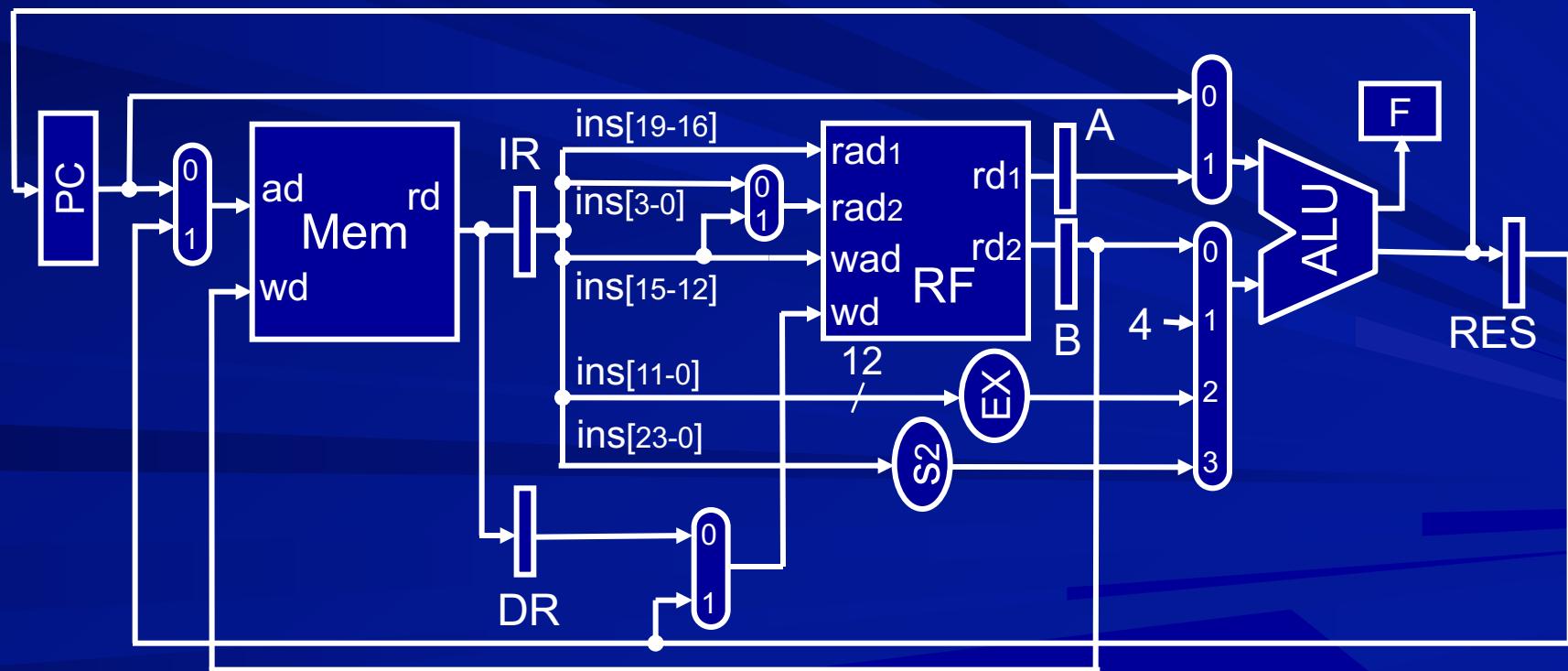
Eliminate adders



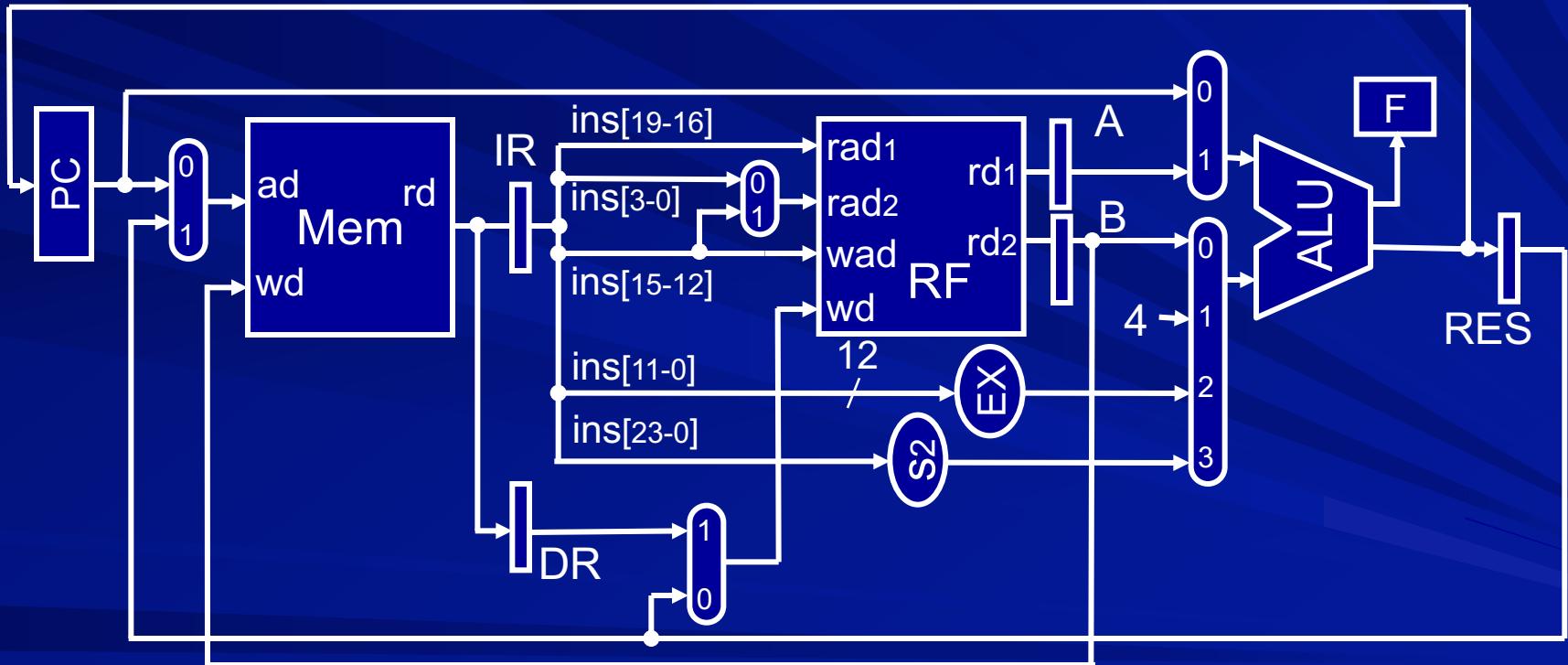
Eliminate adders



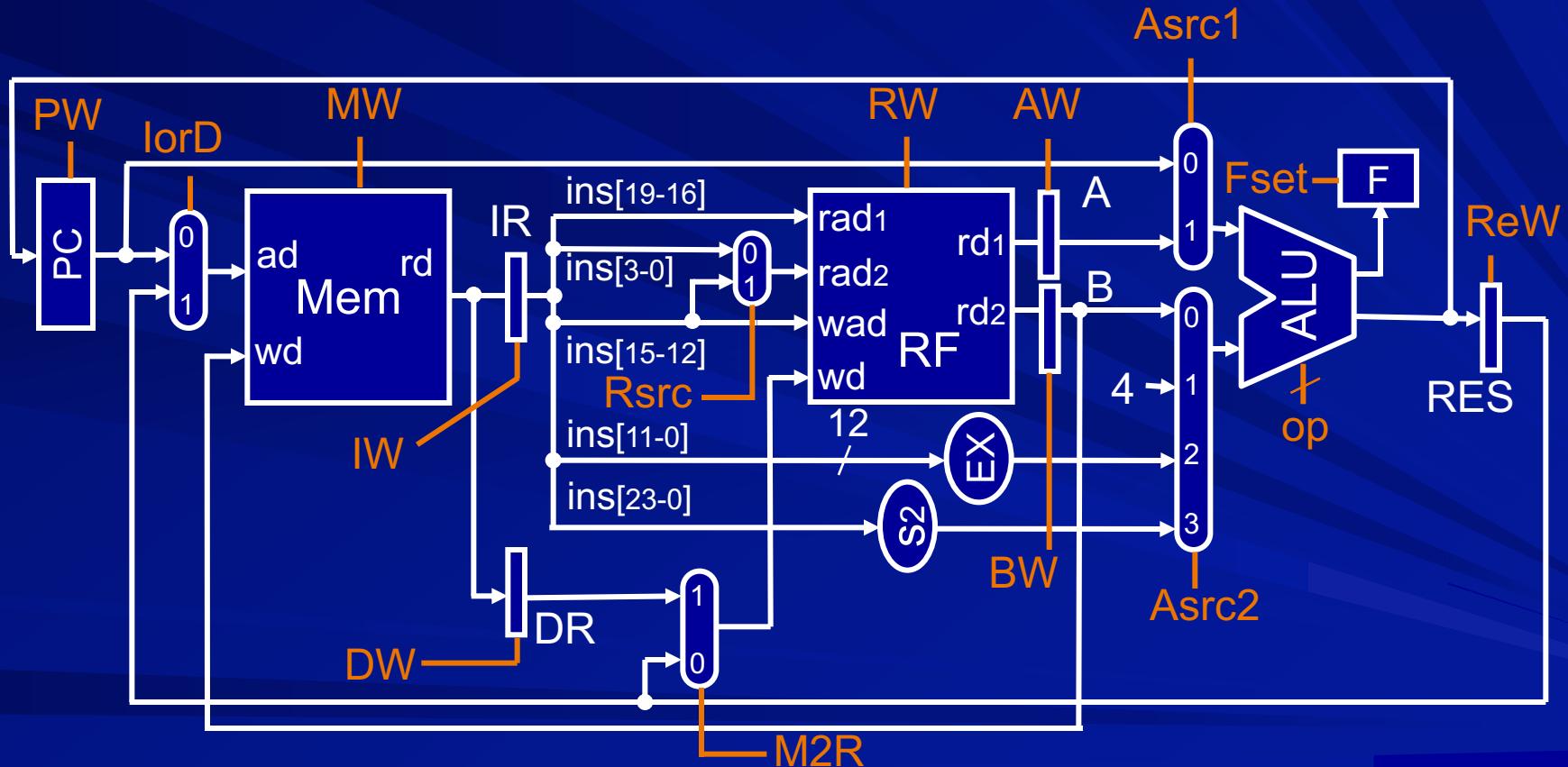
Eliminate adders



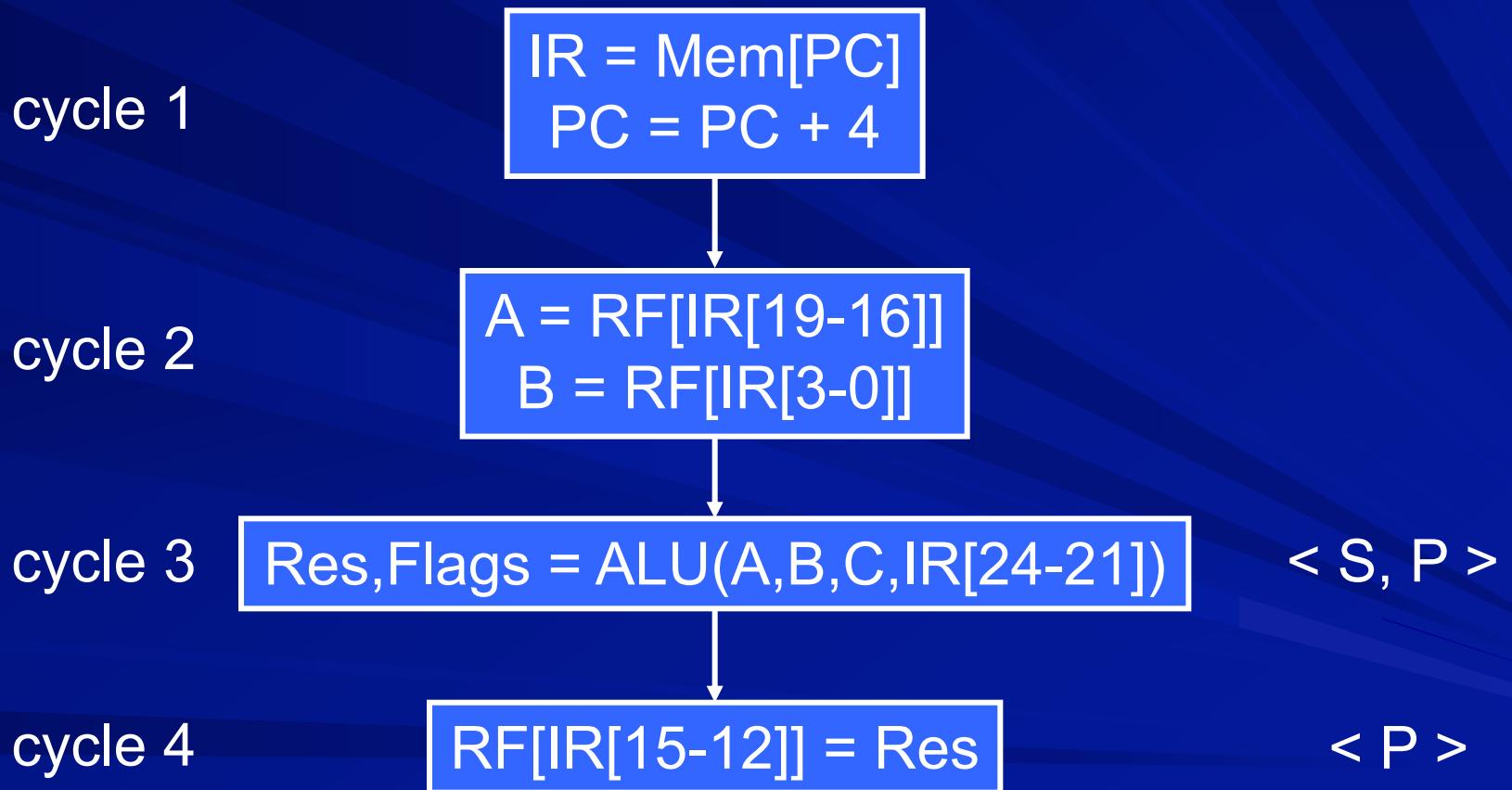
Multi-cycle Datapath



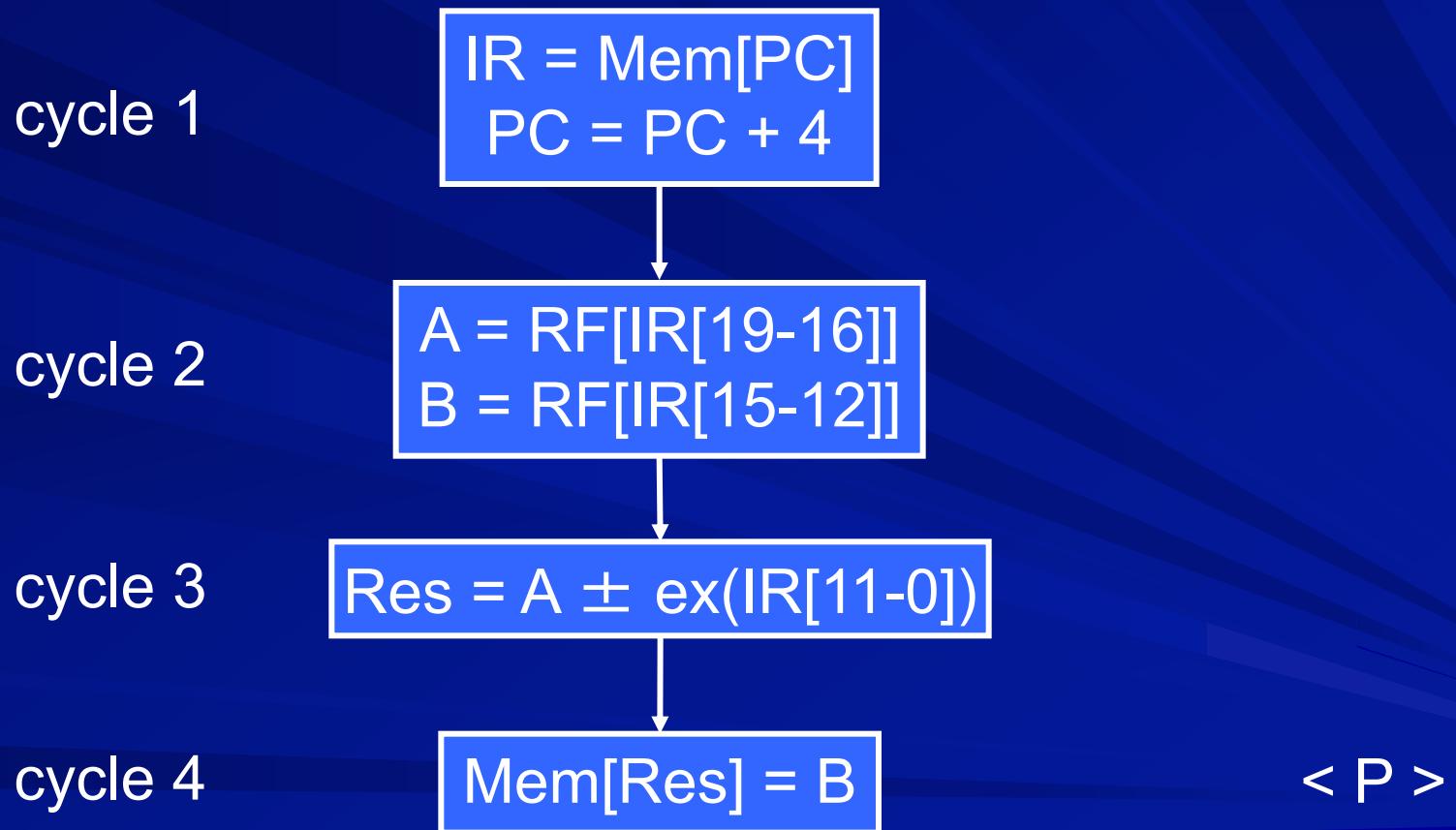
Control signals in multi-cycle datapath



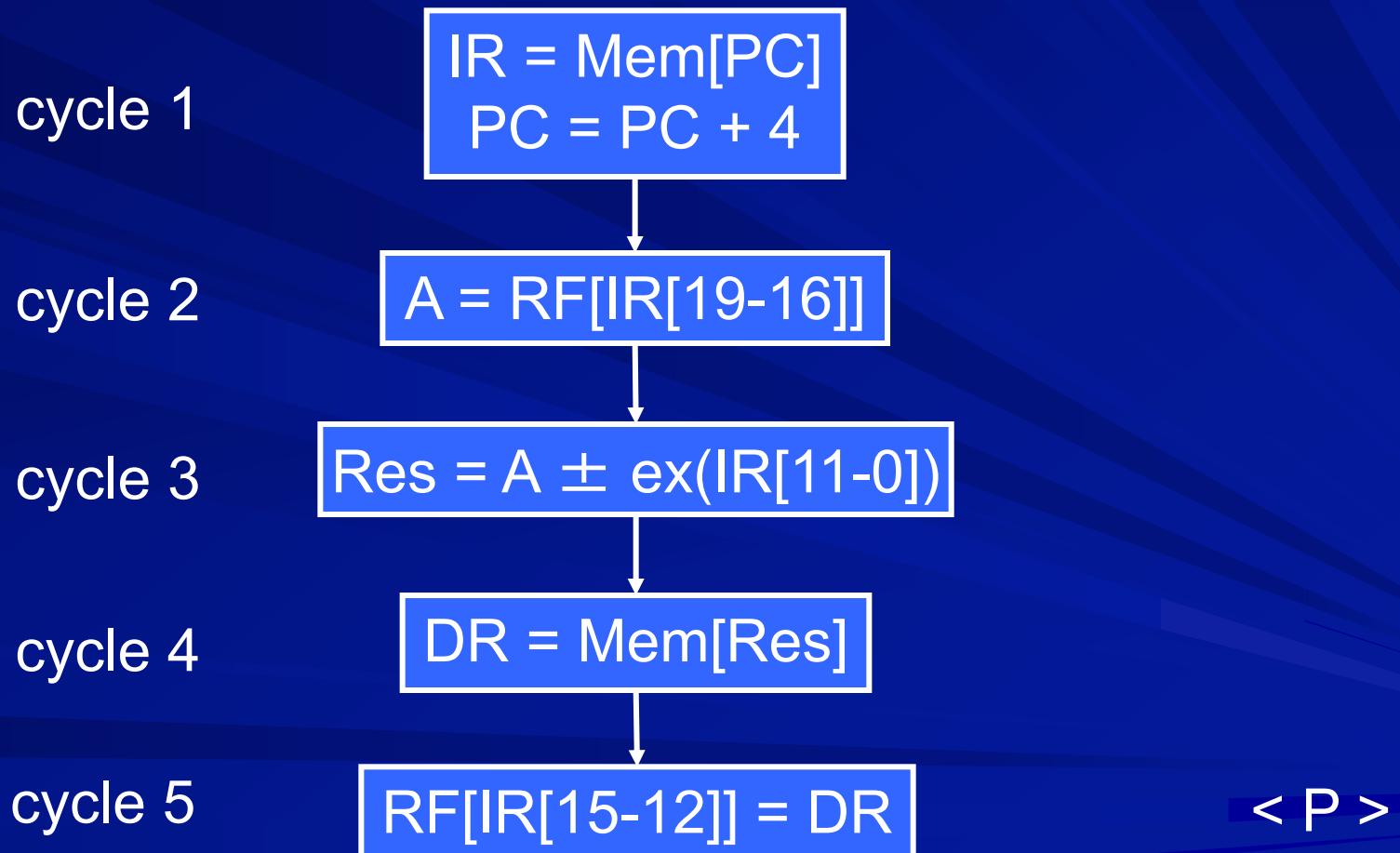
Break Instruction Execution into Cycles: DP instructions



Break Instruction Execution into Cycles: str instruction



Break Instruction Execution into Cycles: ldr instruction



Break Instruction Execution into Cycles: b instruction

cycle 1

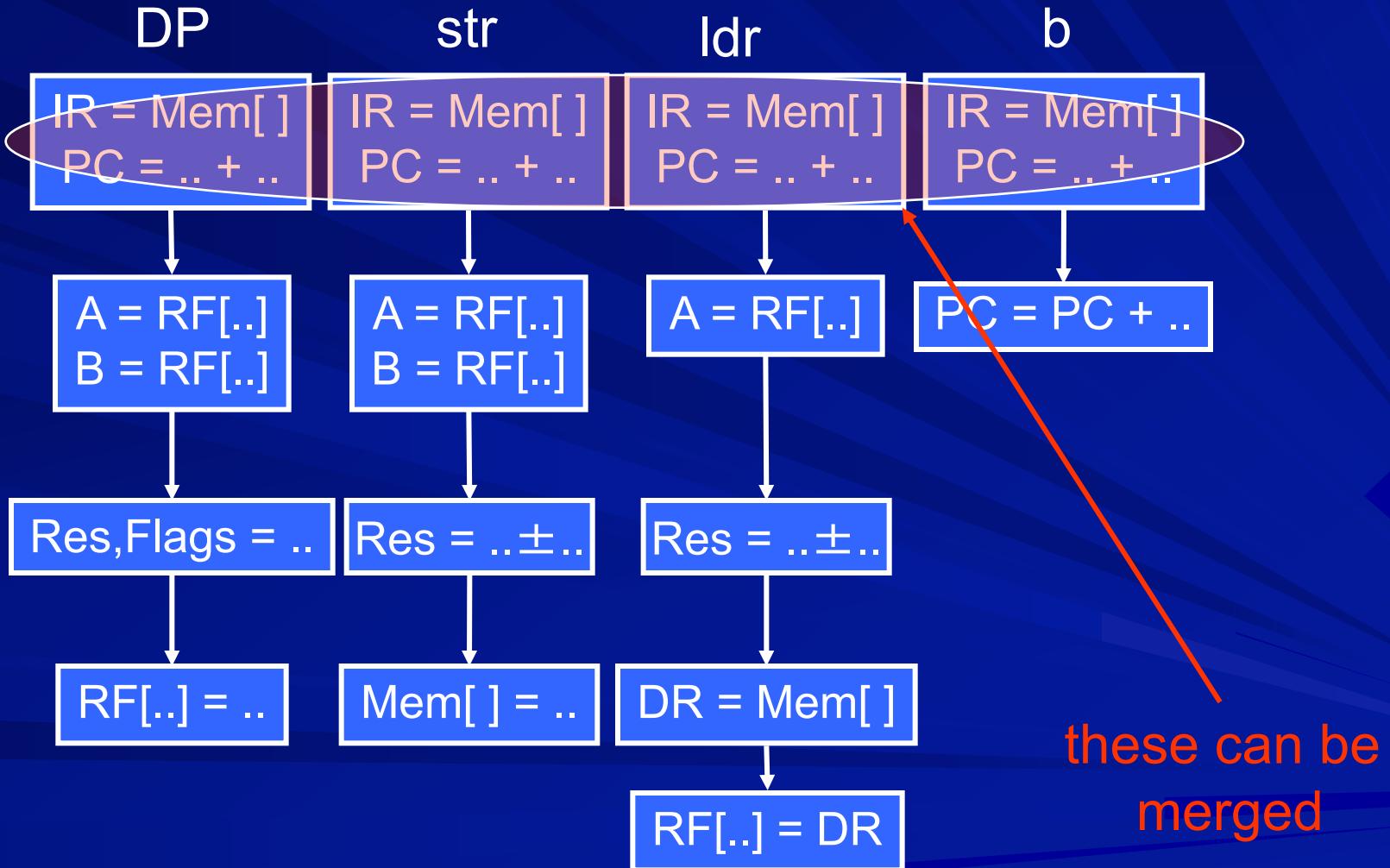
```
IR = Mem[PC]  
PC = PC + 4
```

cycle 2

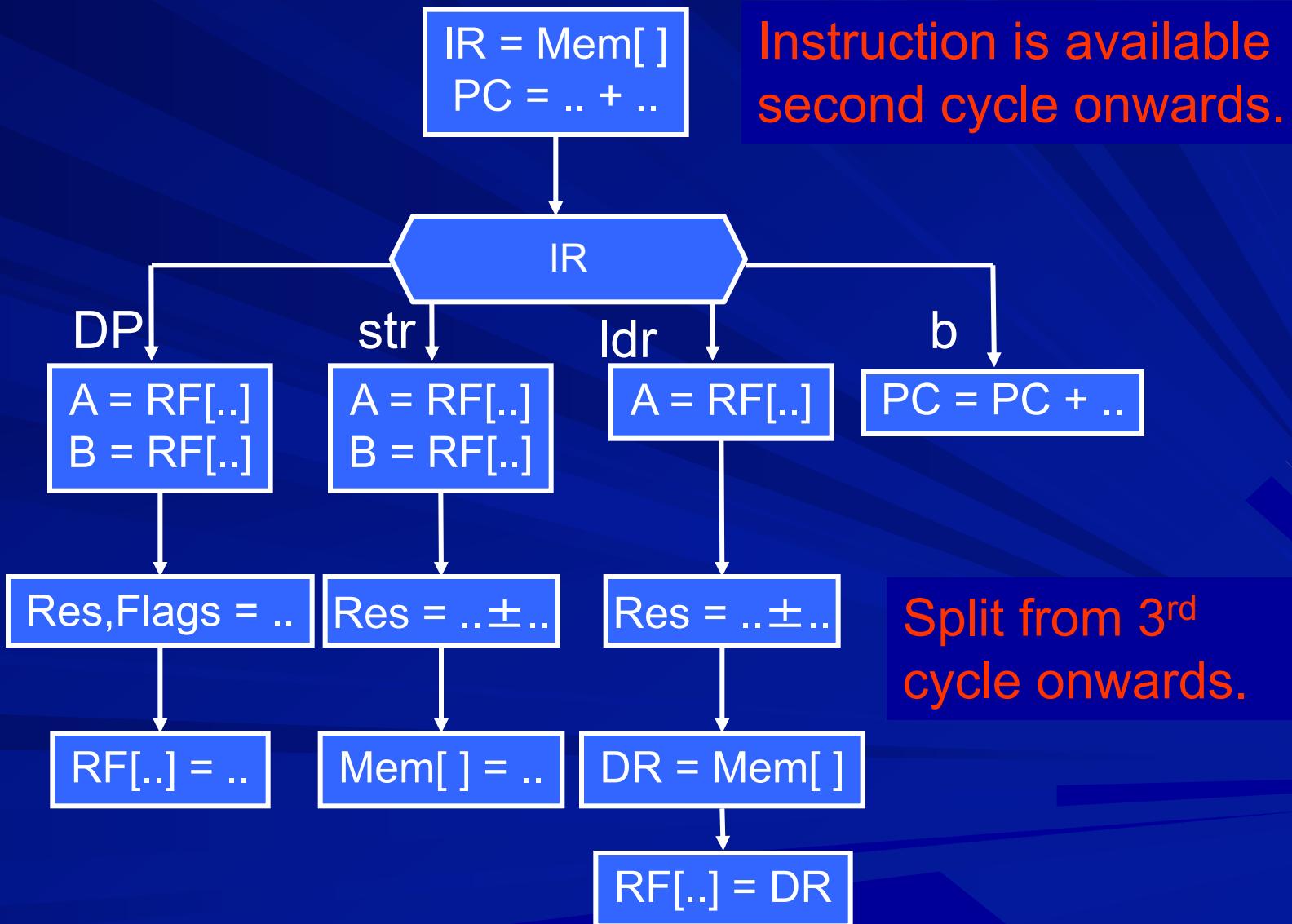
```
PC = PC + S2(IR[23-0]) + 4
```

< P >

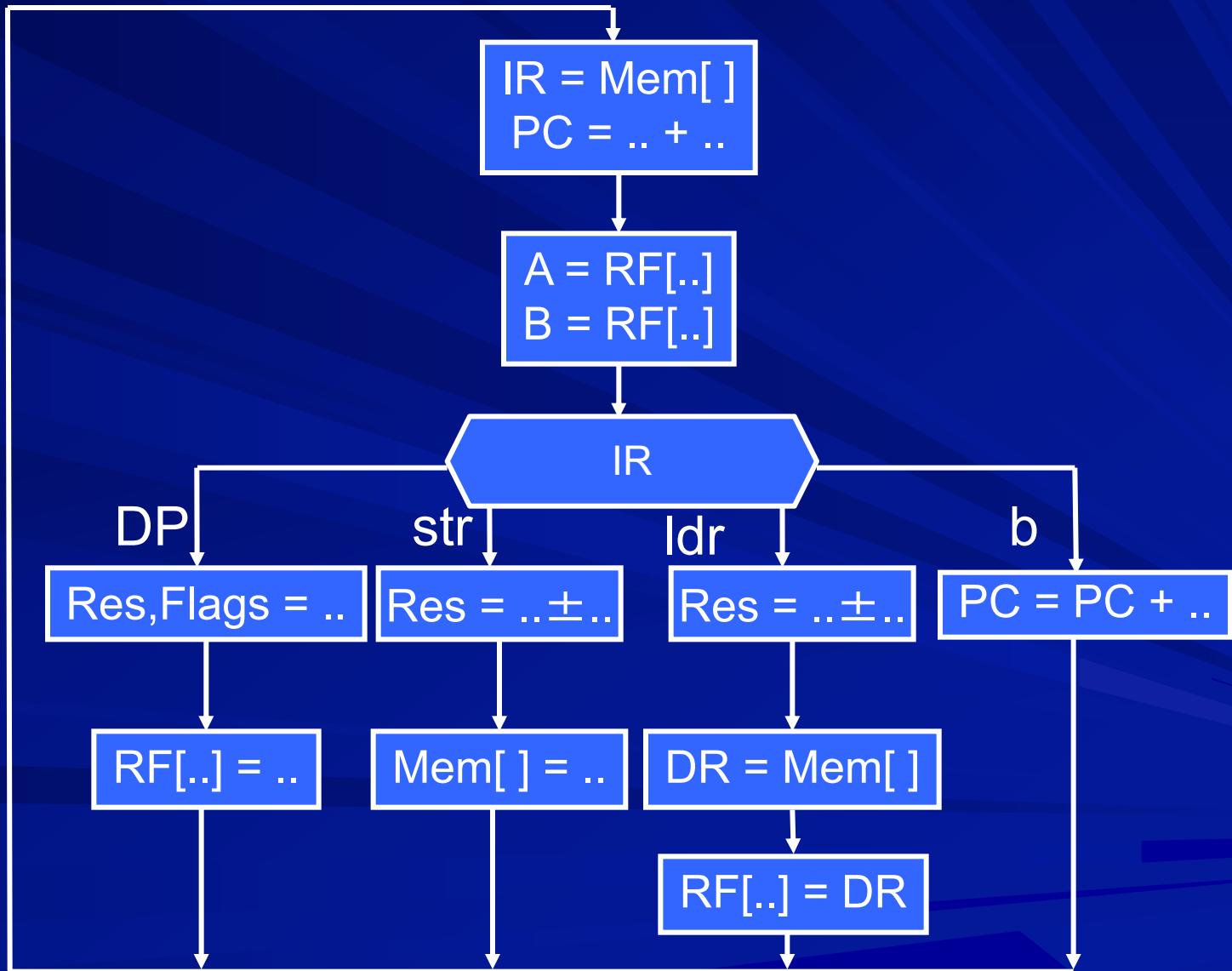
Put cycle sequences together



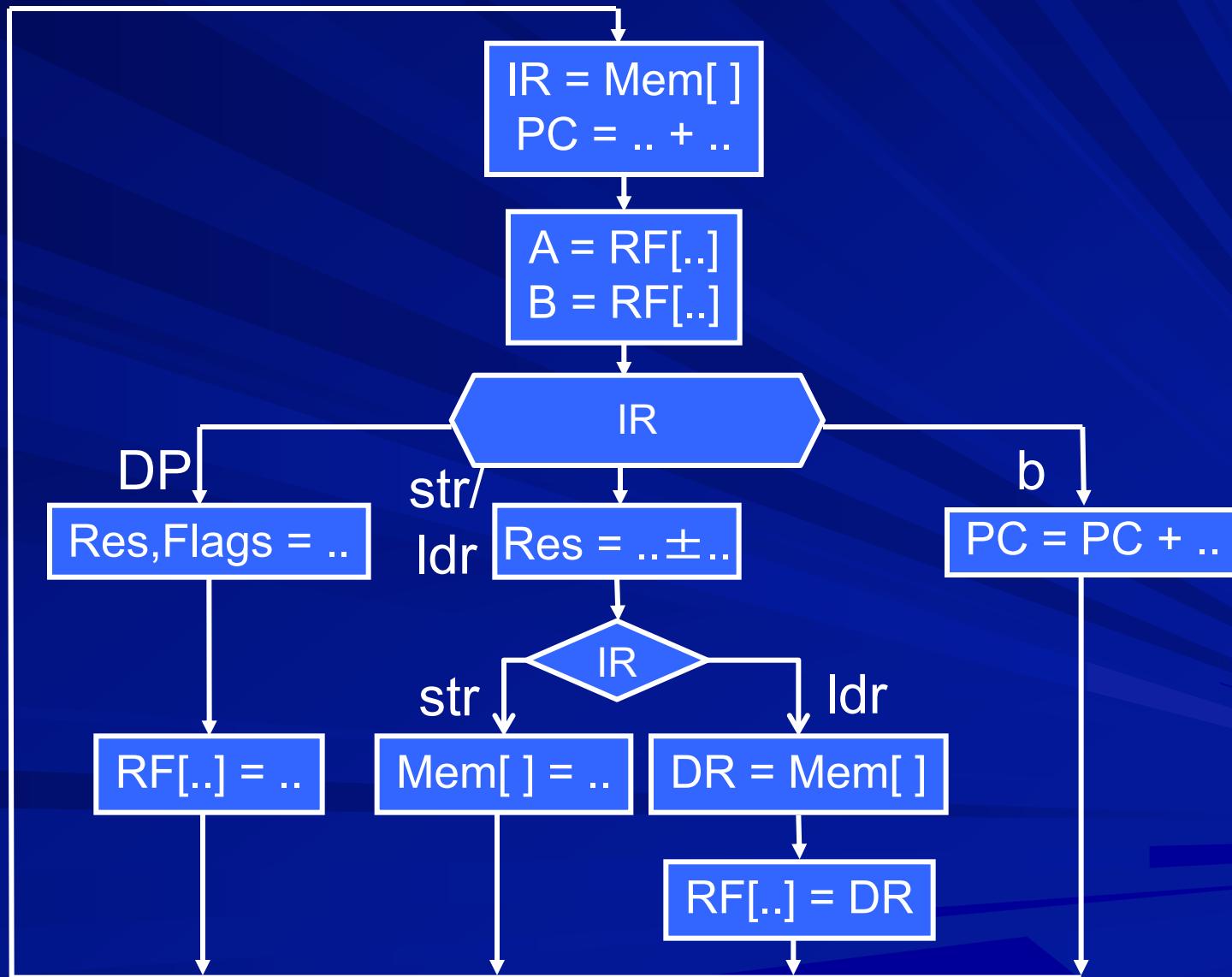
After merging fetch cycle



With a common decoding cycle



ldr, str can split after third cycle



Modified str actions

cycle 1

$$\begin{aligned} \text{IR} &= \text{Mem}[\text{PC}] \\ \text{PC} &= \text{PC} + 4 \end{aligned}$$

cycle 2

$$\begin{aligned} A &= \text{RF}[\text{IR}[19-16]] \\ B &= \text{RF}[\text{IR}[15-12]] \end{aligned}$$

cycle 3

$$\text{Res} = A \pm \text{ex}(\text{IR}[11-0])$$

cycle 4

$$\text{Mem}[\text{Res}] = B$$

$$\begin{aligned} \text{IR} &= \text{Mem}[\text{PC}] \\ \text{PC} &= \text{PC} + 4 \end{aligned}$$

$$\begin{aligned} A &= \text{RF}[\text{IR}[19-16]] \\ B &= \text{RF}[\text{IR}[3-0]] \end{aligned}$$

$$\begin{aligned} \text{Res} &= A \pm \text{ex}(\text{IR}[11-0]) \\ B &= \text{RF}[\text{IR}[15-12]] \end{aligned}$$

$$\text{Mem}[\text{Res}] = B$$

Modified ldr actions

cycle 1

$$\begin{aligned} \text{IR} &= \text{Mem}[\text{PC}] \\ \text{PC} &= \text{PC} + 4 \end{aligned}$$

cycle 2

$$A = \text{RF}[\text{IR}[19-16]]$$

cycle 3

$$\text{Res} = A \pm \text{ex}(\text{IR}[11-0])$$

cycle 4

$$\text{DR} = \text{Mem}[\text{Res}]$$

cycle 5

$$\text{RF}[\text{IR}[15-12]] = \text{DR}$$

$$\begin{aligned} \text{IR} &= \text{Mem}[\text{PC}] \\ \text{PC} &= \text{PC} + 4 \end{aligned}$$

$$\begin{aligned} A &= \text{RF}[\text{IR}[19-16]] \\ B &= \text{RF}[\text{IR}[3-0]] \end{aligned}$$

$$\begin{aligned} \text{Res} &= A \pm \text{ex}(\text{IR}[11-0]) \\ B &= \text{RF}[\text{IR}[15-12]] \end{aligned}$$

$$\text{DR} = \text{Mem}[\text{Res}]$$

$$\text{RF}[\text{IR}[15-12]] = \text{DR}$$

Modified b actions

cycle 1

```
IR = Mem[PC]  
PC = PC + 4
```

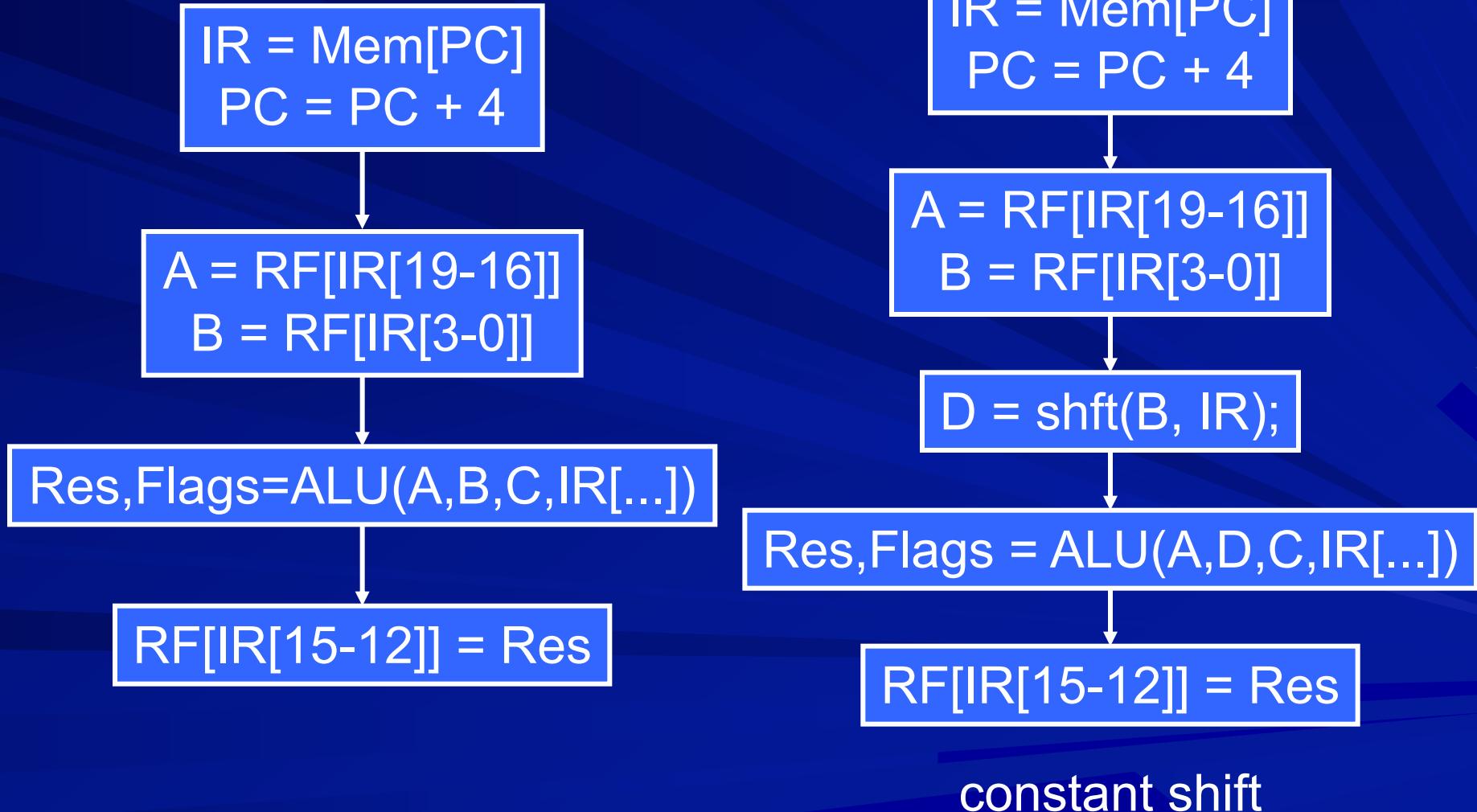
cycle 2

```
A = RF[IR[19-16]]  
B = RF[IR[3-0]]
```

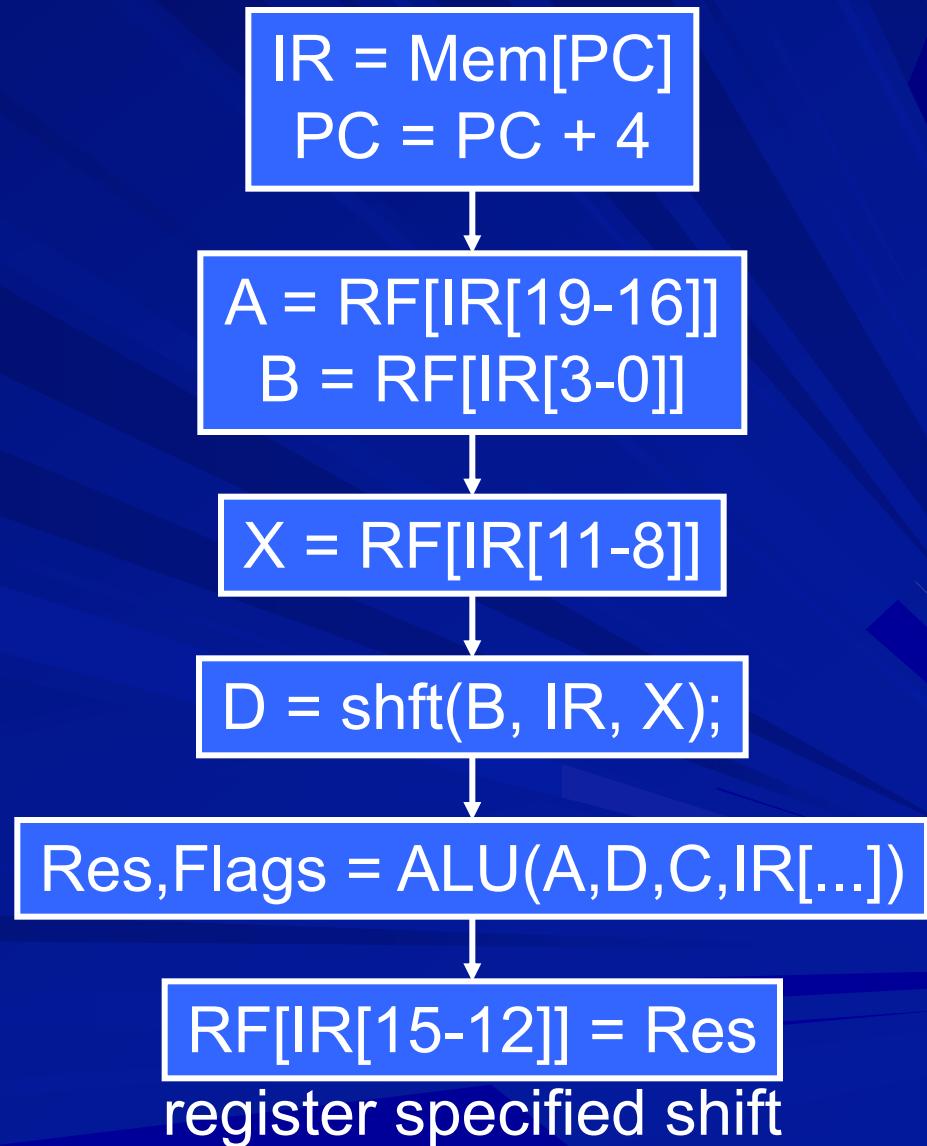
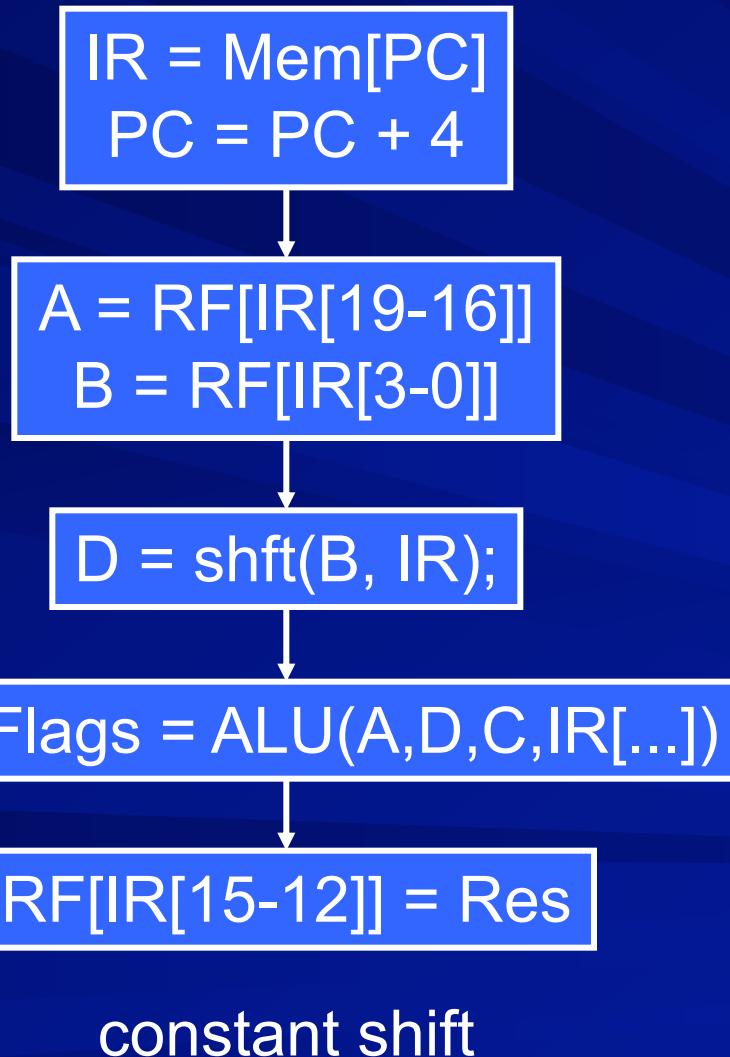
cycle 3

```
PC = PC + S2(IR[23-0]) + 4
```

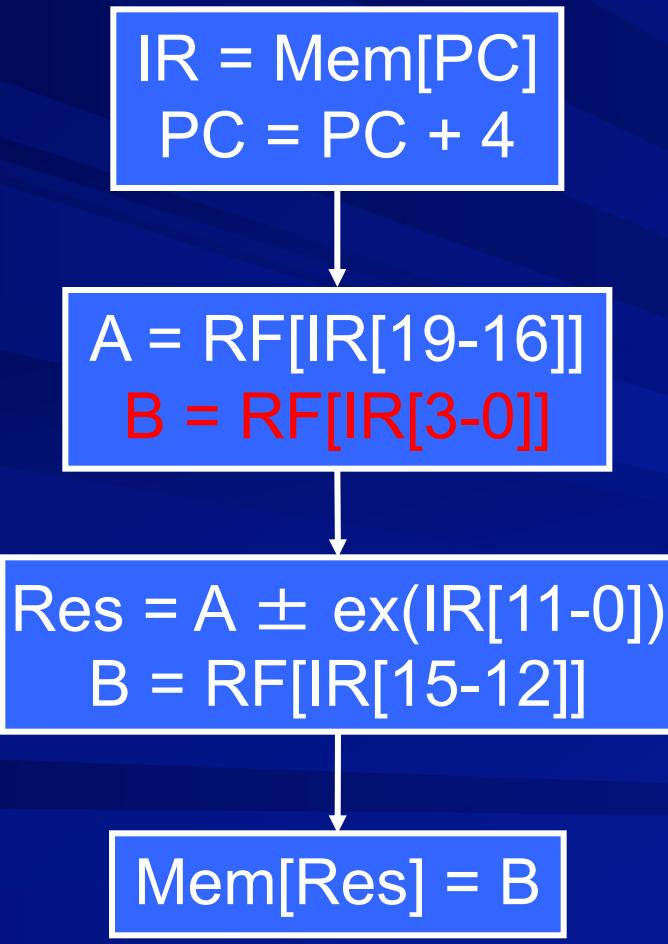
Other forms of DP instructions



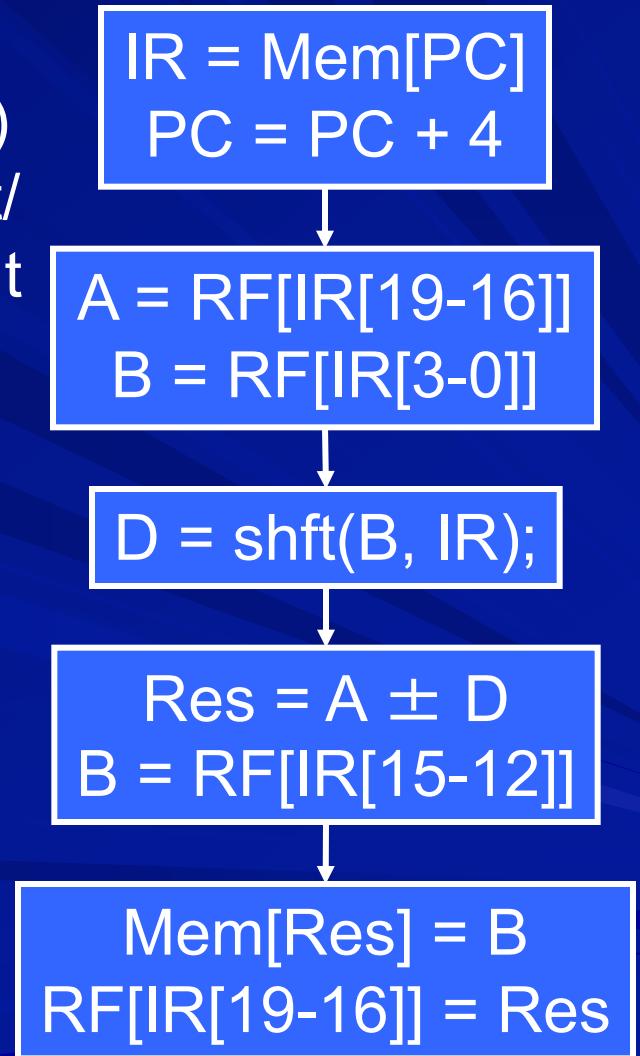
Other forms of DP instructions



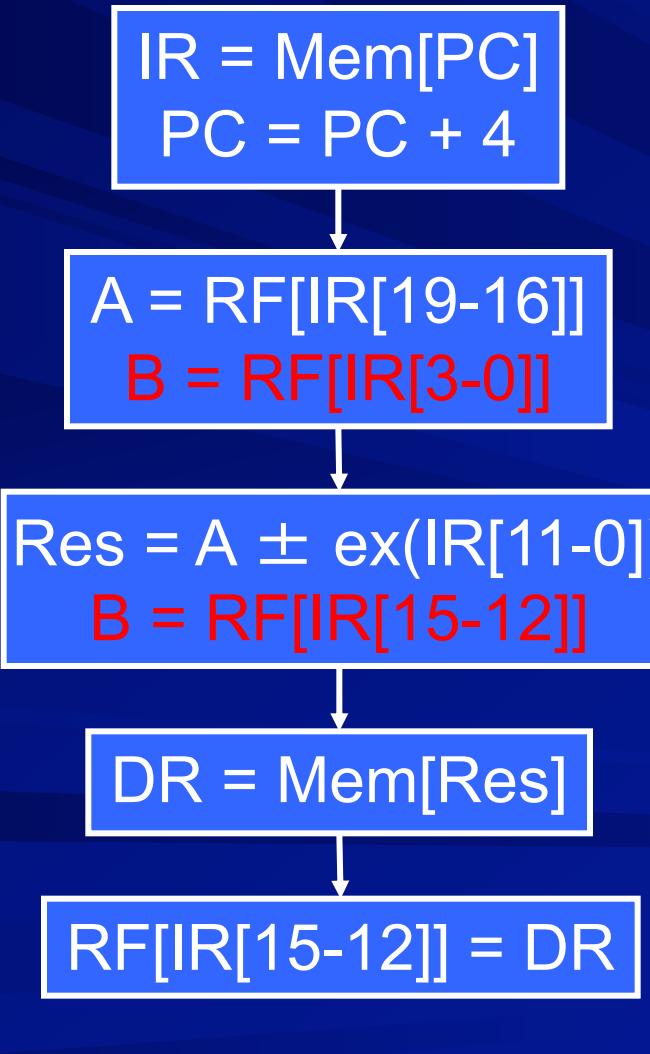
Other forms of str instruction



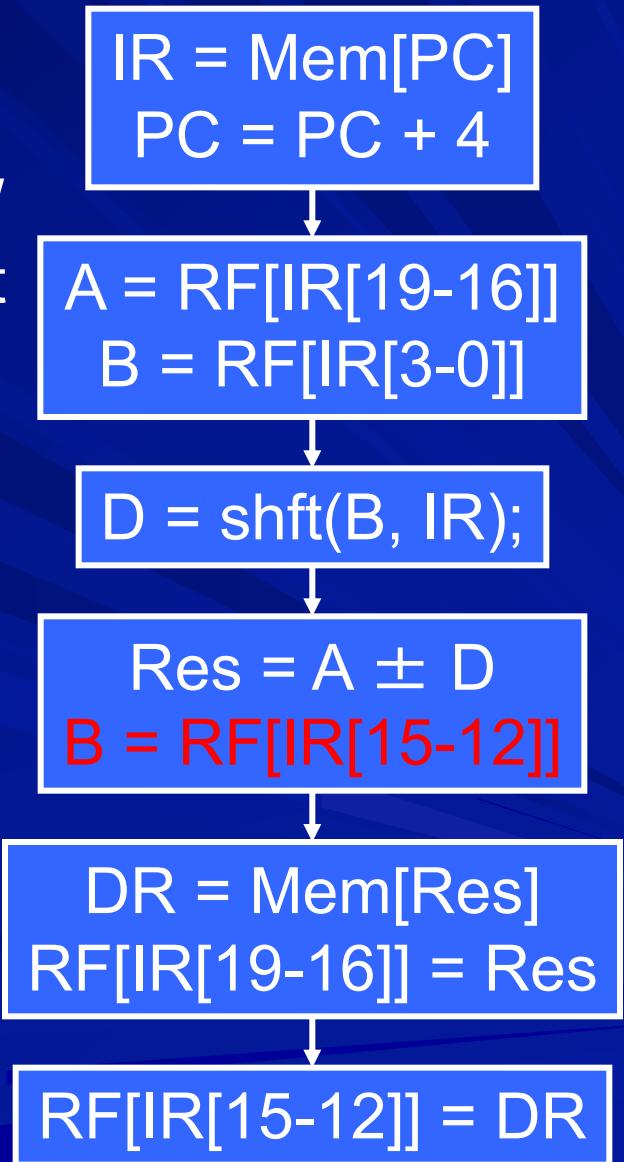
auto (pre)
increment/
decrement
register
offset



Other forms of ldr instruction



auto (pre)
increment/
decrement
register
offset



Thanks

COL216

Computer Architecture

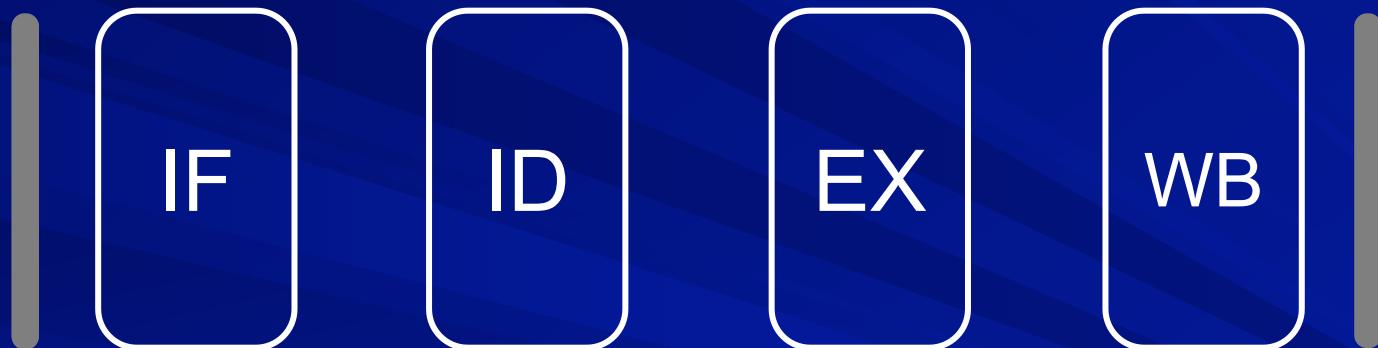
Processor design -
Pipelining

7th February, 2022

Outline of this lecture

- Timings of Single cycle, multi-cycle designs
- Increasing performance with pipelining
- Limitations of pipelined approach
 - Hazards
- Handling hazards
 - Removing hazards
 - Reducing effect of hazards

Single cycle design



Problems with single cycle design

- Slowest instruction pulls down the clock frequency
- Resource utilization is poor
- There are some instructions which are impossible to be implemented in this manner

Multi-cycle design



Features of multi-cycle design

- Actions split into multiple steps
- Registers hold intermediate values
- All instructions need not take same steps
- Clock frequency decided by slowest step
- Resources can be shared across cycles
 - Eliminate adders
 - Use single memory
 - More multiplexing

Improving the design further

If resource saving is not important,
can the performance be improved further?

Pipelined design



Resource usage in different designs

■ Single cycle design

- each resource is tied up for the entire duration of the instruction execution

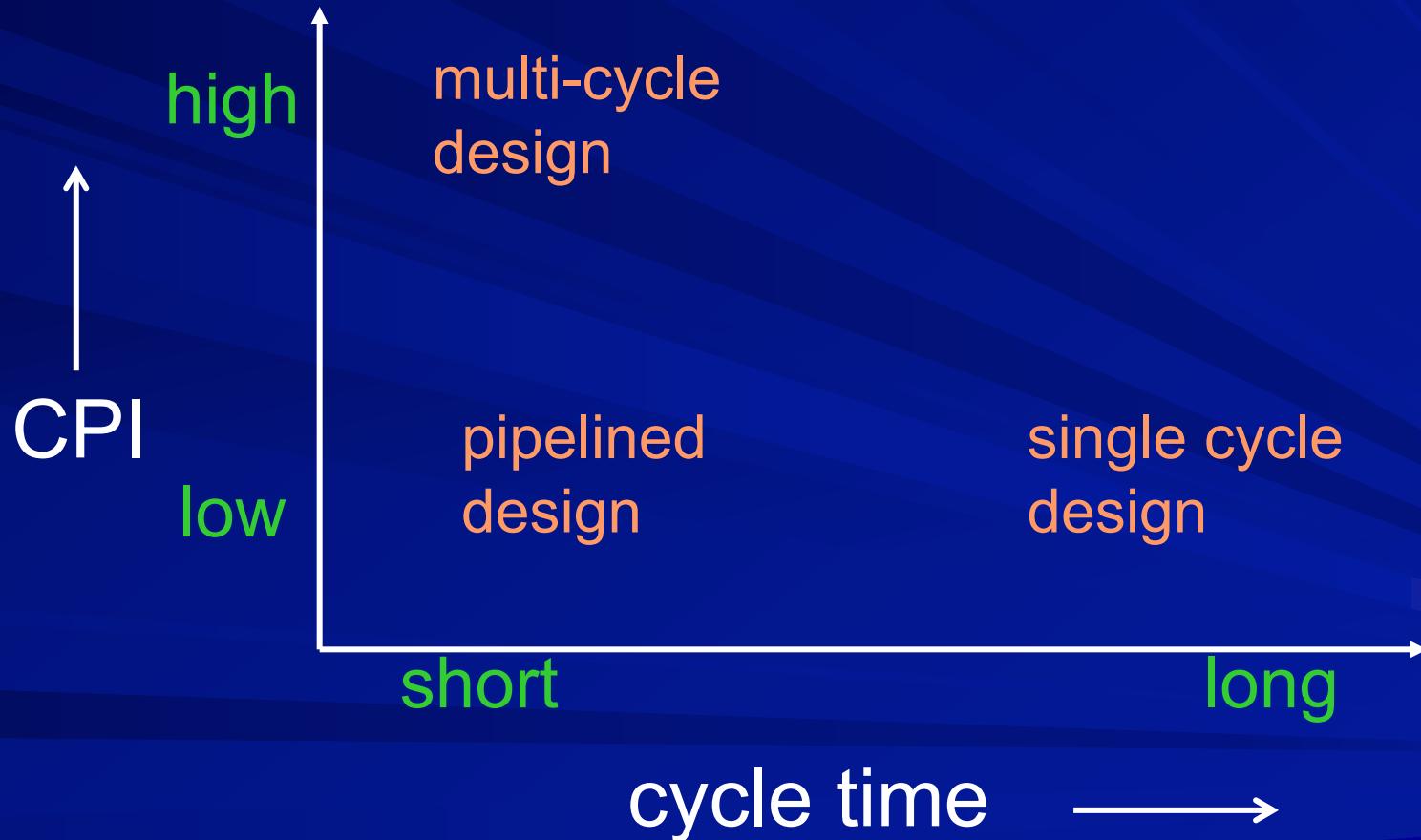
■ Multi-cycle design

- resource utilized in cycle t of instruction I is available again for cycle $t+1$ of instruction I

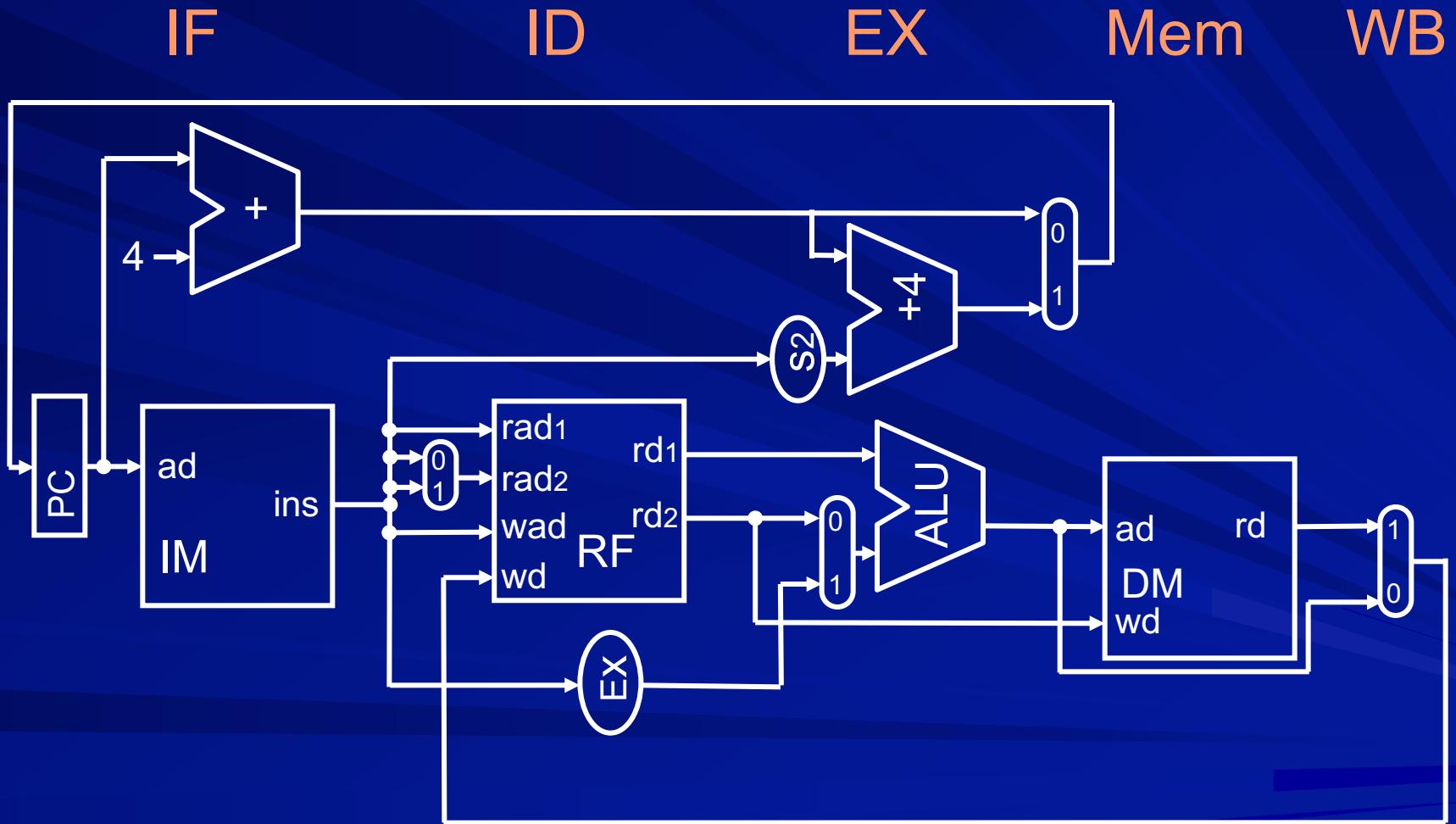
■ Pipelined design

- resource utilized in cycle t of instruction I is available again for cycle t of instruction $I+1$

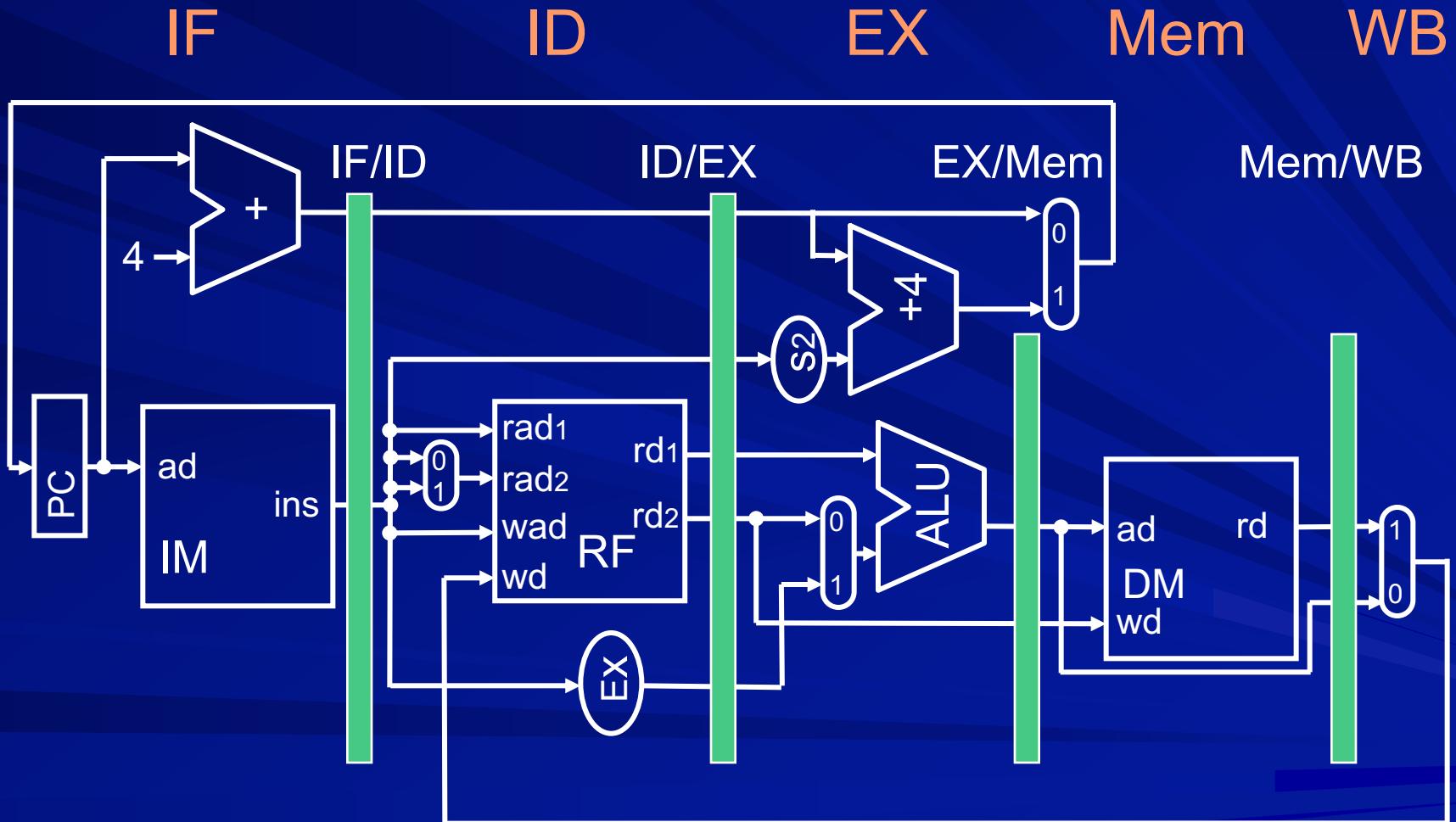
Performance of different designs



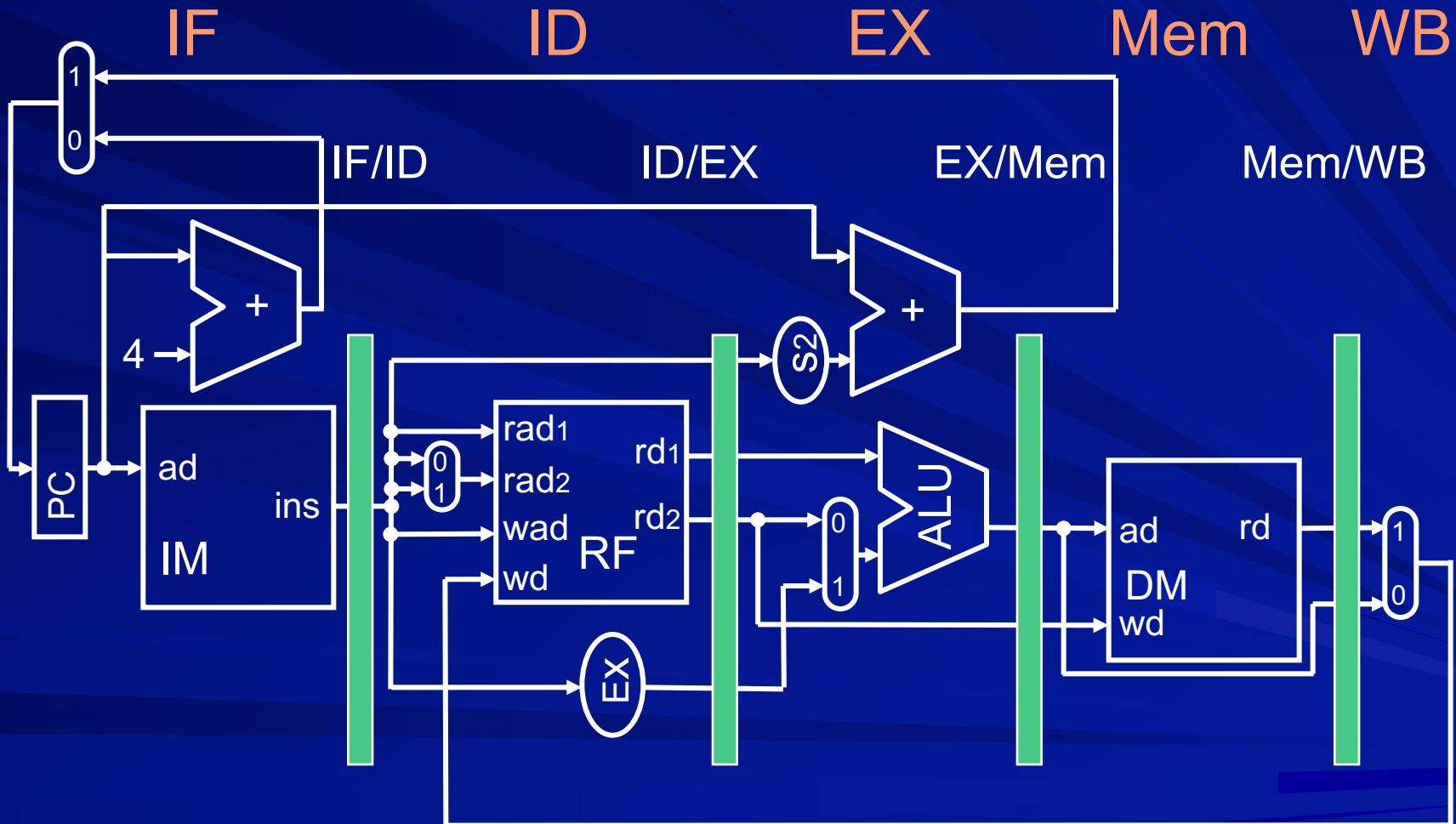
Single cycle datapath



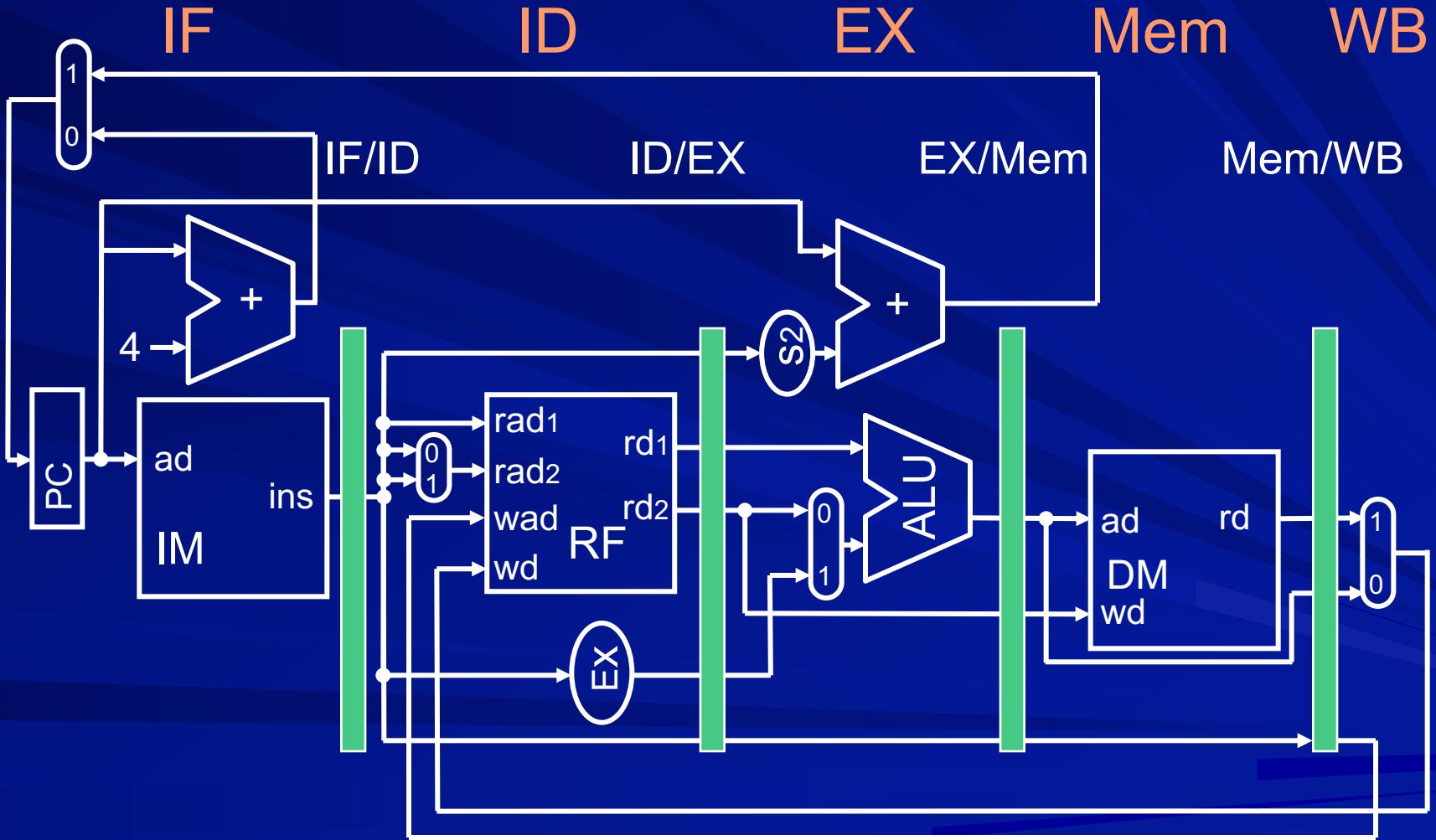
Pipelined datapath



Fetch new instruction every cycle



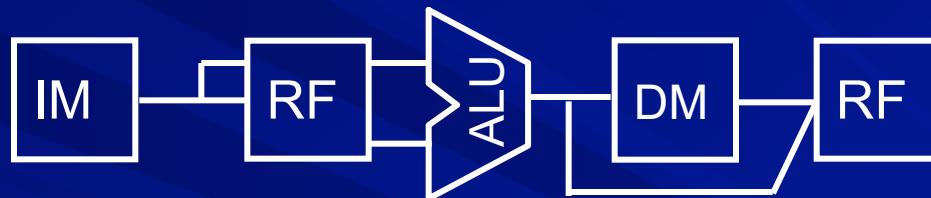
Correction for WB stage



Graphical representation

5 stage pipeline

stages

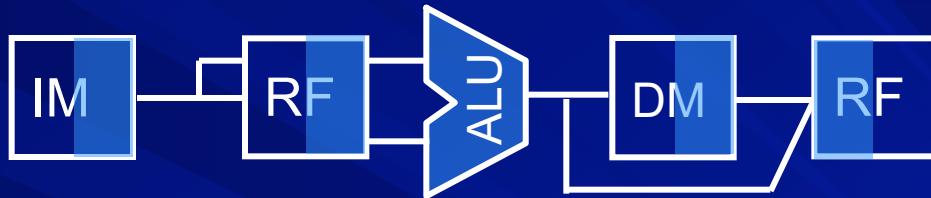


actions

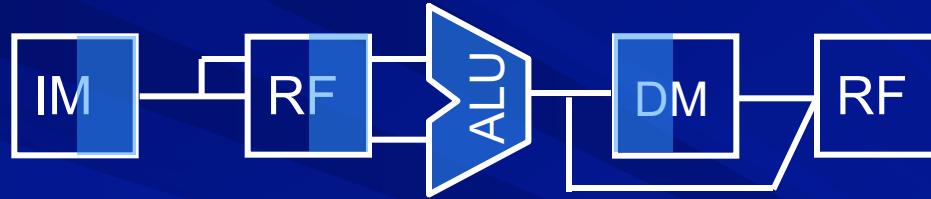


Usage of stages by instructions

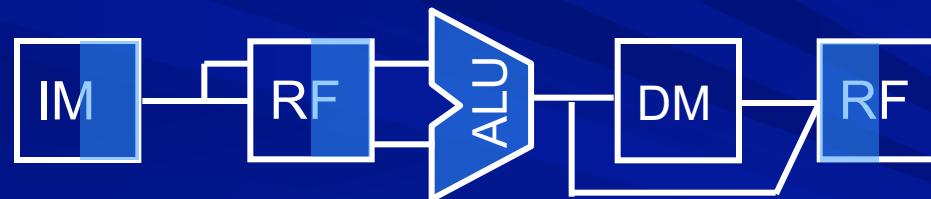
ldr



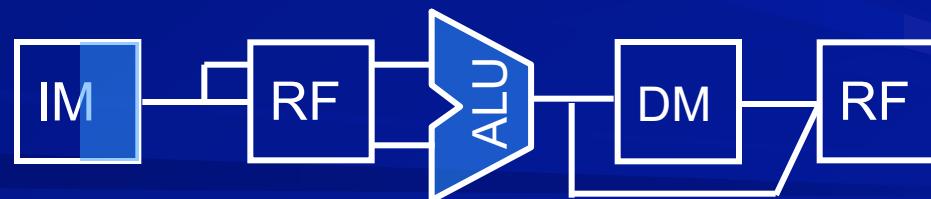
str



add



b



Representing pipelined execution

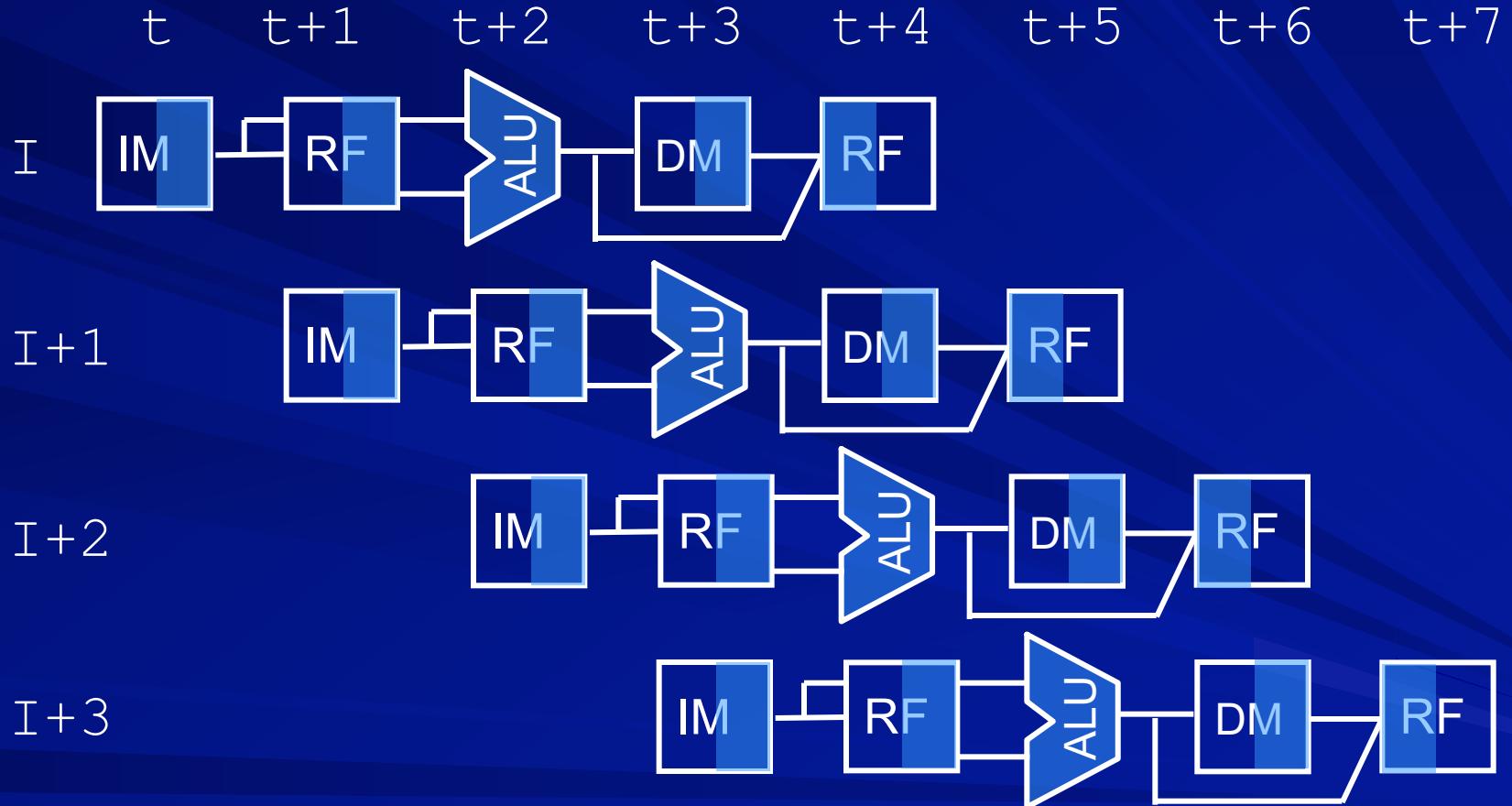
Representation I

- Horizontal axis: time
- Vertical axis: instructions

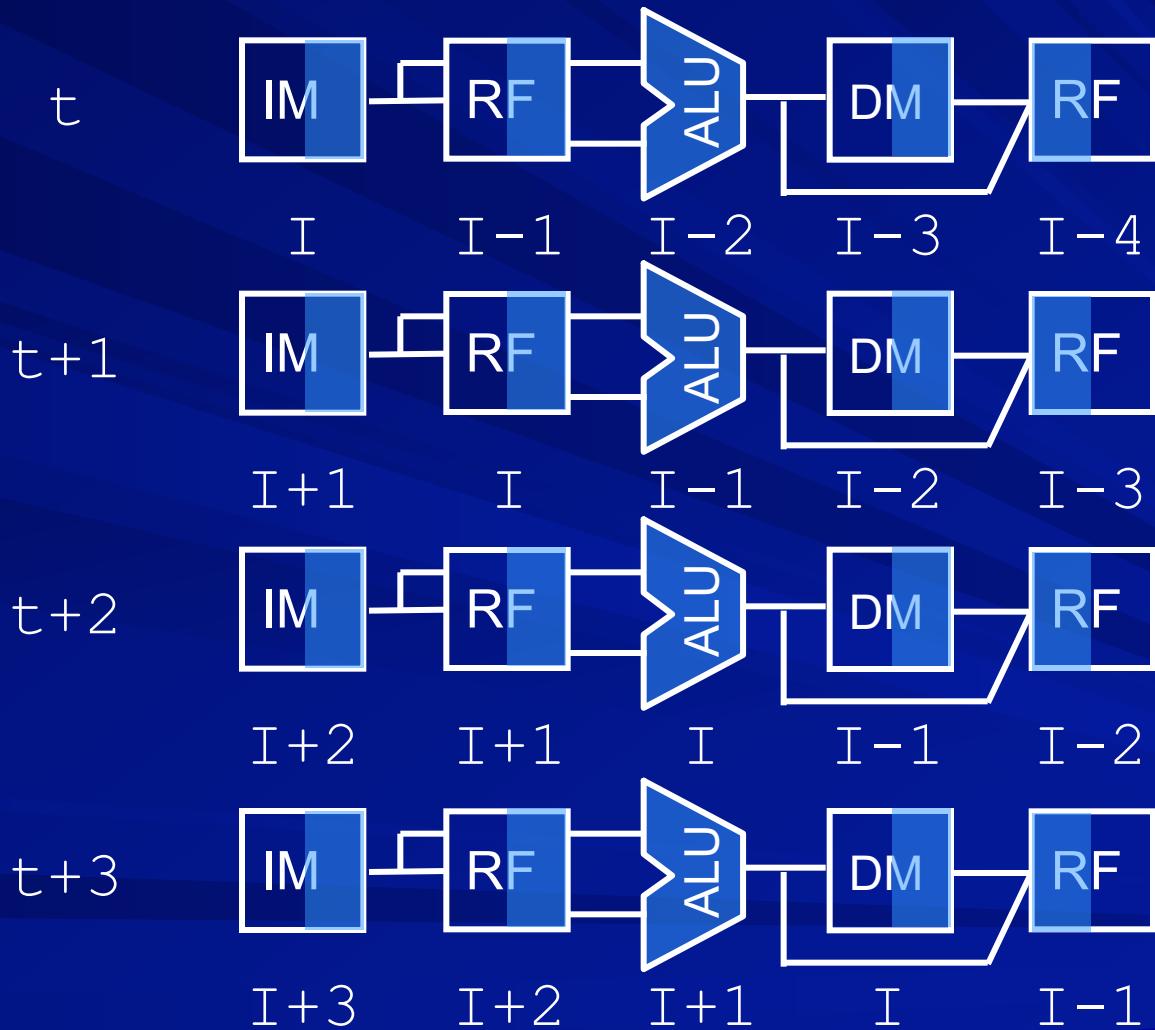
Representation II

- Horizontal axis: pipeline stages
- Vertical axis: time

Representation I



Representation II



Hurdles in instruction pipelining

■ Structural hazards

- Resource conflicts - two instruction require same resource in the same cycle

■ Data hazards

- Data dependencies - one instruction needs data which is yet to be produced by another instruction

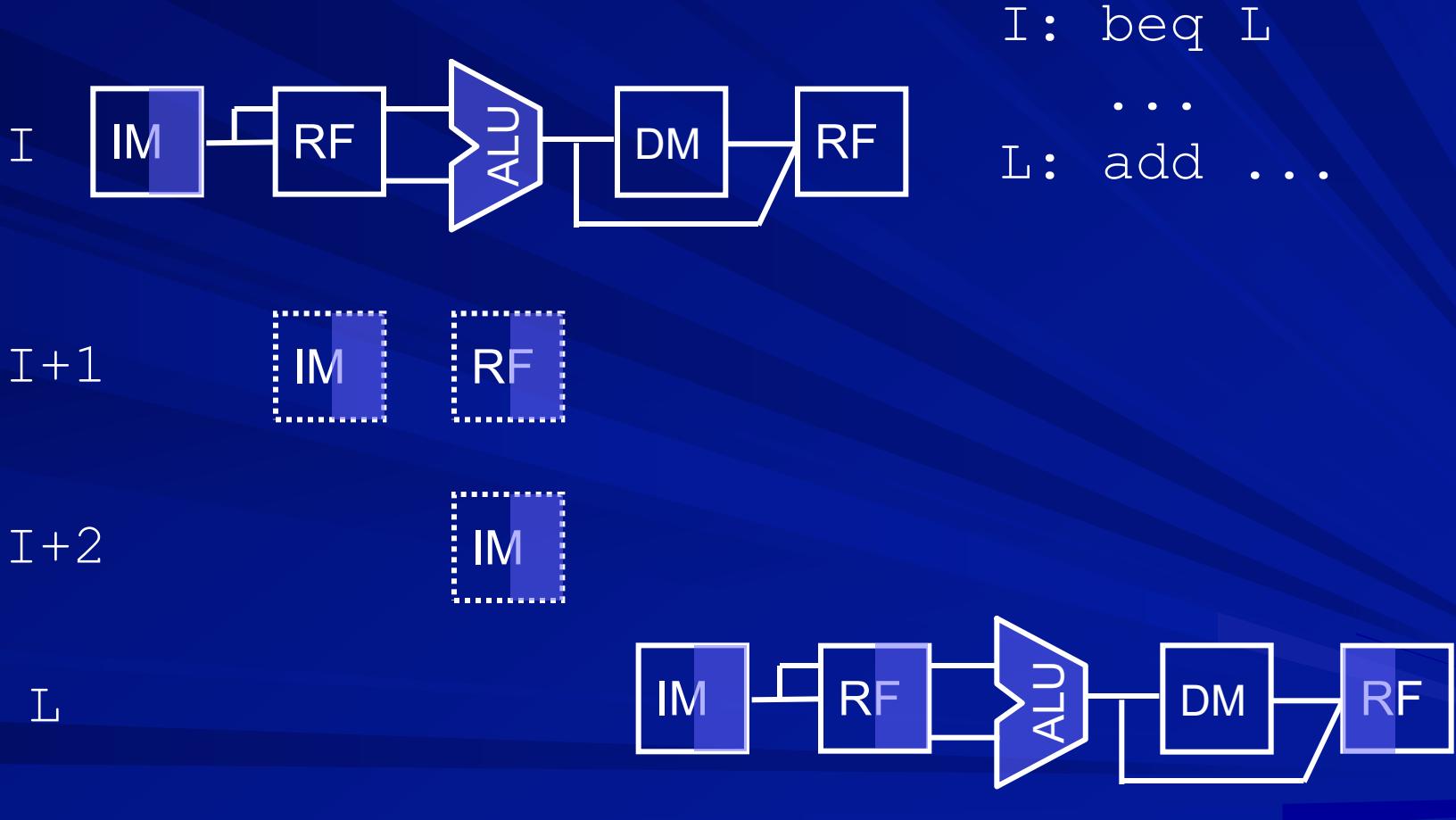
■ Control Hazards

- Decision about next instruction needs more cycles

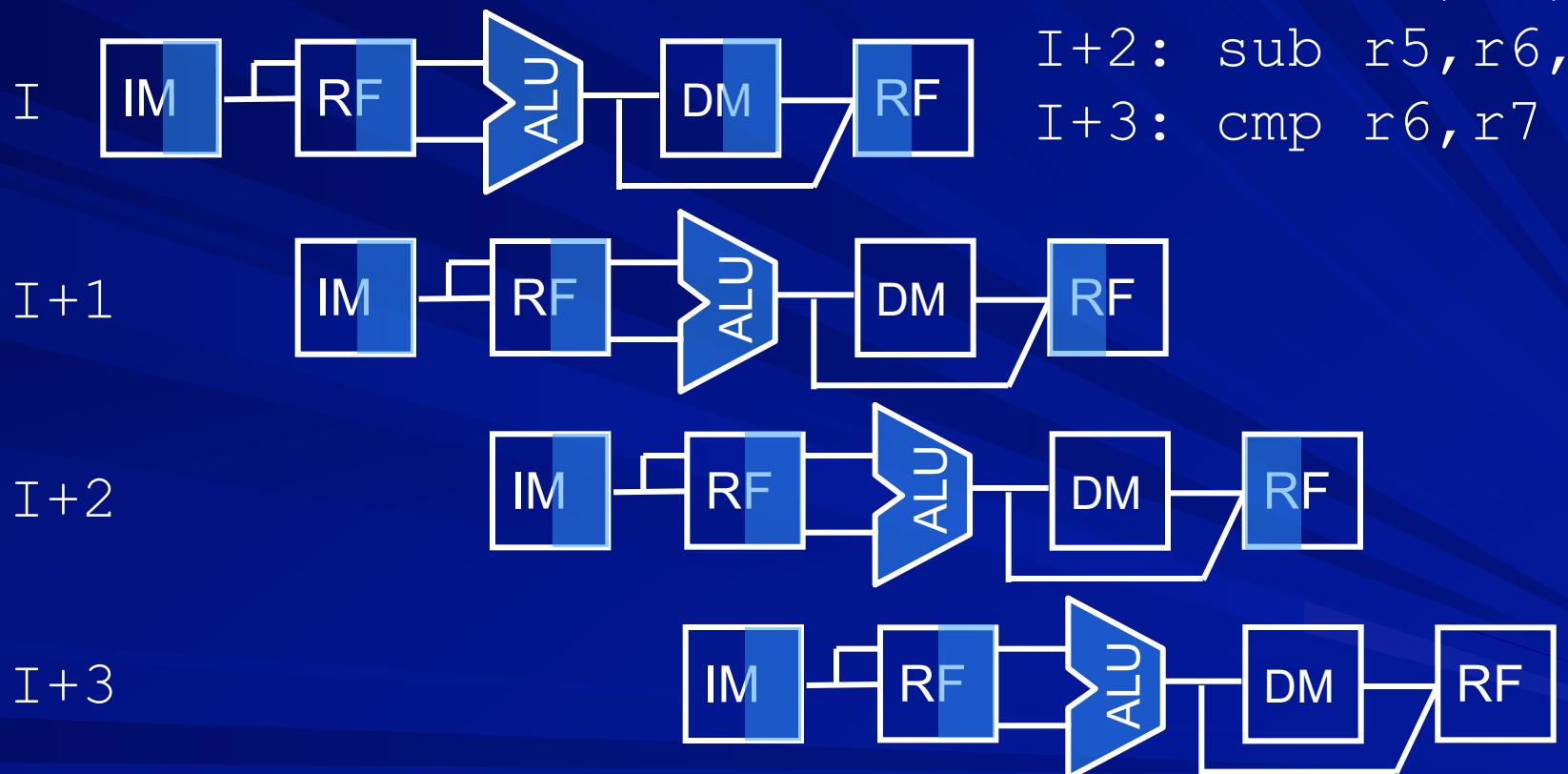
Structural hazards

- No structural hazards in the present design
 - separate instruction and data memories
 - adders for PC increment and offset addition to PC separate from main ALU
 - each instruction uses ALU at most in one cycle
 - one instruction can read from RF while other can write into it in the same cycle

Stalls due to control hazards



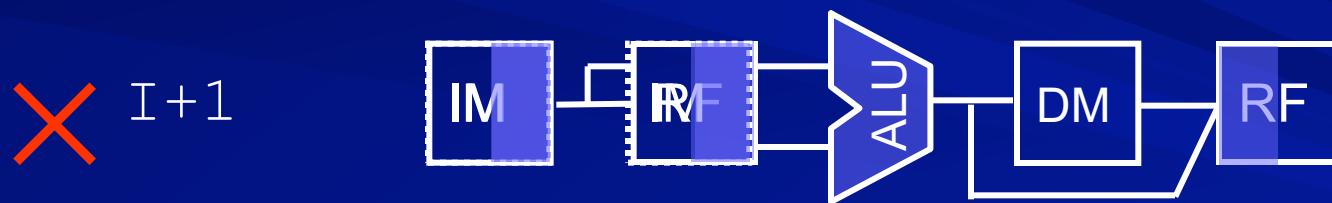
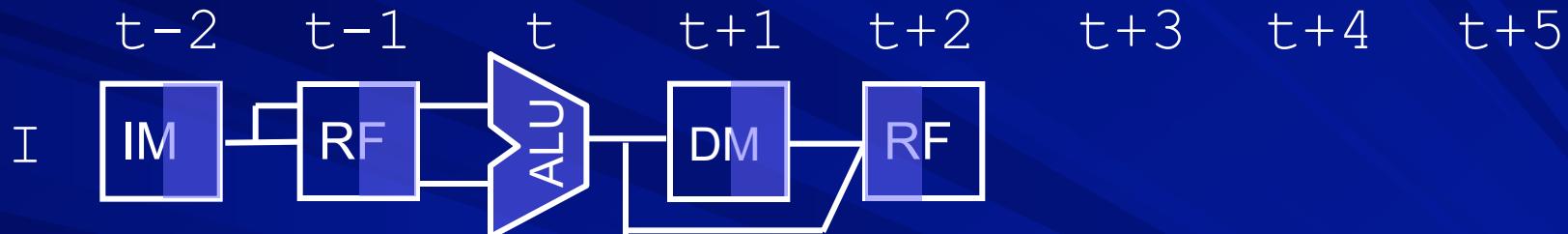
Data hazards



Stalls due to data hazards

instruction view

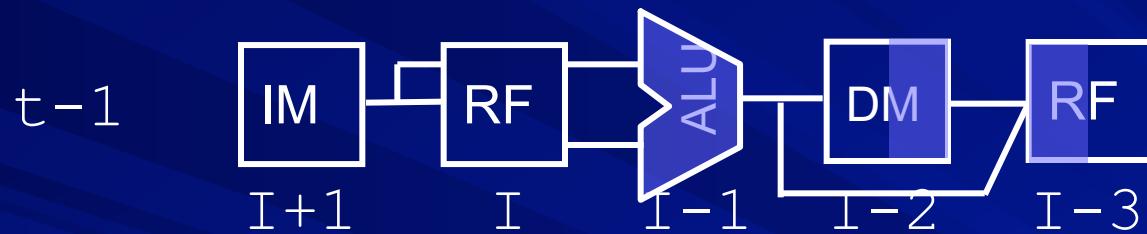
I: ldr r6, ...
I+1: add r4, r6, ...



Stalls due to data hazards

stage-wise view

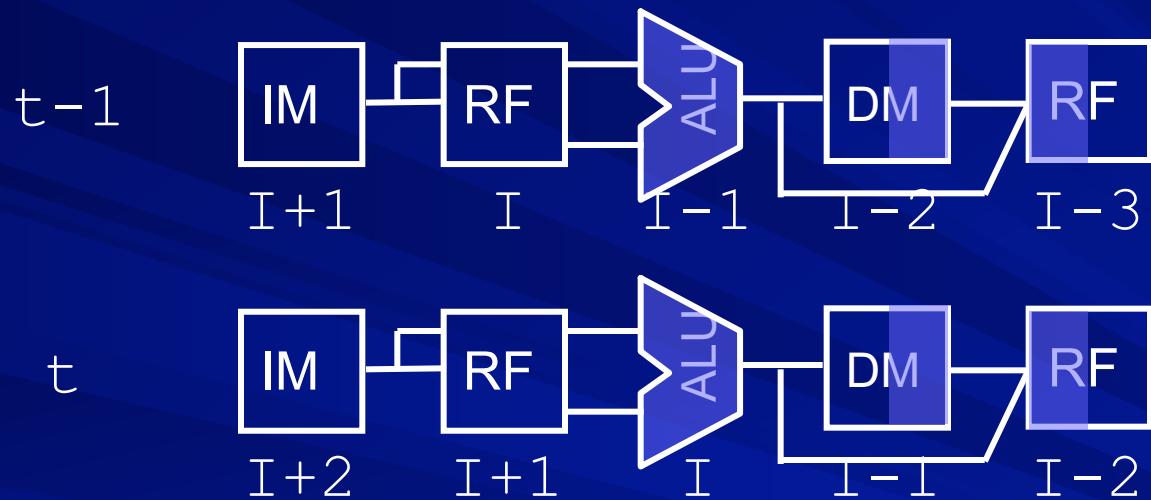
I: ldr r6, ...
I+1: add r4, r6, ...



Stalls due to data hazards

stage-wise view

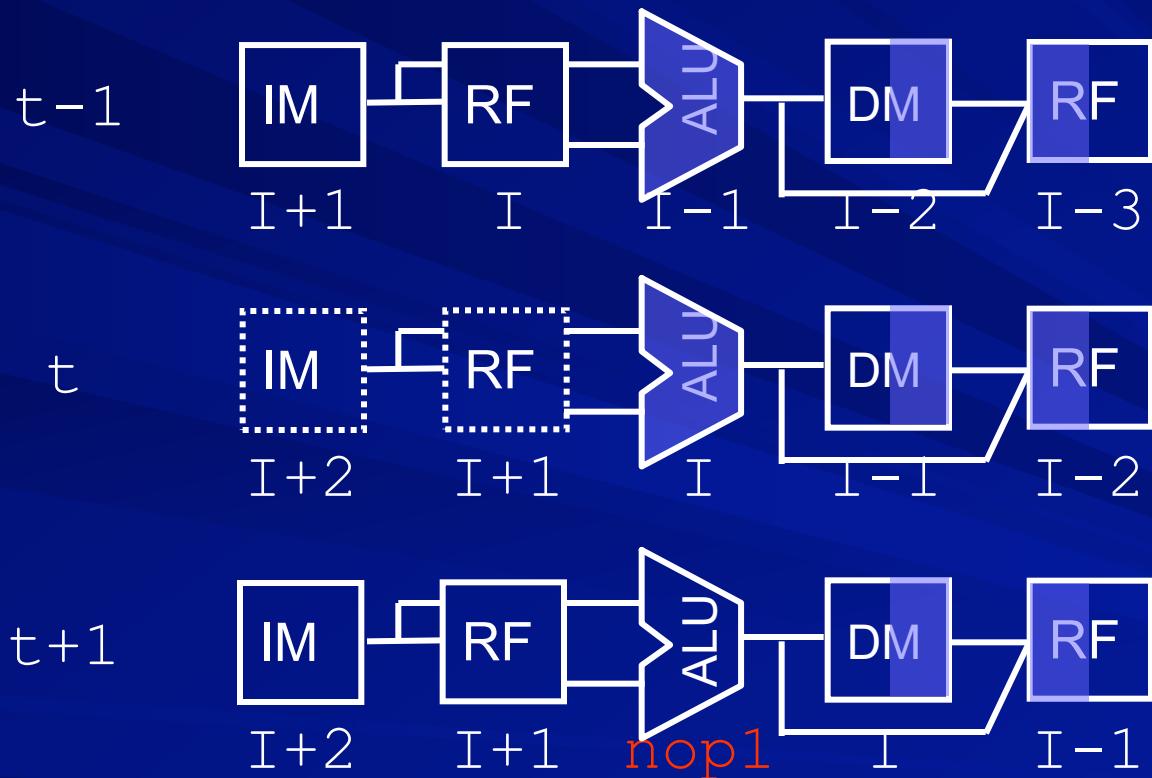
I: ldr r6, ...
I+1: add r4, r6, ...



Stalls due to data hazards

stage-wise view

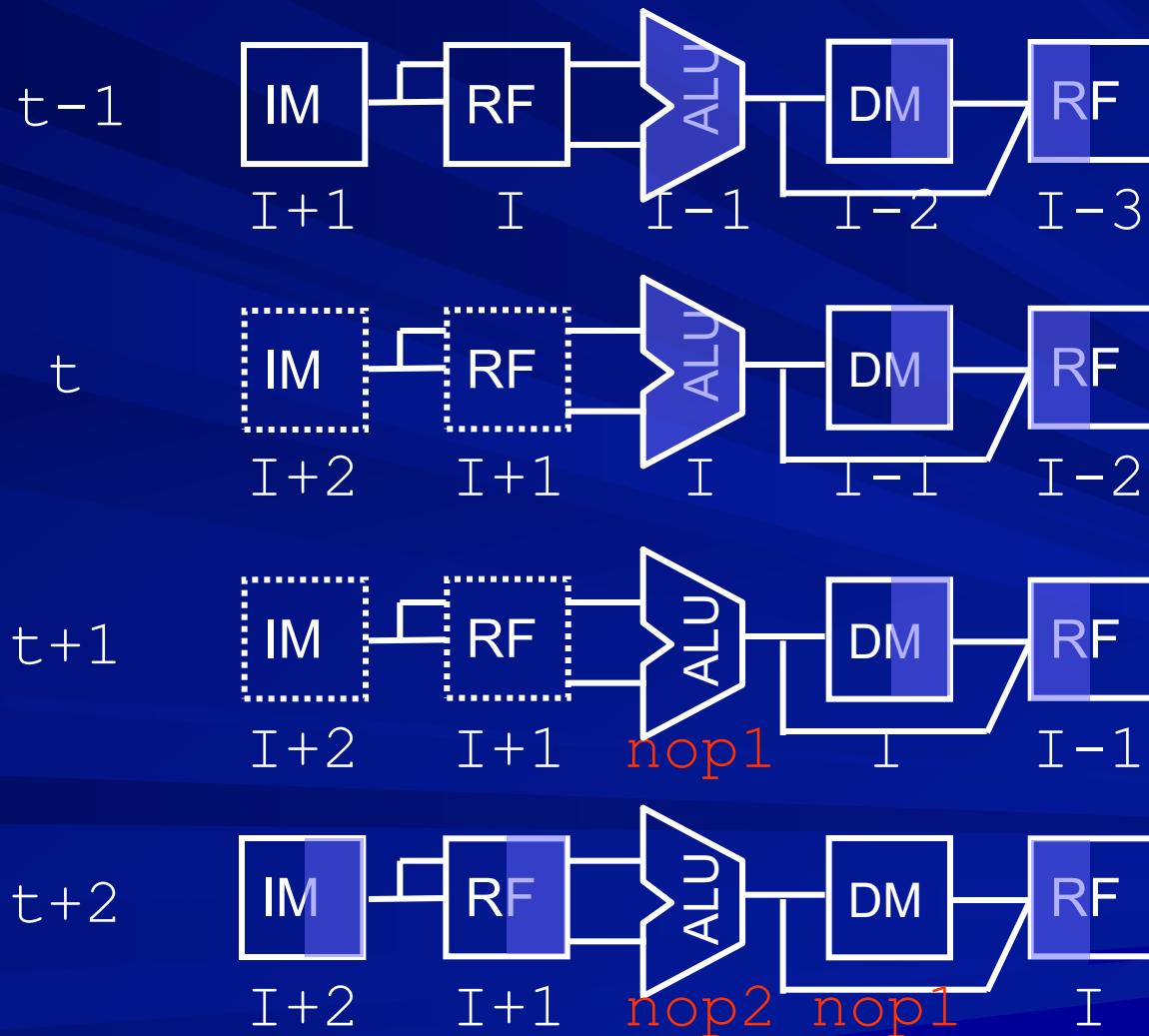
I: ldr r6, ...
I+1: add r4, r6, ...



Stalls due to data hazards

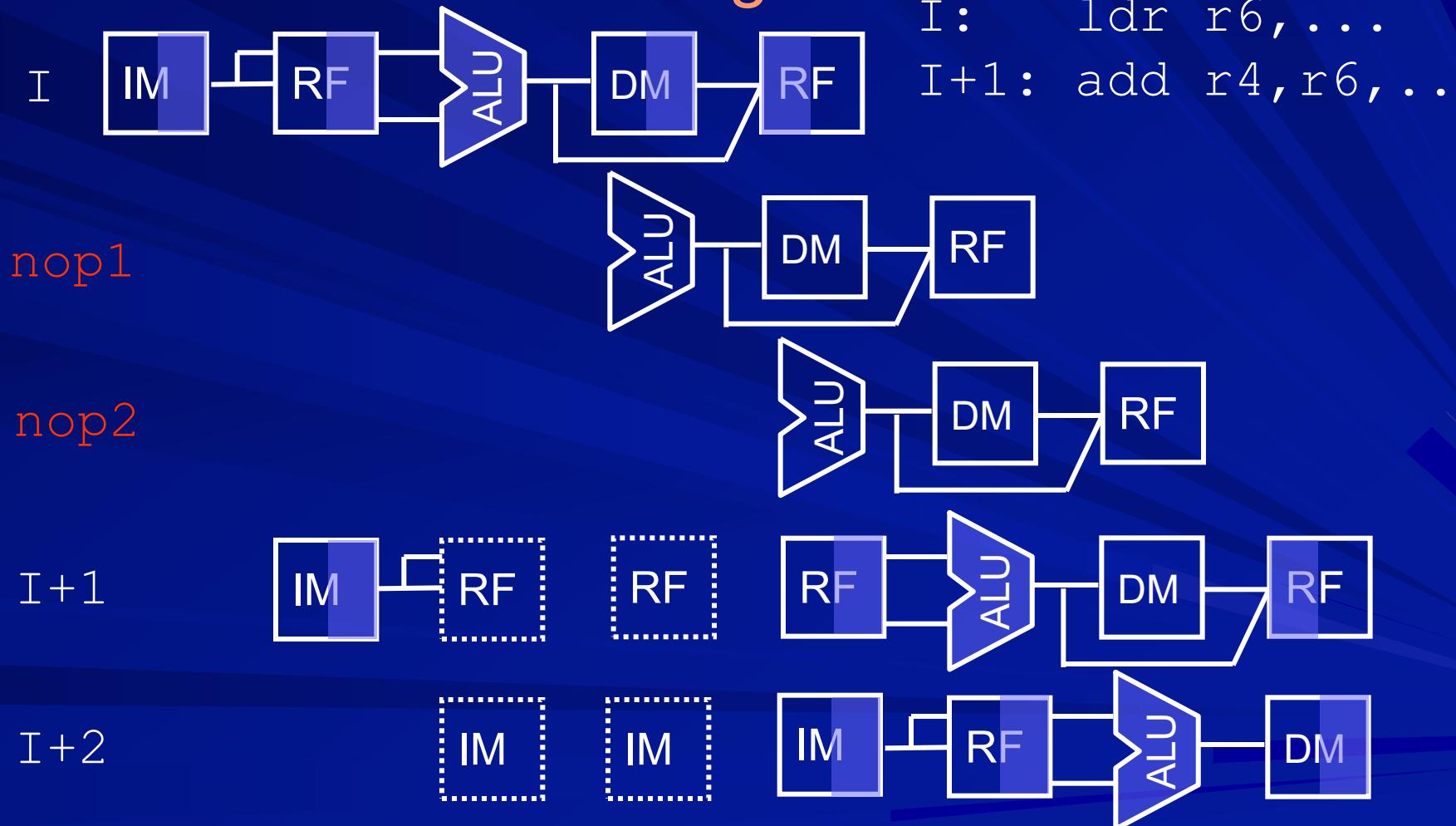
stage-wise view

I: ldr r6, ...
I+1: add r4, r6, ...



Stalls due to data hazards

instruction view again



Handling hazards

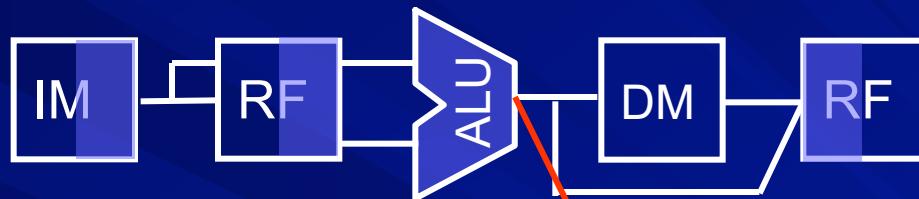
- Data hazards
 - detect instructions with data dependence
 - introduce nop instructions (bubbles) in the pipeline
 - more complex: data forwarding
- Control hazards
 - detect branch instructions
 - flush inline instructions if branching occurs
 - more complex: branch prediction

Are there software solutions?

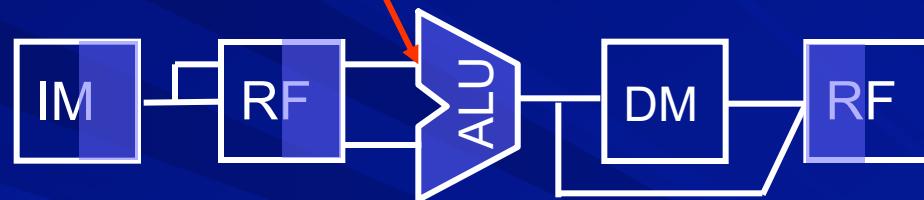
- Separate dependent instructions by reordering code
- Insert nop instructions in worst case
- Treat branches as delayed branches and insert suitable instructions in delay slots

Data forwarding path P1

I:add r6,...

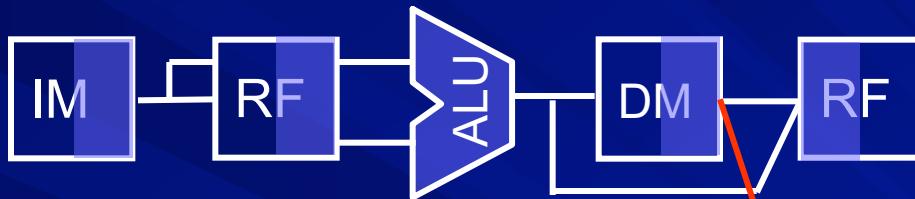


I+1:add r4, r6, ..

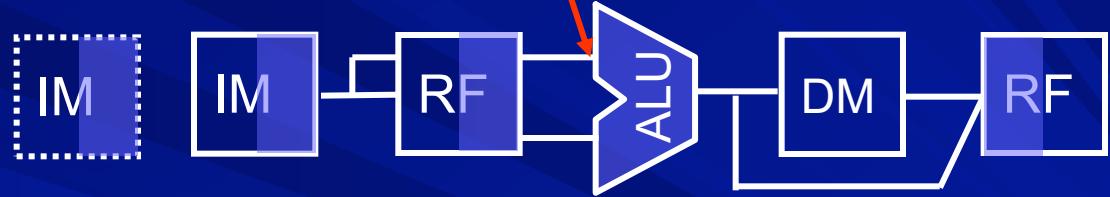


Data forwarding path P2

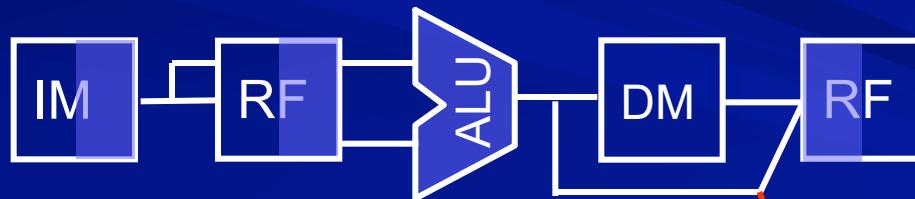
I:ldr r6,..



I+1:add r4,r6,..



I:add r6,..

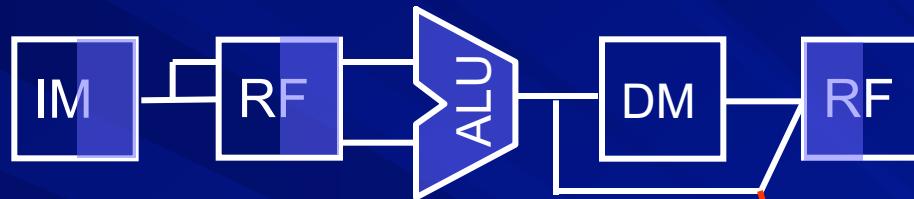


I+2:add r4,r6,..

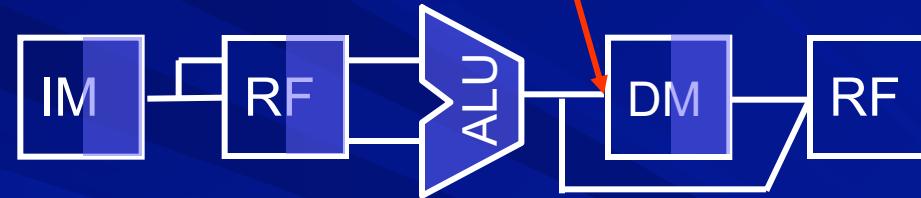


Data forwarding path P3

I:add r6,...



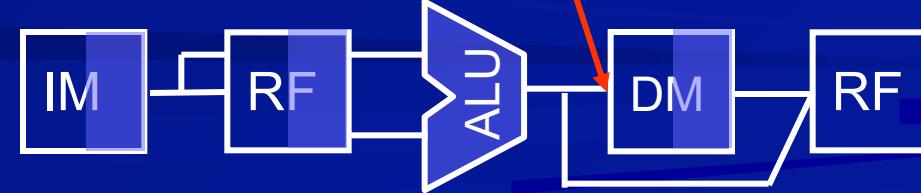
I+1:str r6,...



I:ldr r6,...



I+1:str r6,...



Thanks

COL216

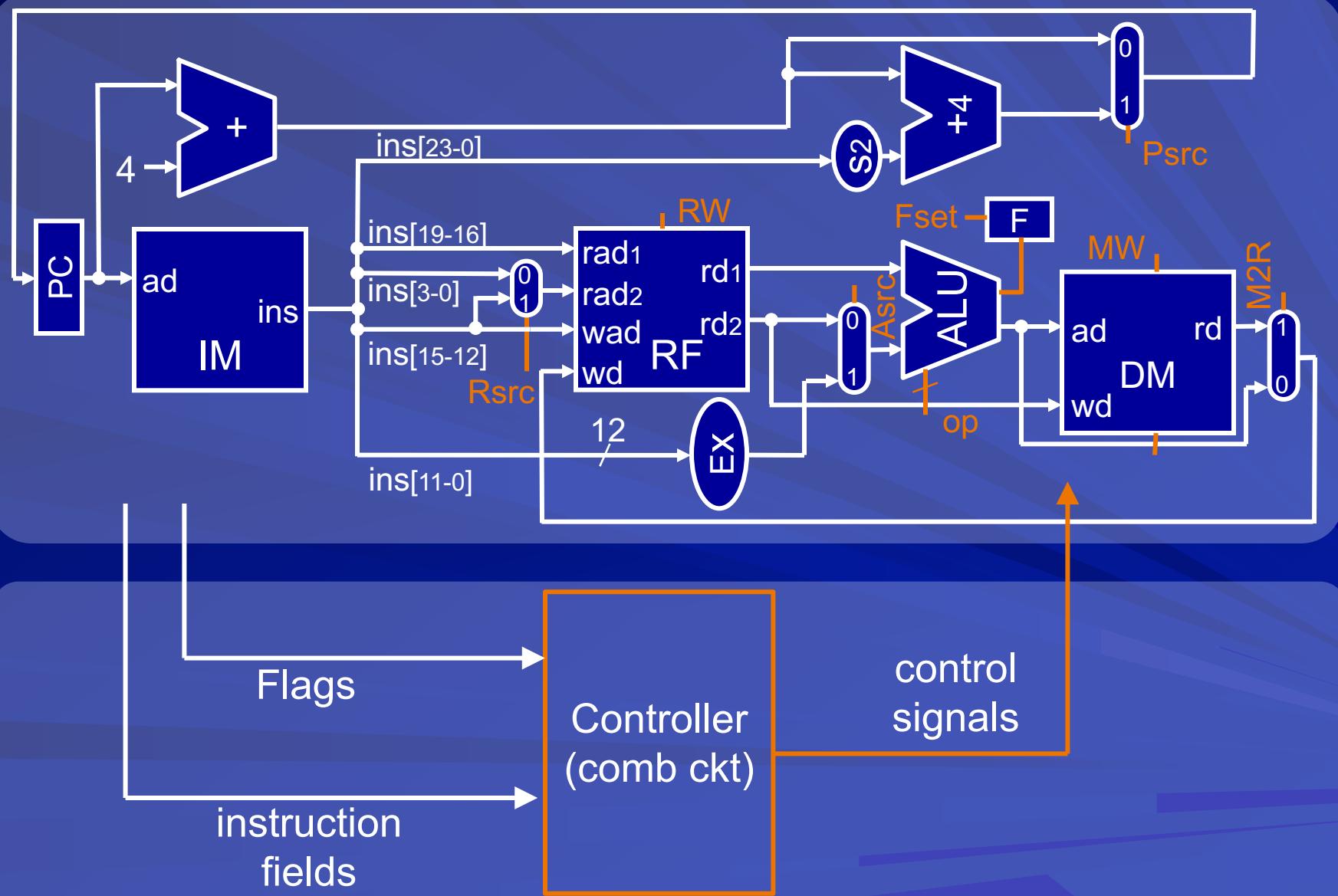
Computer Architecture

Pipelined Processor design –

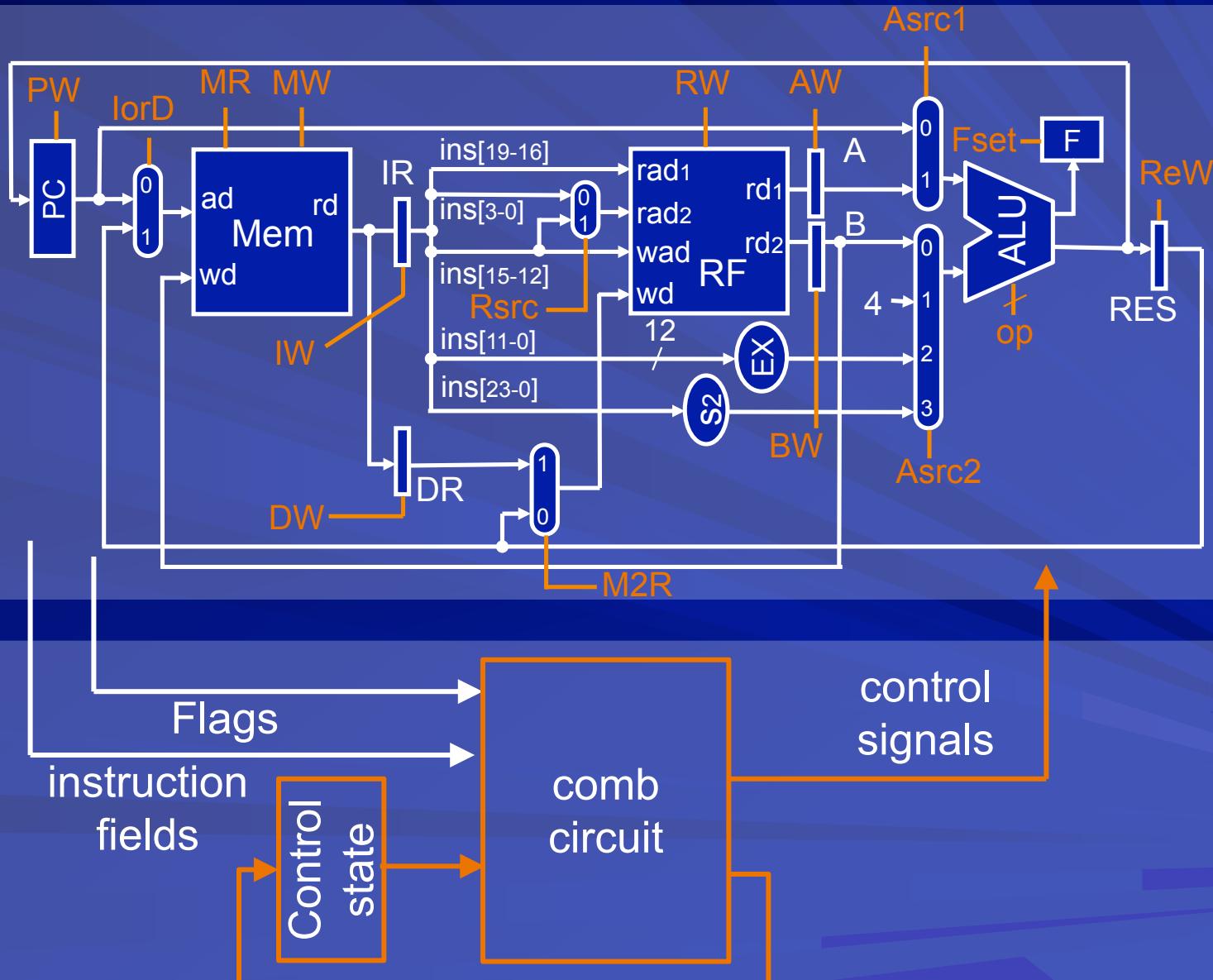
Controller, Data forwarding

10th February 2022

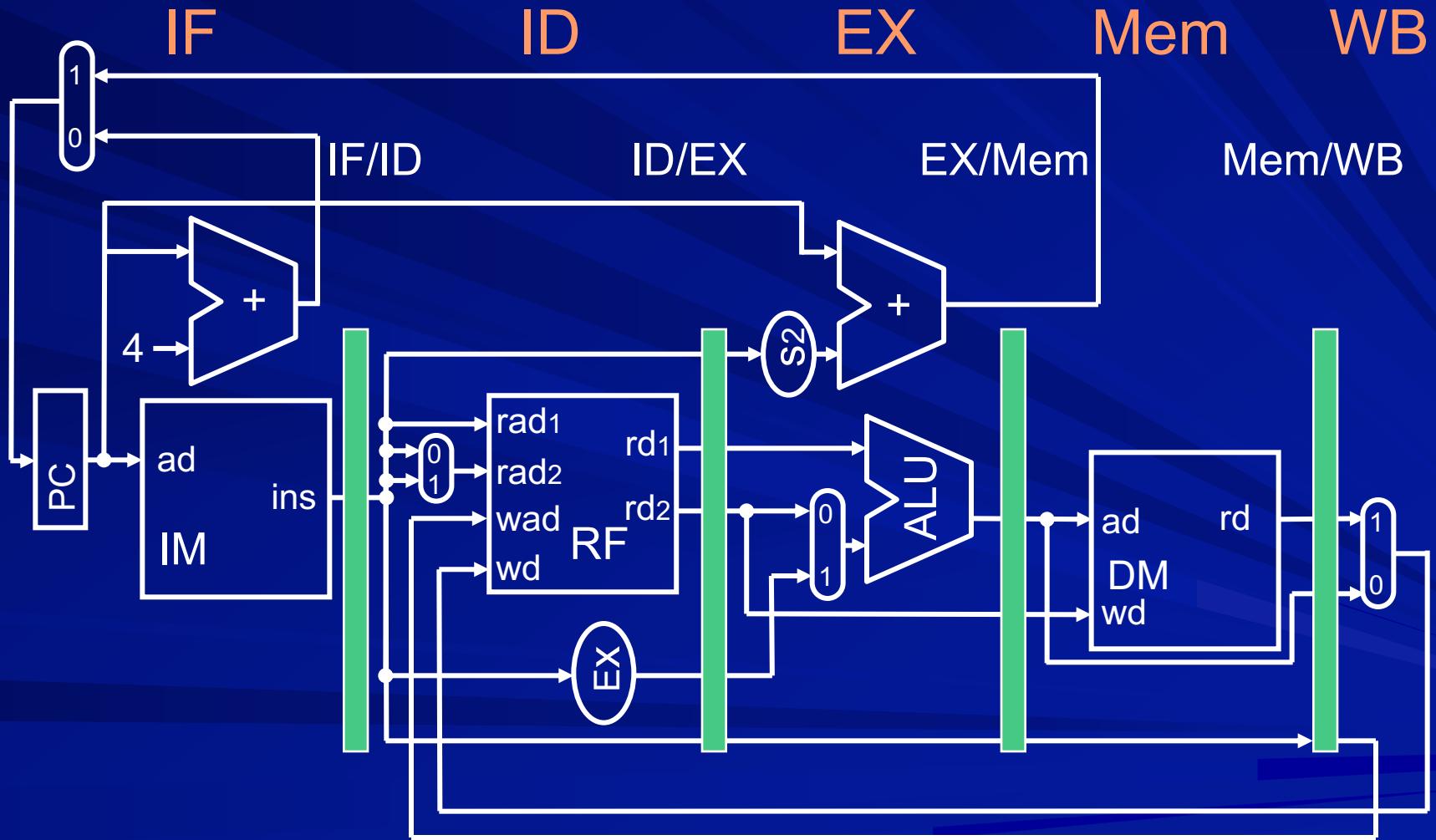
Single Cycle Datapath + Controller



Multi Cycle Datapath + Controller



5 Stage Pipeline



Hurdles in instruction pipelining

■ Structural hazards

- Resource conflicts - two instruction require same resource in the same cycle

■ Data hazards

- Data dependencies - one instruction needs data which is yet to be produced by another instruction

■ Control Hazards

- Decision about next instruction needs more cycles

Handling data hazards

- Assume no data hazards
 - leave it to compiler to remove hazards
- Introduce stalls/bubbles
 - requires hazard detection
(check data dependence among instructions)
 - compiler may still help in reducing hazards
- Do data forwarding
 - this also requires hazard detection
 - stalls may also be required in some cases

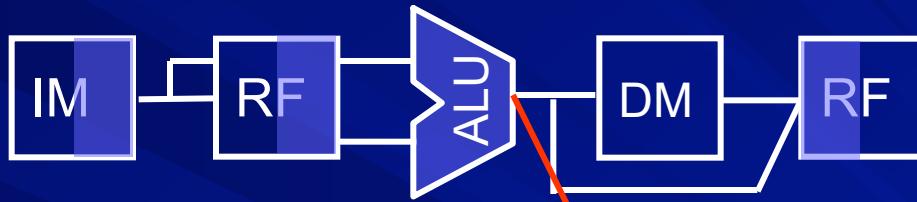
Detecting data hazard

Condition to be checked:

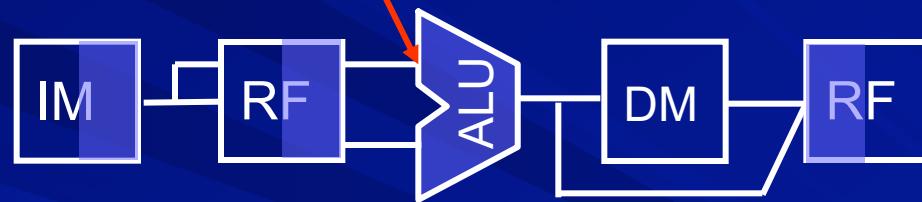
Instruction in RF stage reads from a register in which instruction in ALU stage or DM stage is going to write

Data forwarding path P1

I:add r6,...

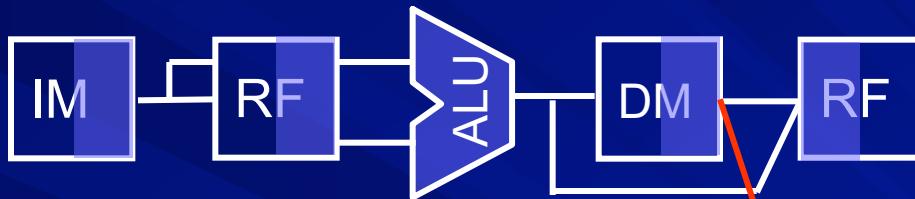


I+1:add r4, r6, ..

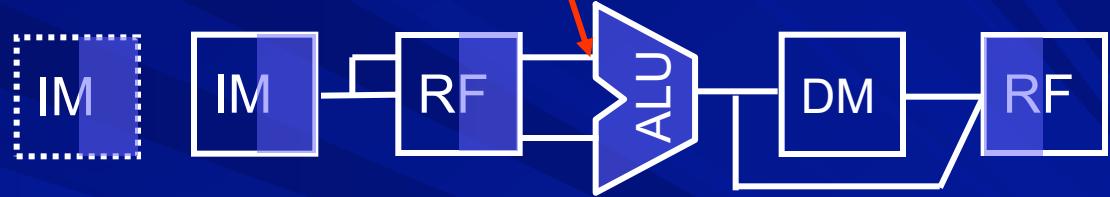


Data forwarding path P2

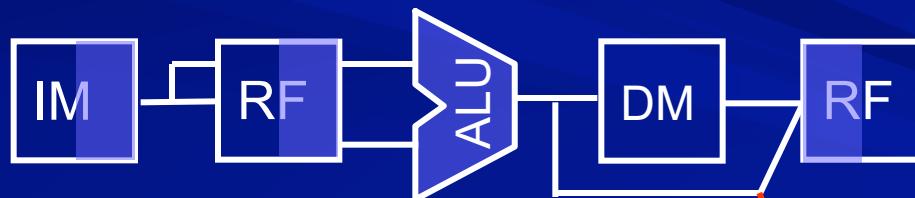
I:ldr r6,..



I+1:add r4,r6,..



I:add r6,..

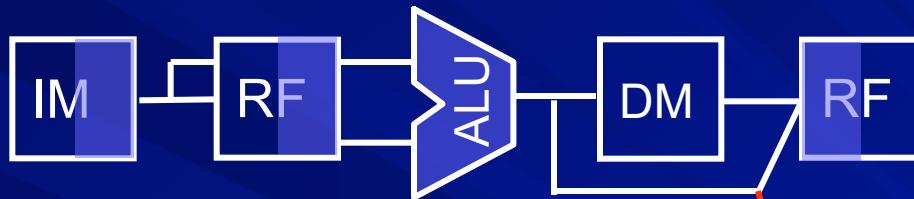


I+2:add r4,r6,..

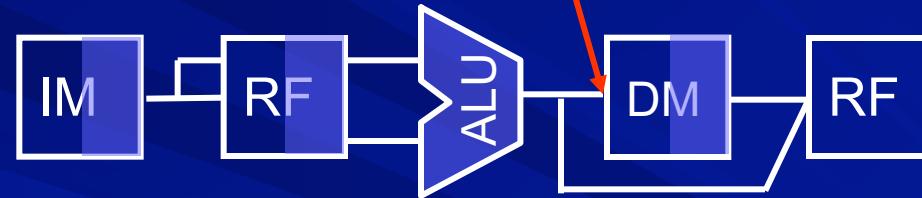


Data forwarding path P3

I:add r6,...



I+1:str r6,...



I:ldr r6,...



I+1:str r6,...



Data forwarding path list

- P1
from ALU out
(EX/DM register) to ALU in1/2
 - P2
from DM/ALU out
(DM/WB register) to ALU in1/2
 - P3
from DM/ALU out
(DM/WB register) to DM in

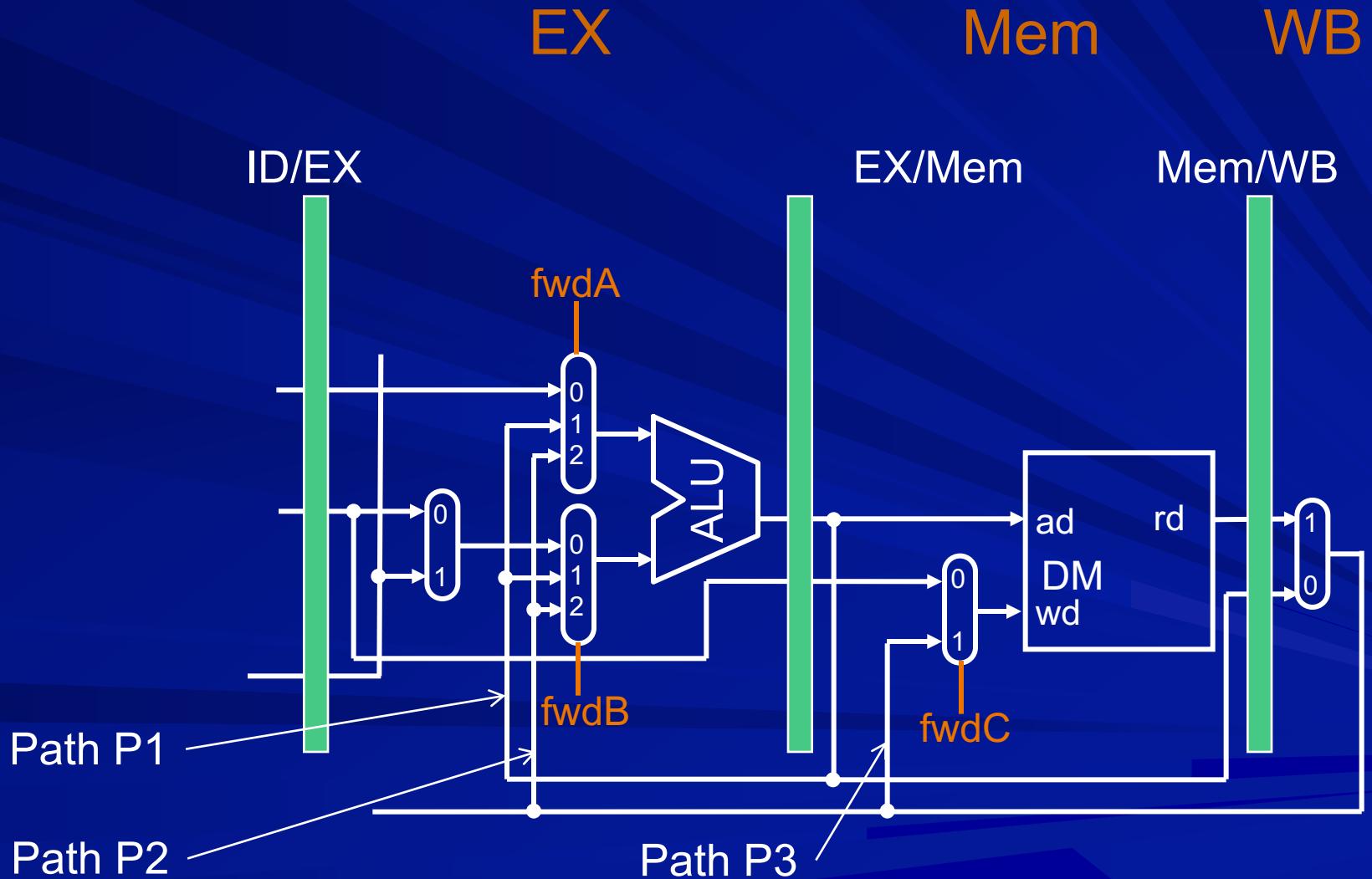
Dependence check logic

Condition to be checked:

Operand of instruction in RF stage is a register in which instruction in ALU stage or DM stage is going to write

We need to ensure that instruction in RF stage actually reads Rn and/or Rm

Data forwarding paths



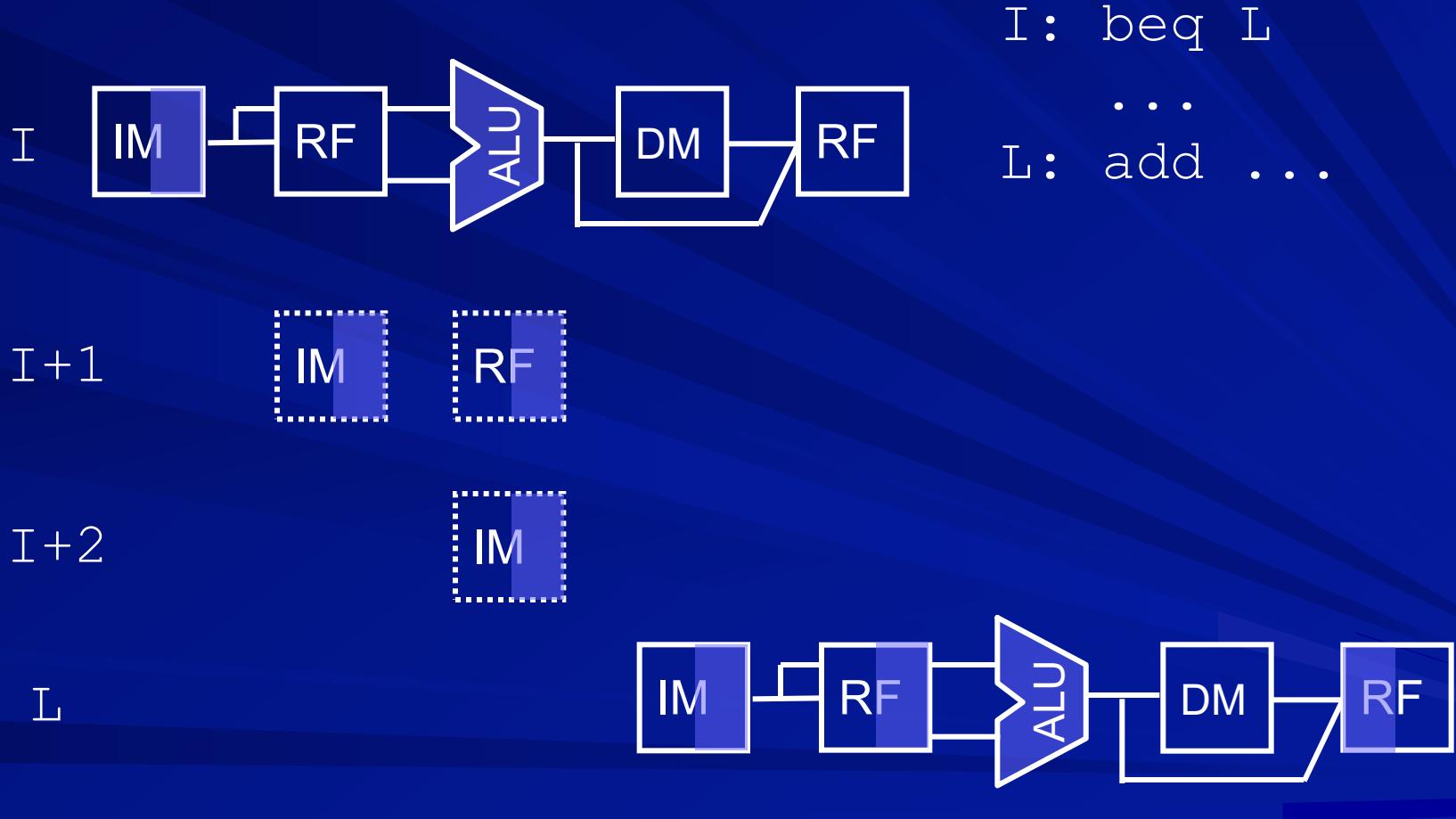
Executing branch instructions

- In which cycle the instruction is found to be a branch instruction?
- In which cycle the branch decision is known?
- In which cycle the target address is computed?

On decoding a branch instruction

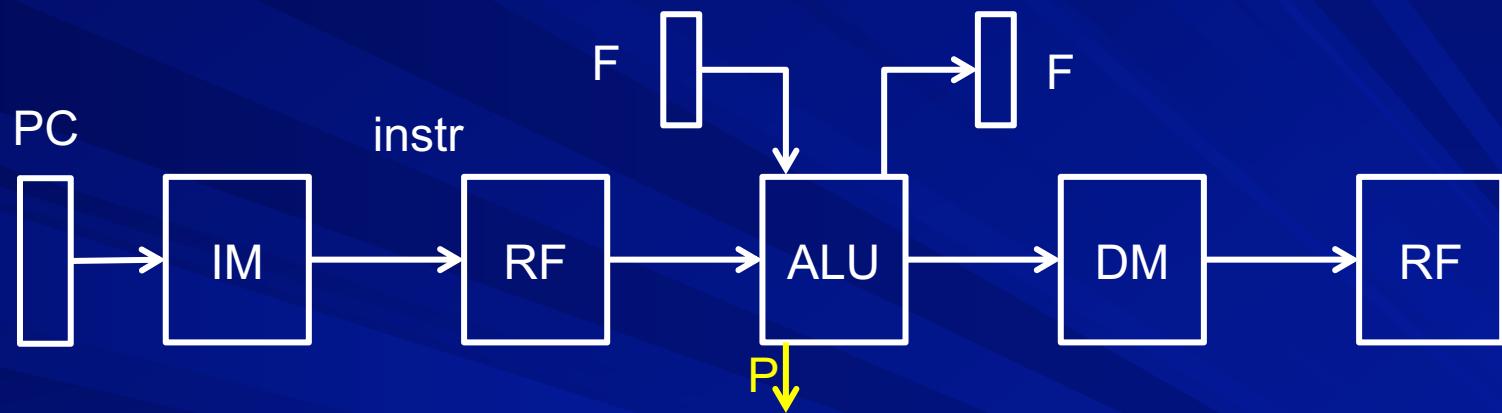
- Flush the inline instructions
- Freeze (stall) the inline instructions
- Allow the inline instructions to continue

Stalls due to control hazards



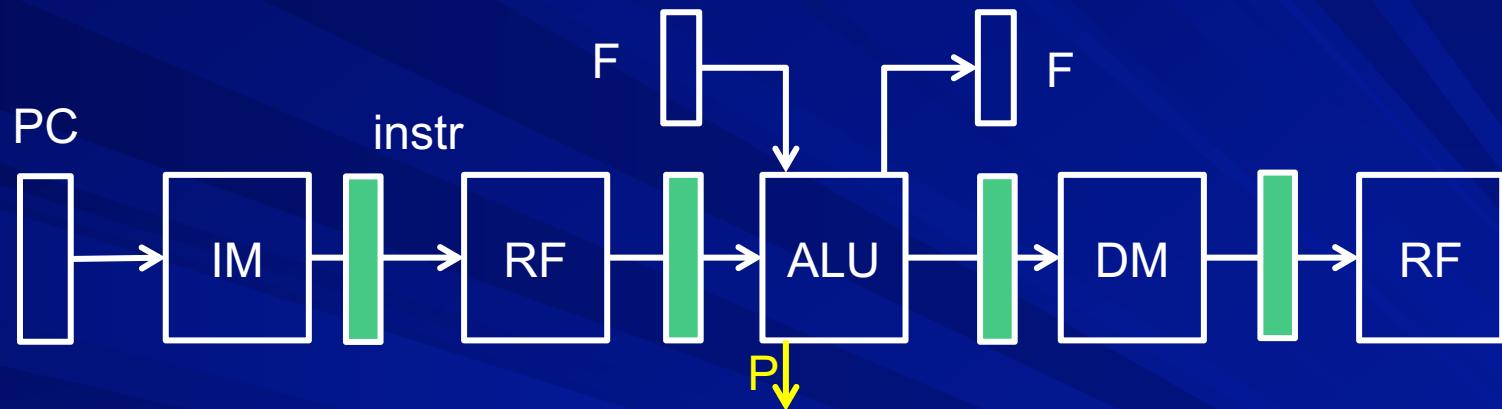
Controller design

Single cycle datapath



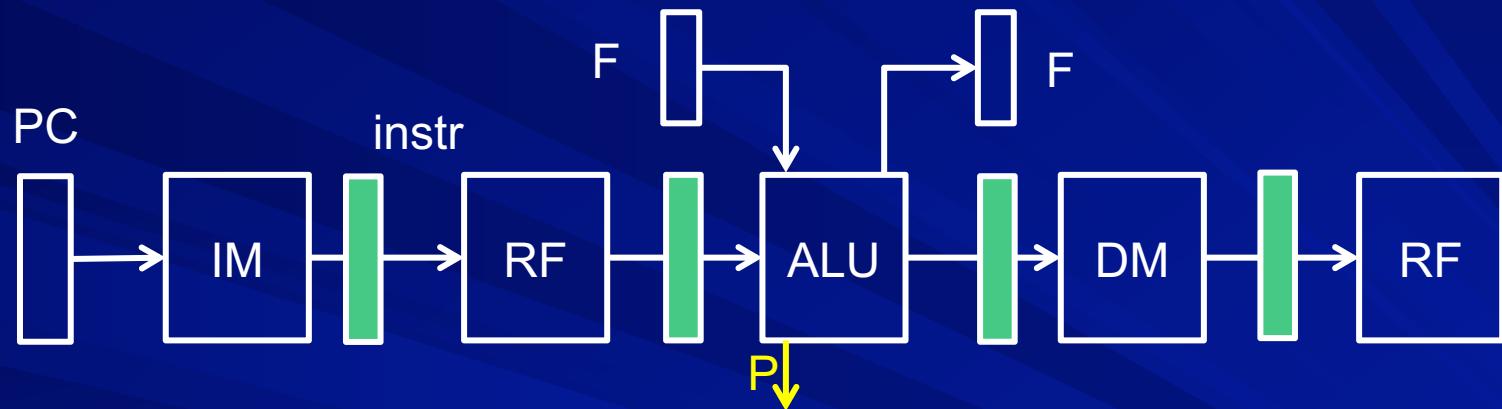
Multi-cycle datapath

Resource sharing possible across cycles

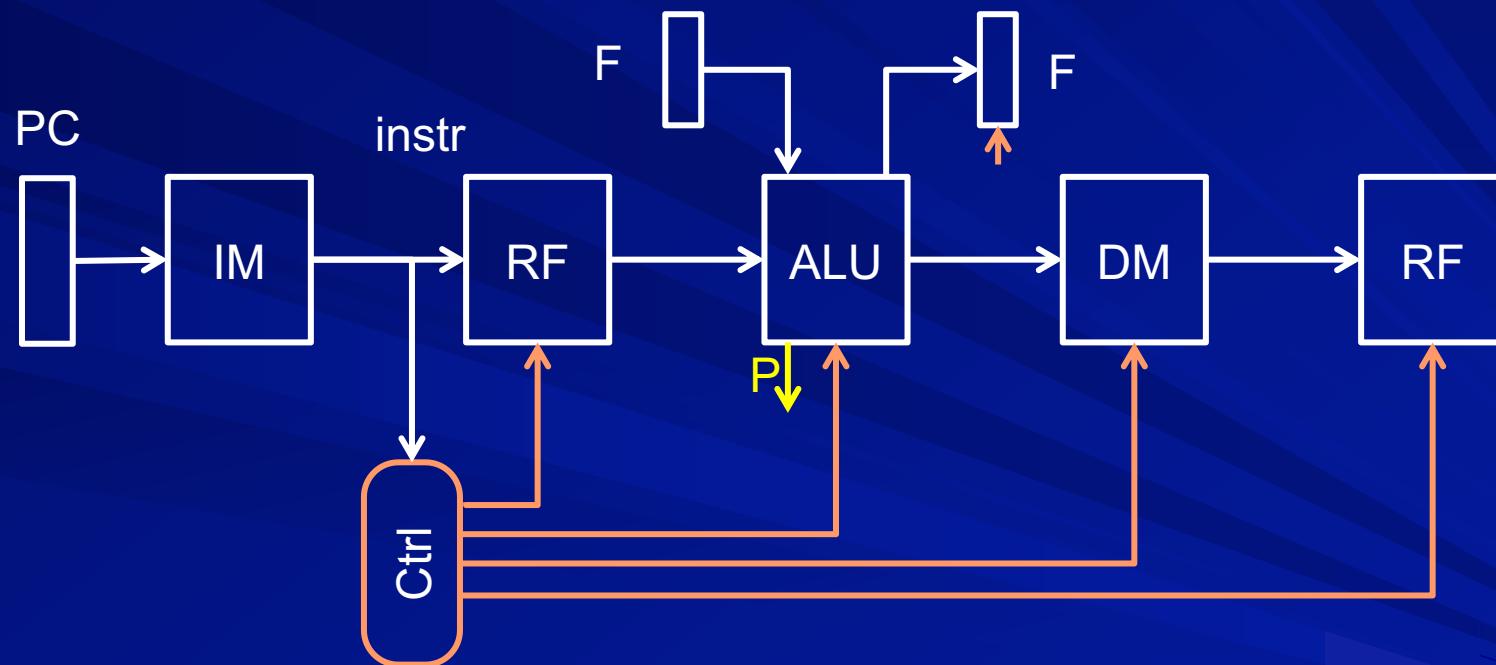


Pipelined datapath

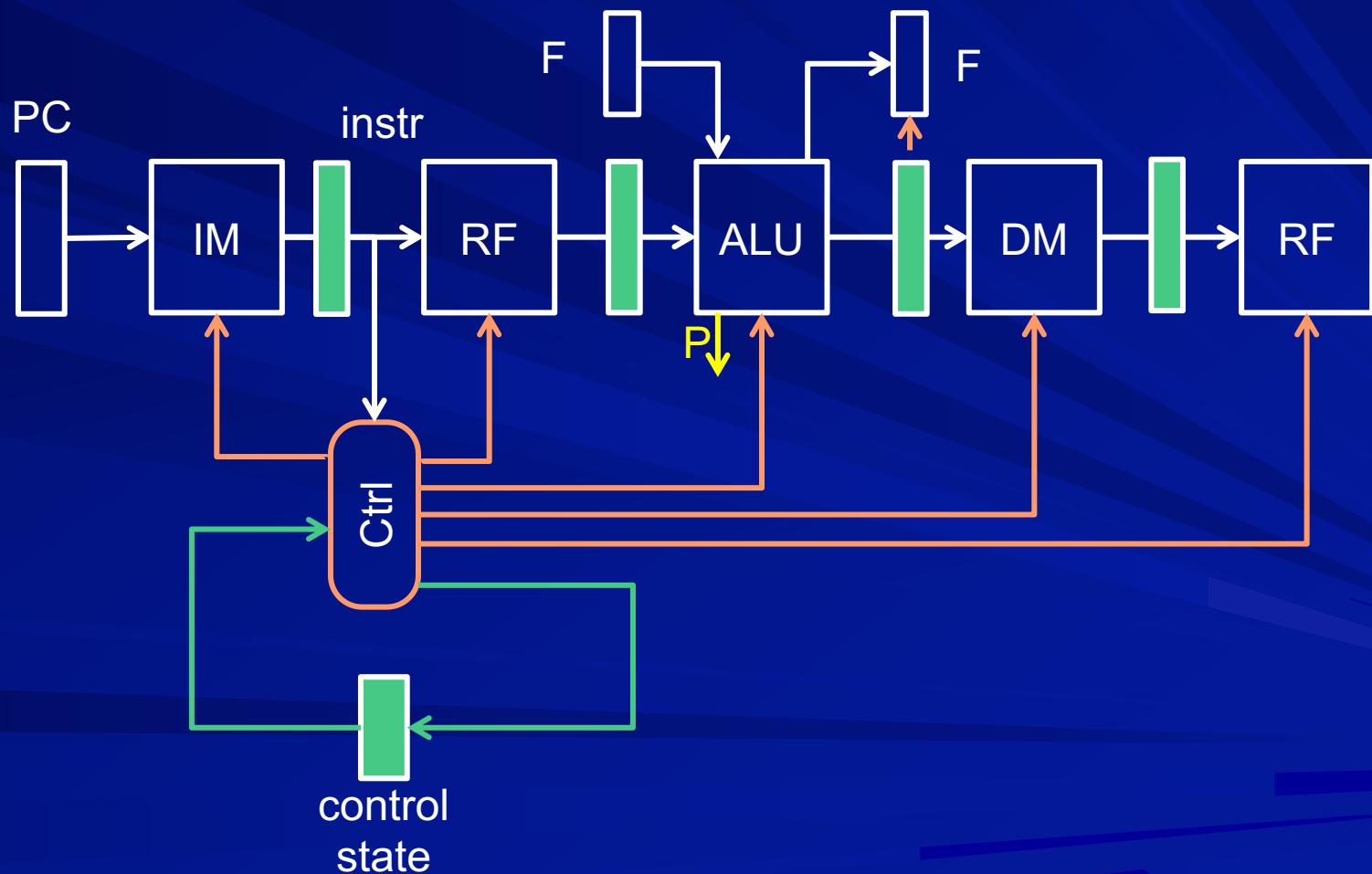
Resource sharing leads to hazards



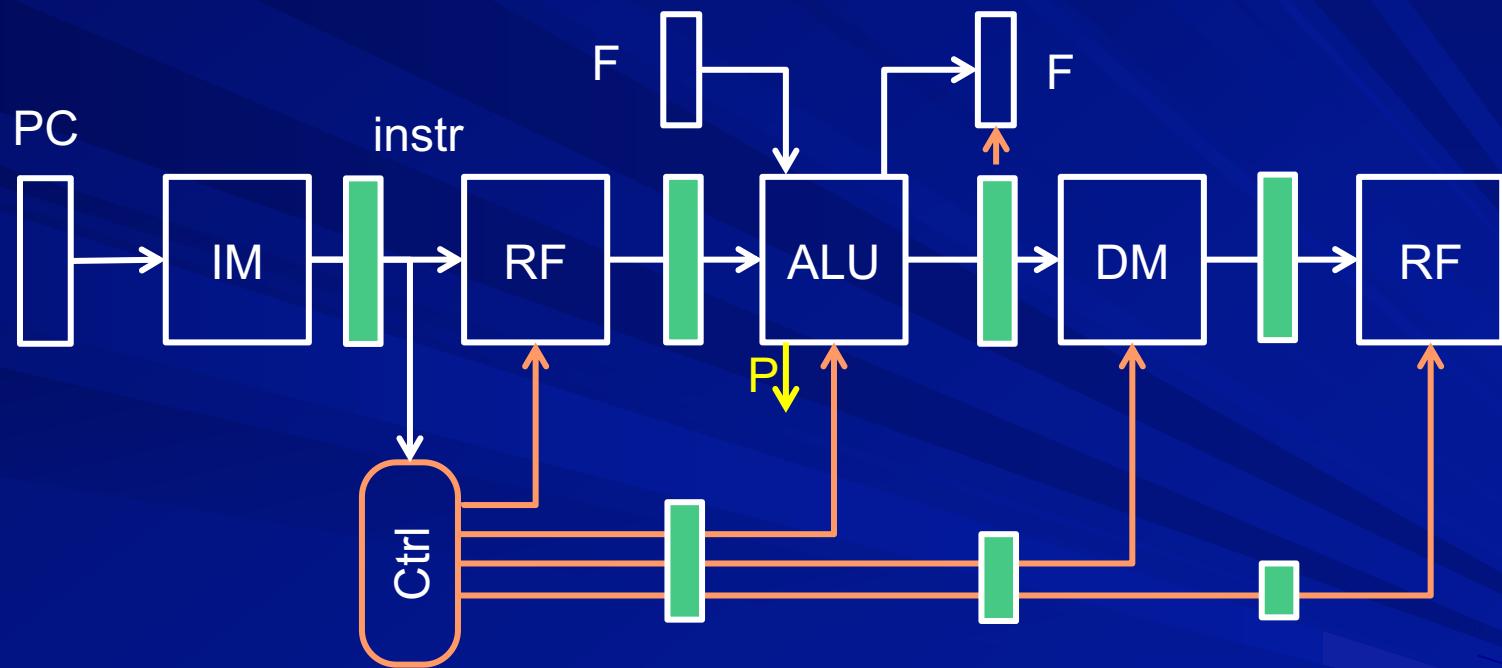
Controller for single cycle DP



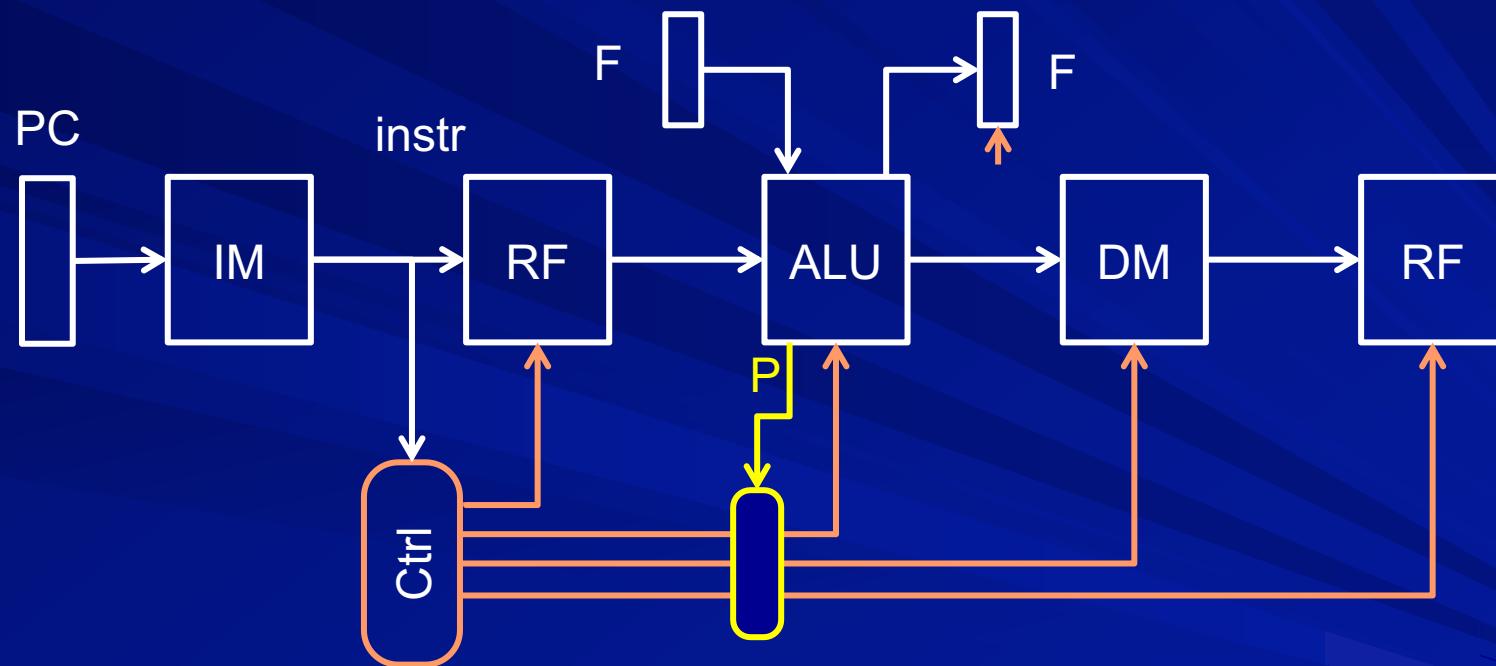
Controller for multi-cycle DP



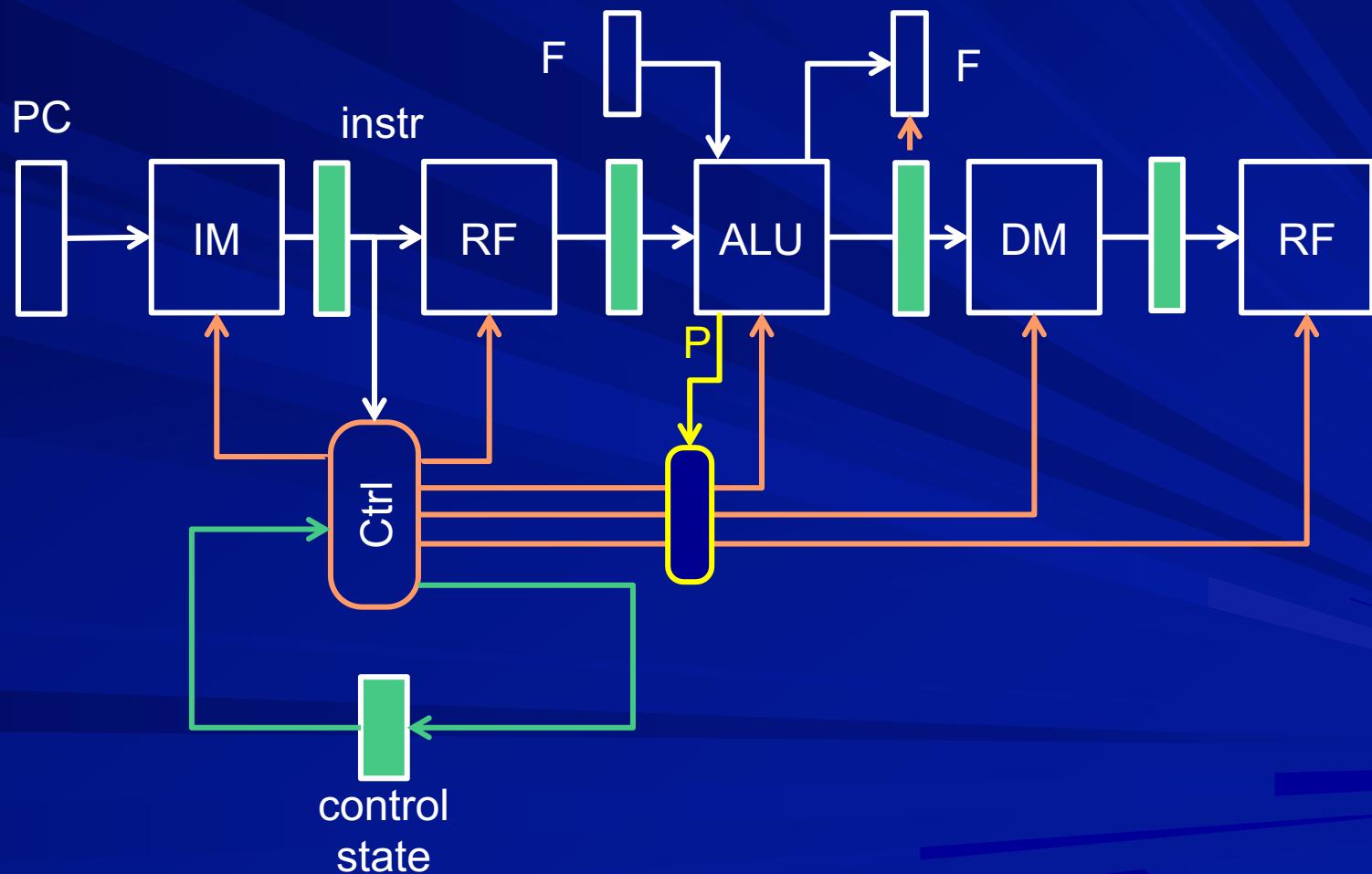
Controller for pipelined DP



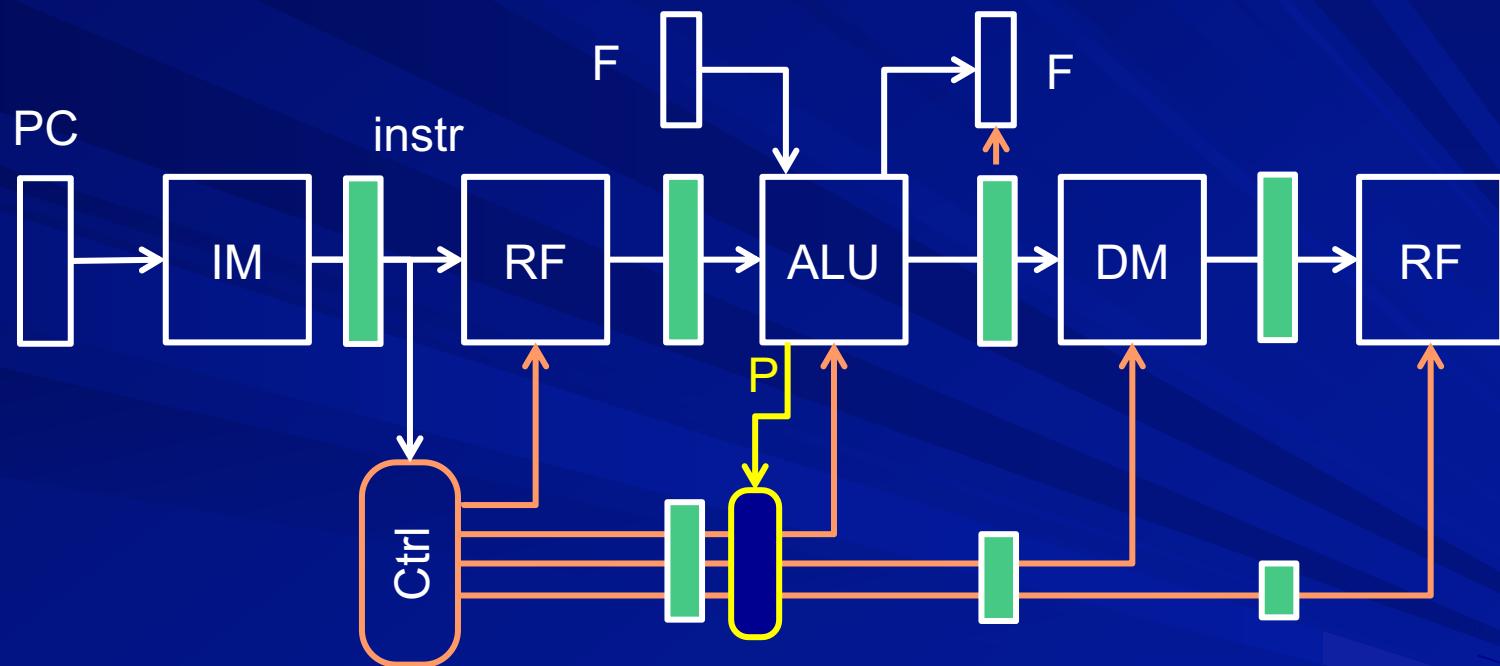
Controller for single cycle DP



Controller for multi-cycle DP



Controller for pipelined DP



Thanks

COL216

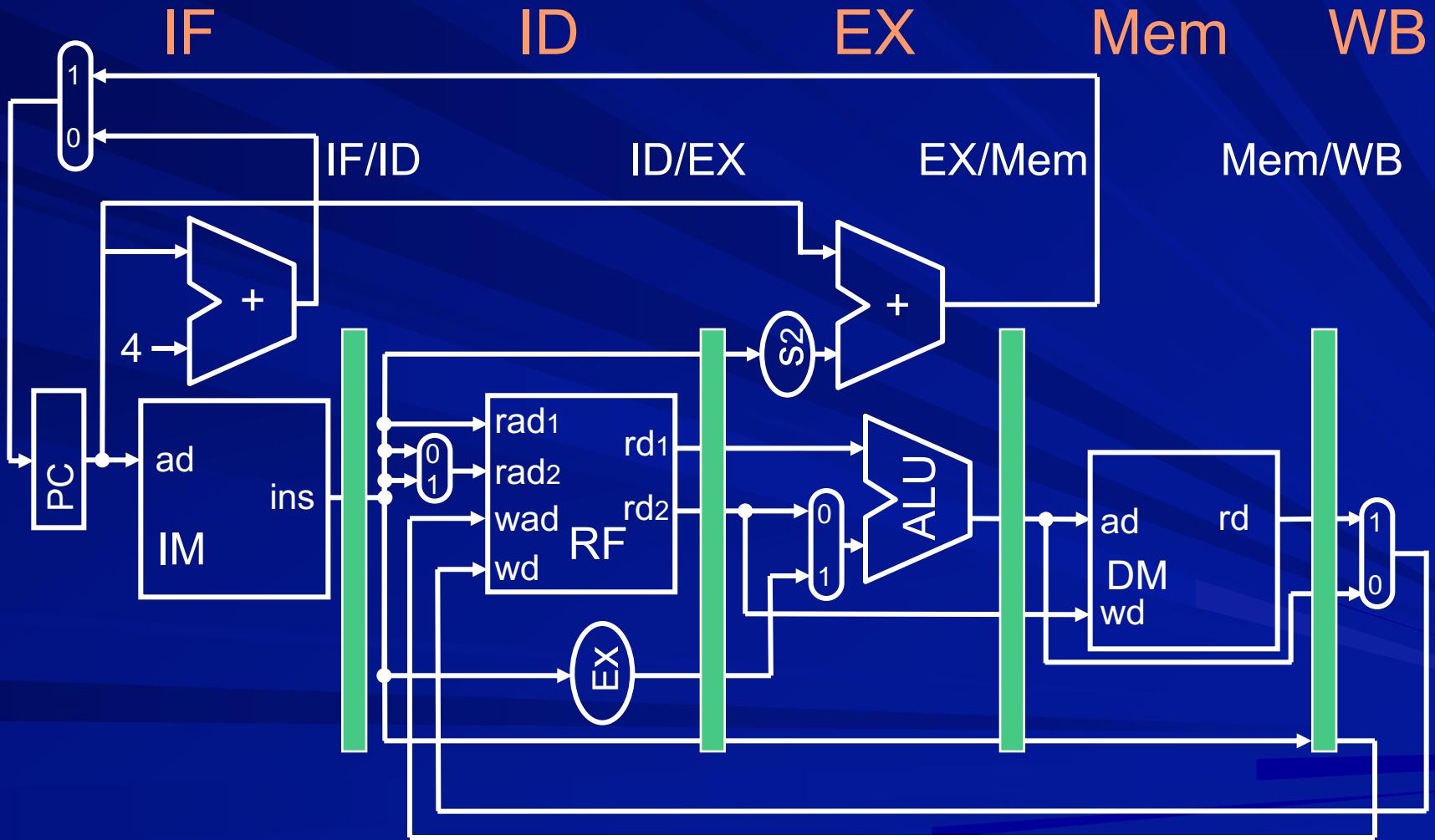
Computer Architecture

Pipelined Processor design –

Controller, Data forwarding

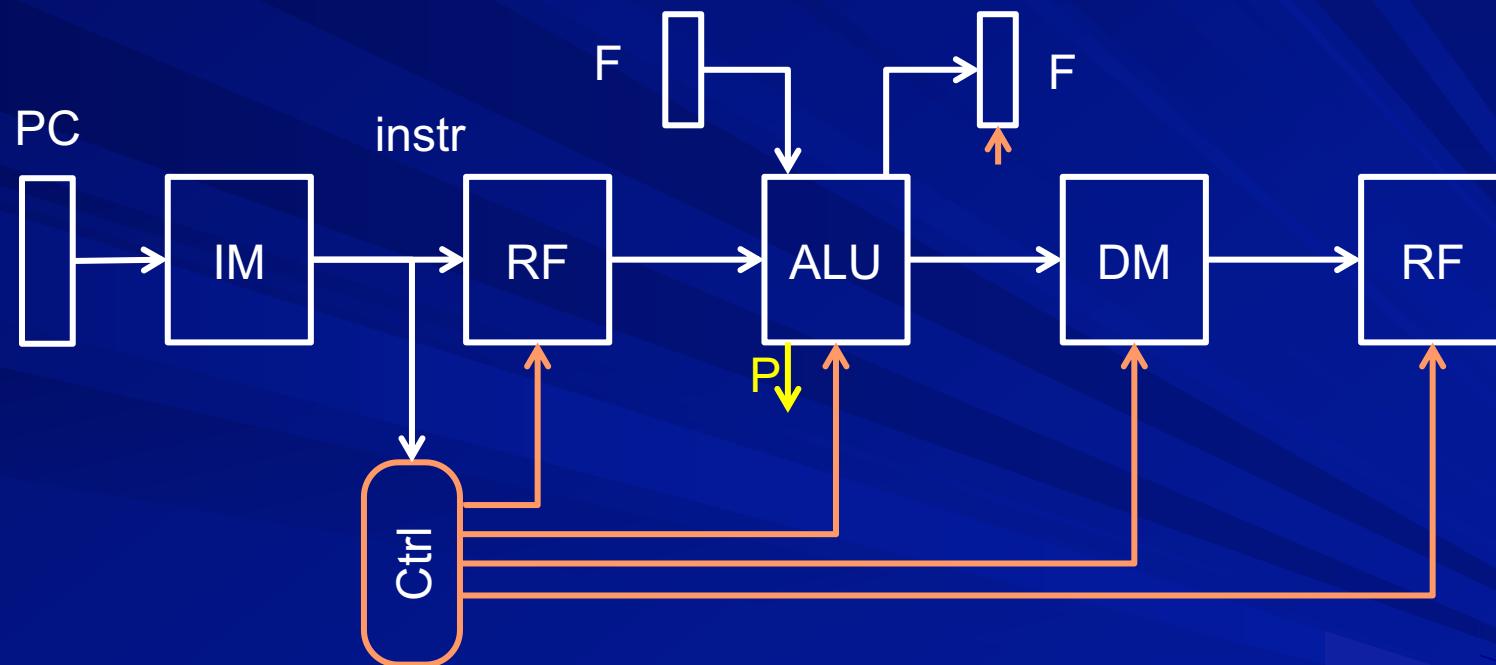
21st February 2022

5 Stage Pipeline

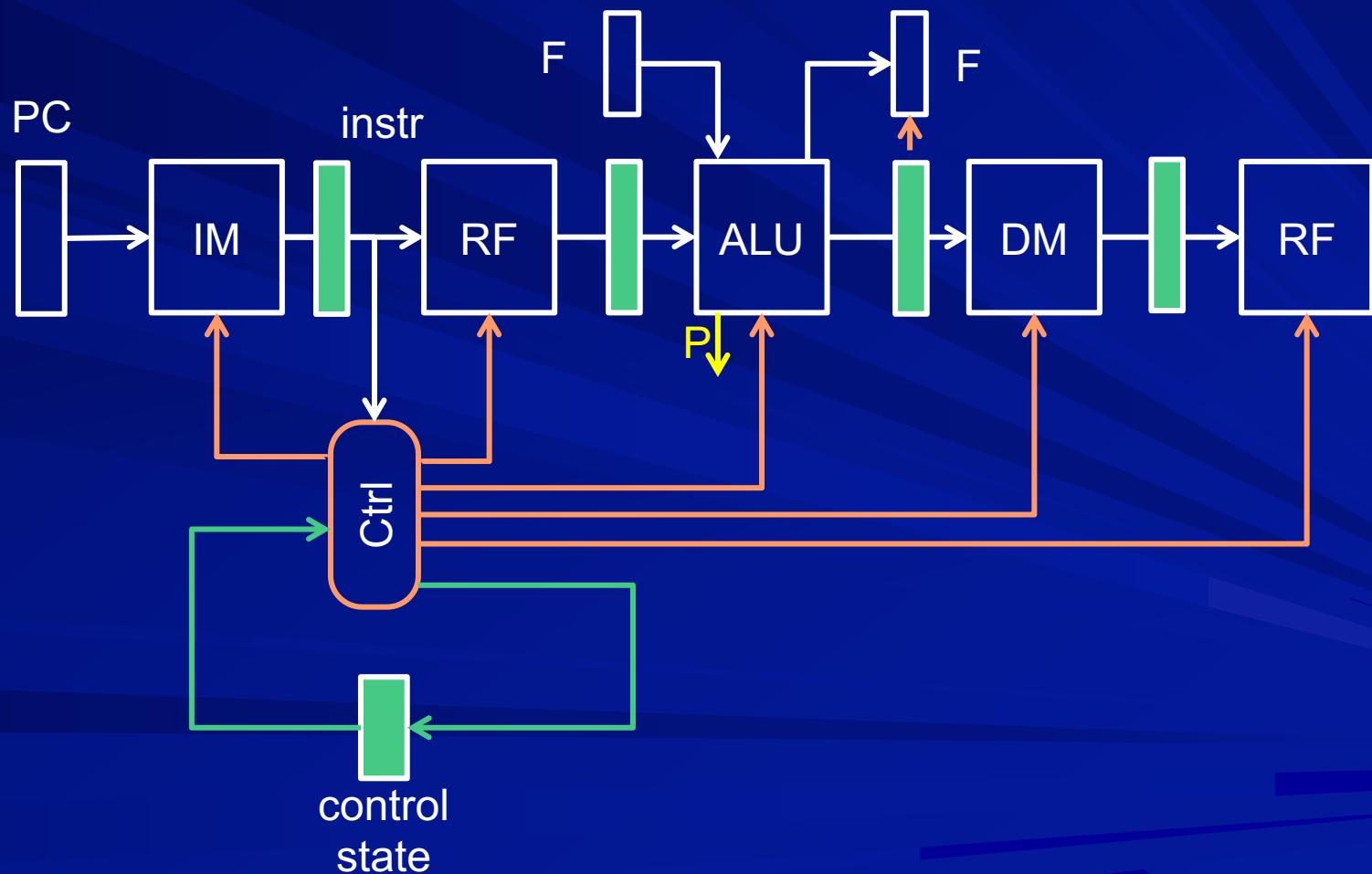


Controllers for different design styles

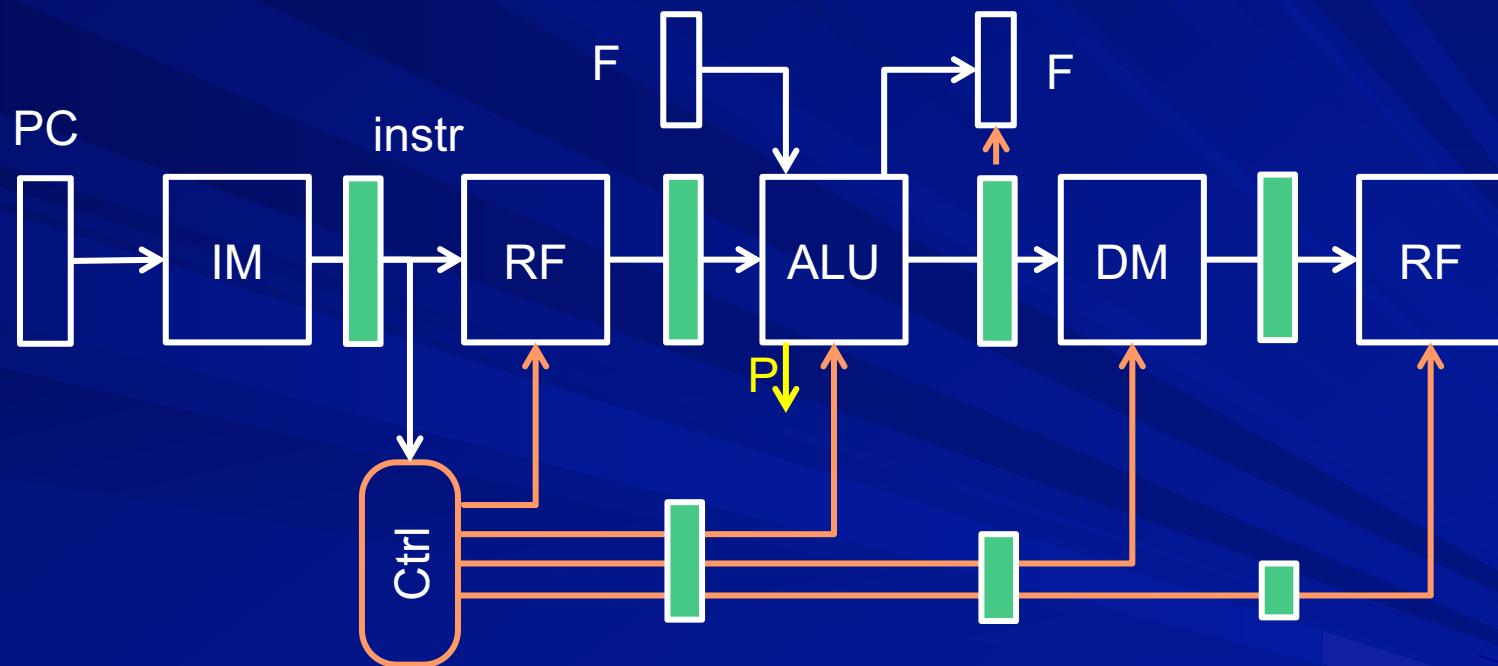
Controller for single cycle DP



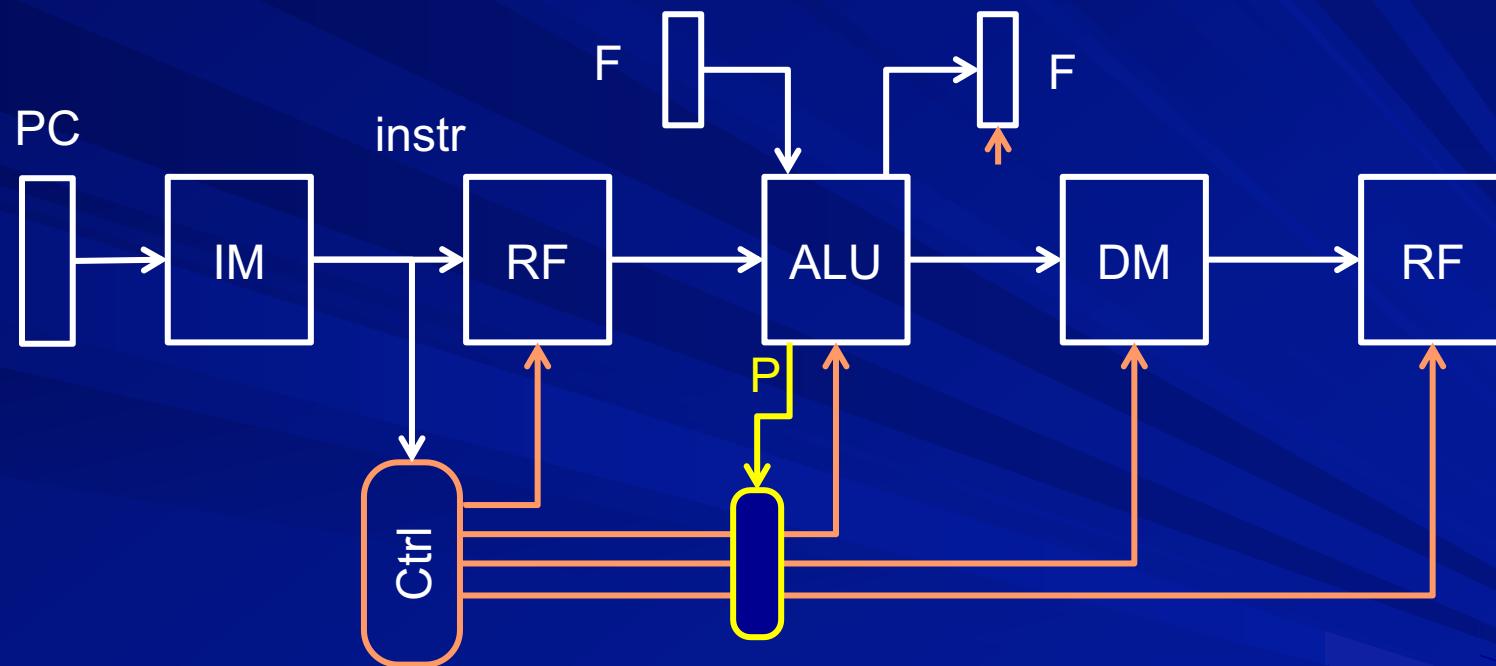
Controller for multi-cycle DP



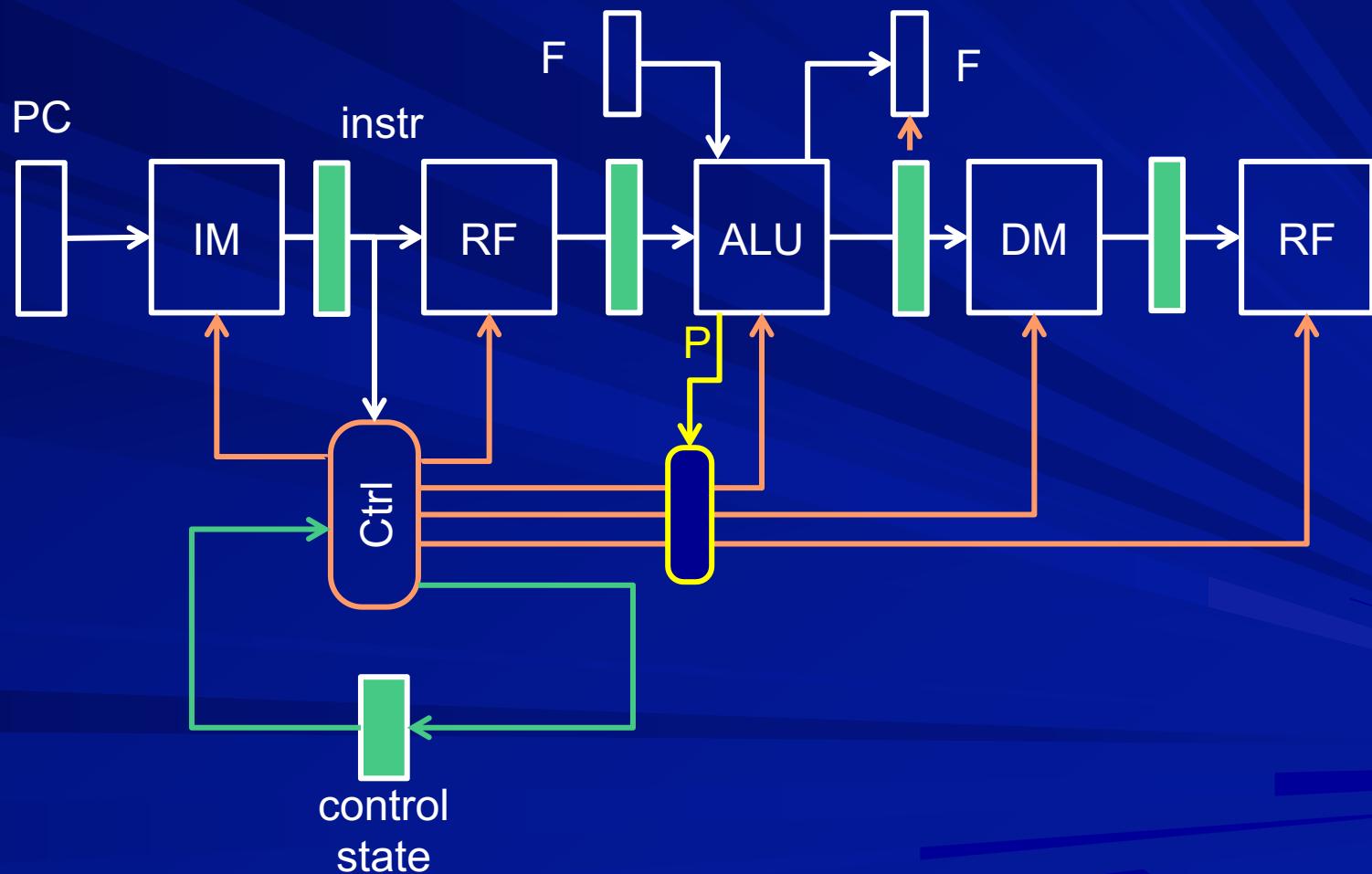
Controller for pipelined DP



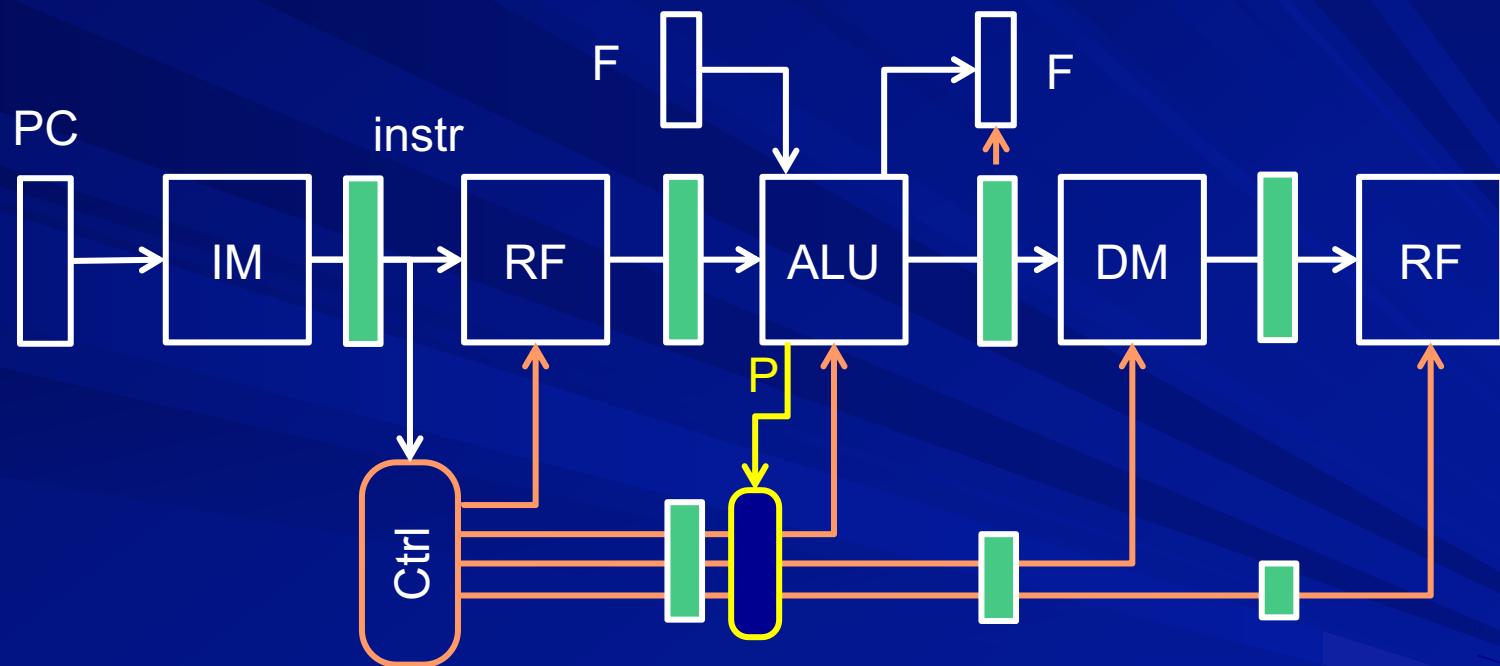
Controller for single cycle DP



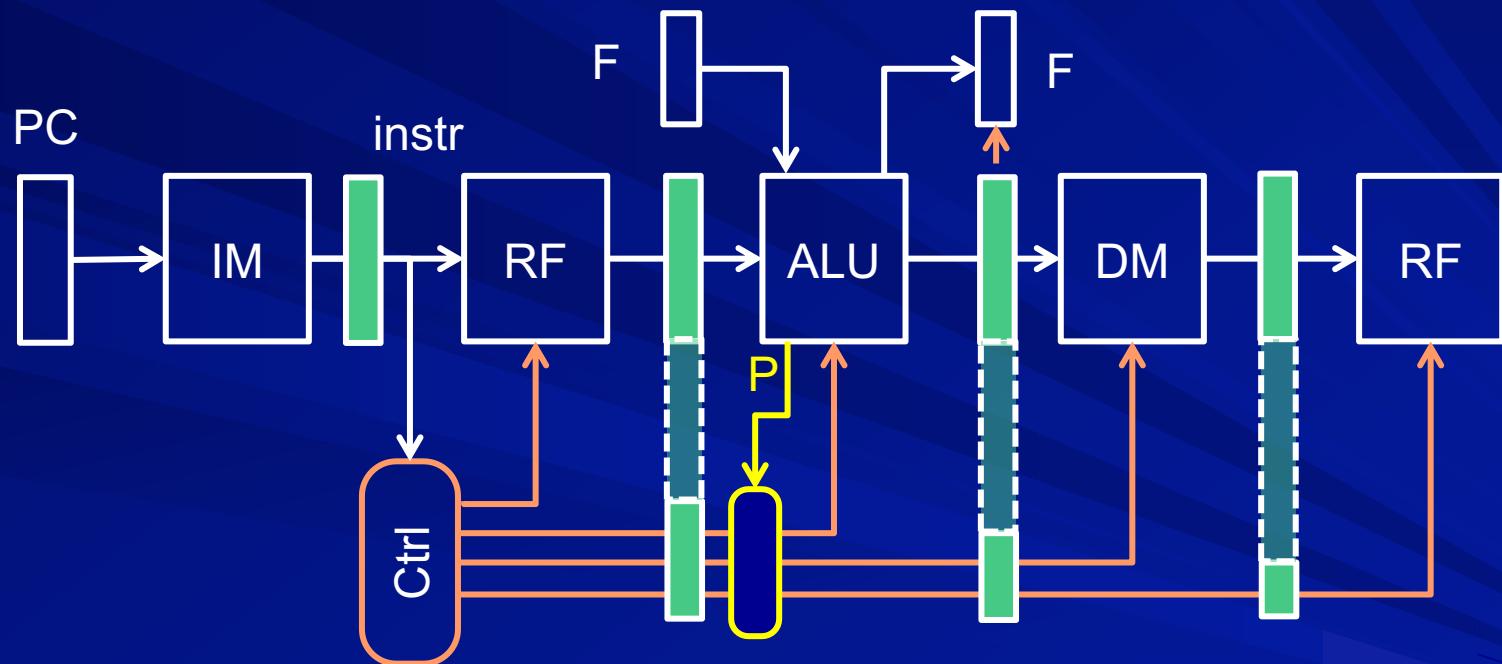
Controller for multi-cycle DP



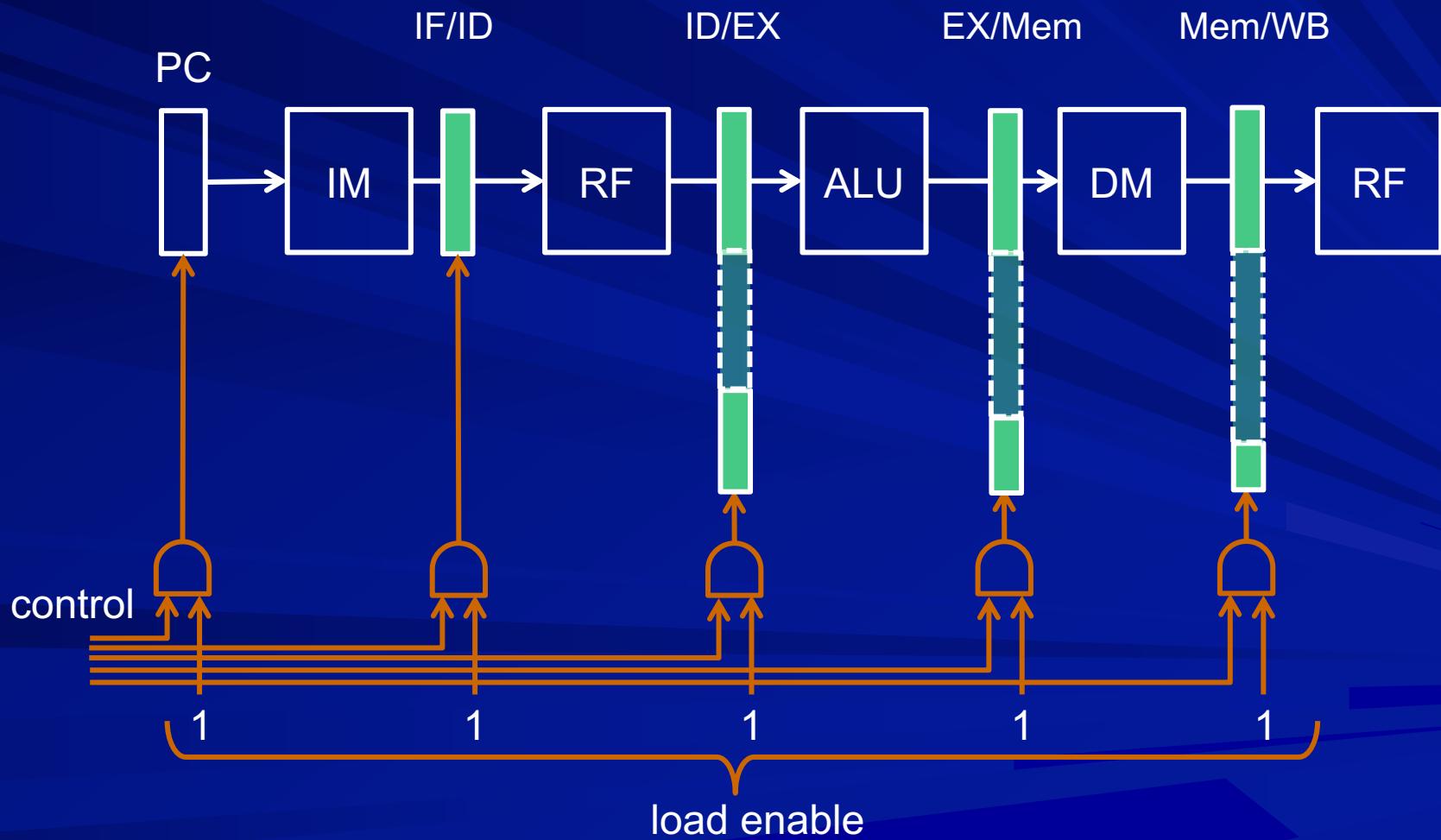
Controller for pipelined DP



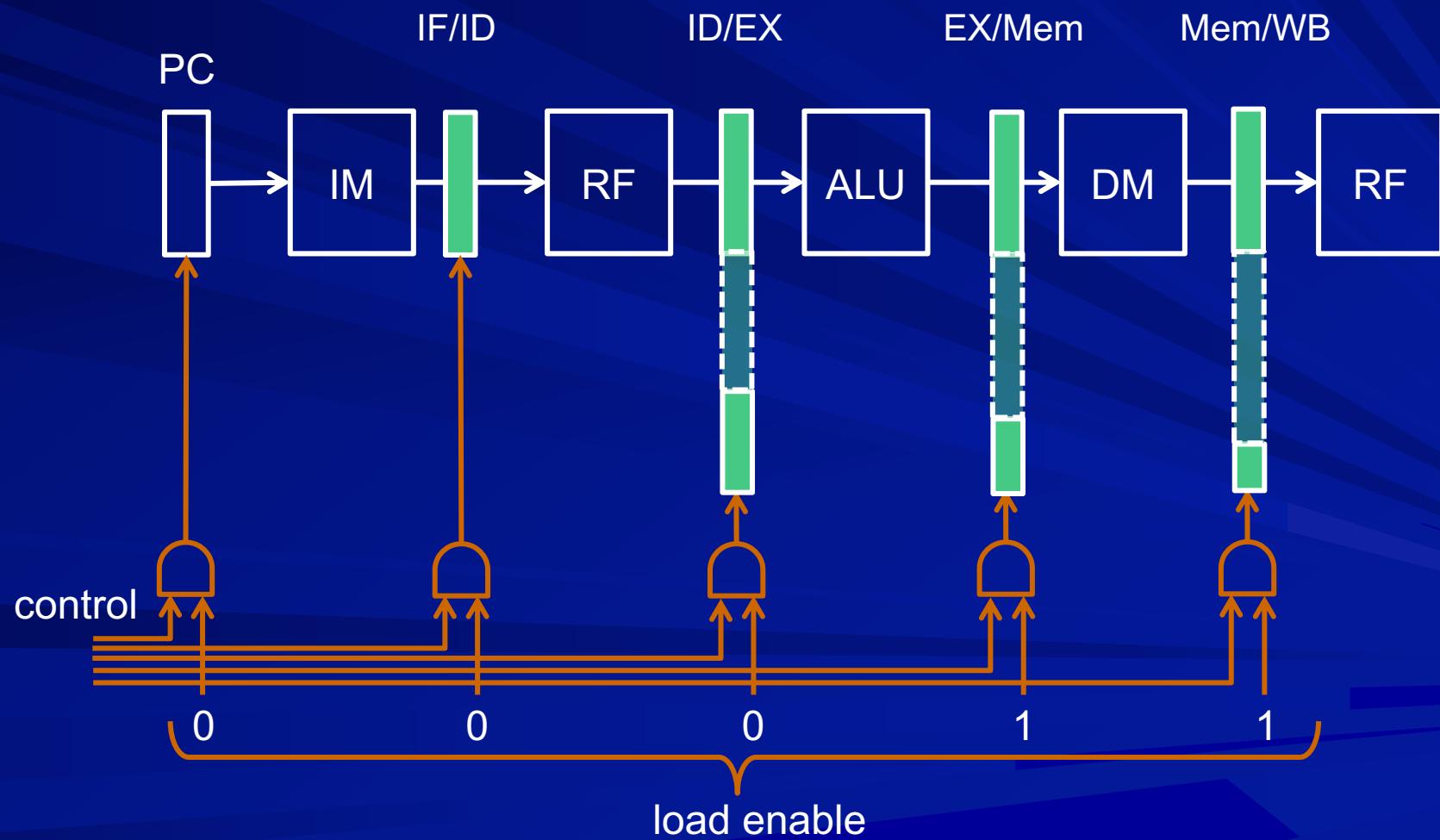
Extending inter-stage registers



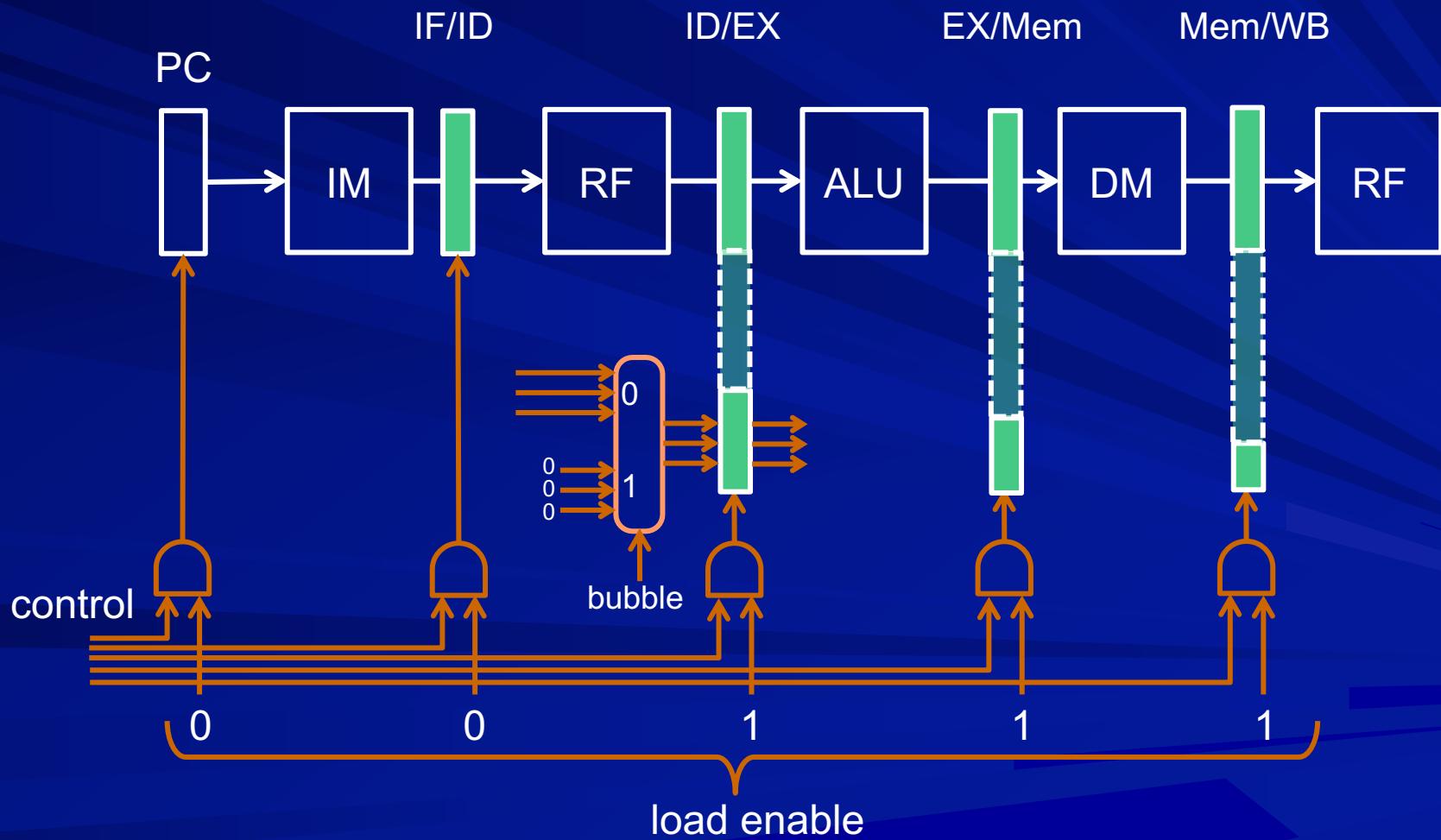
Controlling inter-stage registers



Stalling instructions



Inserting bubbles



Dependence check logic

Condition to be checked:

Operand of instruction in RF stage is a register in
which instruction in ALU stage or DM stage is
going to write

ID/EX.RW = 1 **and**

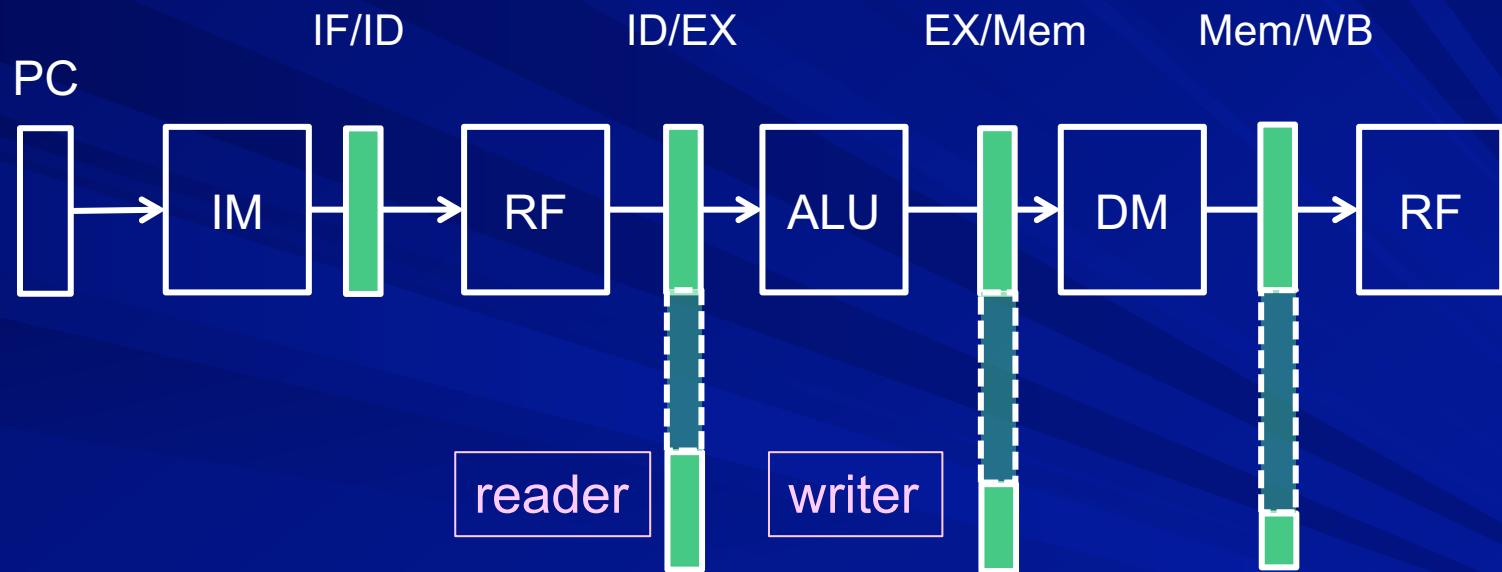
(IF/ID.Rn=ID/EX.Rd **or** IF/ID.Rm=ID/EX.Rd)

EX/Mem.RW = 1 **and**

(IF/ID.Rn=EX/Mem.Rd **or** IF/ID.Rm=EX/Mem.Rd)

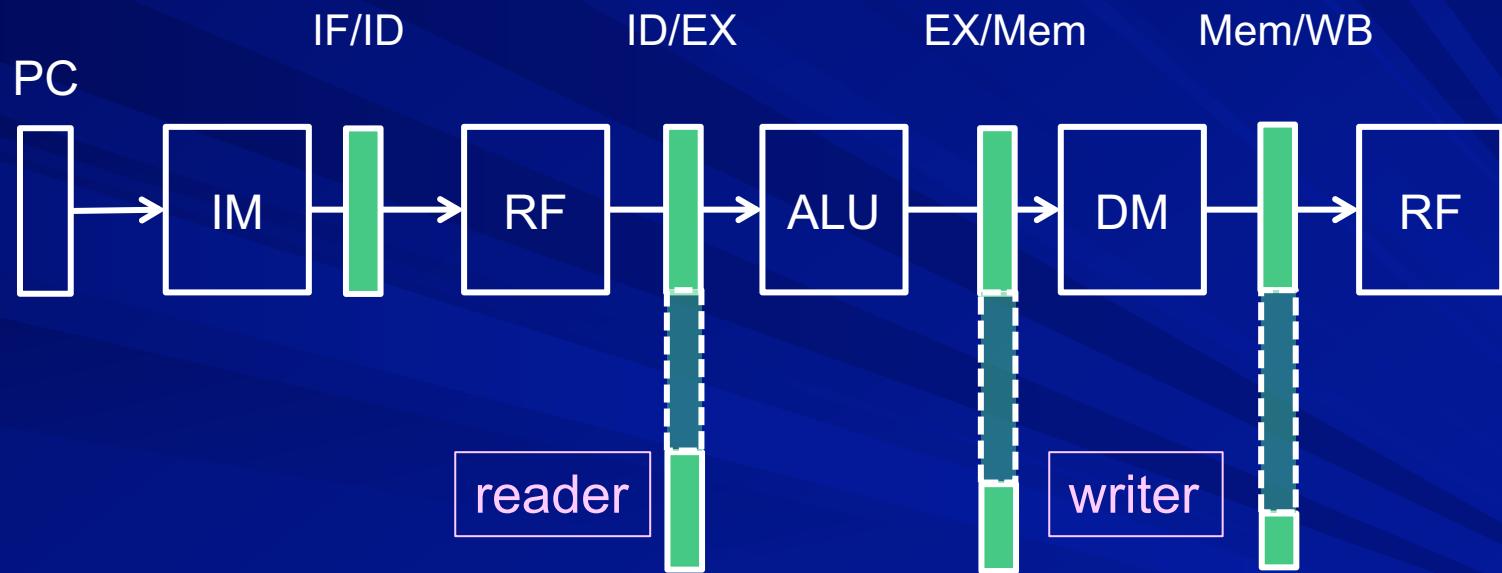
We need to ensure that instruction in RF stage actually
reads Rn and/or Rm (not taken care here)

Dependence check



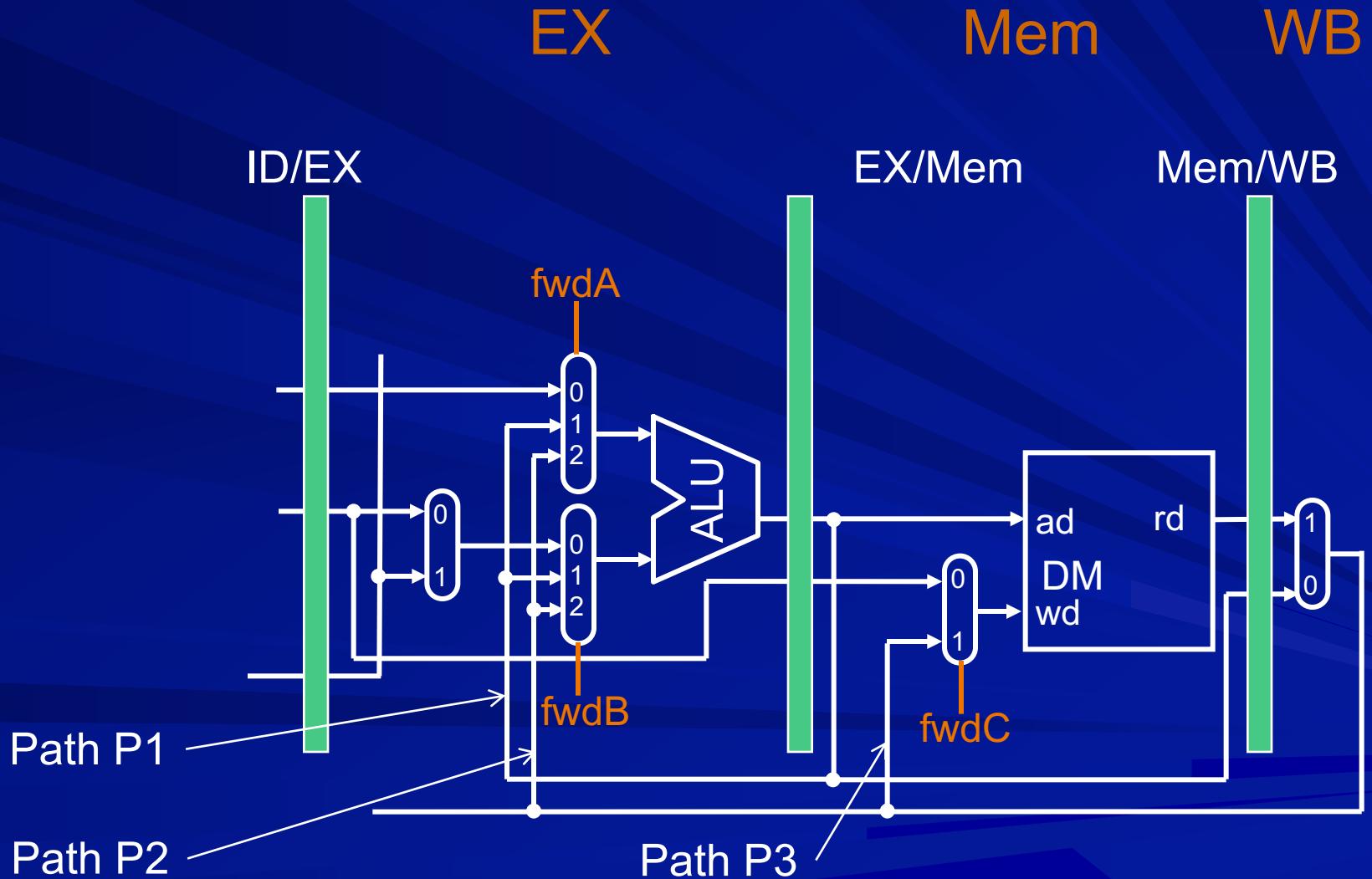
$ID/EX.RW = 1 \text{ and}$
 $(IF/ID.Rn = ID/EX.Rd \text{ or}$
 $IF/ID.Rm = ID/EX.Rd)$

Dependence check



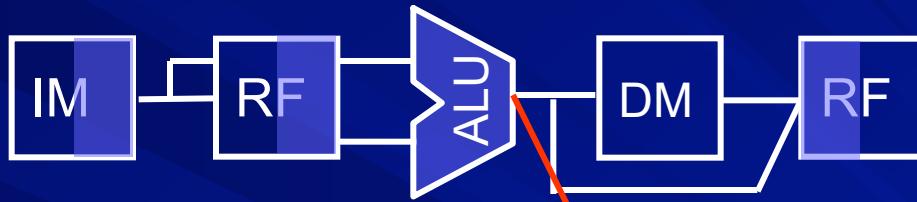
$\text{EX/Mem.RW} = 1$ and
($\text{IF/ID.Rn} = \text{EX/Mem.Rd}$ or
 $\text{IF/ID.Rm} = \text{EX/Mem.Rd}$)

Data forwarding paths

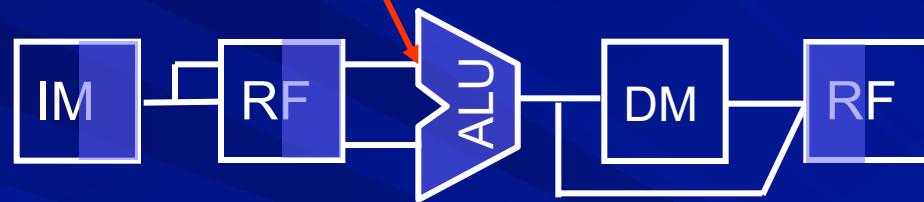


Data forwarding path P1

I:add r6,...



I+1:add r4, r6, ..



Control for forwarding path P1

$fwdA = 1$ if

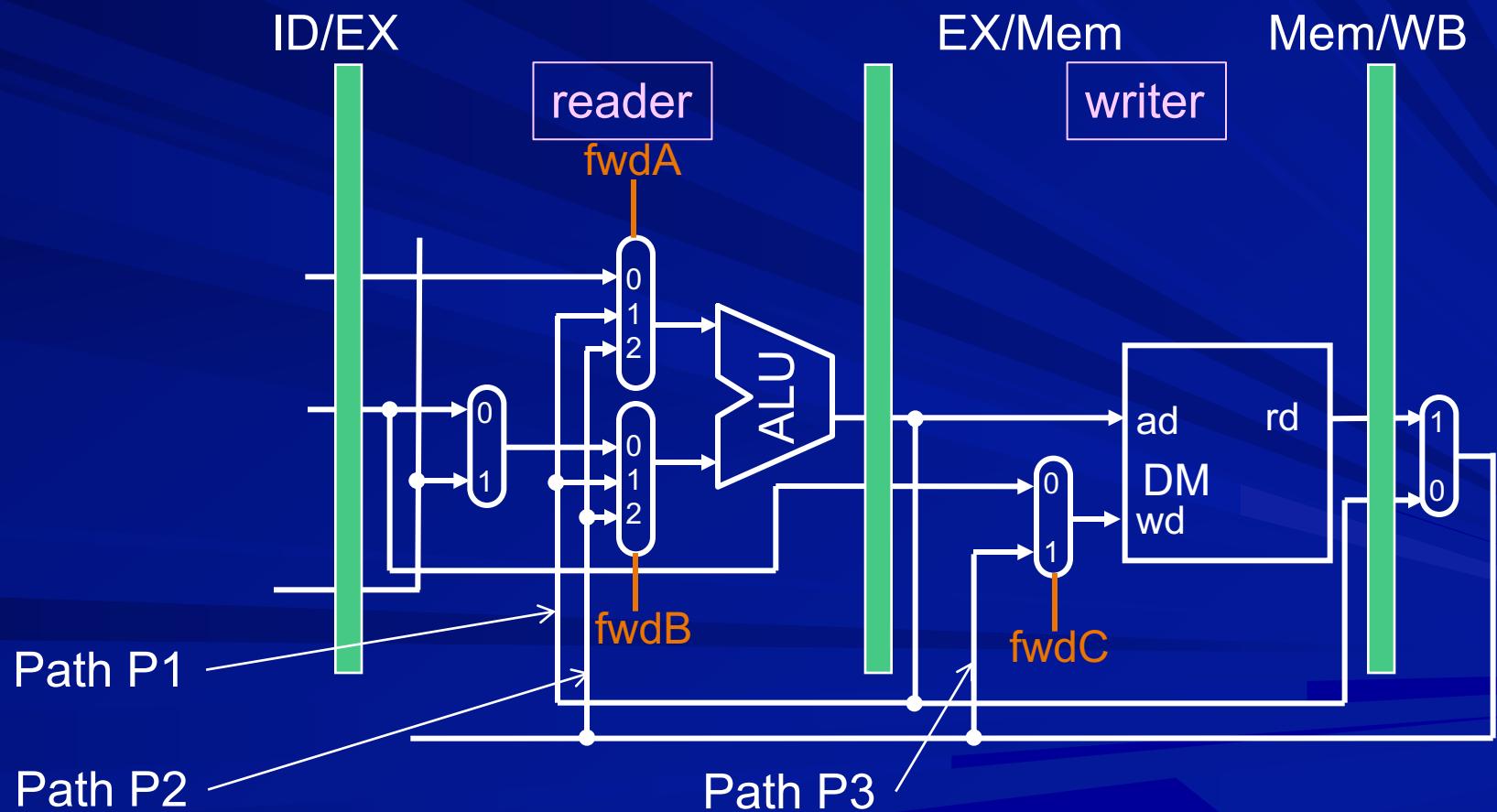
$EX/Mem.RW = 1$ and

$EX/Mem.Rd = ID/EX.Rn$

EX

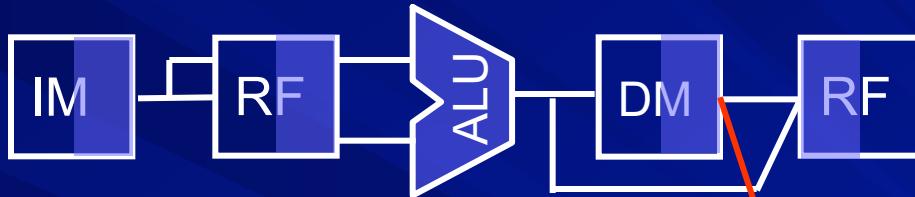
Mem

WB

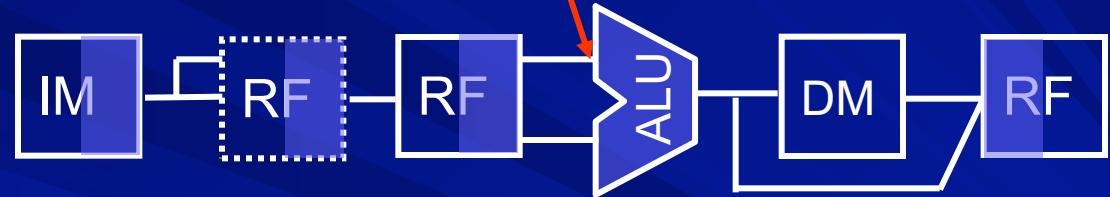


Data forwarding path P2

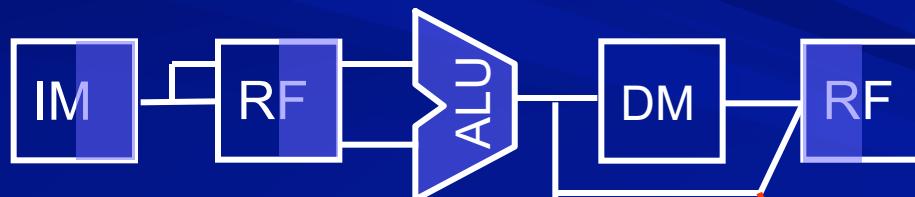
I:ldr r6,..



I+1:add r4,r6,..



I:add r6,..



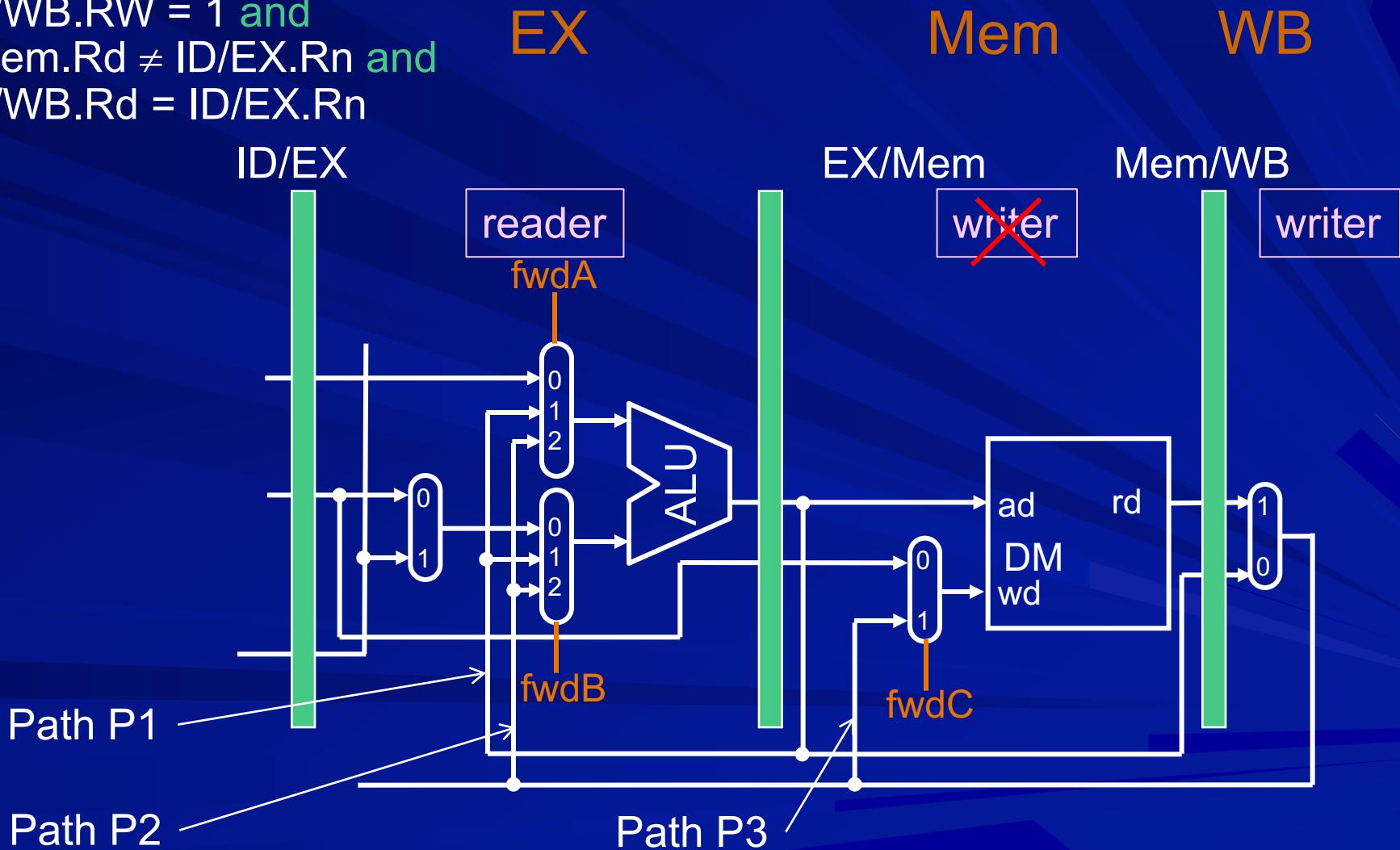
I+2:add r4,r6,..



Control for forwarding path P2

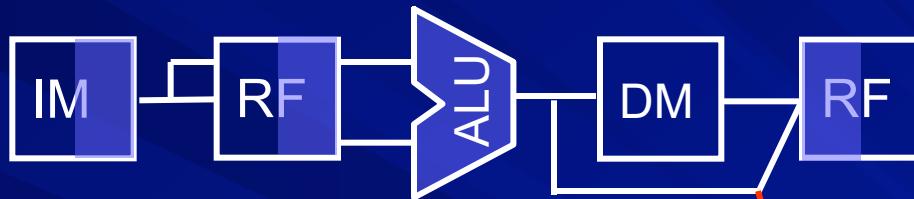
fwdA = 2 if

Mem/WB.RW = 1 and
EX/Mem.Rd \neq ID/EX.Rn and
Mem/WB.Rd = ID/EX.Rn

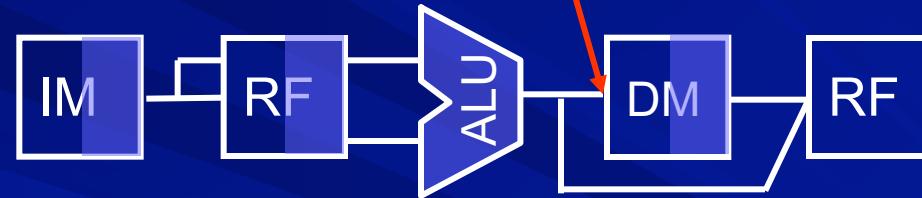


Data forwarding path P3

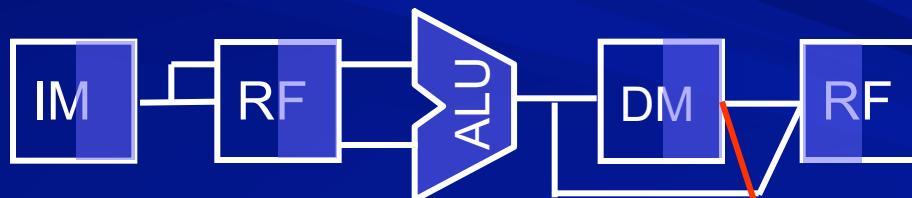
I:add r6,...



I+1:str r6,...



I:ldr r6,...



I+1:str r6,...



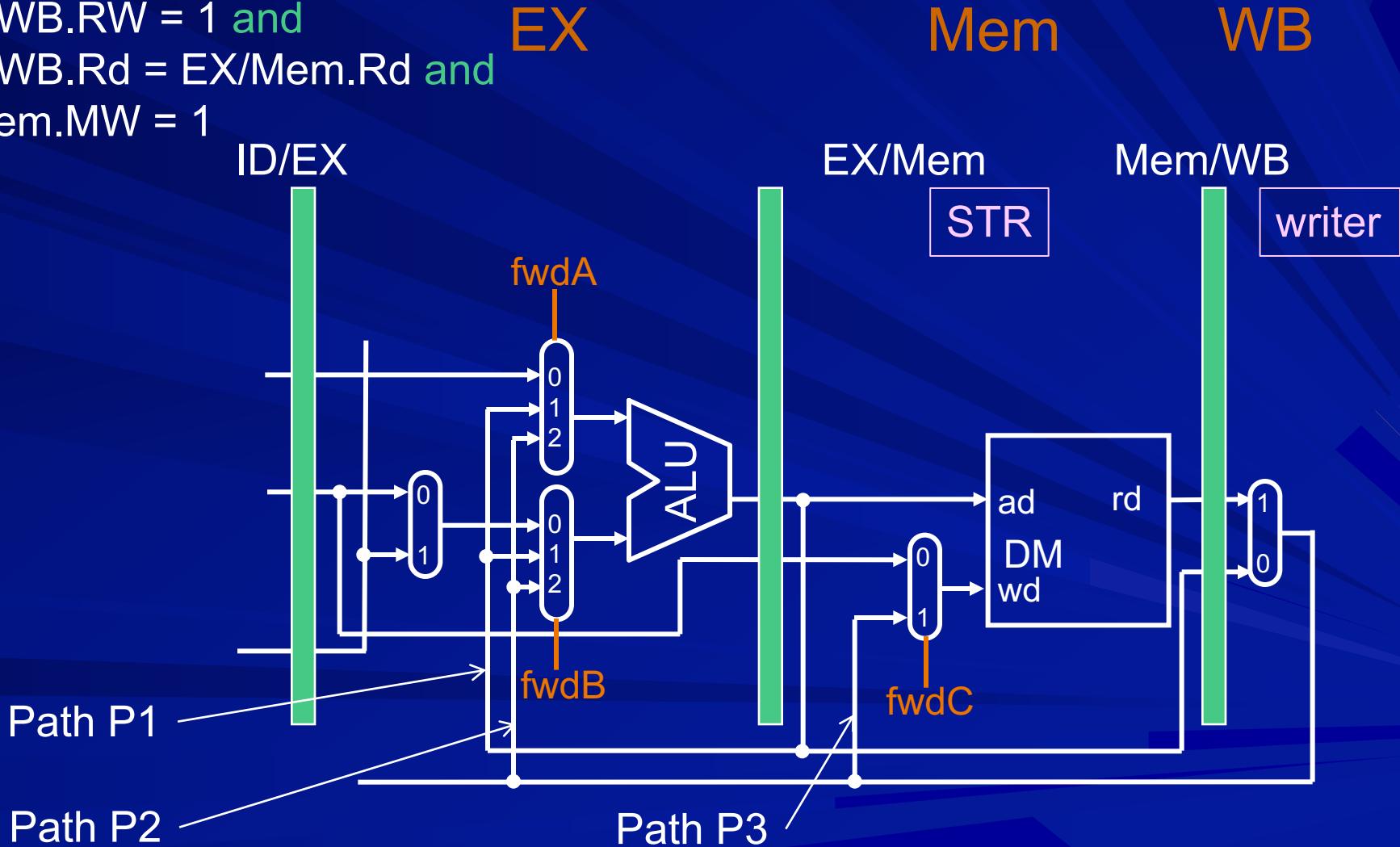
Control for forwarding path P3

fwdC = 1 if

Mem/WB.RW = 1 and

Mem/WB.Rd = EX/Mem.Rd and

EX/Mem.MW = 1



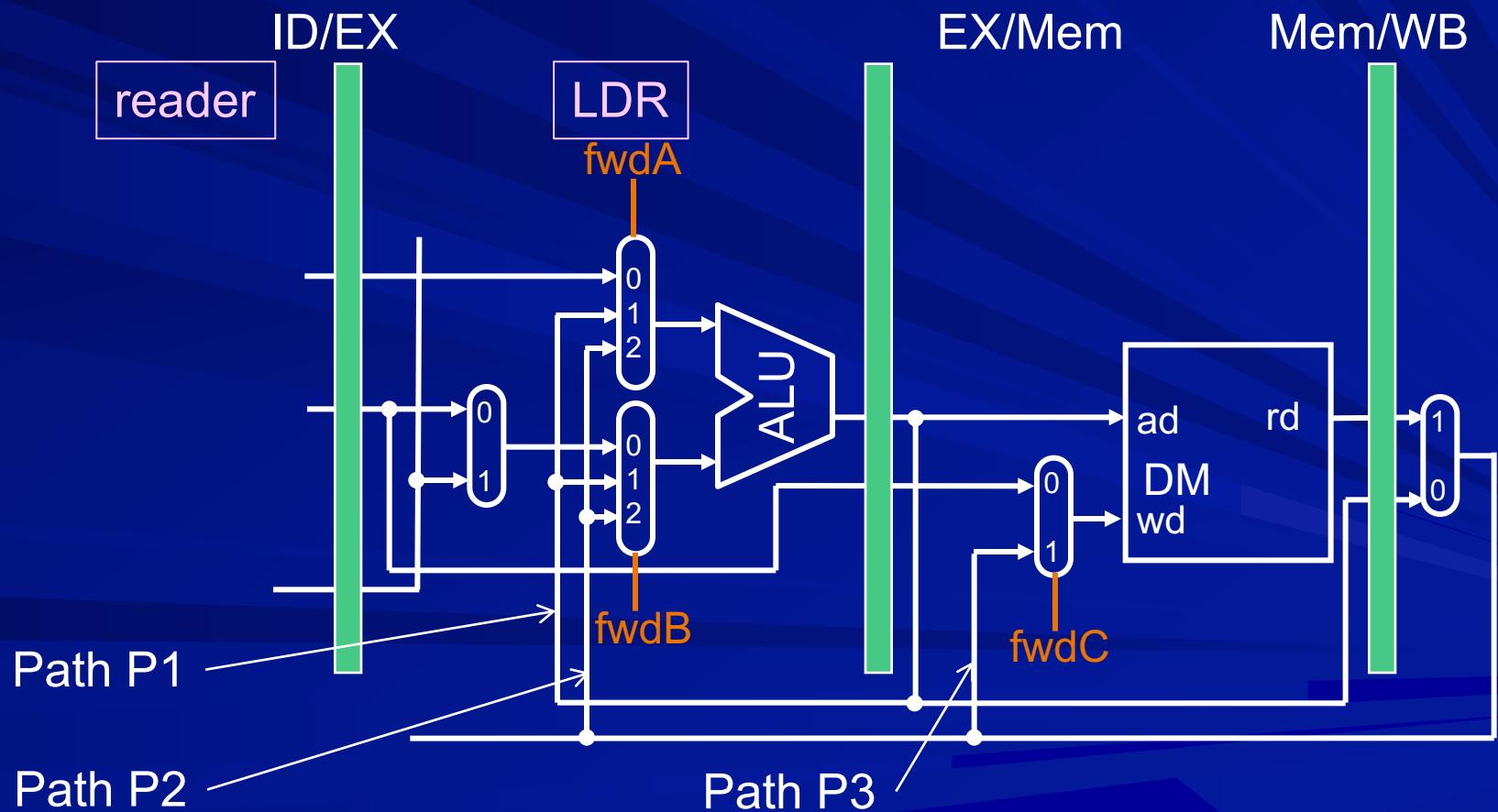
Stalling with data forwarding

ID/EX.MR and
(IF/ID.Rn=ID/EX.Rd or
IF/ID.Rm=ID/EX.Rd)

EX

Mem

WB



Thanks

COL216

Computer Architecture

Pipelined processor : Handling hazards

24th February, 2022

Executing branch instructions

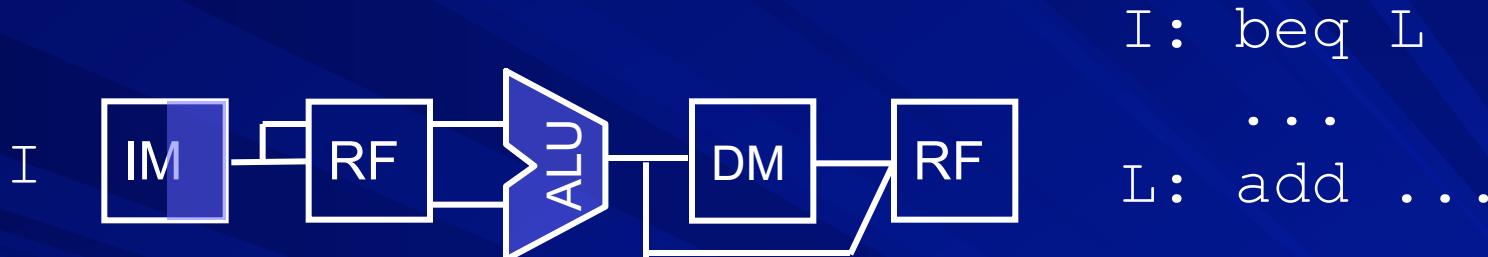
- In which cycle the instruction is found to be a branch instruction? 2
- In which cycle the branch decision is known? 2 or 3
- In which cycle the target address is computed? 3

example

Handling control hazards

- Flush the inline instructions
- Freeze (stall) the inline instructions
- Allow the inline instructions to continue
- Delayed branch
- Predict the branch decision
- Predict the target address

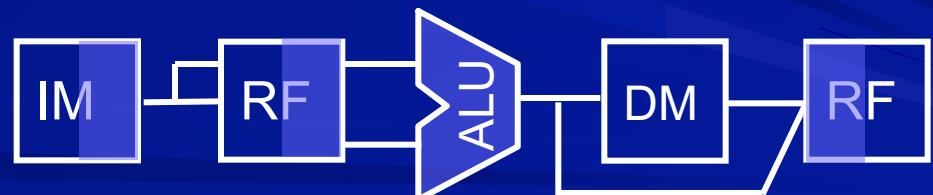
Flush inline instructions



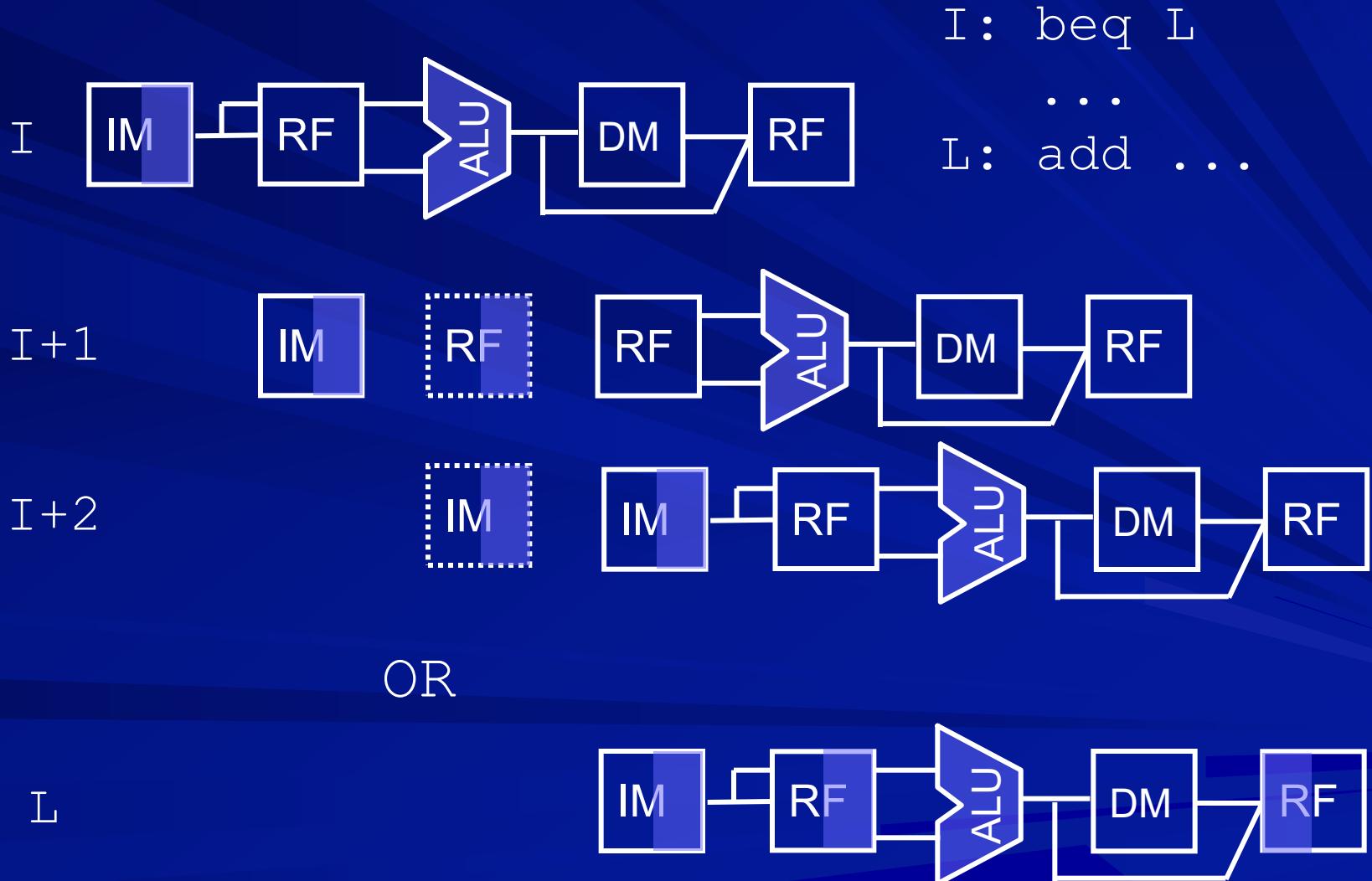
I+1 or L



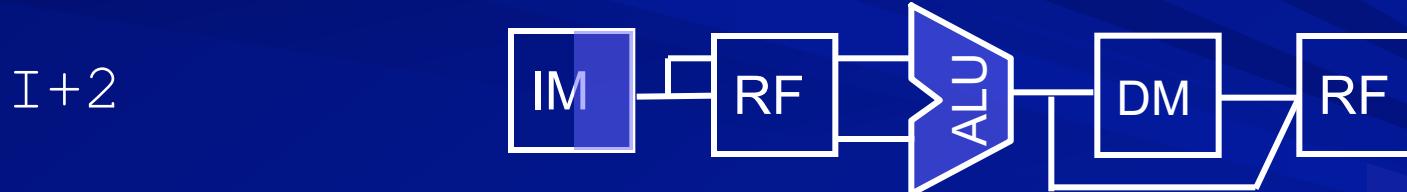
flush these



Freeze inline instructions



Allow inline instructions

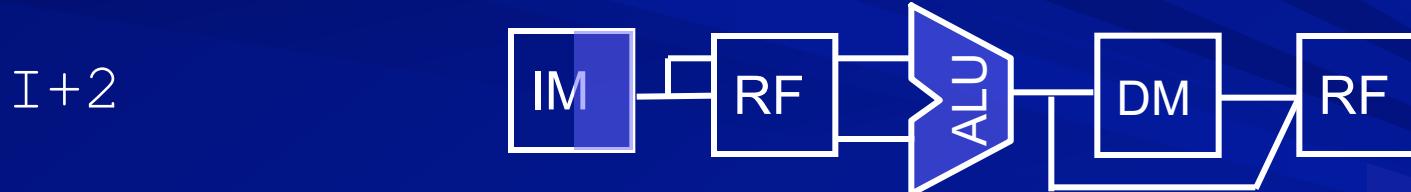
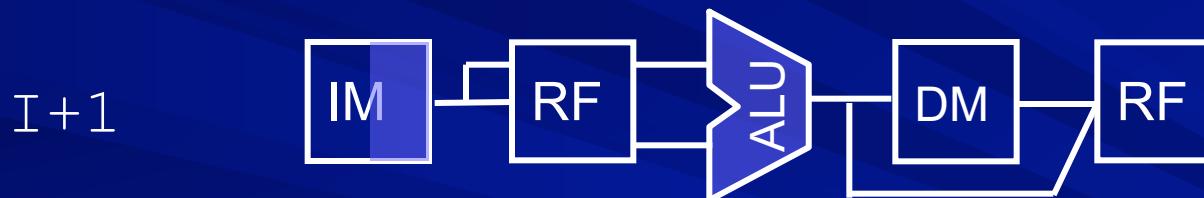


OR

L



Delayed branch



OR

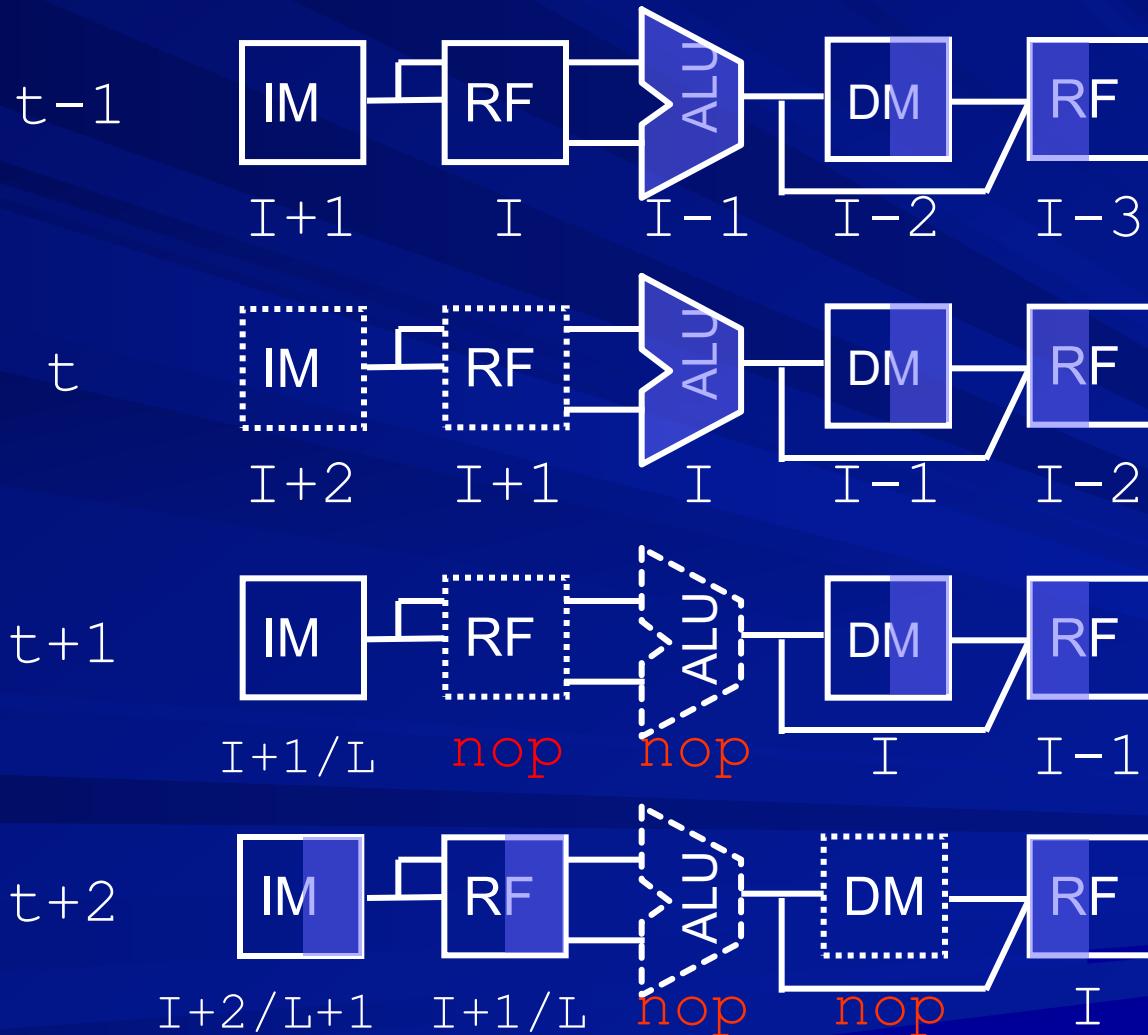
L



Stalls due to control hazards

flush: stage-wise view

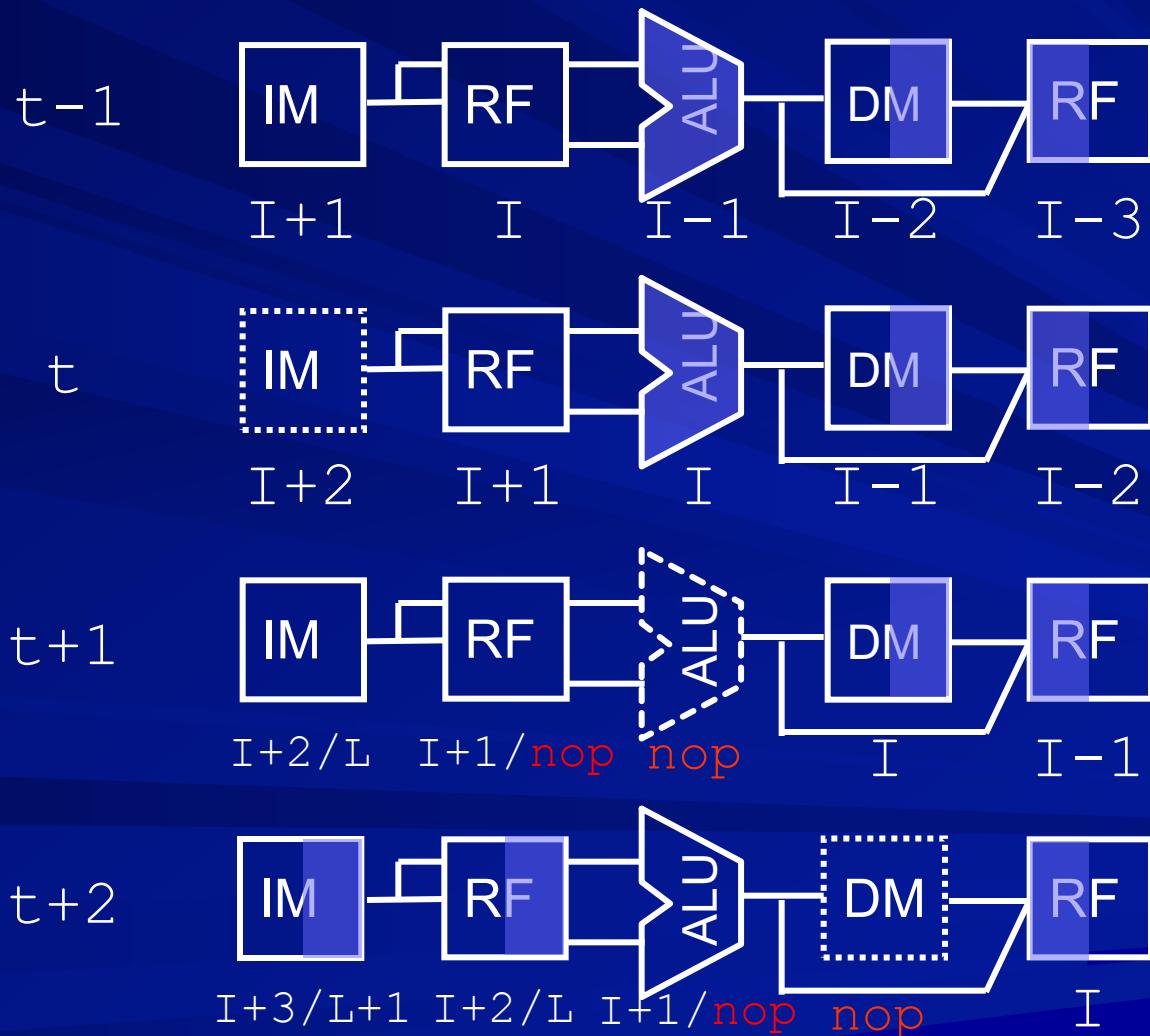
I: beq L



Stalls due to control hazards

freeze: stage-wise view

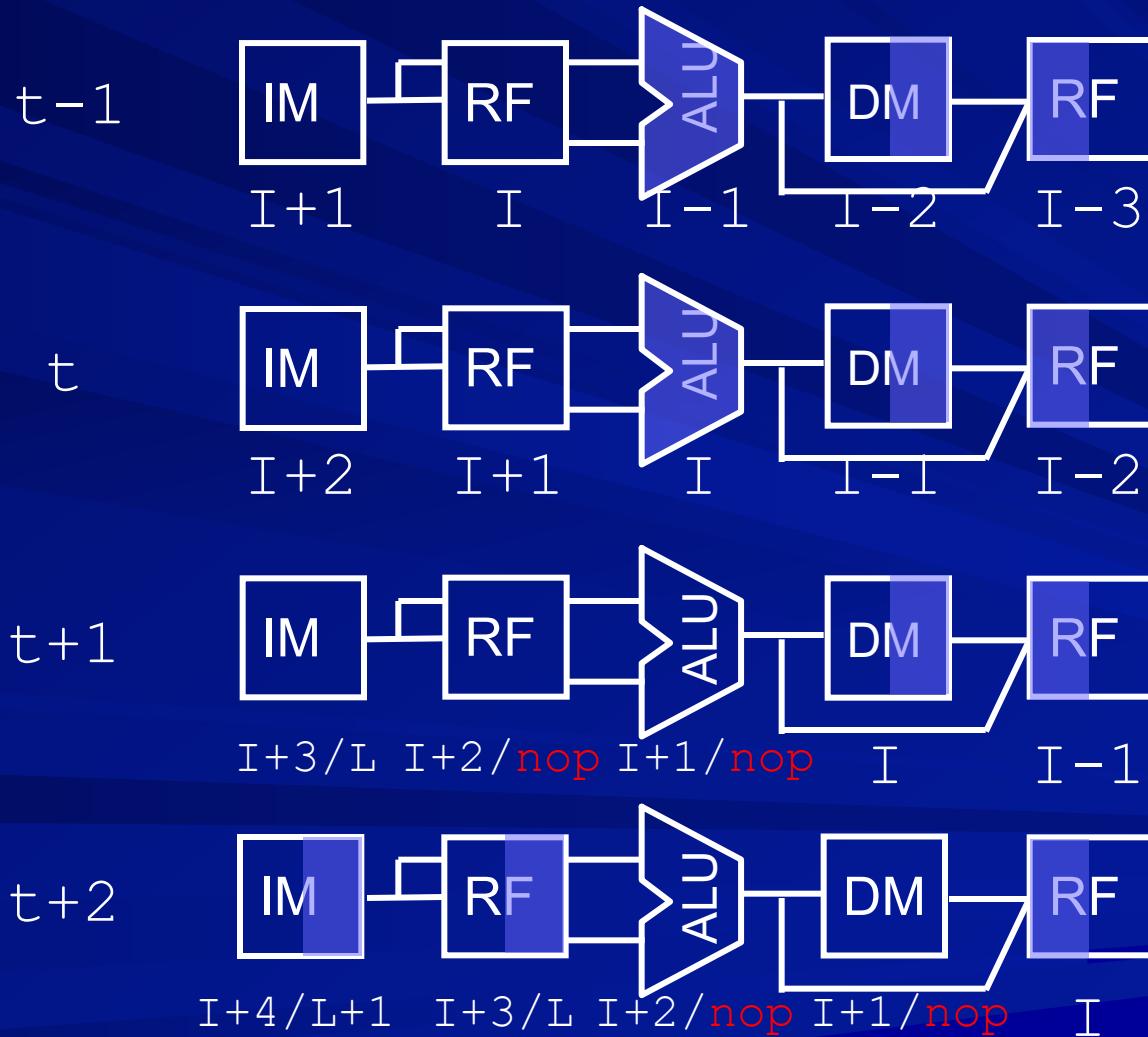
I: beq L



Stalls due to control hazards

continue: stage-wise view

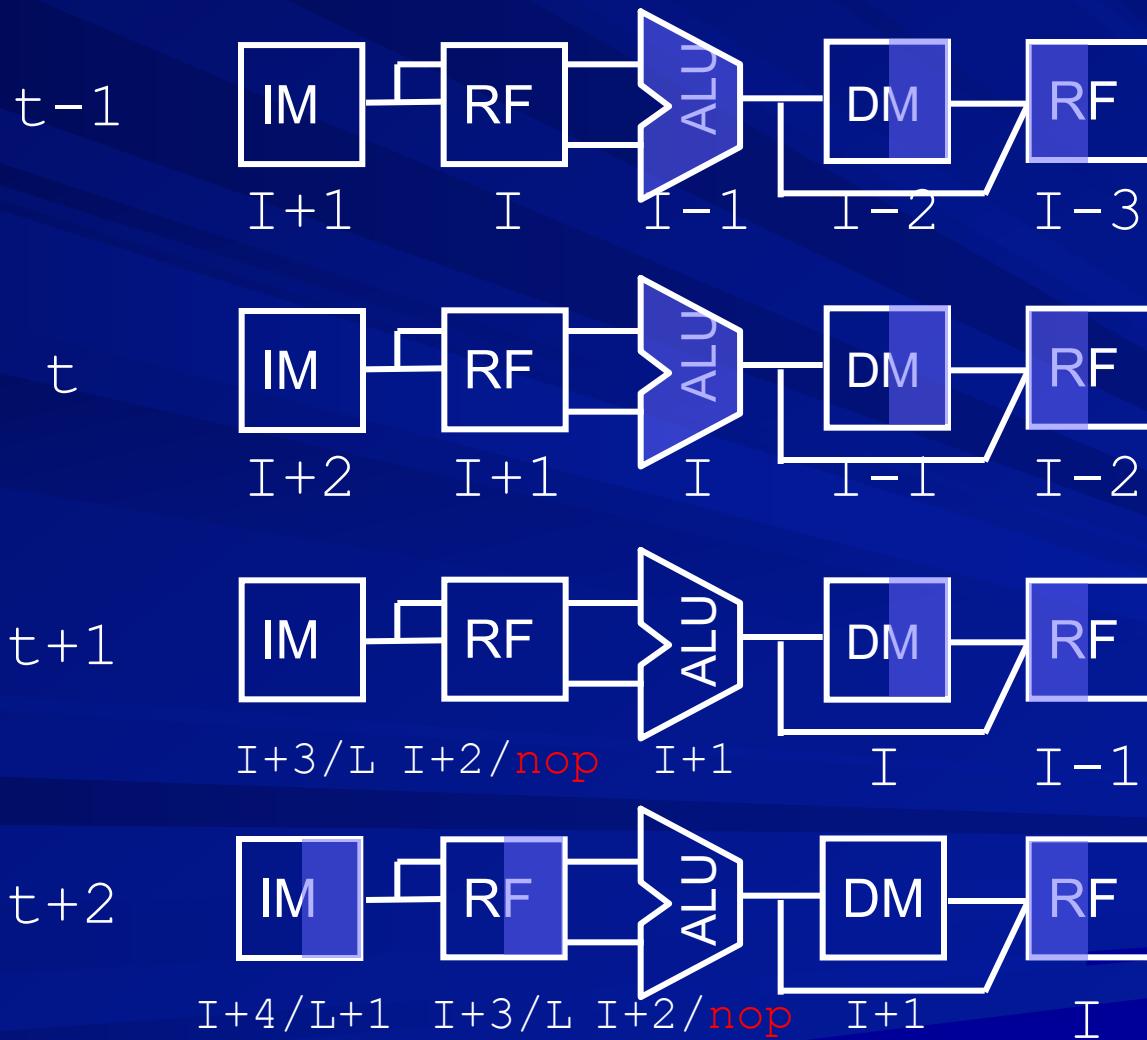
I: beq L



Stalls due to control hazards

delayed branch: stage-wise view

I: beq L



Branch Prediction

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

Strategies:

- Static
- Dynamic

Branch Prediction

- Fixed
 - *always predict inline*
- Static
 - *predict on the basis of instruction type, target address or profiling information*
- Dynamic
 - *predict based on recent history*

Dynamic Branch Prediction - basic idea

Predict based on the history of previous
branch

loop: xxx

xxx

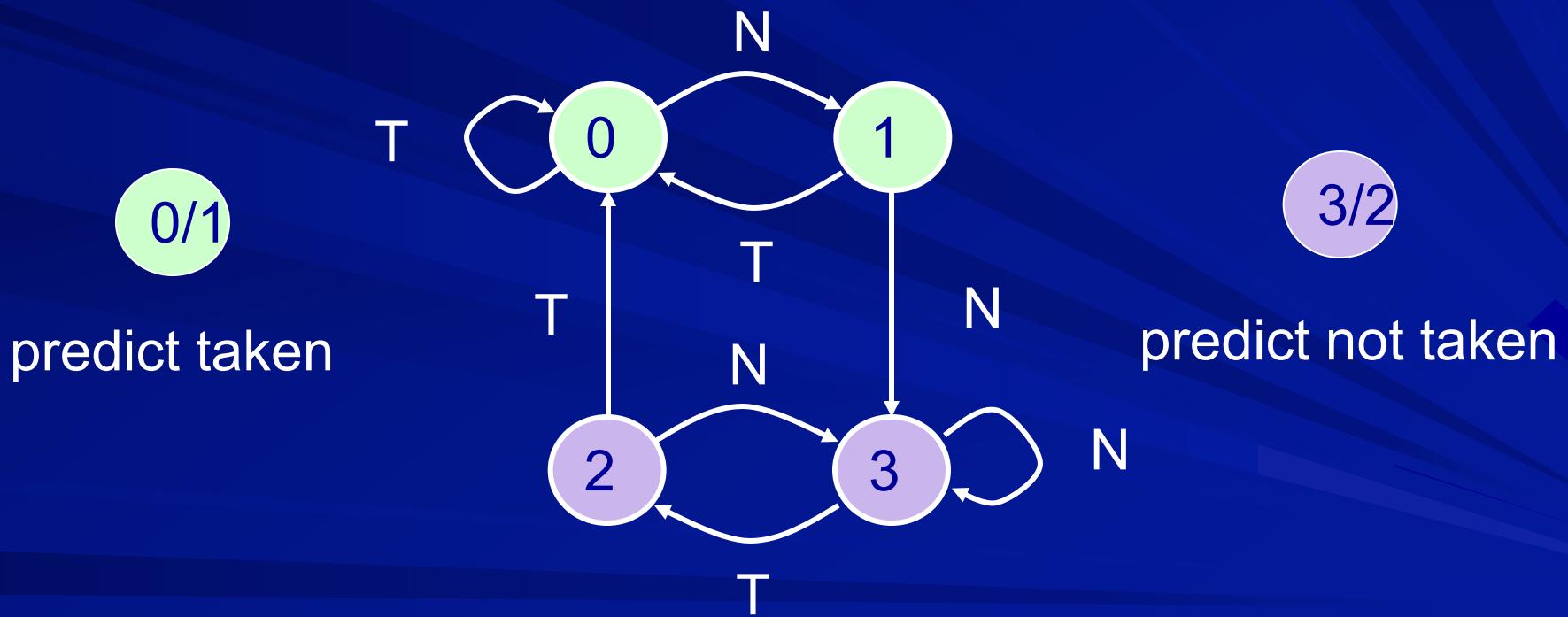
xxx

xxx

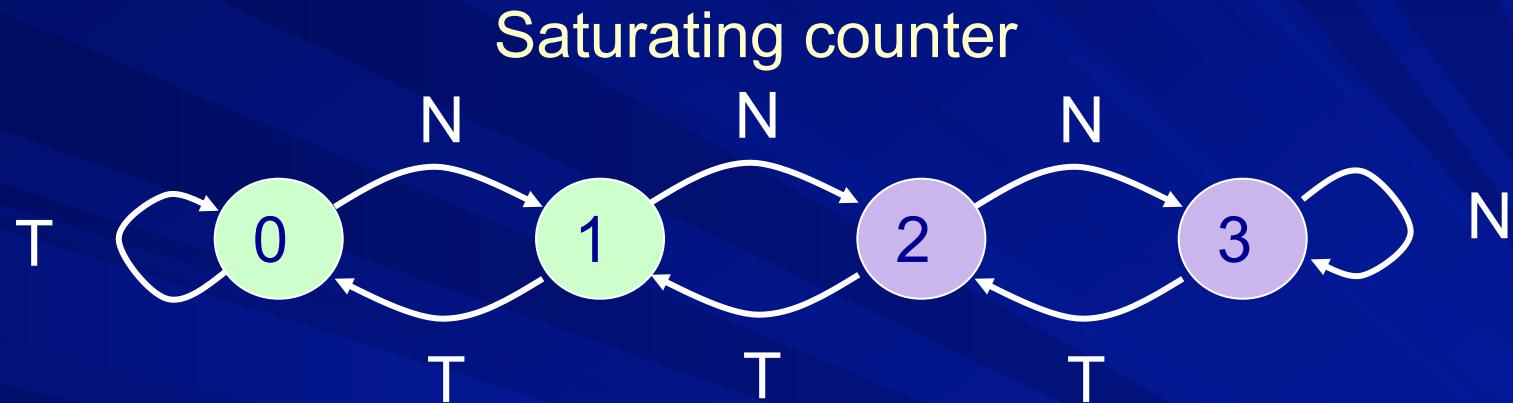
b<cond> loop

2 mispredictions
for every
occurrence of
the loop

Dynamic Branch Prediction - A 2-bit prediction scheme



Another 2-bit prediction scheme



0/1

predict taken

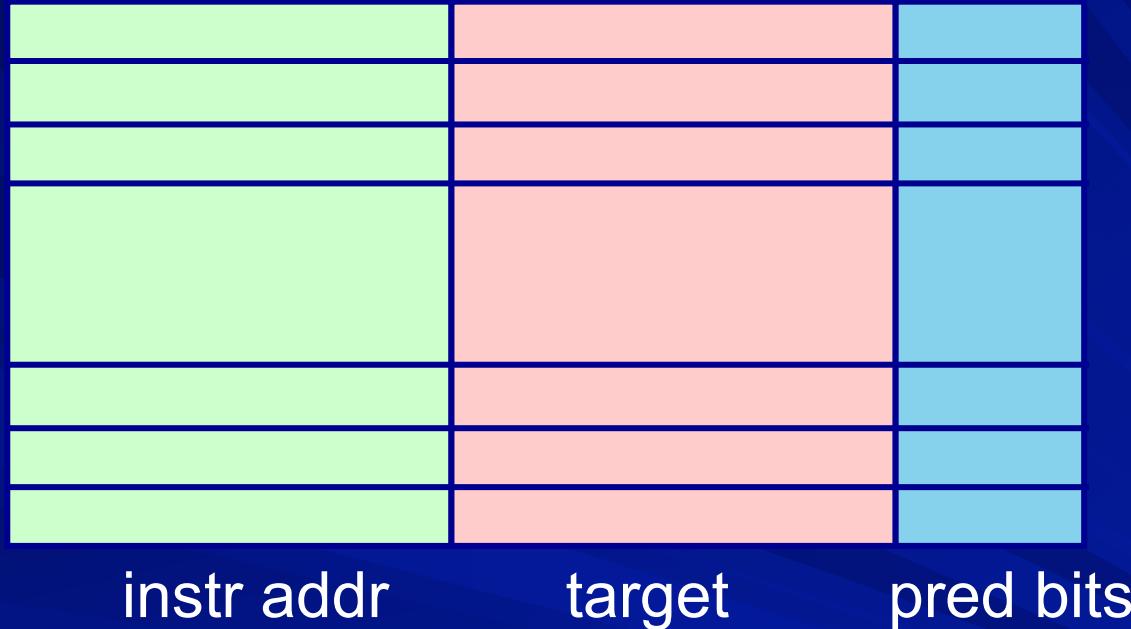
3/2

predict not taken

Dynamic information about branch

- Previous branch decisions
- Previous target address or target instruction
- Stored in
 - Cache
 - Separate buffer
 - Branch History Table (BHT)
 - Branch Target Buffer (BTB)
 - Target Instruction Buffer (TIB)

Branch Target Buffer



- hit \Rightarrow explicit prediction using prediction bits
 - prediction \rightarrow go target (use target info)
 - \rightarrow go inline (ignore target info)
- miss \Rightarrow go inline

Accessing BTB

- In which cycle do you access BTB?
- Before checking condition?
- Before address computation?
- Just after decoding?
- Along with instruction fetch?

Correlation between branches

B1: if (x)

...

B2: if (y)

...

$z = x \&& y$

B3: if (z)

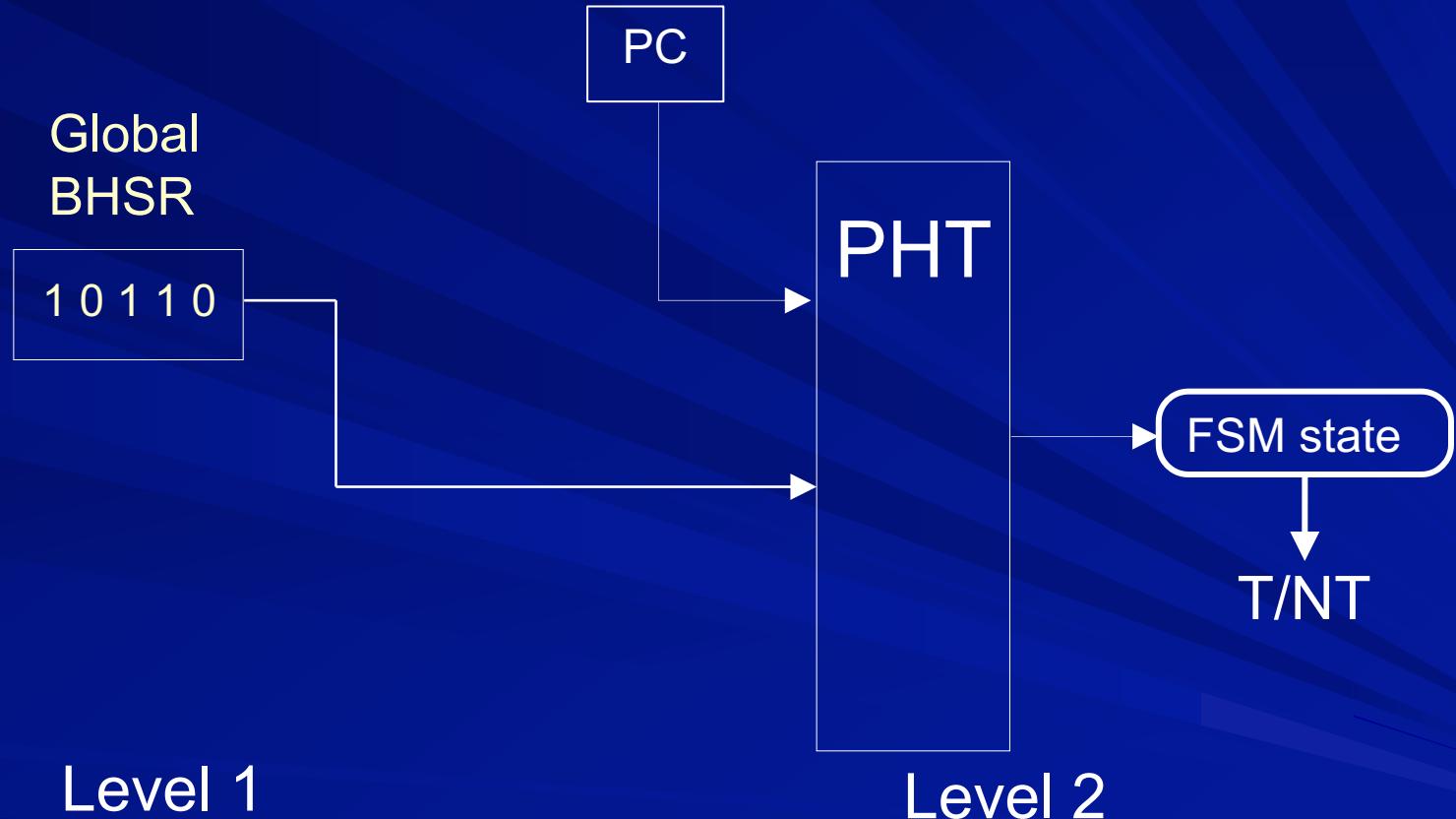
...

- B3 can be predicted with 100% accuracy based on the outcomes of B1 and B2

Two-Level Branch Predictors

- Level 1
 - Branch History Shift Register (BHSR) - last n occurrences
 - Captures patterned behavior of groups of branches
- Level 2
 - Pattern History Table (PHT) - states of predictor FSMs
 - Captures behavior of individual branches

A two-level branch predictor



Bits from PC and BHSR are combined to index PHT

Optimizing programs for pipeline

- Instruction reordering
- Moving instructions across branches
- Delayed branches
- Predication

Running program on pipeline

```
    mov r1, #1
    mov r2, #100
L1:   str r1, [r2, #0]
        add r2, r2, #4
        add r1, r1, #1
        cmp r1, #11
        bne L1
        mov r3, #0
        mov r2, #100
L2:   sub r1, r1, #1
        cmp r1, #0
        beq Over
        ldr r4, [r2, #0]
        add r3, r3, r4
        add r2, r2, #4
        b L2
```

Over:

Running program on pipeline

Without stalls

	mov	r1, #1	1		
	mov	r2, #100	2		
L1:	str	r1, [r2, #0]	3	8	.. 48
	add	r2, r2, #4	4		
	add	r1, r1, #1	5		
	cmp	r1, #11	6		
	bne	L1	7	12	.. 52
	mov	r3, #0	53		
	mov	r2, #100	54		
L2:	sub	r1, r1, #1	55	62	.. 118
	cmp	r1, #0	56		
	beq	Over	57		
	ldr	r4, [r2, #0]	58		
	add	r3, r3, r4	59		
	add	r2, r2, #4	60		
	b	L2	61	68	.. 124
Over:			128		

Running program on pipeline

L1:	mov r1, #1	1	
	mov r2, #100	2	
	str r1, [r2, #0]	5	14 .. 86
	add r2, r2, #4	6	
	add r1, r1, #1	7	
	cmp r1, #11	10	
	bne L1	11	20 .. 92
	mov r3, #0	95	
	mov r2, #100	96	
L2:	sub r1, r1, #1	97	112.. 232
	cmp r1, #0	100	
	beq Over	101	
	ldr r4, [r2, #0]	104	
	add r3, r3, r4	107	
	add r2, r2, #4	108	
	b L2	109	124..244
Over:		254	

Execute inline instructions after branch

L1:	mov r1, #1	1	
	mov r2, #100	2	
	str r1, [r2, #0]	5	14 .. 86
	add r2, r2, #4	6	
	add r1, r1, #1	7	
	cmp r1, #11	10	
	bne L1	11	20 .. 92
	mov r3, #0	95	
	mov r2, #100	96	
L2:	sub r1, r1, #1	97	110.. 214
	cmp r1, #0	100	
	beq Over	101	
	ldr r4, [r2, #0]	102	
	add r3, r3, r4	105	
	add r2, r2, #4	106	
	b L2	107	120..224

Note: this benefit is applicable to instruction “bne L1” as well, but not considered in the calculations shown here.

Over:

234

Reorder instructions

	mov r1, #1	1		
	mov r2, #100	2		
L1:	str r1, [r2, #0]	5	14 .. 86	
	add r2, r2, #4	6		
	add r1, r1, #1	7		
	cmp r1, #11	10		
	bne L1	11	20 .. 92	
	mov r3, #0	95		
	mov r2, #100	96		
L2:	sub r1, r1, #1	97	110.. 214	
	cmp r1, #0	100		
	beq Over	101		
	ldr r4, [r2, #0]	102		
	add r3, r3, r4	105		
	add r2, r2, #4	106		
	b L2	107	120..224	
Over:		234		

After
reordering

	mov r1, #1	1	
	mov r2, #100	2	
L1:	str r1, [r2, #0]	5	13 .. 77
	add r1, r1, #1	6	
	add r2, r2, #4	7	
	cmp r1, #11	9	
	bne L1	10	18 .. 82
	mov r3, #0	85	
	mov r2, #100	86	
L2:	sub r1, r1, #1	87	99 .. 195
	cmp r1, #0	90	
	beq Over	91	
	ldr r4, [r2, #0]	92	
	add r2, r2, #4	93	
	add r3, r3, r4	95	
	b L2	96	108..204
Over:		214	

Further reordering

	mov	r1, #1	1		
	mov	r2, #100	2		
L1:	str	r1, [r2, #0]	5	13 .. 77	
	add	r1, r1, #1	6		
	add	r2, r2, #4	7		
	cmp	r1, #11	9		
	bne	L1	10	18 .. 82	
	mov	r3, #0	85		
	mov	r2, #100	86		
L2:	sub	r1, r1, #1	87	99 .. 195	
	cmp	r1, #0	90		
	beq	Over	91		
	ldr	r4, [r2, #0]	92		
	add	r2, r2, #4	93		
	add	r3, r3, r4	95		
	b	L2	96	108..204	
Over:			214		

After
reordering

	mov	r1, #1	1	
	mov	r2, #100	2	
L1:	str	r1, [r2, #0]	5	13 .. 77
	add	r1, r1, #1	6	
	add	r2, r2, #4	7	
	cmp	r1, #11	9	
	bne	L1	10	18 .. 82
	sub	r1, r1, #1	85	
	mov	r3, #0	86	
	mov	r2, #100	87	
L2:	cmp	r1, #0	88	97 .. 169
	beq	Over	89	
	ldr	r4, [r2, #0]	90	
	add	r2, r2, #4	91	
	sub	r1, r1, #1	92	
	add	r3, r3, r4	93	
	b	L2	94	103..175
Over:			182	

Delayed branch

	mov	r1, #1	1	
	mov	r2, #100	2	
L1:	str	r1, [r2, #0]	5	13 .. 77
	add	r1, r1, #1	6	
	add	r2, r2, #4	7	
	cmp	r1, #11	9	
	bne	L1	10	18 .. 82
	sub	r1, r1, #1	85	
	mov	r3, #0	86	
	mov	r2, #100	87	
L2:	cmp	r1, #0	88	97 .. 169
	beq	Over	89	
	ldr	r4, [r2, #0]	90	
	add	r2, r2, #4	91	
	sub	r1, r1, #1	92	
	add	r3, r3, r4	93	
	b	L2	94	103..175
Over:			182	

Delayed branch

Instructions
in delay
slots

L1:	mov	r1, #1	1	
	mov	r2, #100	2	
	str	r1, [r2, #0]	5	
	add	r1, r1, #1	6	13 .. 69
	add	r2, r2, #4	7	
	cmp	r1, #11	9	
	bne	L1	10	
	str	r1, [r2, #0]	11	18 .. 74
	sub	r1, r1, #1	75	
	mov	r3, #0	76	
	mov	r2, #100	77	
	cmp	r1, #0	78	
L2:	beq	Over	79	87 .. 151
	ldr	r4, [r2, #0]	80	
	add	r2, r2, #4	81	
	sub	r1, r1, #1	82	
	add	r3, r3, r4	83	
	b	L2	84	
	cmp	r1, #0	85	93 .. 157
			162	

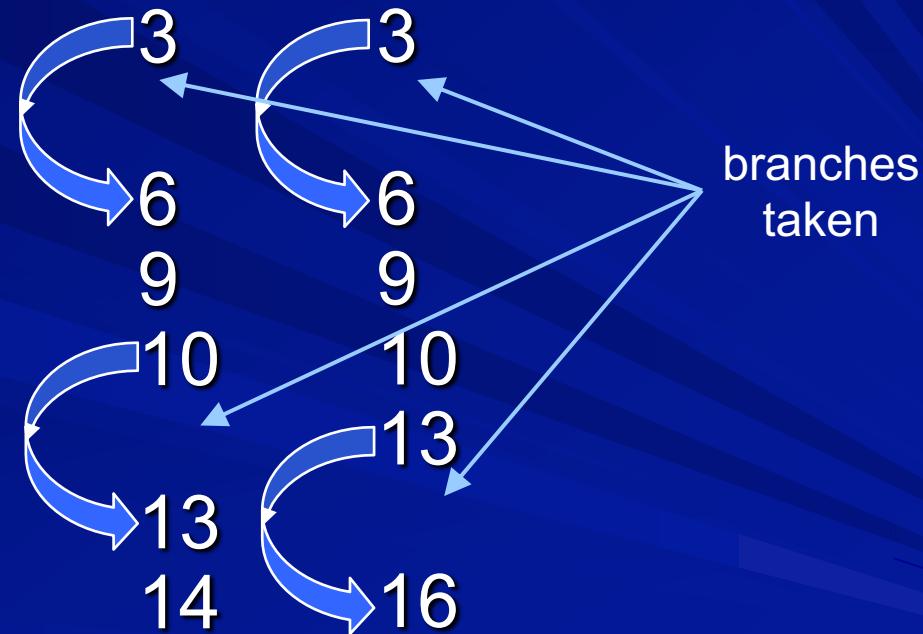
Summary

■ Original code without stalls	128
■ Original code with stalls	254
■ Execute inline instr after branch	234
■ After first re-ordering	214
■ After second reordering	182
■ Using delayed branch	162

Branch elimination example

(find max element in array)

func:	str	lr, [sp, #-4]!	1	1
	cmp	r2, #0	2	2
	bne	L1		
	b	Ret		
L1:	ldr	r3, [r1]	3	3
	cmp	r0, r3	6	6
	blo	L2	9	9
	b	L3		
L2:	mov	r0, r3	10	10
L3:	add	r1, r1, #4	13	13
	sub	r2, r2, #1	14	16
	bl	func	15	17
Ret:	ldr	pc, [sp], #4	16	18



Branch elimination example

func: str lr, [sp, #-4]!

cmp r2, #0

bne L1

b Ret



beq Ret

L1: ldr r3, [r1]

cmp r0, r3

blo L2

b L3

L2: mov r0, r3



L2: movlo r0, r3

L3: add r1, r1, #4

sub r2, r2, #1

bl func

Ret: ldr pc, [sp], #4

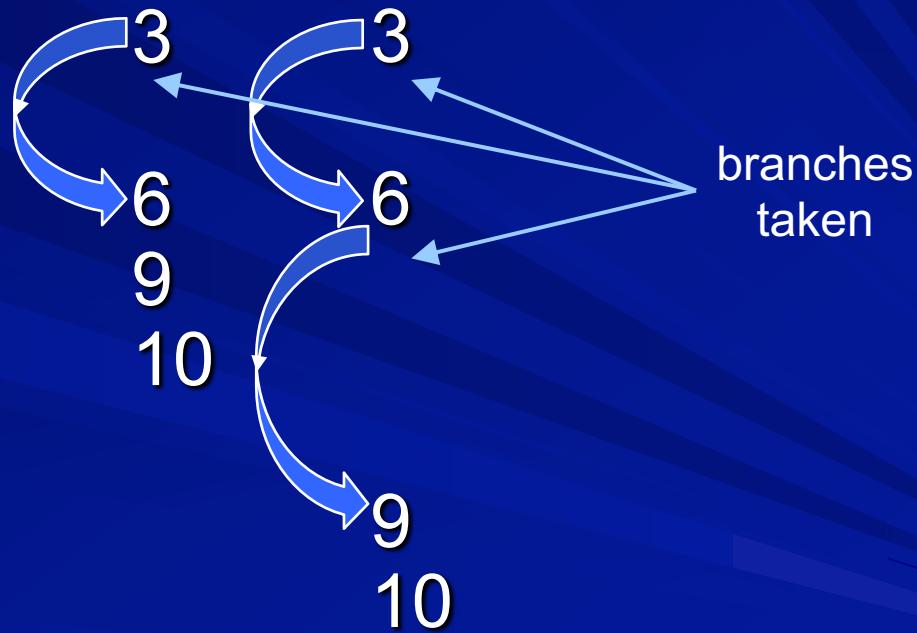
Reduced branches

func:	str	lr, [sp, #-4]!	1
	cmp	r2, #0	2
	beq	Ret	3
L1:	ldr	r3, [r1]	6
	cmp	r0, r3	9
L2:	movlo	r0, r3	10
L3:	add	r1, r1, #4	11
	sub	r2, r2, #1	12
	bl	func	13
Ret:	ldr	pc, [sp], #4	

Another example

(GCD)

```
func: str  lr, [sp, #-4]!  1   1
      cmp r0, r1    2   2
      bne L1
      b    Ret
L1:  bhs L2
      sub r1, r1, r0  9
      bl  func
      b    Ret
L2:  sub r0, r0, r1  10
      bl  func
Ret: ldr pc, [sp], #4
```



Another example

```
func: str lr, [sp, #-4]!
```

```
    cmp r0, r1
```

```
    bne L1
```

```
    b Ret
```

```
L1: bhs L2
```

```
    sub r1, r1, r0
```

```
    bl func
```

```
    b Ret
```

```
L2: sub r0, r0, r1
```

```
    bl func
```

```
Ret: ldr pc, [sp], #4
```



```
beq Ret
```



```
L2: sublo r1, r1, r0  
     subhs r0, r0, r1  
     bl func
```

Reduced branches

func:	str	lr, [sp, #-4]!	1
	cmp	r0, r1	2
	beq	Ret	3
L2:	sublo	r1, r1, r0	6
	subhs	r0, r0, r1	9
	bl	func	10
Ret:	ldr	pc, [sp], #4	

data
dependence

Moving “sublo r1,r1,ro” up

func:	str	lr, [sp, #-4]!	1
	cmp	r0, r1	2
	sublo	r1, r1, r0	3
	beq	Ret	4
L2:	subhs	r0, r0, r1	7
	bl	func	8
Ret:	ldr	pc, [sp], #4	

Put “sublo r1,r1,ro” in delay slot

func:	str	lr, [sp, #-4]!	1
	cmp	r0, r1	2
	beq	Ret	3
	sublo	r1, r1, r0	4
L2:	subhs	r0, r0, r1	7
	bl	func	8
Ret:	ldr	pc, [sp], #4	

THANKS

COL216

Computer Architecture

Performance

3rd March, 2022

Comparing Computers

- Functionality
- Performance/speed
- Power/energy consumption
- Cost
- Reliability
- Size/weight

Different Perspectives

- Individual user's perspective
- Service provider's perspective
- CPU designer's perspective
- Embedded system designer's perspective

Multiple parameters may be important

What are implications of architectural choices on these parameters?

Questions related to performance

- Can one system be better for one user whereas other system be better for another user ?
- Is the system performance only dependent on hardware? What about software?
- How does the machine's instruction set affect performance?

Defining Performance

- How do you define performance?
- Is there a unique way of defining it?

Two notions of “performance”

- Time (response time, execution time, latency)
 - How long does it take for my job to run?
 - How long must I wait for my query?
- Throughput (tasks per unit time)
 - How many jobs can the machine run at once?
 - What is the average execution rate?

Latency vs. Throughput

- Latency and throughput may often go together, but some time they may be in opposition - why?
- When is throughput more important than execution time?
- When is execution time more important than throughput?

Understanding Performance

- How are the following likely to effect response time and throughput?
 - Upgrade a machine with a new processor with faster clock.
 - Increasing the number of jobs (e.g., having a single computer service multiple users).
 - Adding a new machine in the lab.
 - Increasing the number of processors (e.g., in a network of ATM machines).

Performance Definitions

- Performance is in units of things-per-second
 - bigger is better
- If we are primarily concerned with response time

$$\text{performance} = \frac{1}{\text{execution_time}}$$

Relative Performance

" X is n times faster than Y" means

$$\frac{\text{performance}(X)}{\text{performance}(Y)} = \frac{\text{execution_time}(Y)}{\text{execution_time}(X)}$$

$$n = \frac{\text{performance}(X)}{\text{performance}(Y)} = \frac{\text{execution_time}(Y)}{\text{execution_time}(X)}$$

Comparing Performance

- If an Intel processor runs a program in 8 seconds and an ARM processor runs the same program in 10 seconds, how many times faster is the Intel processor?
- $n = 10 / 8 = 1.25$ times faster (or 25% faster)
- Why might someone choose to buy the ARM processor in this case?

Definitions of Time

- Defined in different ways:
 - **Response time** : total time = CPU exec time + disk access time + waiting time.
 - CPU execution time : total time spent by CPU
 - User CPU time : time spent in user program
 - System CPU execution time : time spent by OS.
- For example, a program may have a **system CPU time** of **22 sec.**, a **user CPU time** of **90 sec.**, a **CPU execution time** of **112 sec.**, and a **response time** of **162 sec..**

Computing CPU time

- The time to execute a given program can be computed as
 $\text{CPU time} = \text{CPU clock cycles} \times \text{clock cycle time}$ or
 $\text{CPU time} = \text{CPU clock cycles} / \text{clock rate}$
- $\text{CPU clock cycles} = (\text{instr/program}) \times (\text{clock cycles/instruction})$
= Instruction count x CPI

which gives

$$\text{CPU time} = \text{Instruction count} \times \text{CPI} \times \text{clock cycle time}$$

$$\text{CPU time} = \text{Instruction count} \times \text{CPI} / \text{clock rate}$$

- The units for this are

$$\frac{\text{seconds}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{clock cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{clock cycle}}$$

Example of computing CPU time

- If clock rate = 500 MHz, find execute time for a program that executes 1,000 instructions, if the CPI for the program = 3.5?

CPU time = instruction count x CPI / clock rate

$$\text{CPU time} = 1000 \times 3.5 / (500 \times 10^6) \text{ sec} = 7 \mu\text{s}$$

- If clock rate increases from 500 MHz to 600 MHz and the other factors remain the same, how many times faster will the computer be?

$$\frac{\text{CPU time old}}{\text{CPU time new}} = \frac{\text{clock rate new}}{\text{clock rate old}} = \frac{600 \text{ MHz}}{500 \text{ MHz}} = 1.2$$

Computing CPI

- CPI = avg. no. of cycles per instruction.

$$CPI = \sum_{i=1}^n CPI_i \times F_i$$

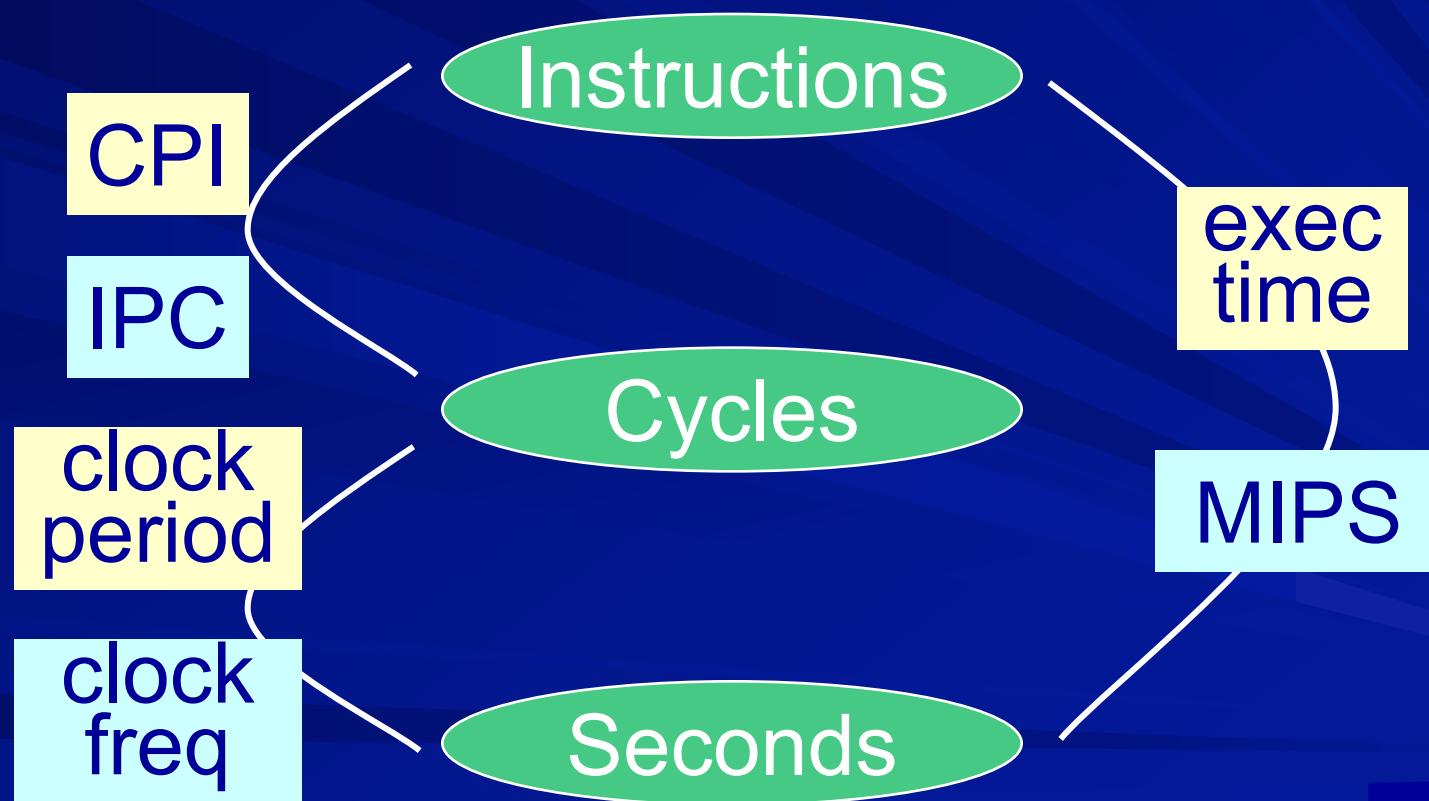
Example:

Op	F _i	CPI _i	CPI _i × F _i	% Time
DP	50%	4	2.0	50%
Load	20%	5	1.0	25%
Store	10%	4	0.4	10%
Branch	20%	3	0.6	15%
Total	100%		4.0	100%

Different numbers of cycles for different instructions

- Multiplication takes longer than addition
- Floating point operations take longer than integer ones
- Accessing memory takes more time than accessing registers
- Changing cycle time often changes number of cycles required for various instructions

Instructions↔Cycles↔Seconds



Performance

- Performance is determined by execution time
- Common pitfall: Use one of these variables as indicator of performance
 - # of cycles to execute program?
 - # of instructions in program?
 - # of cycles per second?
 - average # of cycles per instruction?
 - average # of instructions per second?

Example

- Computer A, (400 MHz clock), a program runs in 10 sec.
- Target: build a new machine B, the program runs in 6 sec.
- New (more expensive!) technology available to substantially increase the clock rate.
- This change will affect the rest of the CPU design, causing B to require 1.2 times as many clock cycles as A for the same program.
- What clock rate will make it possible for the designer to achieve the target?

Solution

- Two implementations (A and B) of the same instruction set architecture (ISA).
- If two machines have the same ISA which of our quantities (e.g., clock rate, CPI, execution time, # of instructions, MIPS) will always be identical?

Solution continued

A has clock cycle time = 2.5 ns

B has clock cycle time = T ns

■ For given program,

A has CPI = c , execution time = 10 s

B has CPI = 1.2 c , execution time = 6 s

No. of instructions executed = $N \cdot 10^9$ for both

$$10 = N * c * 2.5$$

$$6 = N * 1.2 * c * T$$

$$T^{-1} = 1.2 * 10 / (2.5 * 6) = .8 \text{ GHz} = 800 \text{ MHz}$$

Compiler factor

A compiler designer is to decide between two code sequences.

■ First code sequence:

- 2 of A, 1 of B, 2 of C

■ Second code sequence:

- 4 of A, 1 of B, 1 of C

3 classes of instructions:

- A : 1 cycle
- B : 2 cycles
- C : 3 cycles

■ Which sequence will be faster?

- How much?
- CPI for each?

Compiler and MIPS

- Two compilers being tested for 100 MHz machine with 3 classes of instructions:
 - A (1 cycle), B (2 cycles), and C (3 cycles)
 - Compiler 1: 5M A, 1M B, 1M C instructions
 - Compiler 2: 10M A, 1M B, 1M C instructions
- Which sequence has higher MIPS?
- Which sequence has lower execution time?

Programs to test performance

- Performance best determined by running a real application
 - Use programs typical of expected workload,
or
 - Typical of expected class of applications
e.g., compilers/editors, scientific applications,
graphics, etc.

Computer Benchmarks

- Benchmark = program (s) used to evaluate computer performance.
- Allow a user to make performance comparisons
- Benchmarks should
 - Be representative of the type of applications
 - Not be overly dependent on one or two features
- Benchmarks can vary greatly in terms of their complexity and their usefulness.

Sources of Benchmarks

- Synthetic benchmarks
 - small benchmarks can be easily written
 - nice for architects and designers
 - easy to use, but can be abused
- SPEC (Standard Performance Evaluation Corporation)
<http://www.spec.org>

"An ounce of honest data is worth a pound of marketing hype"

 - benchmarks derived from real programs

SPEC Benchmark Categories

- Cloud
- CPU
- Graphics/Workstations
- ACCEL/MPI/OMP
- Java Client/Server
- Mail Servers
- Storage
- Power
- Virtualization
- Web Servers

SPEC CPU Benchmarks

- SPEC CPU 92 => SPEC CPU 95 =>
SPEC CPU 2000 => SPEC CPU 2006 =>
SPEC CPU 2017
- SPEC CPU 2017: 43 benchmarks
organized into 4 suites
 - SPECspeed 2017 Integer
 - SPECspeed 2017 Floating Point
 - SPECCrate 2017 Integer
 - SPECCrate 2017 Floating Point

Integer benchmarks

- Perl interpreter, GNU C compiler
- Route planning
- Network simulation
- Video compression, Data compression
- Tree search in games like chess
- Recursive solution generator (Sudoku)

Floating point benchmarks

- Modeling of physical phenomena like explosion, molecular dynamics, ocean, atmosphere etc.
- Optical tomography
- Ray tracing, 3-d rendering, animation
- Computational electromagnetics
- Image manipulation

Amdahl's Law

$$\text{Speedup} = \frac{\text{ExTime old}}{\text{ExTime new}} = \frac{\text{Performance new}}{\text{Performance old}}$$



Suppose that an enhancement accelerates a fraction
Fraction_{enhanced} of the task by a factor Speedup_{enhanced}

Execution Time After Improvement =
time unaffected + time affected / improvement

Amdahl's Law

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

Example:

A program runs in 100 seconds on a machine, with multiplication responsible for 80%. How much do we have to improve the speed of multiplication if we want the program to run 4 times faster?

How about making it 5 times faster?

Example

- Suppose we enhance a machine making all floating-point instructions run five times faster.
- Let the execution time of some benchmark before the floating-point enhancement be 10 seconds.
- What will the speedup be if half of the 10 seconds is spent executing floating-point instructions?

$$\text{Speedup}_{\text{enhanced}} = 5$$

$$\text{ExTime}_{\text{old}} = 10$$

$$\text{Fraction}_{\text{enhanced}} = 0.5$$

$$\text{ExTime}_{\text{new}} =$$

$$10 (1 - .5 + .5/5) = 6$$

$$\text{Speedup} = 10/6 = 1.67$$

Example continued

- We are looking for a benchmark to show off the new FP unit.
- Want the overall benchmark to show a speedup of 3.
- Benchmark under consideration runs for 100 seconds with the old floating-point hardware.
- How much of the execution time would FP instructions have to account for in this program in order to yield our desired speedup on this benchmark?

$$\text{Speedup}_{\text{enhanced}} = 5$$

$$\text{Speedup} = 3$$

$$\text{ExTime}_{\text{old}} = 100$$

$$\text{ExTime}_{\text{new}} = 33.3$$

$$\text{Fraction}_{\text{enhanced}} = f$$

$$33.3 = 100(1-f + f/5)$$

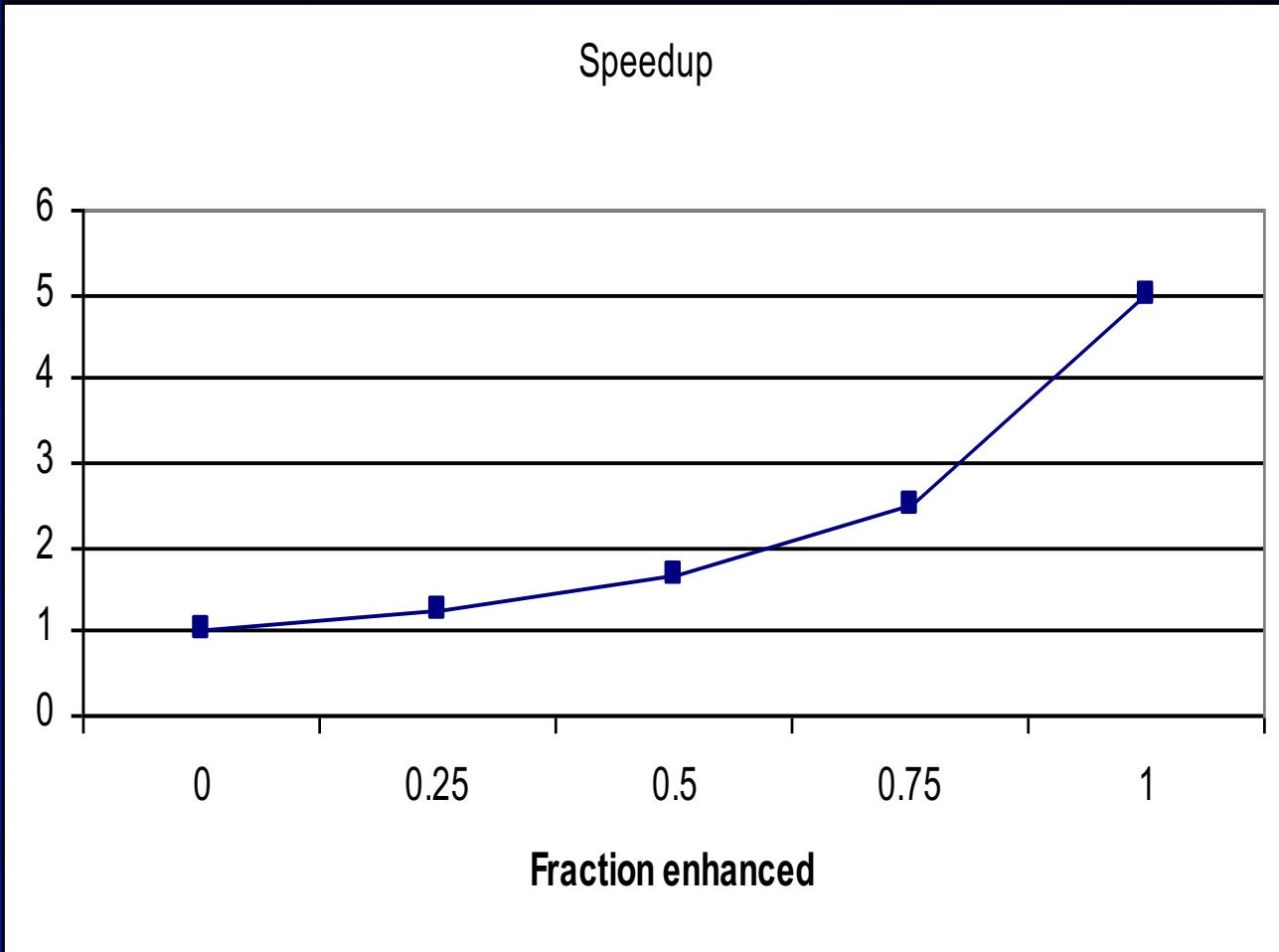
$$1 = 3 (1 - 4 f / 5)$$

$$f = 5 / 6$$

$$100 * 5/6 = 83.3$$

Example continued

Frac _{enh}	Speedup
0.00	1.00
0.25	1.25
0.50	1.67
0.75	2.50
1.00	5.00



Example: FP h/w vs s/w

C Clock freq
1000 MHz

	Instr mix in prog P	Cycles in MFP	Int instr in MNFP
FP multiply	10%	6	30
FP add	15%	4	20
FP divide	5%	20	50
Int instructions	70%	2	
CPI			

$$10\% * 6 + 15\% * 4 + 5\% * 20 + 70\% * 2 \\ = 0.6 + 0.6 + 1.0 + 1.4 = 3.6$$

Example: FP h/w vs s/w

C Clock freq 1000 MHz	Instr mix in prog P	Cycles in MFP	Int instr in MNFP
FP multiply	10%	6	30
FP add	15%	4	20
FP divide	5%	20	50
Int instructions	70%	2	
CPI		3.6	2.0
MIPS = C/CPI		277.8	500.0
N		300M	

$$\begin{aligned}
 & 300M(10\% * 30 + 15\% * 20 + 5\% * 50 + 70\% * 1) \\
 & = 300M(3 + 3 + 2.5 + 0.7) = 300M * 9.2 = 2760M
 \end{aligned}$$

Example: FP h/w vs s/w

C Clock freq 1000 MHz	Instr mix in prog	Cycles in P	Int instr in MFP	Int instr in MNFP
FP multiply	10%	6	30	
FP add	15%	4	20	
FP divide	5%	20	50	
Int instructions	70%	2		
CPI		3.6	2.0	
MIPS = C/CPI		277.8	500.0	
N		300M	2760M	
ExTime = N*CPI/C		1.08	5.52	
MFLOPS = 30%*300M/ExTime		90/1.08	90/5.52	

Thanks

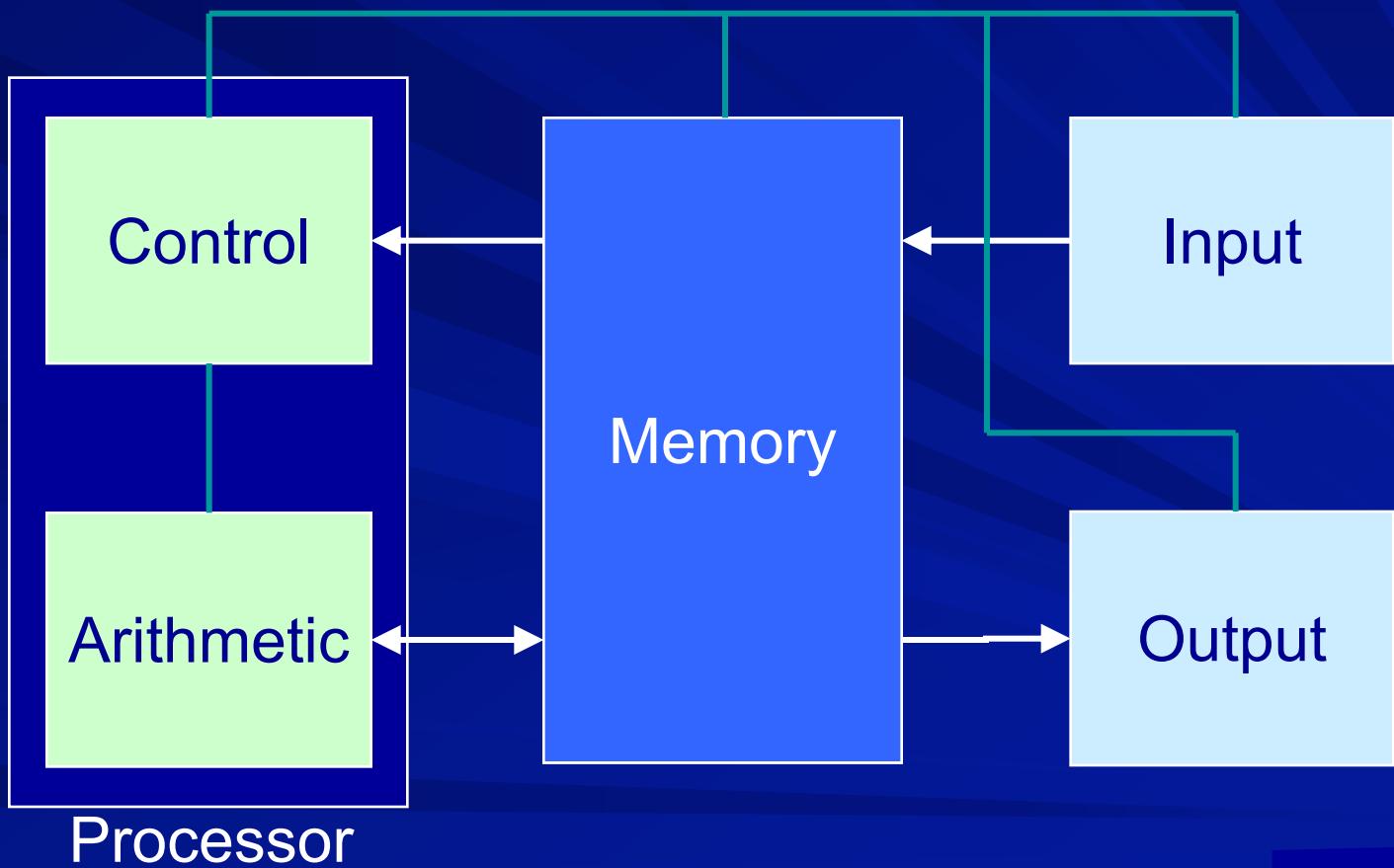
COL216

Computer Architecture

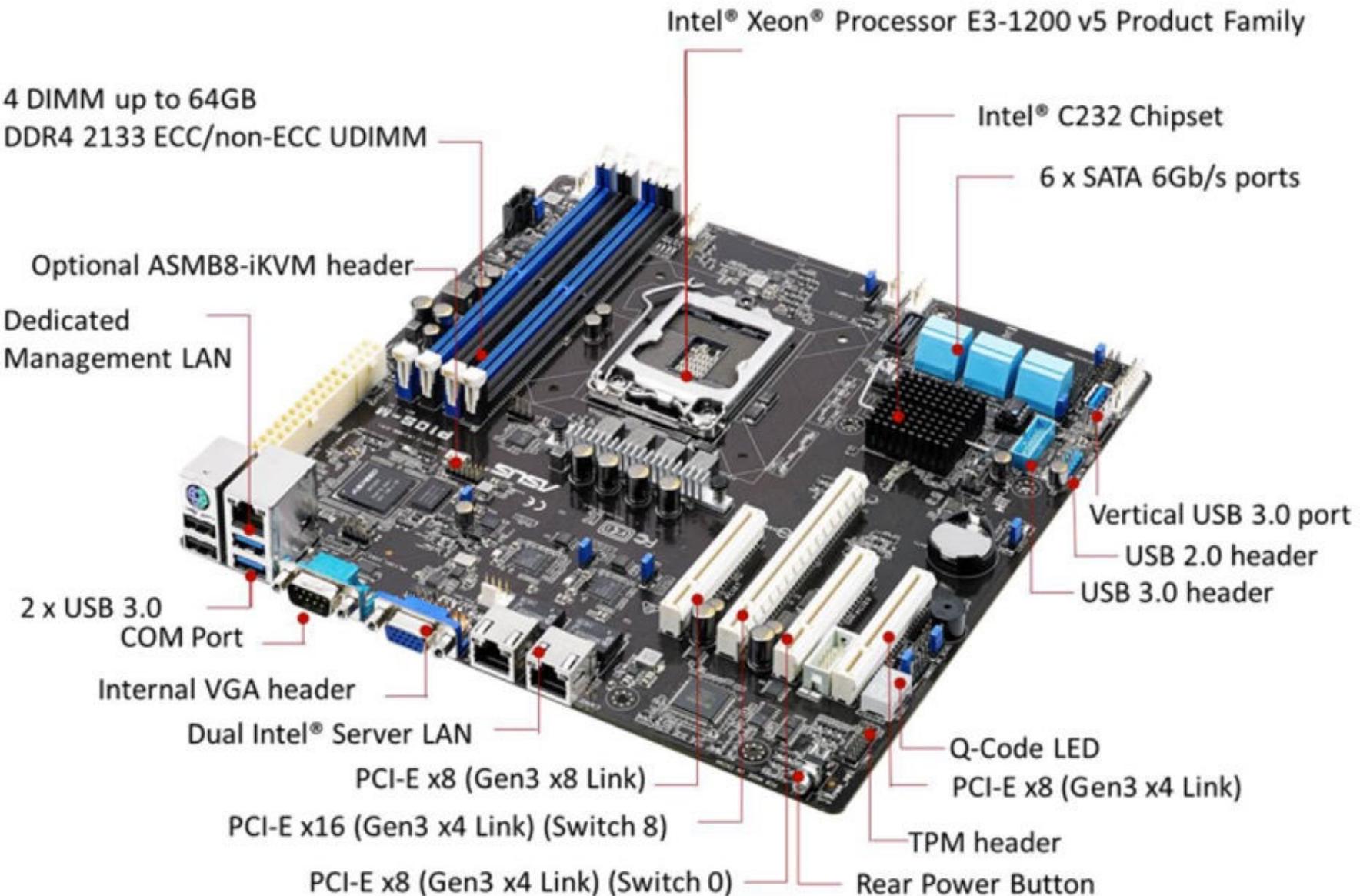
Memory Organization:
Hierarchy

7th March 2022

Computer System: Typical Block Diagram



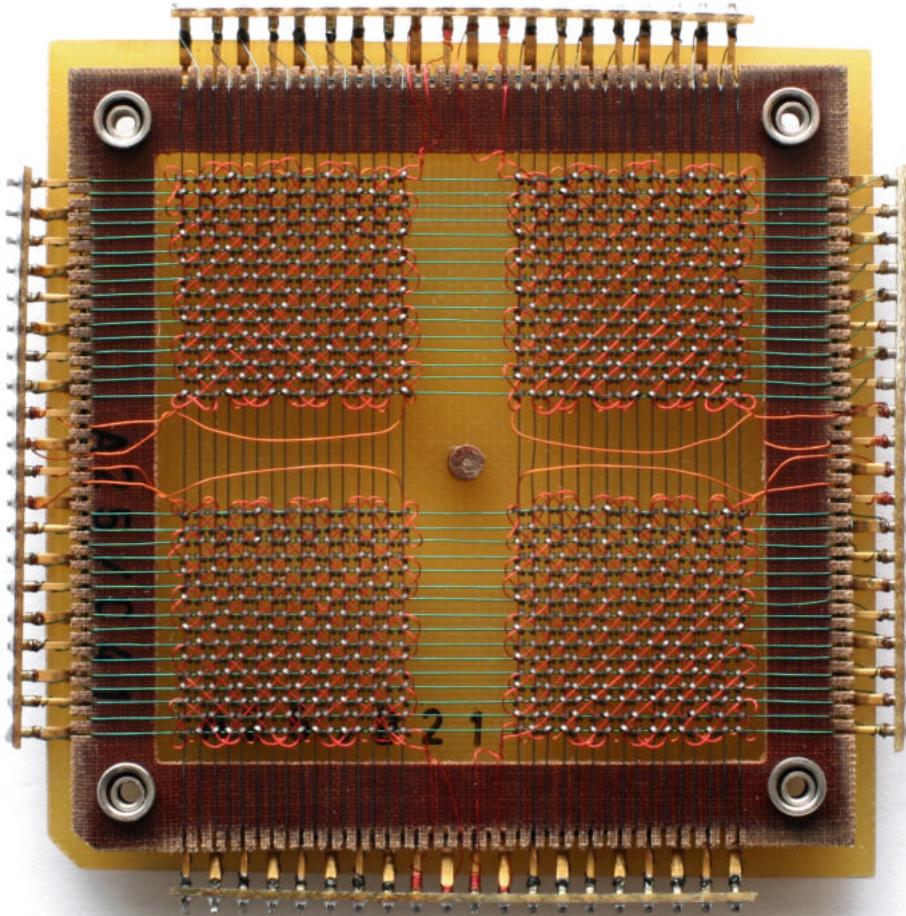
A server board



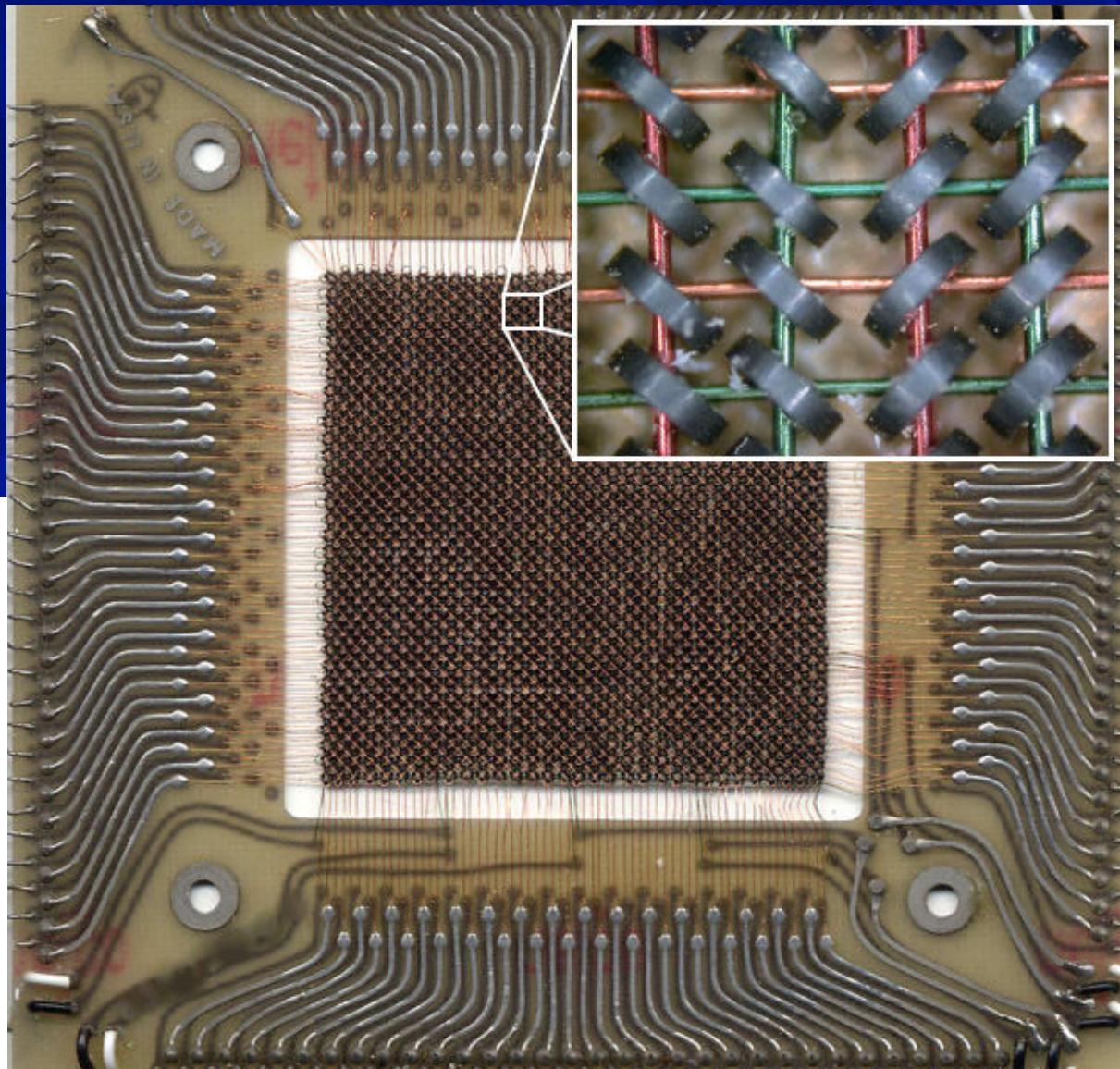
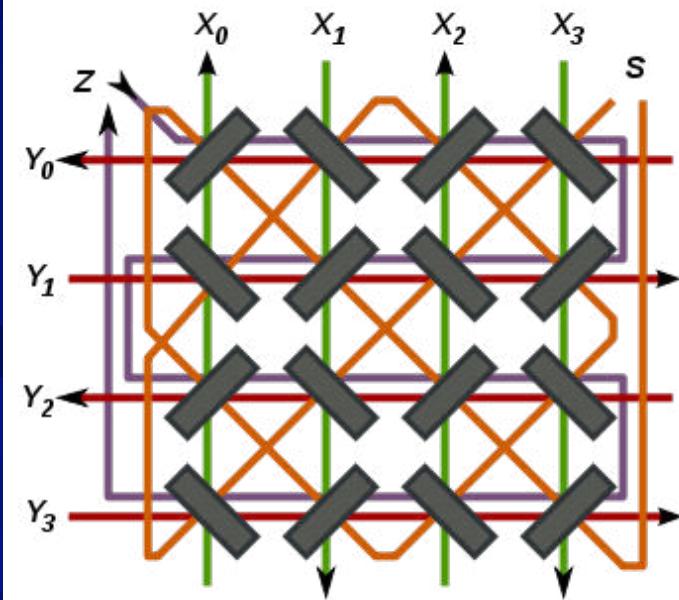
Memory Modules



Ferrite core memory



Ferrite core memory



Memory Technologies

- Semiconductor

- Registers
- SRAM
- DRAM
- FLASH

- Magnetic

- FDD
- HDD

- Optical

- CD
- DVD
- Blu Ray



Random Access

Random + Sequential

Speed vs size|cost

Speed

Fastest

Slowest

Size

Smallest

Cost / bit

Highest

Lowest

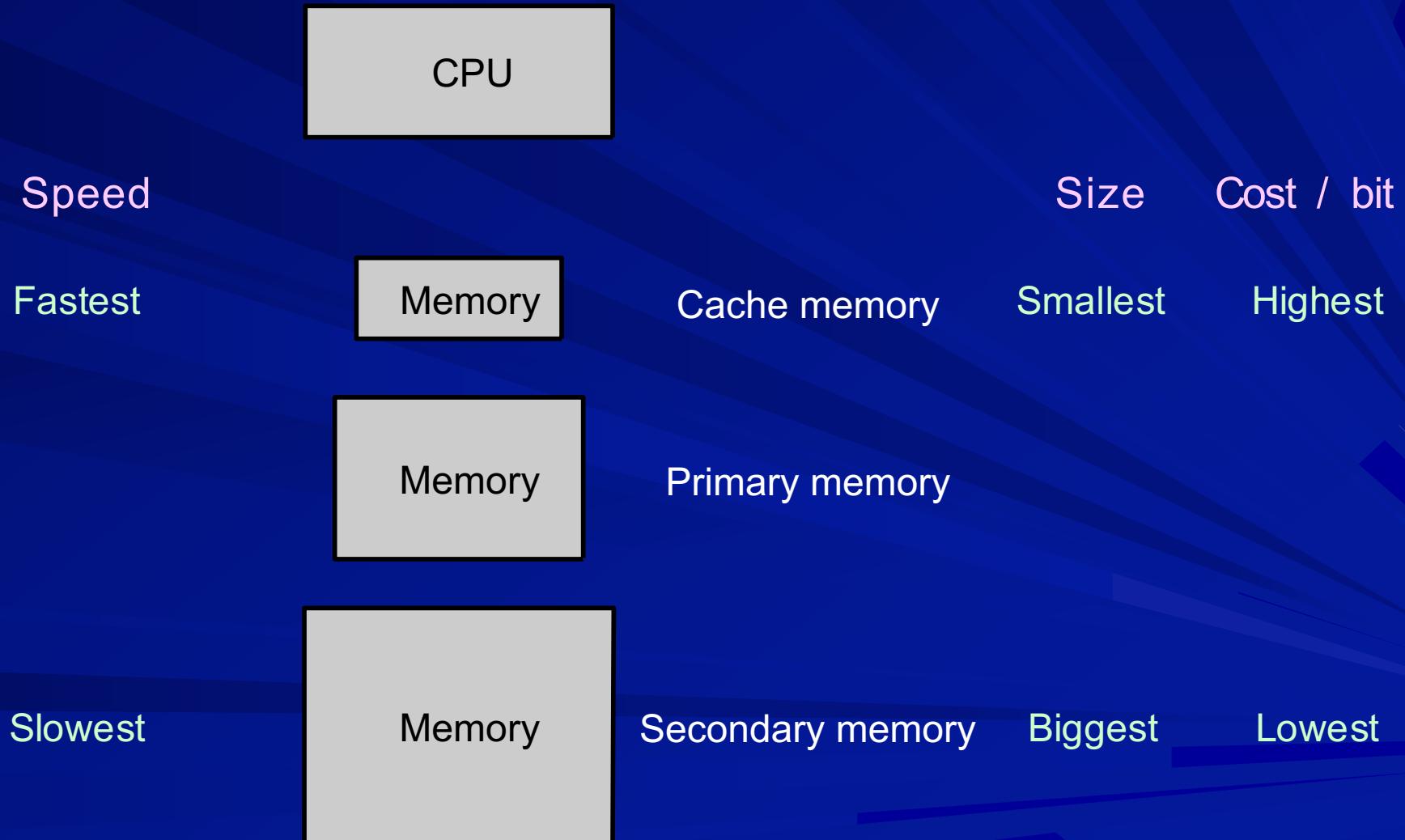
Memory

Memory

Memory



Hierarchical Organization



Primary Memory

- Also called Main memory
- Made using DRAM
- Volatile
- Separate from processor chip (off-chip)

Cache Memory

- 1 to 3 levels
- Made using SRAM
- Volatile
- On processor chip (on-chip)

Cache Memory

- 1 to 3 levels
 - L1 : split L2 : unified L3 : unified
 - L1 : private L2 : priv/sh L3 : shared
- Made using SRAM
- Volatile
- On processor chip (on-chip)

Cache example

Intel Xeon E7 8870

- Level 1 cache

- 64 KB (split I + D)

- Level 2 cache

- 256 KB

- Level 3 cache

- 30 MB (shared by 10 cores)

- External address space

- 16 TB

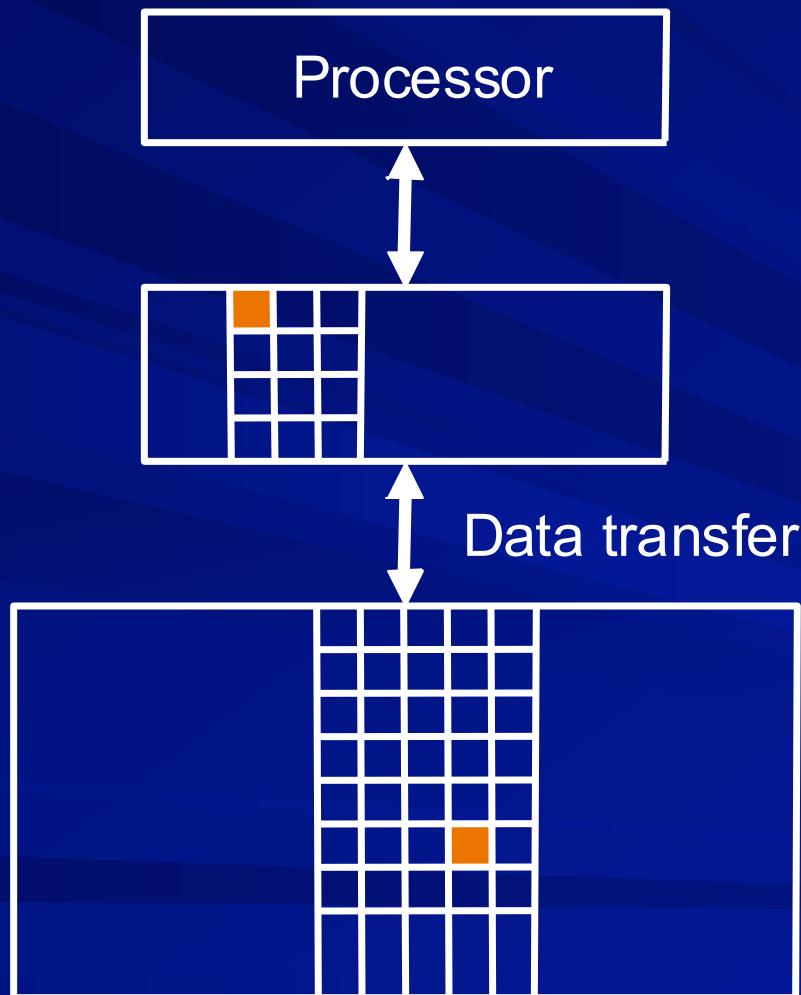
Secondary Memory

- Flash or Disc drive
- Not random access
- Non-volatile
- Like a peripheral device

Back-up Memory

- Used for archival
- Optical Discs, HDD, Cloud
- External, Removable

How CPU accesses hierarchy?



access

hit

miss

unit of transfer = block

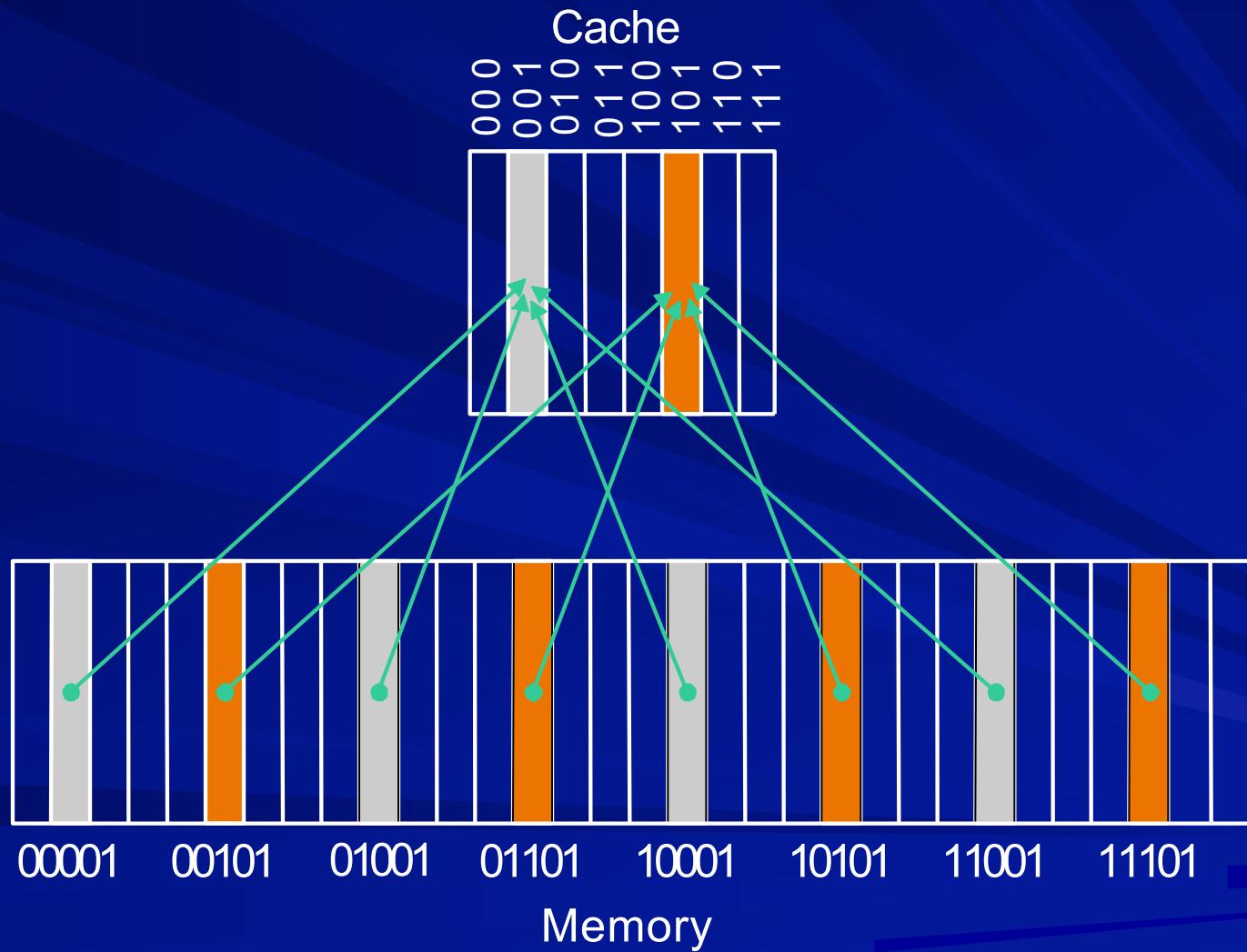
Why does the hierarchy work?

- Temporal Locality
- Spatial Locality

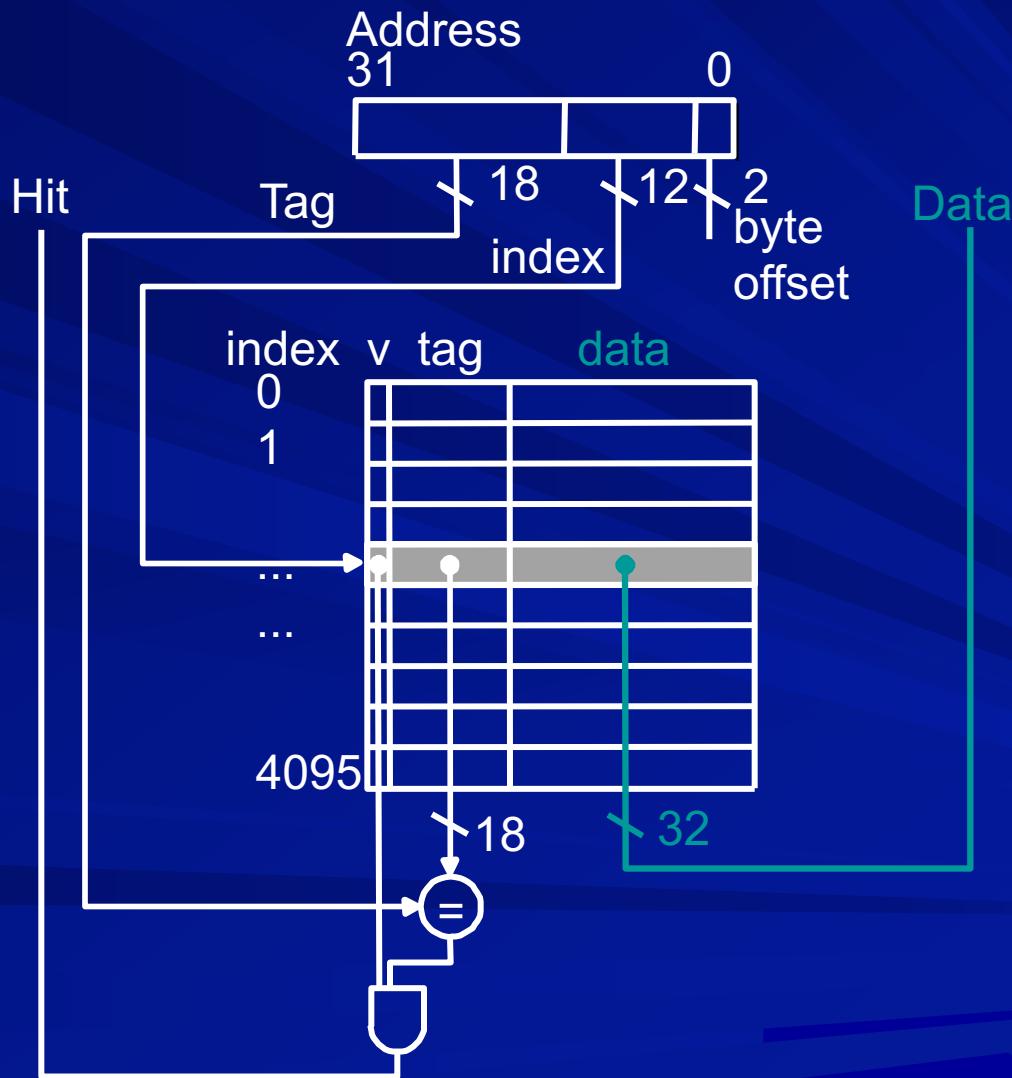
Why does the hierarchy work?

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

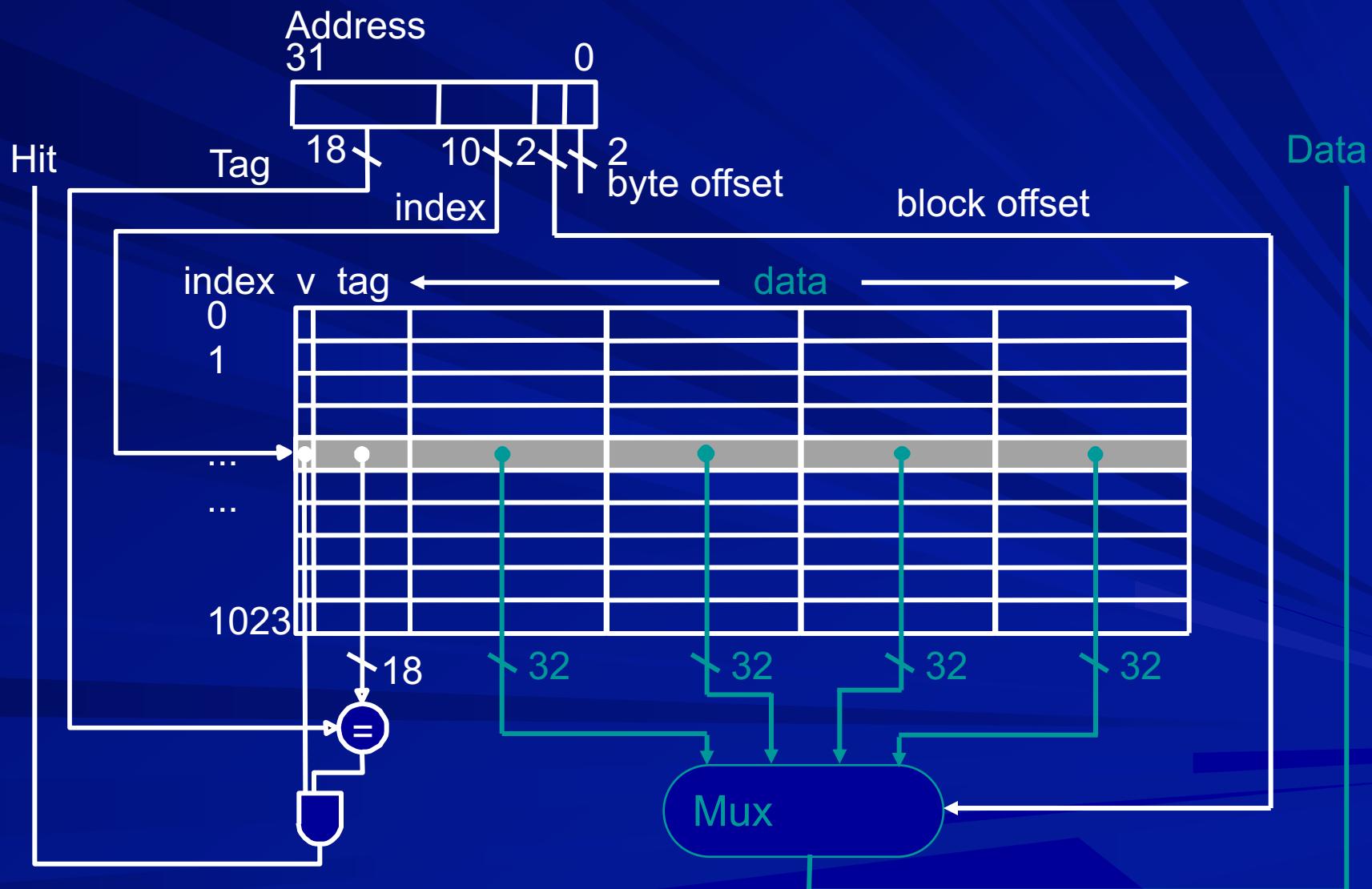
Direct mapped cache



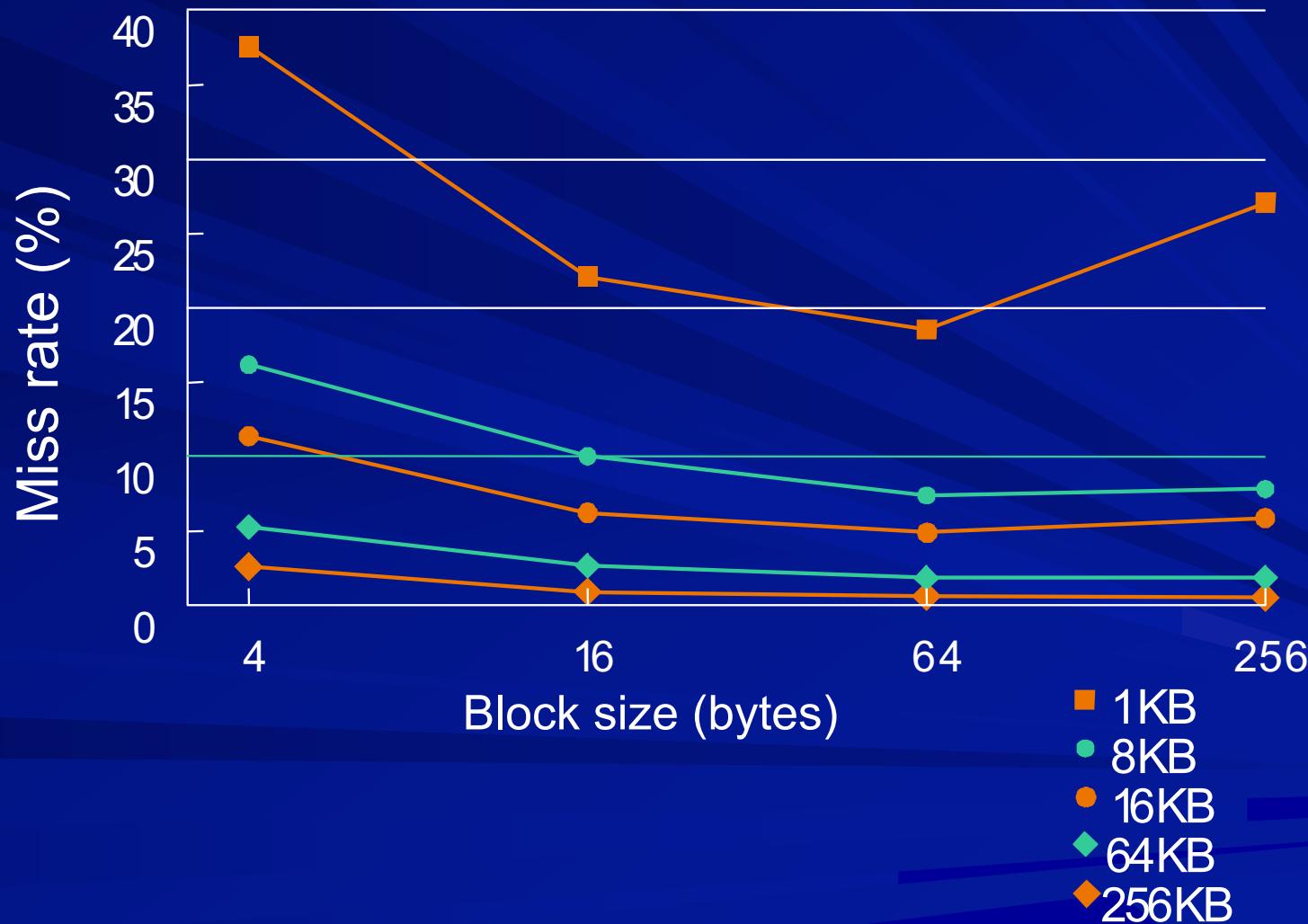
Cache access mechanism



Cache with 4 word blocks

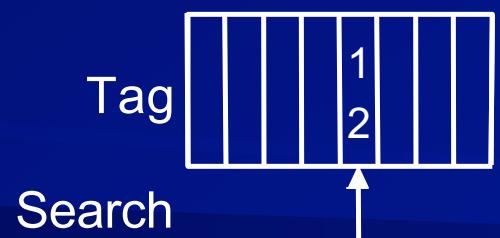


Miss rate vs block size

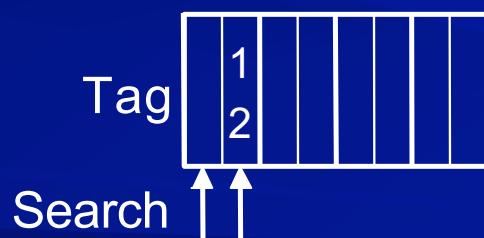


More flexible block placement

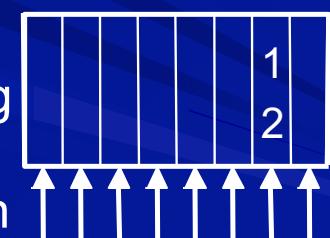
Direct mapped



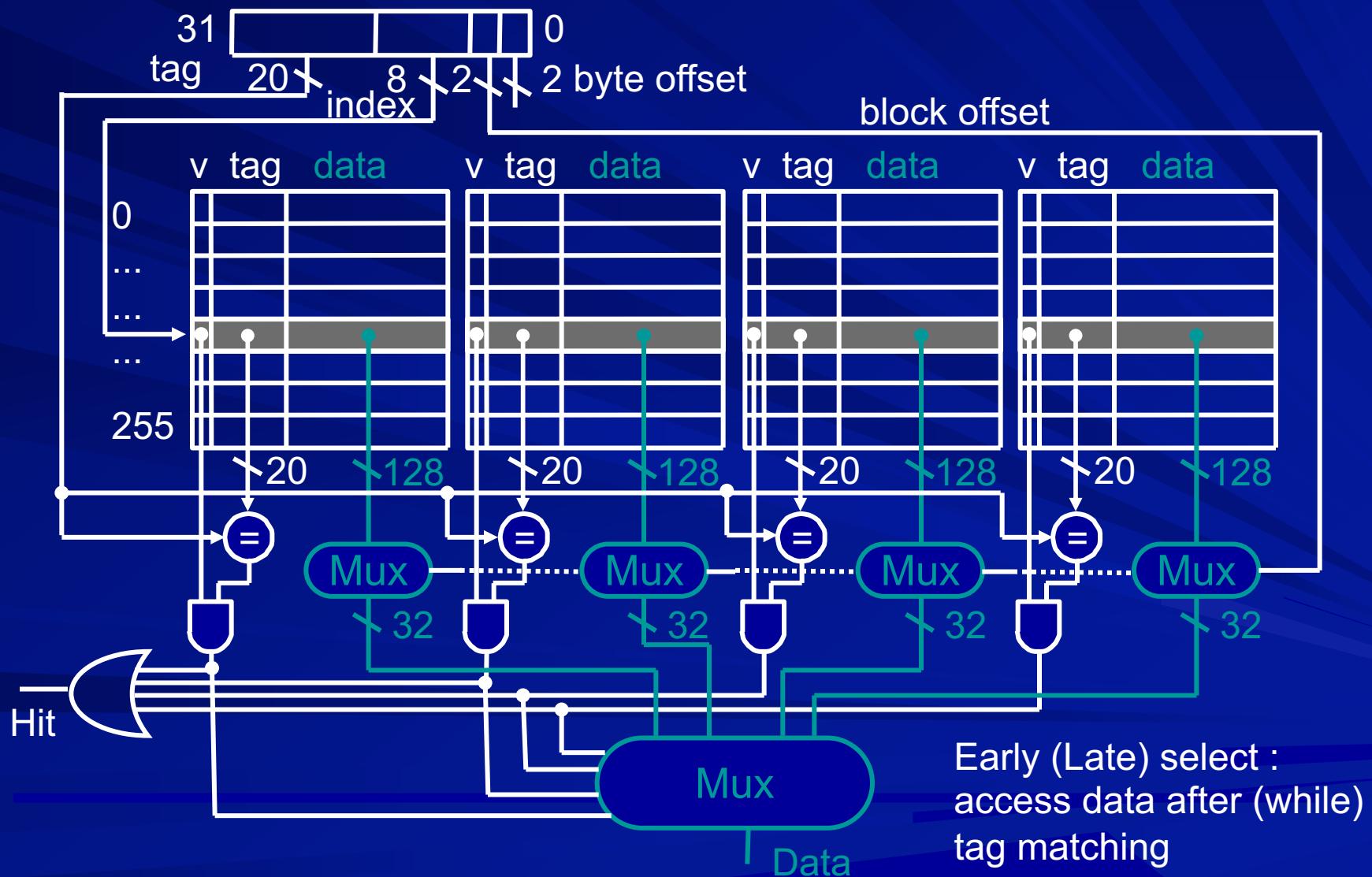
Set associative



Fully associative



4-way set associative cache



Sizes and bits (Direct mapped cache)

Memory = $M = 2^m$ bytes

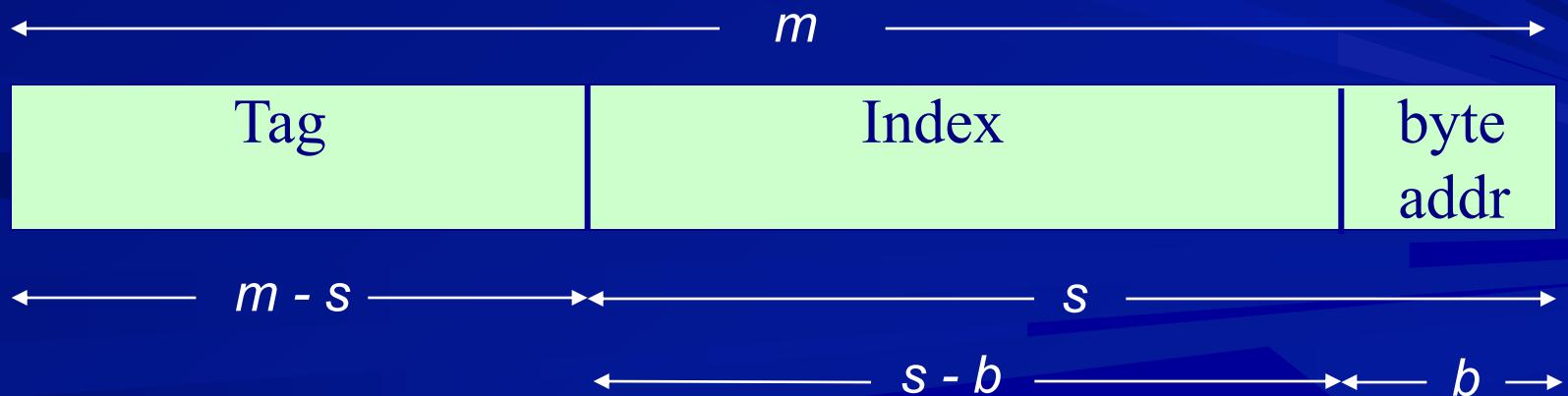
Cache = $S = 2^s$ bytes

Block = $B = 2^b$ bytes No. of cache blocks = $\frac{S}{B} = 2^{s-b}$

No. of possible tag values = No. of mem blocks ^{B}

that map to same cache block = $\frac{M}{S} = 2^{m-s}$

No. of tag bits per block = $m - s$



Sizes and bits (Set assoc. cache)

Degree of S.A. = $A = 2^a$

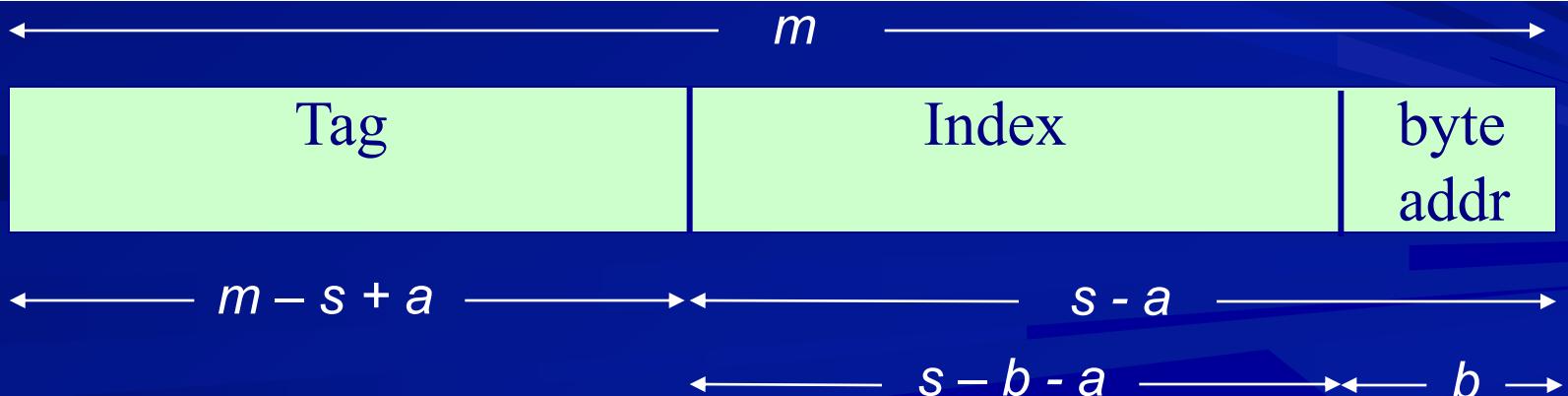
No. of sets = $\frac{S}{A \cdot B}$

Index bits = $\log_2\left(\frac{S}{A \cdot B}\right) = s - a - b$

No. of possible tag values = No. of mem blocks

that map to same cache set = $\frac{M}{A} = 2^{m-s+a}$

Tag size = $m - s + a$ Total tag bits = $(m - s + a) \times \frac{S}{B}$



Cache Policies

- Placement what gets placed where?
- Read when? from where?
- Load order of bytes/words?
- Replacement which one?
- Fetch when to fetch new block?
- Write when? to where?

When to initiate memory access?

- Sequential

- initiate memory access only after detecting a miss

- Concurrent

- initiate memory access along with cache access in anticipation of a miss

Where CPU gets data from?

- Without data forwarding
 - give data to CPU after filling the missing block in cache
- With forwarding
 - forward data to CPU as it gets filled in cache

Order of bytes/words in a block

- Block in cache may be loaded starting from the missing word, in a wrap around manner
 - useful when data is forwarded to CPU as it gets filled in cache

Which block to replace?

- Least Recently Used (LRU)
- Least Frequently Used (LFU)
- First In First Out (FIFO)
- Random

Cache Policies

- Placement direct / associative / set assoc
- Read sequential / concurrent,
 with/without forwarding
- Load with/without wrap around
- Replacement LRU / LFU / FIFO / Random
- Fetch ??
- Write ??

Fetch Policies

- Demand fetching
 - fetch on a miss
- Prefetch
 - fetch in anticipation
- Prefetch approaches
 - hardware driven prefetch
 - software driven prefetch

Write Hit

■ Write in cache ?

- Obviously yes

■ Write in main memory ?

- May be

- Writing in memory has an overhead
- Writing will keep cache and memory consistent
- Can we handle inconsistent data?

Write Miss

■ Write in cache ?

- May be
 - Where to write?

■ Write in main memory ?

- Definitely yes, if answer above is no
 - Writing in memory has an overhead
 - Writing will keep cache and memory consistent
 - Can we handle inconsistent data?

Thanks

COL216

Computer Architecture

Memory Organization:
Cache performance
10th March 2022

Cache Policies

- Placement direct / associative / set assoc
- Read sequential / concurrent,
 with/without forwarding
- Load with/without wrap around
- Replacement LRU / LFU / FIFO / Random
- Fetch Demand fetch / Pre-fetch
- Write ??

Write Hit

■ Write in cache ?

- Obviously yes

■ Write in main memory ?

- May be

- Writing in memory has an overhead
- Writing will keep cache and memory consistent
- Can we handle inconsistent data?

Write Miss

■ Write in cache ?

- May be

- Where to write?

■ Write in main memory ?

- Definitely yes, if answer above is no

- Writing in memory has an overhead

- Writing will keep cache and memory consistent

- Can we handle inconsistent data?

Write Policies

■ Write Back

- Do not write in memory immediately
- Write a block in memory when it is getting evicted from cache

■ Write Through

- Write word in memory immediately
- Two choices in case of a write miss
 - Write Allocate
 - No Write Allocate

Write Policies - Comparison

	WB	WTWA	WTNWA
wr hit	-	word write	word write
wr miss	block rd*	block rd word write	word write
rd miss	block rd*	block rd	block rd
Miss rate	lower	lower	higher

* Write back displaced block if “dirty bit” set

Write Policies : Timings

- Write time depends upon
 - number of blocks transferred
 - block transfer time
 - number of words transferred
 - word transfer time
- Timings as seen by the CPU vs timings as seen by BUS
 - CPU time can be minimized by using a write buffer

Write Buffers

- Processor writes into a buffer and proceeds without waiting
- Transfer from buffer to memory takes place in background
- Possible with WT as well as WB
- What happens if a read comes up for data still in buffer?

Cache Performance

Average memory access time =

$$\text{Hit time} + \text{Miss rate} * \text{Miss penalty}$$

Mem stalls / Instr =

$$\text{Miss rate} * \text{Miss Penalty} * \text{Mem accesses / Instr}$$

Performance Improvement

- Reducing miss penalty
- Reducing miss rate
- Reducing hit time

Performance Improvement

- Reducing miss penalty
- Reducing miss rate
- Reducing hit time

Miss rate depends on

- cache size, block size, associativity, policies

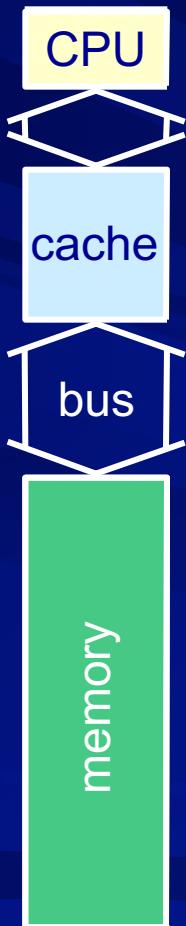
Miss penalty depends on

- Memory access time
- cache – memory interface

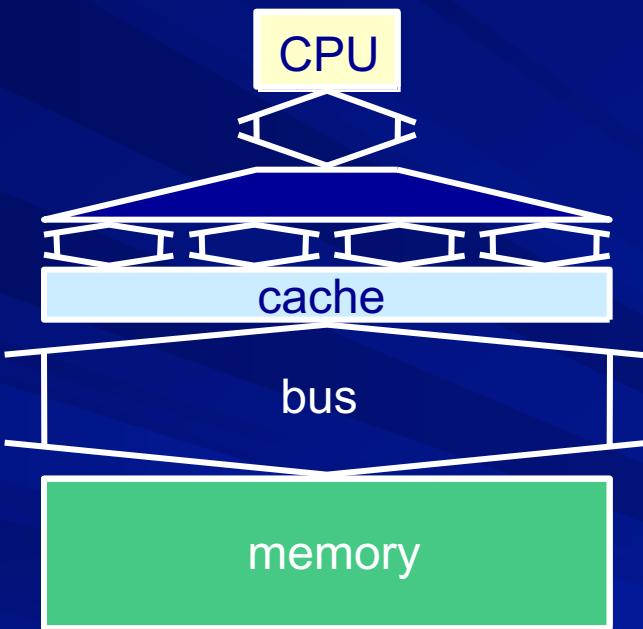
Types of misses

- compulsory miss
- capacity miss
- conflict miss

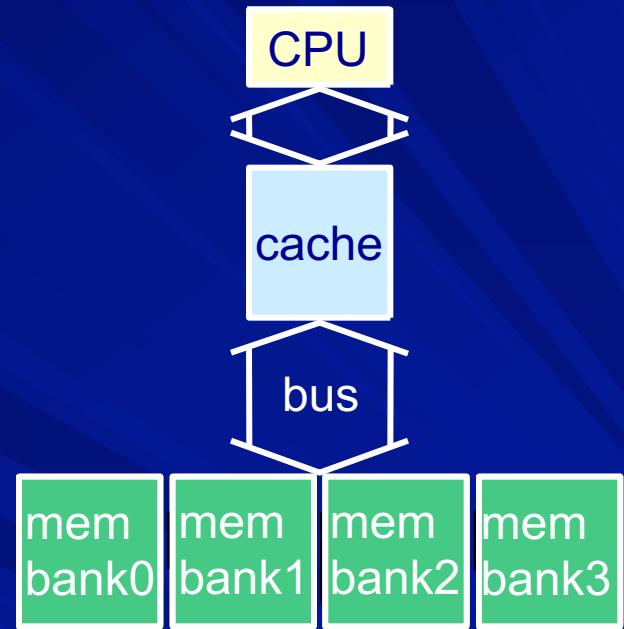
Transferring blocks to/from memory



a. one word wide
memory

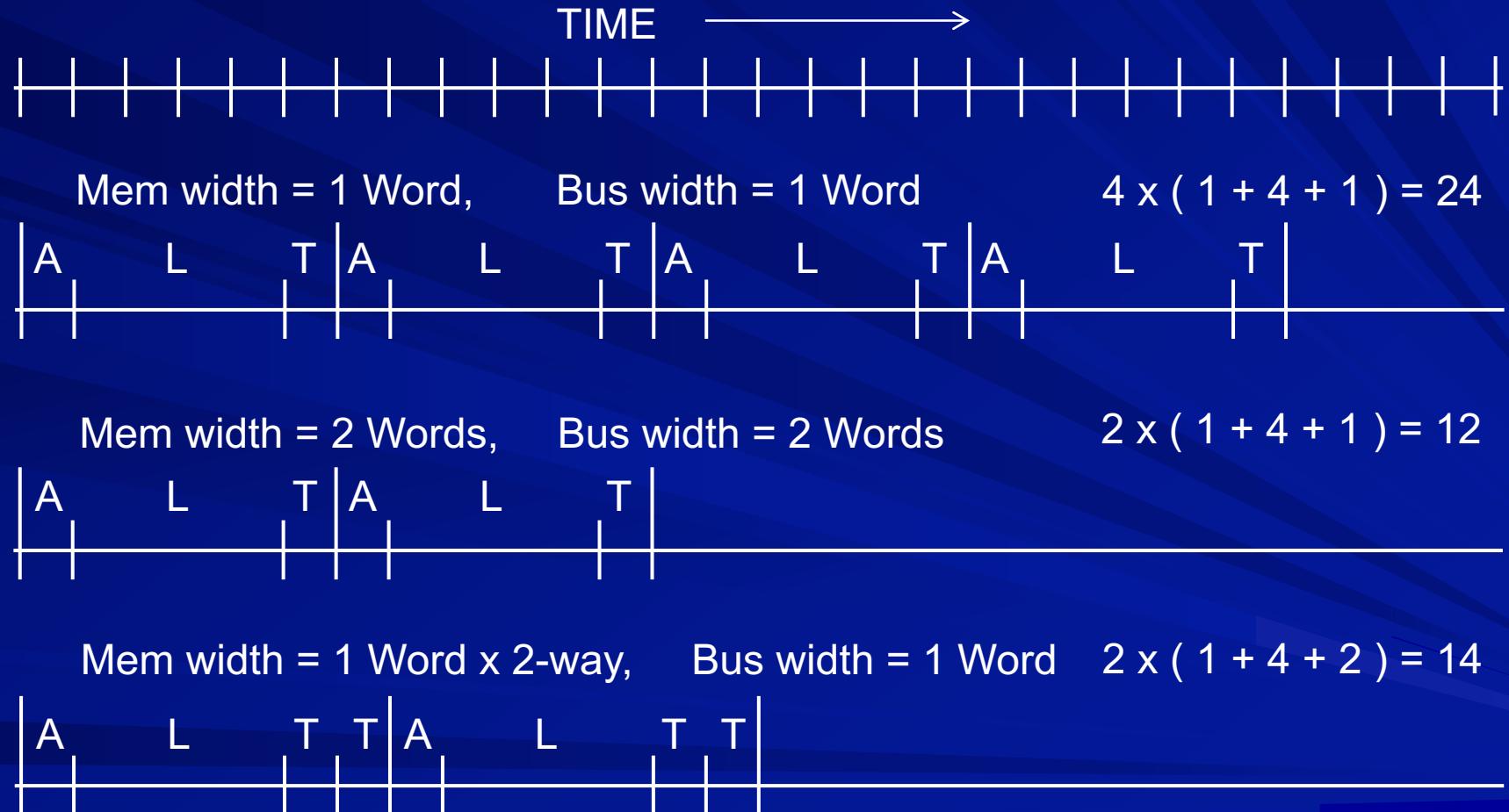


b. four word wide
memory



c. interleaved
memory

Miss penalty



Miss penalty example

- 1 clock cycle to send address
- 15 cycles for RAM access
- 1 cycle for sending data
- block size = 4 words

Compute miss penalty

Answer:

case (a): $4(1 + 15 + 1) = 68$ or $1 + 4(15 + 1) = 65$

case (b): $1 + (15 + 1) = 17$

case (c): $1 + 15 + 4 \times 1 = 20$

DRAM with page mode

- Memory cells are organized as a 2-D structure
- Entire row is accessed at a time internally and kept in a buffer
- Reading multiple bits from a row can be done very fast
 - sequentially, without giving address again
 - randomly, giving only the column addresses

Performance analysis example

$$CPI_{\text{eff}} = CPI + \text{Miss rate} * \text{Miss Penalty} * \frac{\text{Mem accesses}}{\text{Instr}}$$

CPI = 1.2 Miss rate = 0.5%

Block size = 16 w

Miss penalty??

Assume mem access / Instr = 1

Miss penalty calculation

Data / address transfer time = 1 cycle

Memory latency = 10 cycles

a) Miss penalty = $16 * (1 + 10 + 1) = 192$

b) Miss penalty = $4 * (1 + 10 + 1) = 48$

c) Miss penalty = $4 * (1 + 10 + 4 * 1) = 60$

Back to CPI calculation

$$CPI_{\text{eff}} = 1.2 + .005 * \text{miss penalty} * 1.0$$

a) $1.2 + .005 * 192 * 1.0 = 1.2 + .96$

$$= 2.16$$

b) $1.2 + .005 * 48 * 1.0 = 1.2 + .24$

$$= 1.44$$

c) $1.2 + .005 * 60 * 1.0 = 1.2 + .30$

$$= 1.50$$

What makes cache work?

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

Locality Example

Code 1

```
for ( i = 0; i < 8000; i++)  
    for ( j = 0; j < 8; j++)  
        A [ i ] [ j ] = B [ j ] [ 0 ] + A [ j ] [ i ];
```

Code 2

```
for ( j = 0; j < 8; j++)  
    for ( i = 0; i < 8000; i++)  
        A [ i ] [ j ] = B [ j ] [ 0 ] + A [ j ] [ i ];
```

Arrays are stored
row-wise in C,
column-wise in
Matlab

Identify spatial locality and temporal locality

Is cache managed by HW or SW?

Check for hit or miss

Hit: Read/write cache

Miss: Possible actions

1. write back a block in memory
2. read a block from memory
3. write a word in memory

Are these actions performed by
hardware or by software?

Hit time, miss rate, miss penalty

How do these parameters vary with

1. cache size?
2. degree of associativity?
3. block size?
4. memory access time?
5. degree of memory interleaving?

Indicate the type of miss that is affected in each case.

Cache comparison example

Cache	mapping	block size	I-miss	D-miss	CPI
1	direct	1 word	4%	8%	2.0
2	direct	4 word	2%	5%	??
3	2-way s.a.	4 word	2%	4%	??

Miss penalty = 6 + block size

50% instructions have a data reference

Solution:

Stall cycles: cache1: $7 * (.04 + .08 * .5) = .56$

cache2: $10 * (.02 + .05 * .5) = .45$

cache3: $10 * (.02 + .04 * .5) = .40$

Solution continued

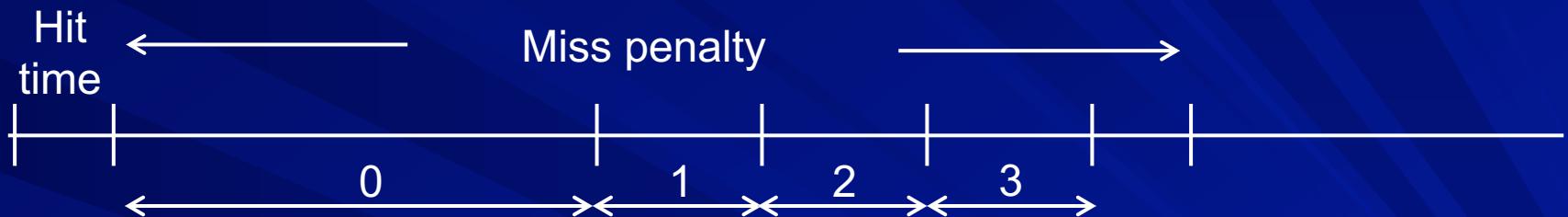
Cache	CPI	clock period	time/instr
1	2.0	2.0	4.0
2	$2.0 - .56 + .45 = 1.89$	2.0	3.78
3	$2.0 - .56 + .40 = 1.84$	2.4	4.416

Hit time, miss rate, miss penalty

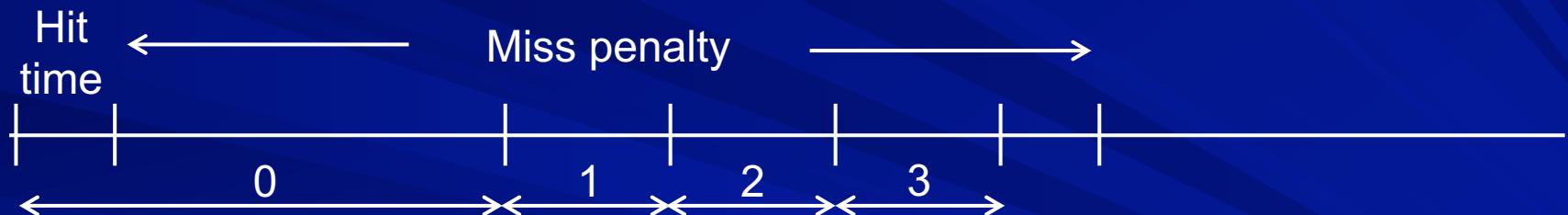
How do these parameters vary with

1. read policy?
2. load policy?
3. replacement policy?
4. fetch policy?
5. write policy?

Miss penalty



Concurrent read



Wrap around load + forwarding



Cache plus various buffers

- Buffer for block(s)/word(s) waiting to be written into memory
 - write buffer
- Buffer for block(s) fetched in anticipation
 - prefetch buffer
- Buffer for displaced blocks waiting for re-admission or permanent disposal
 - victim cache

Which write policy works better?

- Clustered and frequent writes
 - get the block into cache
 - do many writes into it
 - write back
- Sparse writes
 - just write into main memory
- Sparse writes followed by reads
 - get the block
 - write into both

Improved buffer for write through

- Effective for Write Through cache when there are multiple writes to same block
- Buffer size = 1 block, rather than 1 word
- Writes belonging to same block are collected in buffer
- Write from buffer to memory when
 - Buffer gets full or
 - Write for a different block arrives

Split cache vs. Unified, cache

Compare

1. access conflicts
2. space utilization
3. cost.

On-chip cache vs. off-chip cache

Compare

1. access time
2. capacity.

2-Level Cache Performance

Average memory access time =

$$\text{Hit time} + \text{Miss rate} * \text{Miss penalty}$$

Mem stalls / Instr =

$$\text{Miss rate} * \text{Miss Penalty} * \text{Mem accesses / Instr}$$

Average memory access time = Hit time +

$$\begin{aligned} & \text{Miss rate1} * \text{Miss penalty1} + \\ & \text{Miss rate2} * \text{Miss penalty2} \end{aligned}$$

Mem stalls / Instr = $(\text{Miss rate1} * \text{Miss penalty1} +$

$$\text{Miss rate2} * \text{Miss penalty2}) *$$

Mem accesses / Instr

Miss rates in 2 level cache

Consider 2 level cache

L1 miss rate = no. of L1 misses/ no. of requests at L1

L2 miss rate = no. of L2 misses/ no. of requests at L2?

- Global Miss rate
 - no. of misses / no. of requests, on the whole
- Solo Miss rate
 - miss rate if only L2 cache was present
- Relationship among these?

Inclusive & Exclusive caches

- Inclusive:

- Every information that is there in L1 cache is also in L2 cache

- Not vice versa (of course)

- Exclusive:

- No information is duplicated

- Neither inclusive, nor exclusive

What is required to ensure the above properties?

2-level cache example

CPI with no miss = 1.0 Clock = 500 MHz

Main mem access time = 200 ns

Miss rate = 5%

Adding L2 cache (20 ns) reduces miss to 2%. Find performance improvement.

Solution:

Miss penalty (mem) = $200/2 = 100$ cycles

Effective CPI with L1 = $1 + 5\% * 100 = 6$

Solution continued

Miss penalty (L2) = $20/2 = 10$ cycles

Total CPI = Base CPI + stalls due to L1 miss
+ stalls due to L2 miss

$$= 1.0 + 5\% * 10 + 2\% * 100$$

$$= 1.0 + 0.5 + 2.0 = 3.5$$

Performance ratio = $6.0/3.5 = 1.7$

Question

Why many modern processors have 3 levels of cache, while a decade ago it was common to have only one level?

COL216

Computer Architecture

Memory Organization

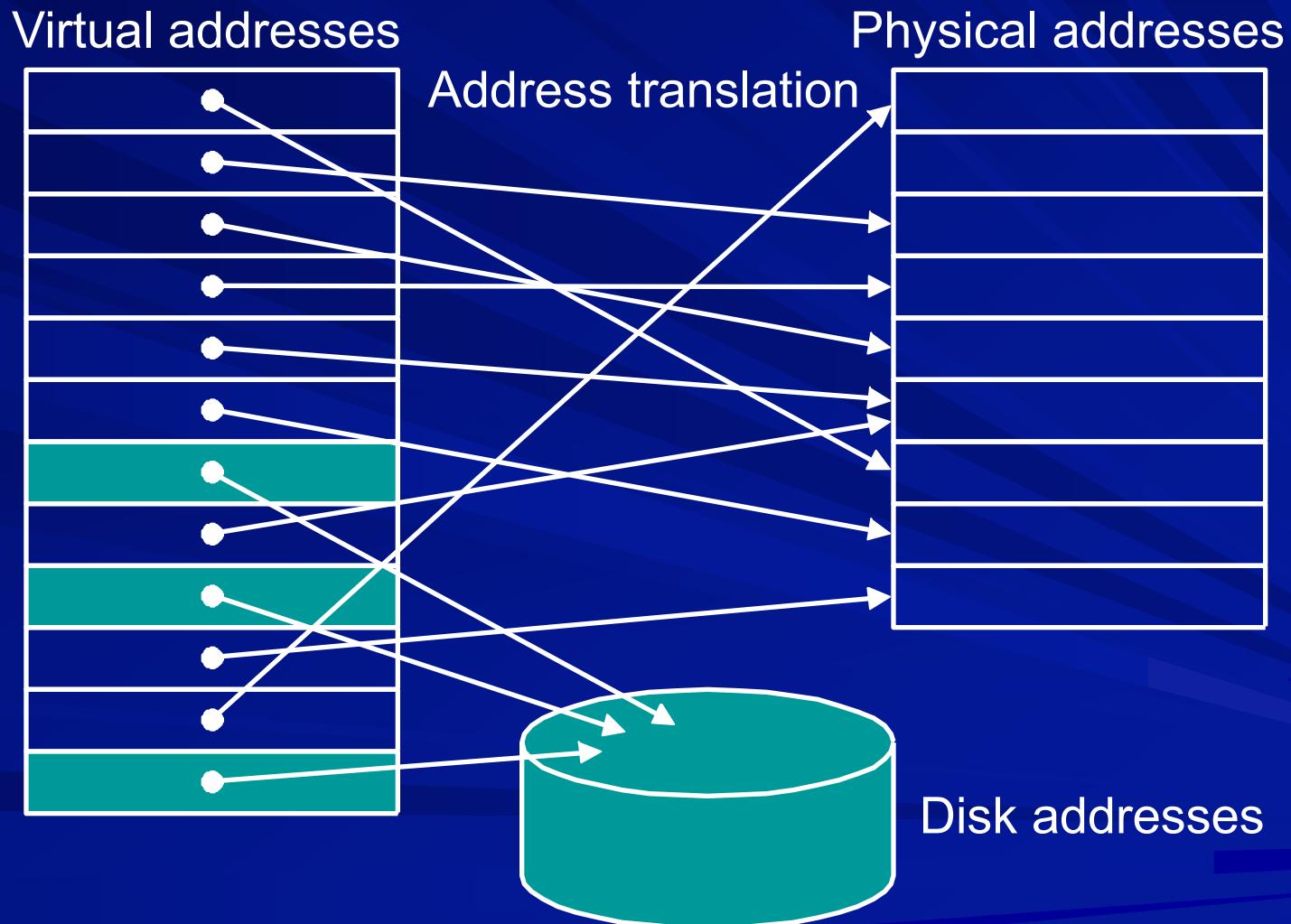
Virtual Memory

12th March 2022

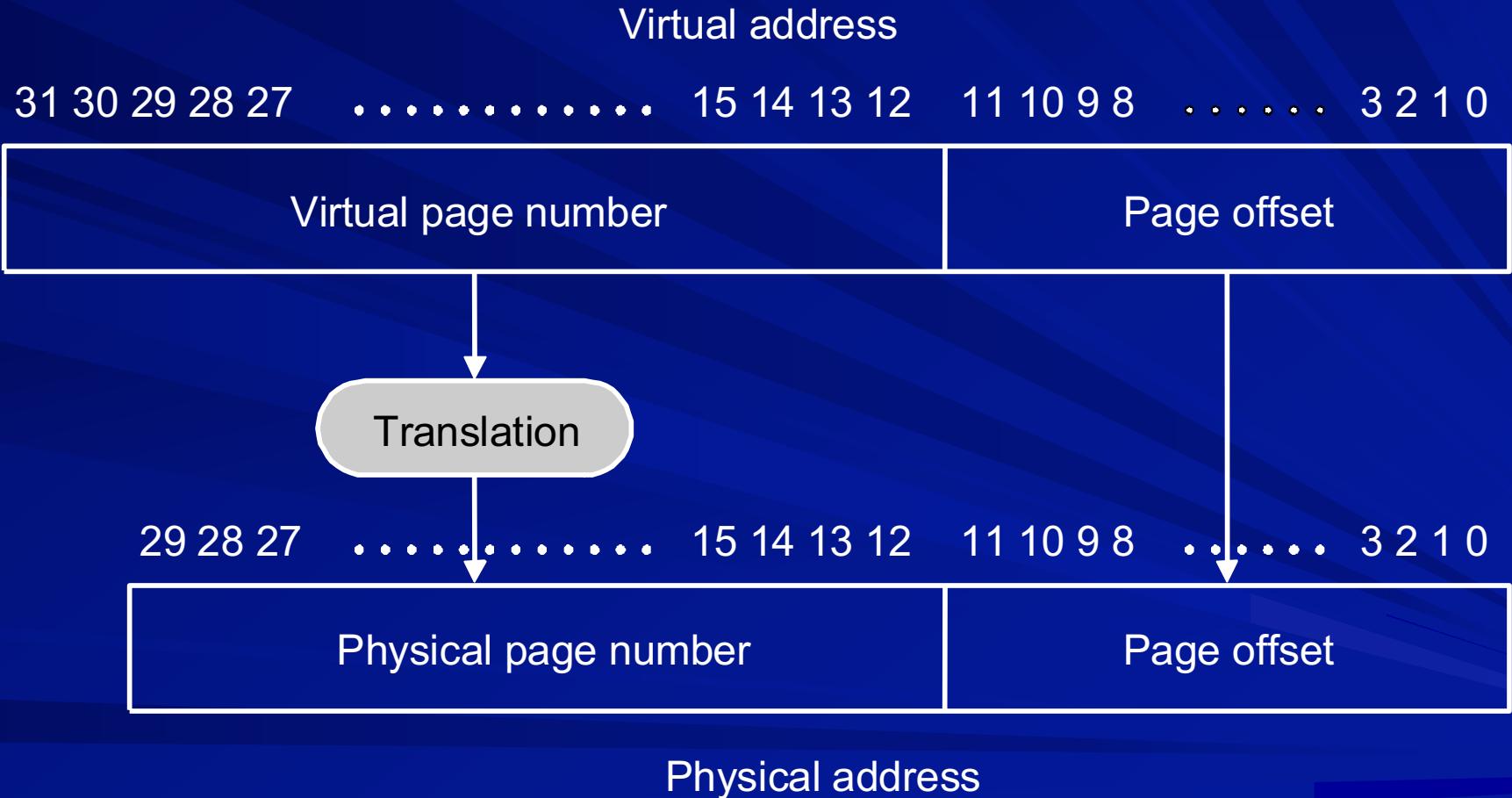
Virtual memory objectives

- To overcome the limitation of the physical memory size
- To allow multiple programs to share memory
 - Protection
 - Program relocation

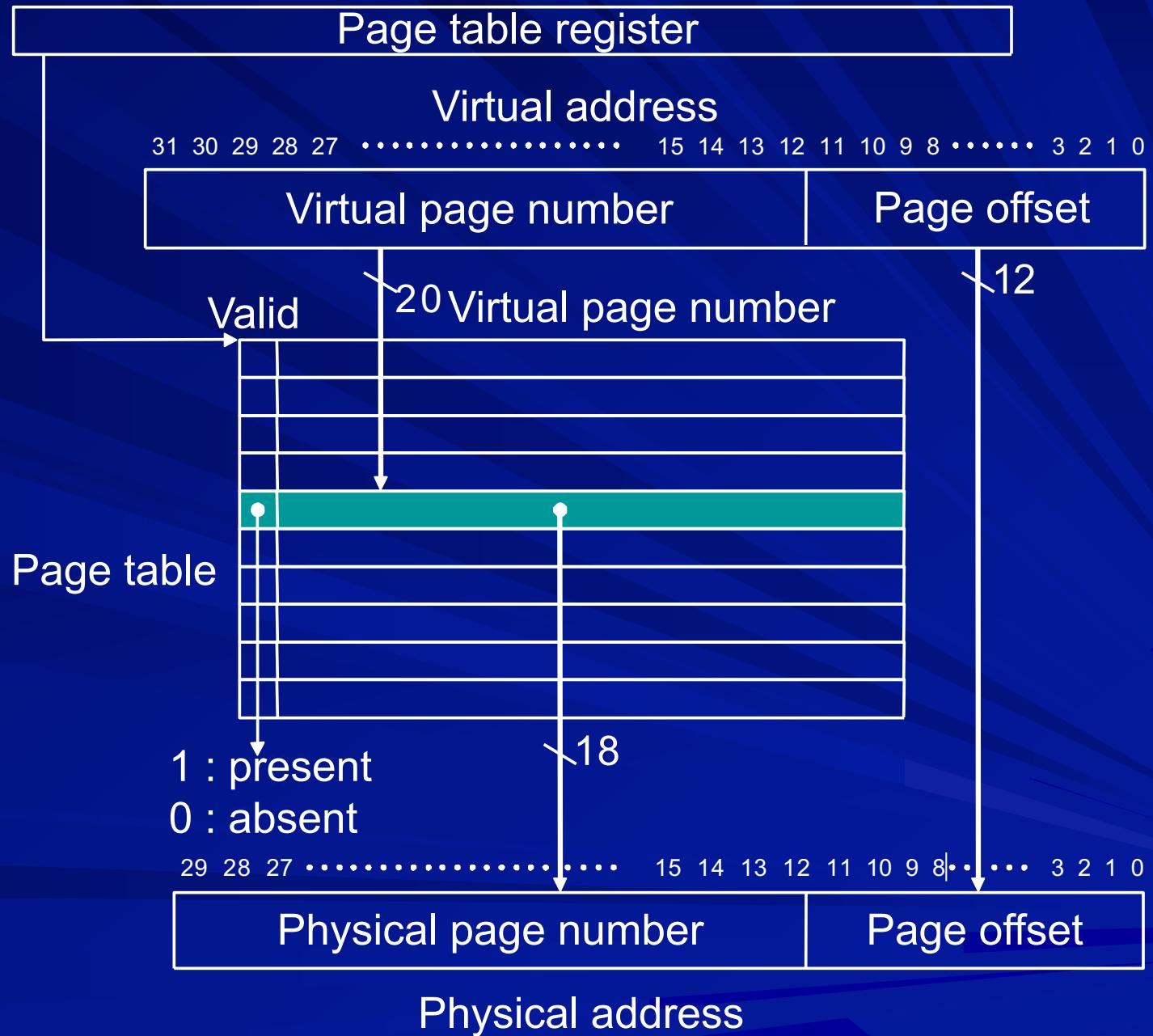
Virtual to physical mapping



Address translation



Page table



Mapping with page table

virtual page number

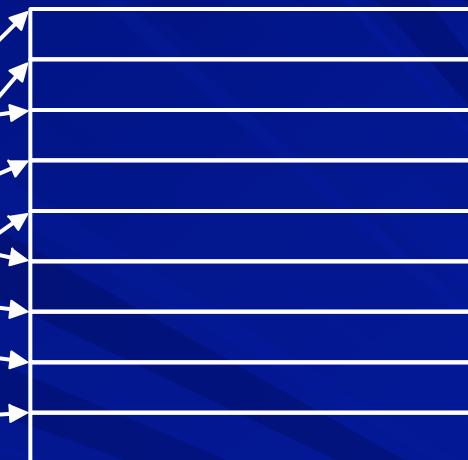


Page table:

v physical page
or disk addr

1	•
1	•
1	•
1	•
0	•
1	•
1	•
0	•
1	•
1	•
0	•
1	•

Physical memory



Disk storage



Virtual memory vs cache : similar but important differences

- Speed difference
 - cache and main memory : one order of magnitude
 - main memory and HDD : several orders of magnitude
- Response to a miss
 - cache : must be handled by h/w, can't afford to switch context
 - VM : can't keep CPU waiting, must switch context, can be conveniently handled by s/w
- Terminology differences
 - page vs block(cache line), page fault vs miss, page table vs cache directory etc

Differences contd.

■ Miss rate

- VM can only afford extremely low miss rates
- main memory much larger than cache, implies much lower miss rate

■ Policies

- pages much larger than blocks (~4KB - 16 KB) : capture large spatial locality, amortize transfer time
- fully flexible mapping (assoc mem not used)
- always write back
- good approximation to LRU

Back to page table

virtual page number



Page table:
physical page
or disk addr

1	•
1	•
1	•
1	•
0	•
1	•
1	•
0	•
1	•
1	•
0	•
1	•

Physical memory



Disk storage



Where is page table stored?

How big is it?

- Suppose virtual address = 32 bits, page size = 4 KB, page table entry = 4 B
- then number of page table entries = number of pages = $2^{32} / 2^{12} = 2^{20}$
- page table size = $2^{20} \times 2^2 = 4 \text{ MB}$
- hundreds of processes

Store in dedicated memory? main memory?

Handling large page tables

- Bound the page table size
- Exploit sparseness
- Use multiple levels
- Page the page table

Bounding the page table size

- Keep a bound register
 - Allocate as much space as necessary
 - Address space expands in one direction
- Split the virtual space into two parts, each with its own bound, e.g. heap + stack
 - Two segments can grow independently
- Sparseness not exploited

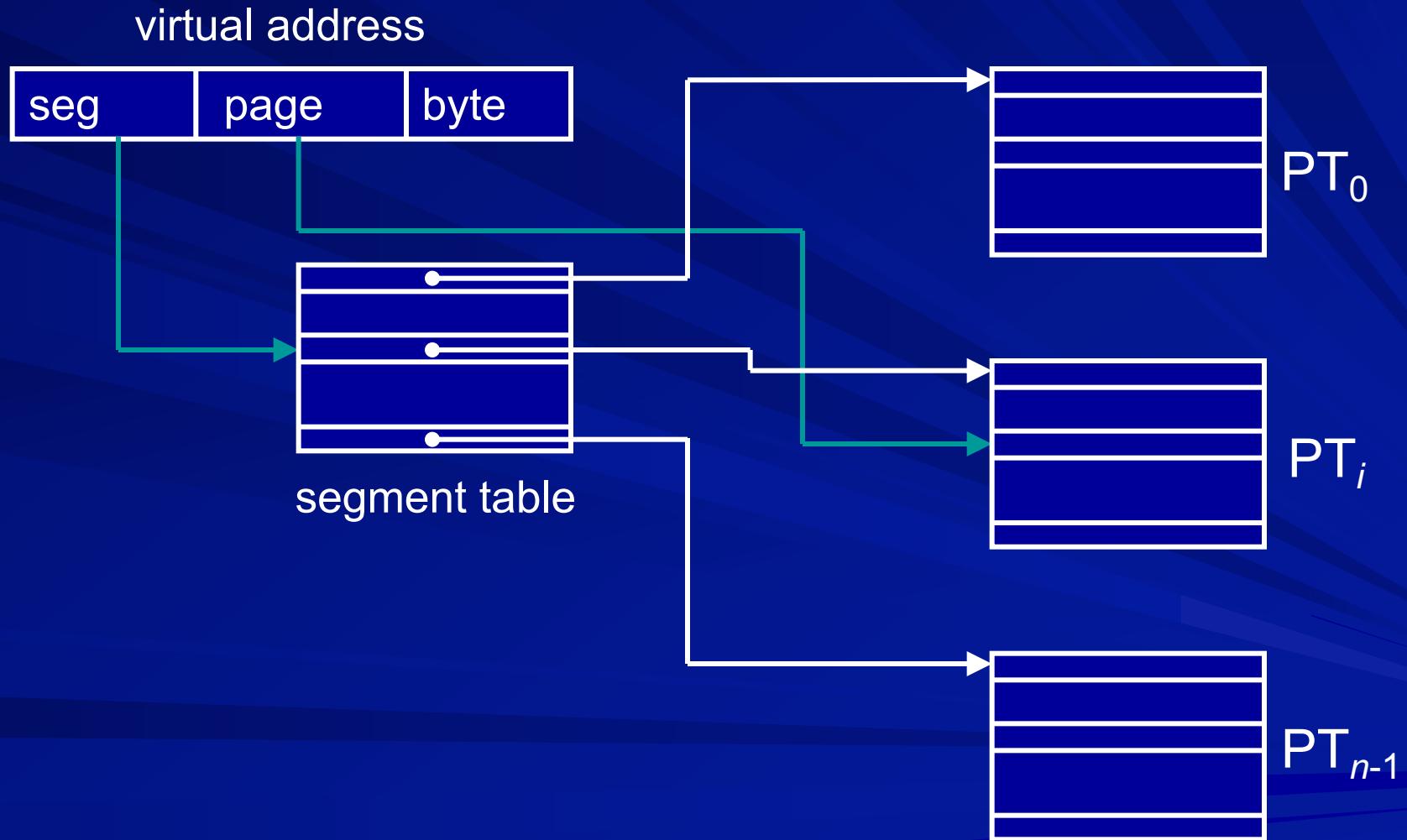
Exploit sparseness

- Page table: one entry per virtual page,
Cache: one entry per cache block
 - Associative memory impractical for page table
- Hashing can be used - inverted page table
 - Access is more complex than simply indexing into the page table

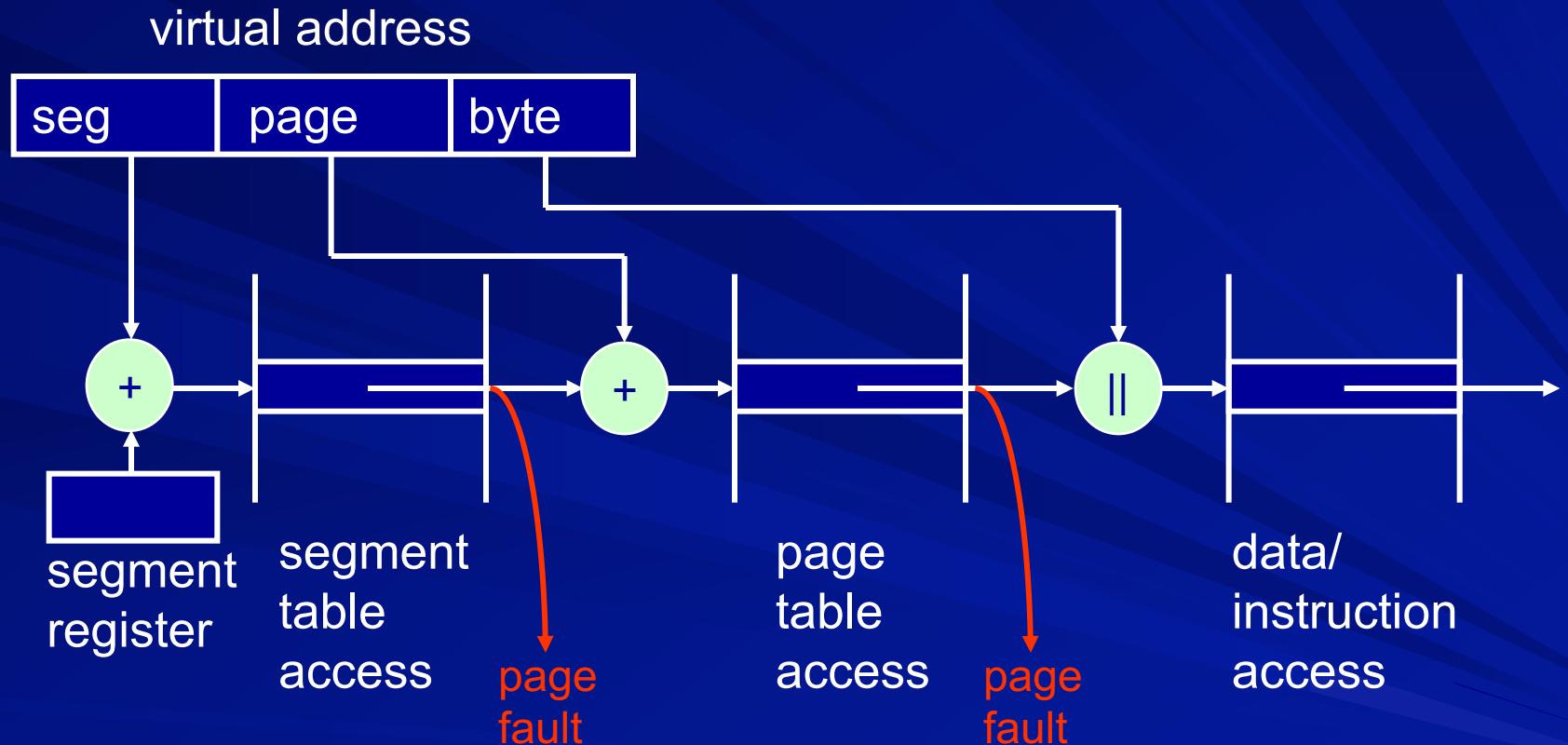
Two level page tables

- Virtual space divided into segments
- A smaller page table for each segment
- All page tables need not be in physical memory
- A segment table keeps track of where the page tables of different segments are located

2 Level page table structure



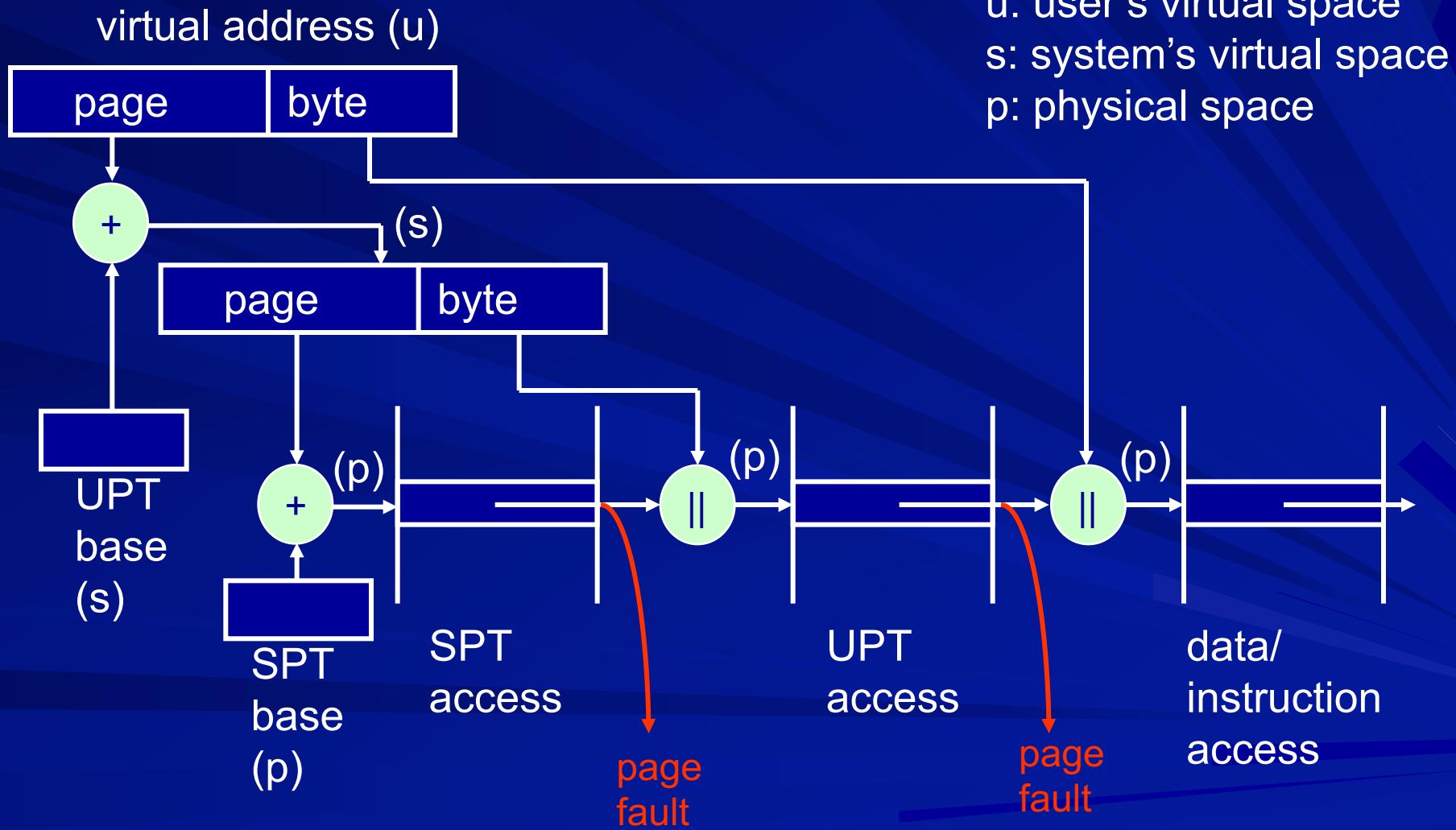
Memory access with 2 level page table



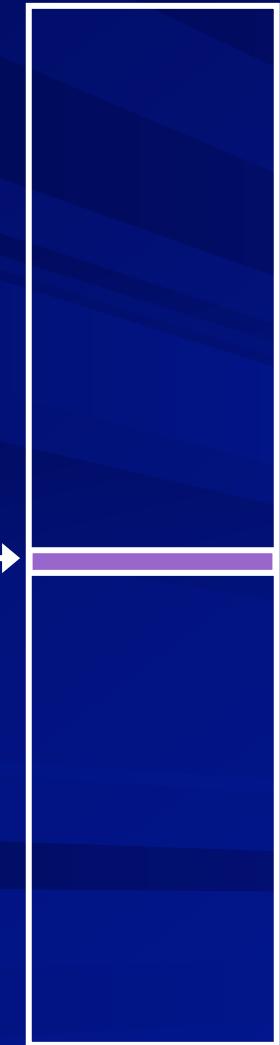
Keep page table in virtual space

- Page table is paged!
- Only active pages of page table are brought into physical memory
- How do you locate these pages? - need another page table!
- User page tables are kept in system's virtual space
- System's page table (or at least a part of it) must be kept in physical memory and accessed directly

Memory access with paged page table



user
virtual space



physical
space



user
virtual space

X →



virtual
page

physical
space

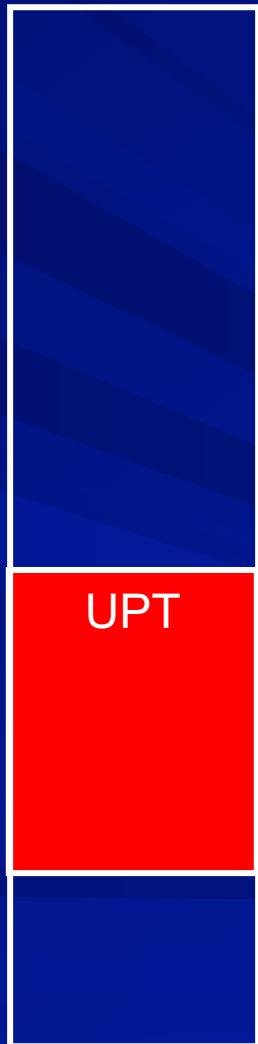


physical
page

user
virtual space



system
virtual space



physical
space

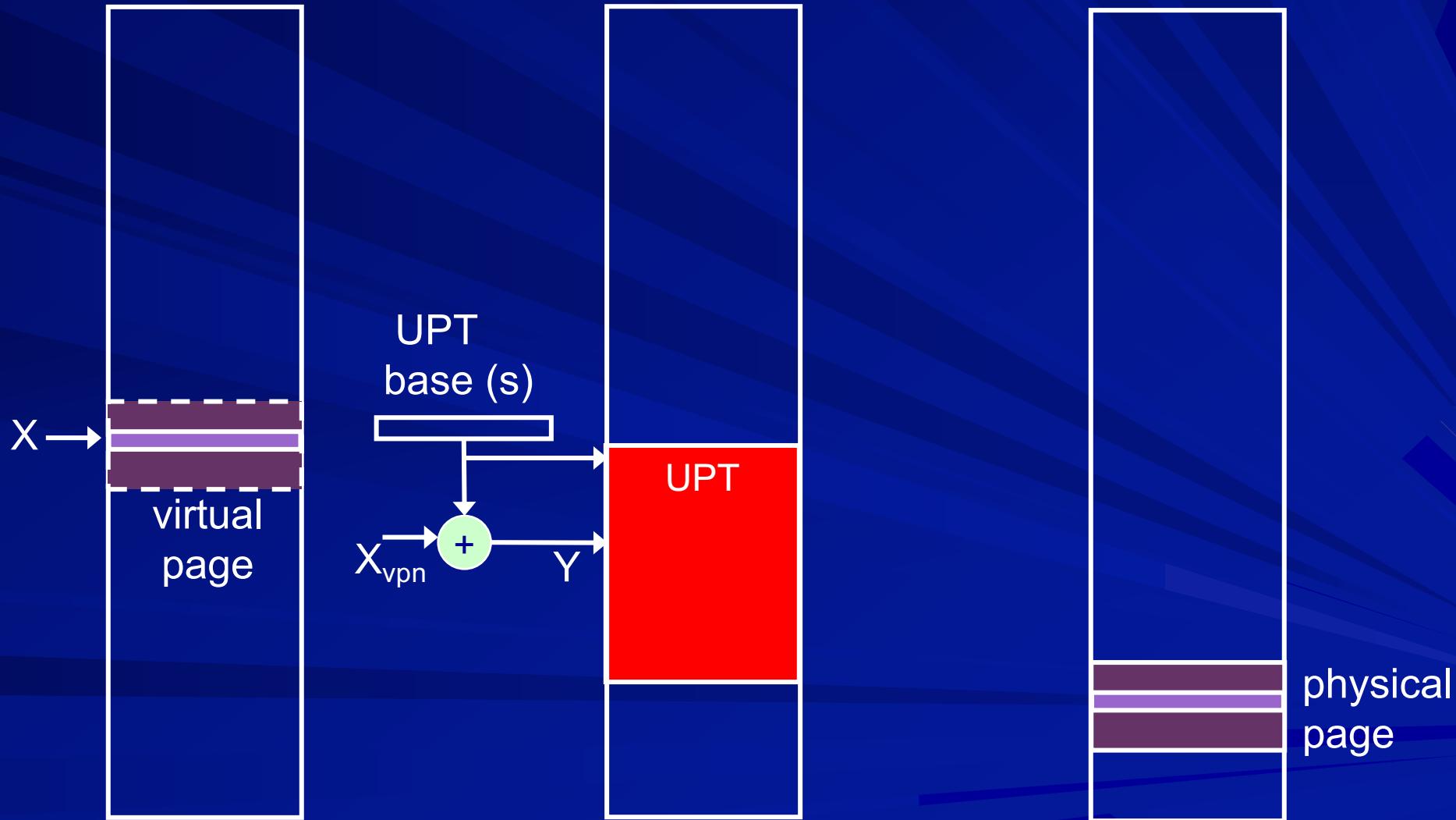


physical
page

user
virtual space

system
virtual space

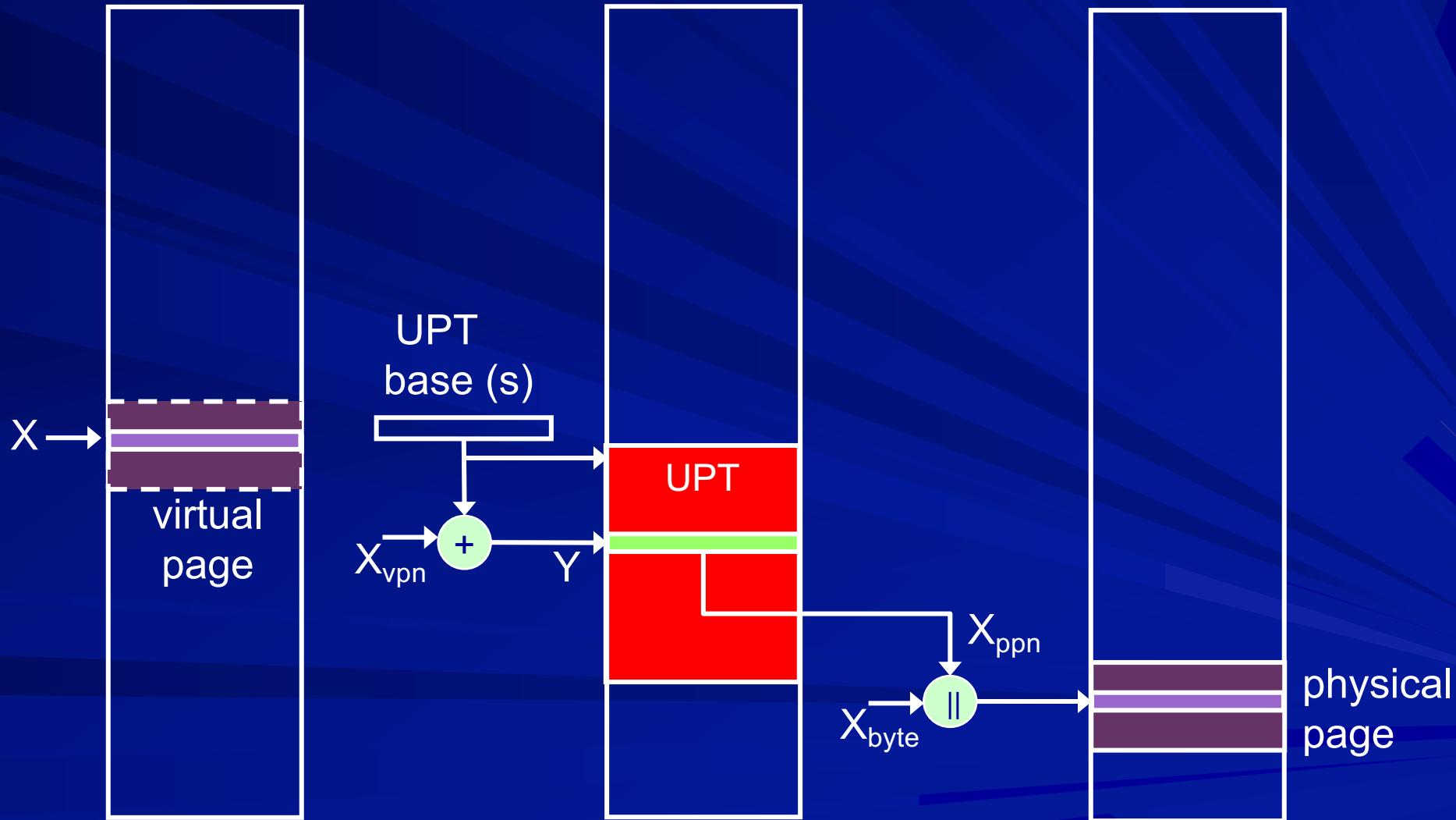
physical
space



user
virtual space

system
virtual space

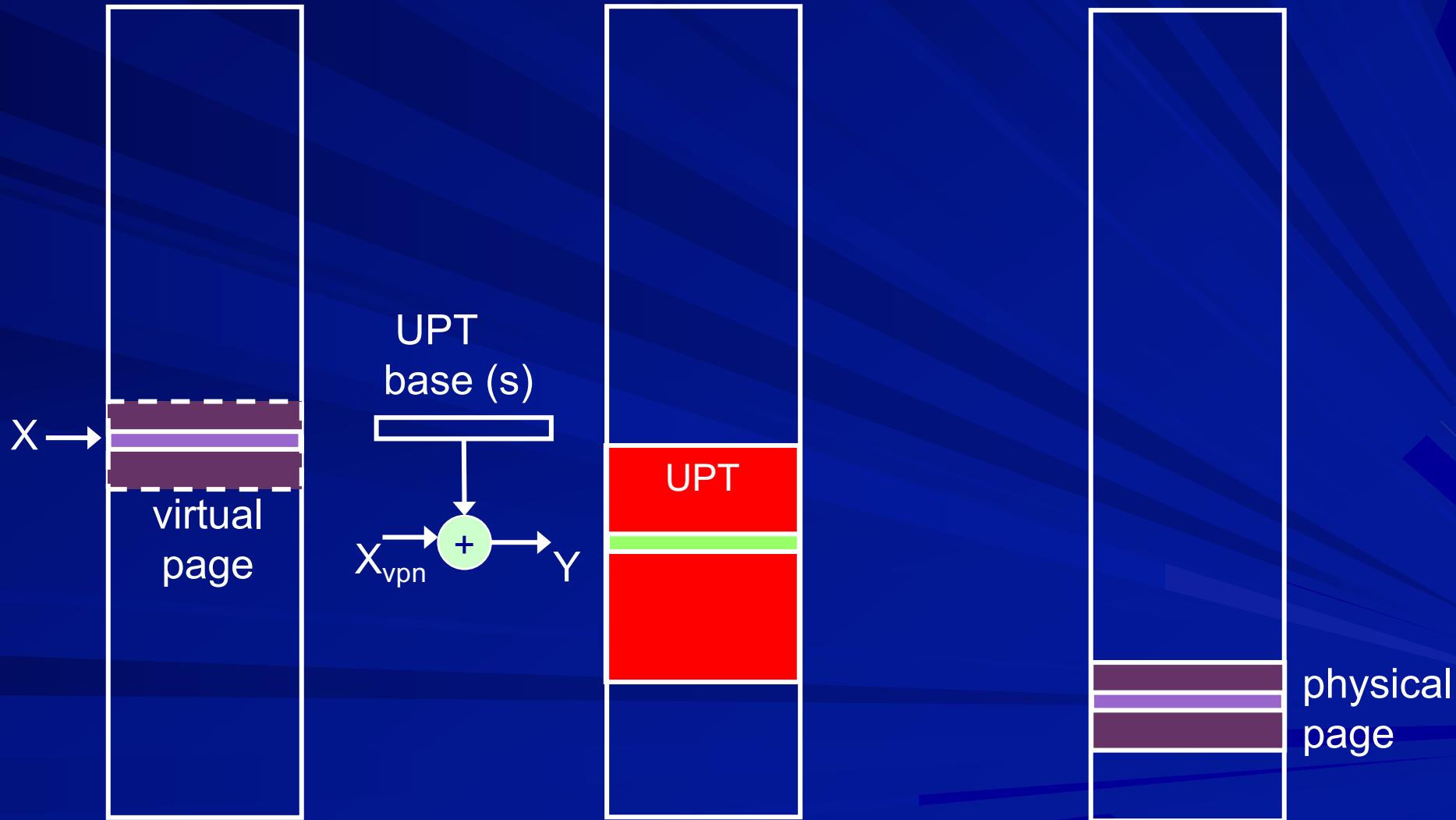
physical
space



user
virtual space

system
virtual space

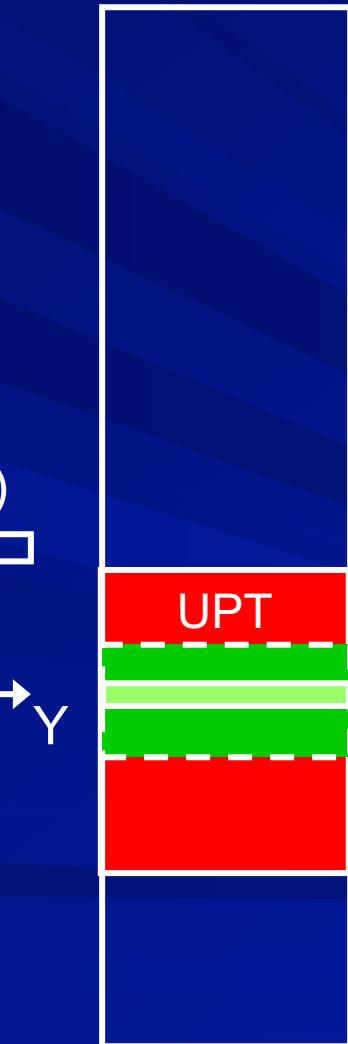
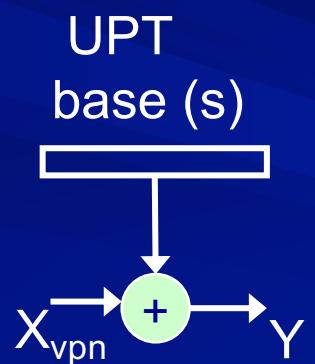
physical
space



user
virtual space



system
virtual space



physical
space

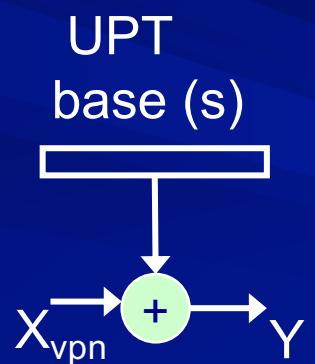


physical
page
physical
page

user
virtual space



system
virtual space



virtual
page

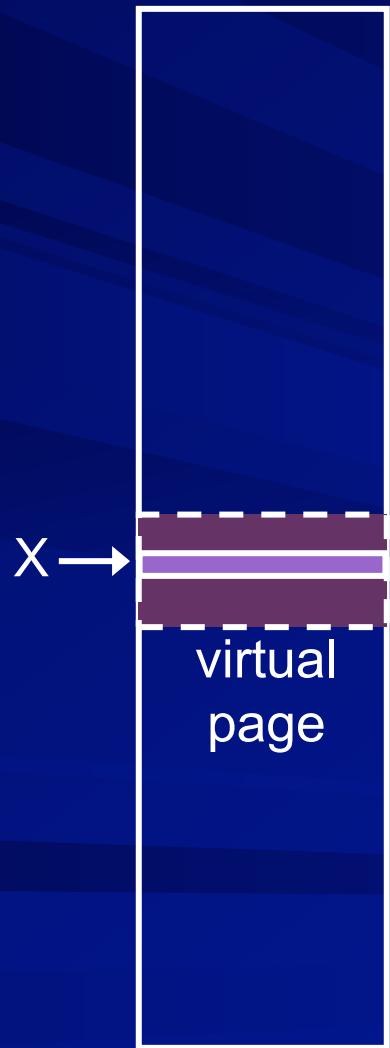
physical
space



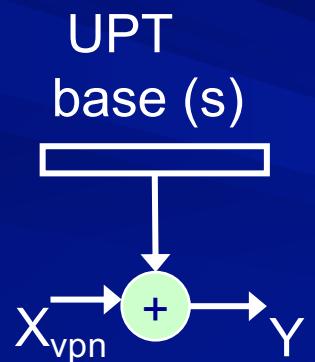
physical
page

physical
page

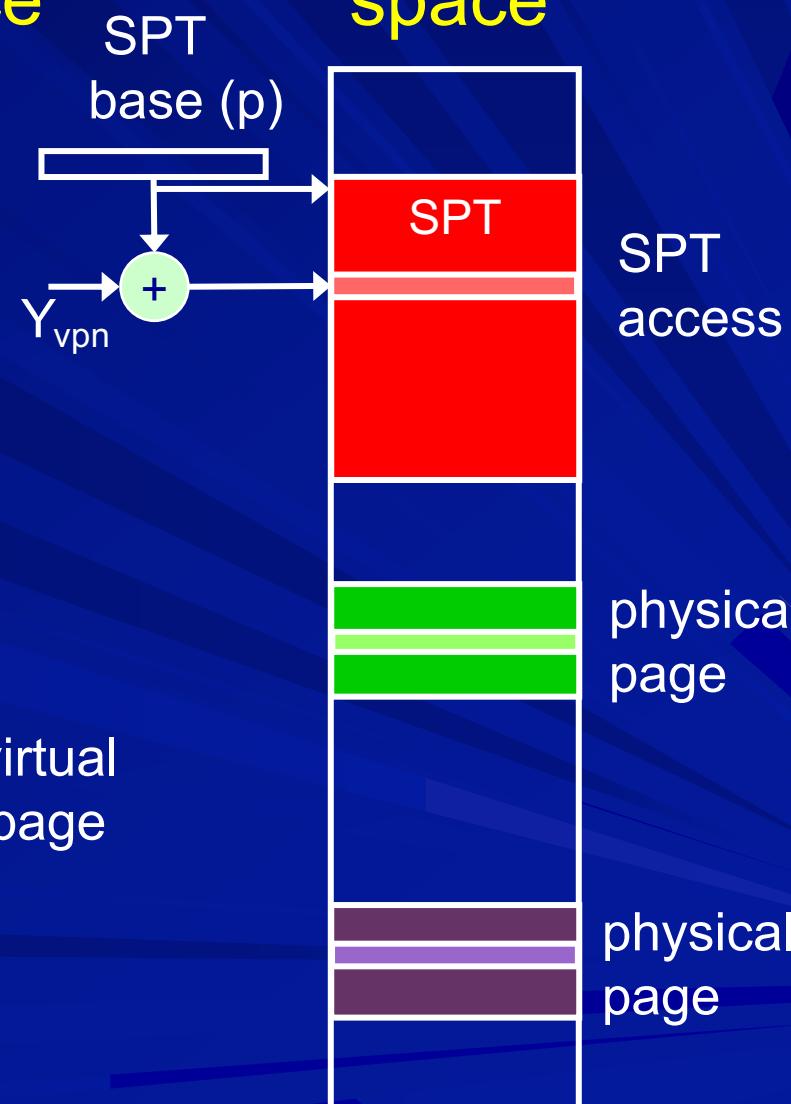
user virtual space



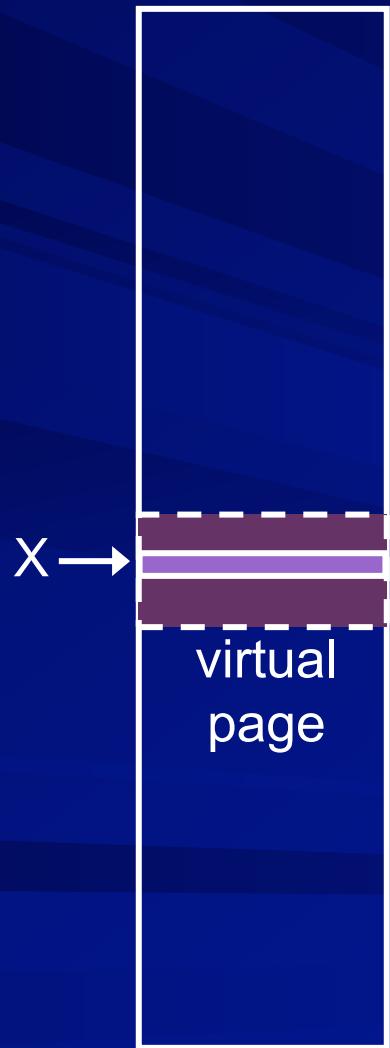
system virtual space



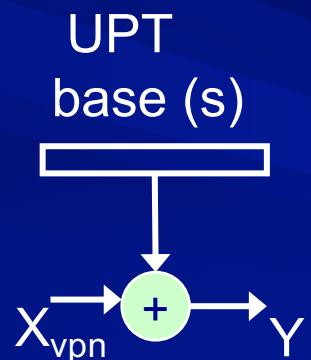
physical space



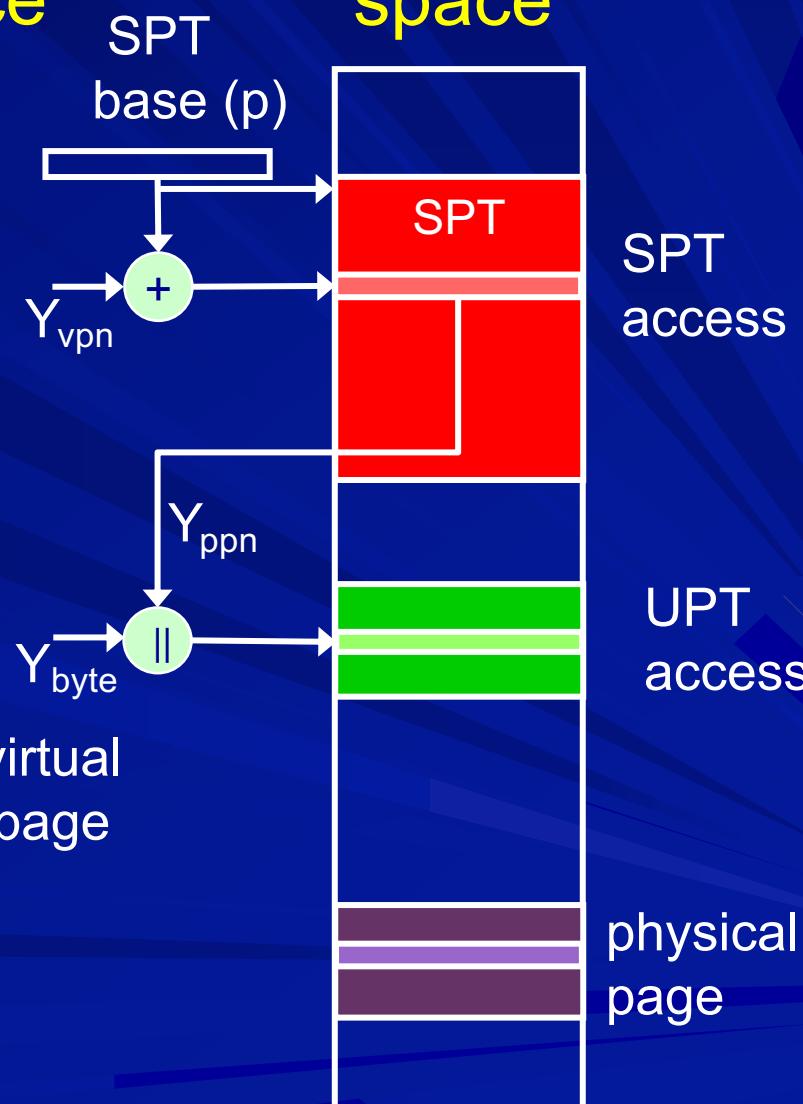
user virtual space



system virtual space



physical space

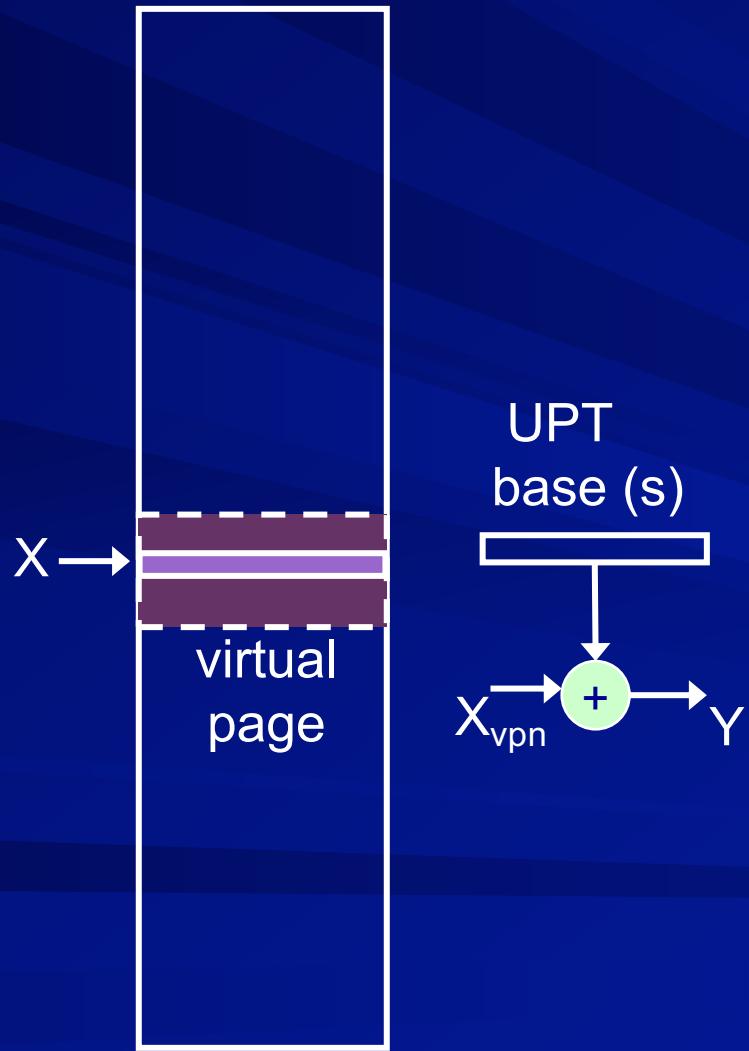


SPT
access

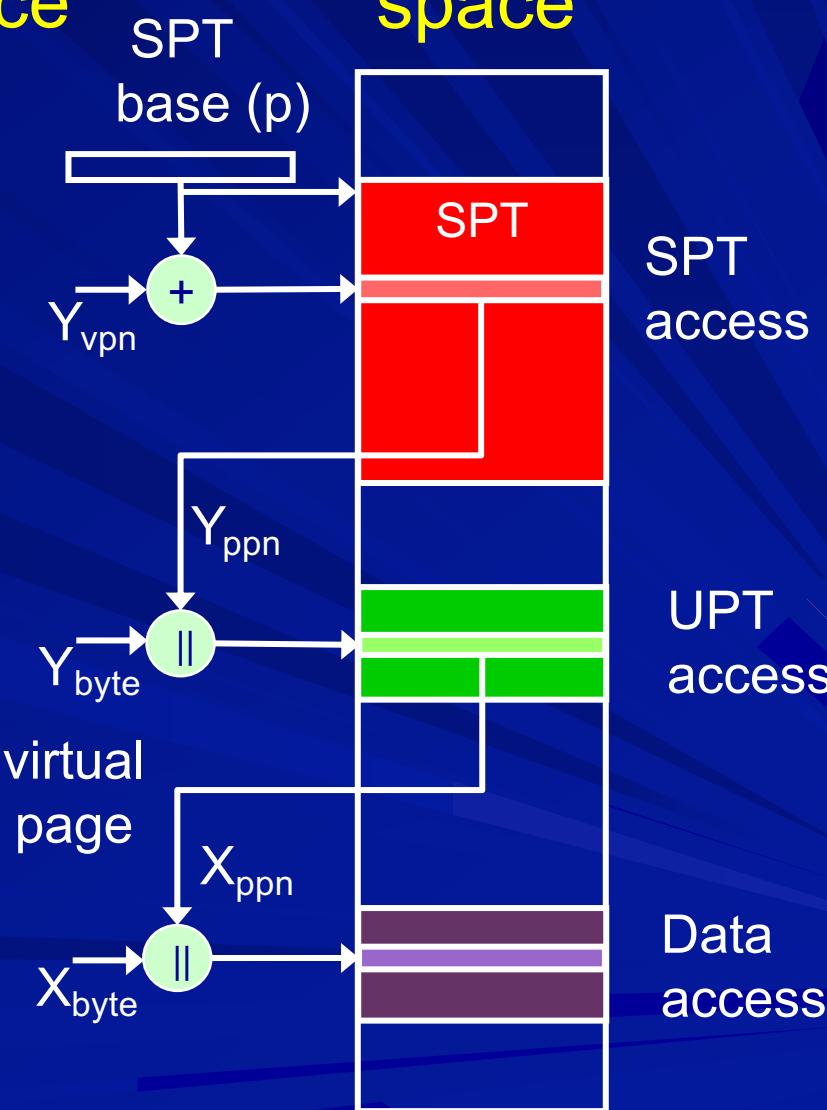
UPT
access

physical
page

user virtual space



system virtual space



physical space

SPT
access

UPT
access

Data
access

Memory protection with VM

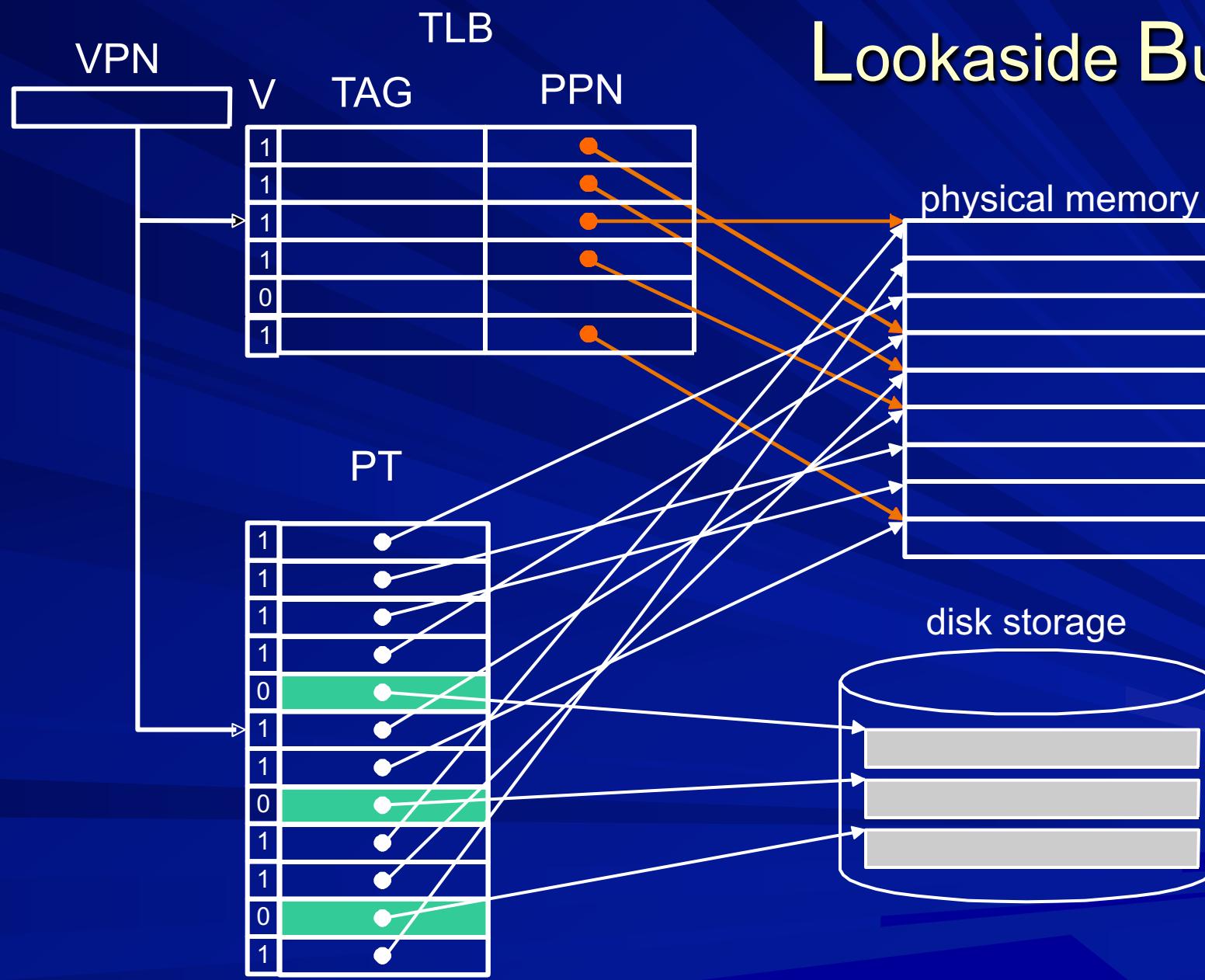
- A process can access only those physical pages which are pointed to by its page table
- Page tables are initialized by the OS and are updated automatically. User processes can't modify these.
- Programs run in user mode or privileged mode. Certain operations are possible in privileged mode only.

Virtual memory hit time

- simple page table :
 $2 \times$ physical memory access time
- 2 level page table :
 $3 \times$ physical memory access time
- paged page table :
 $3 \times$ physical memory access time

Can this be better??

Translation Lookaside Buffer



How is TLB Miss handled?

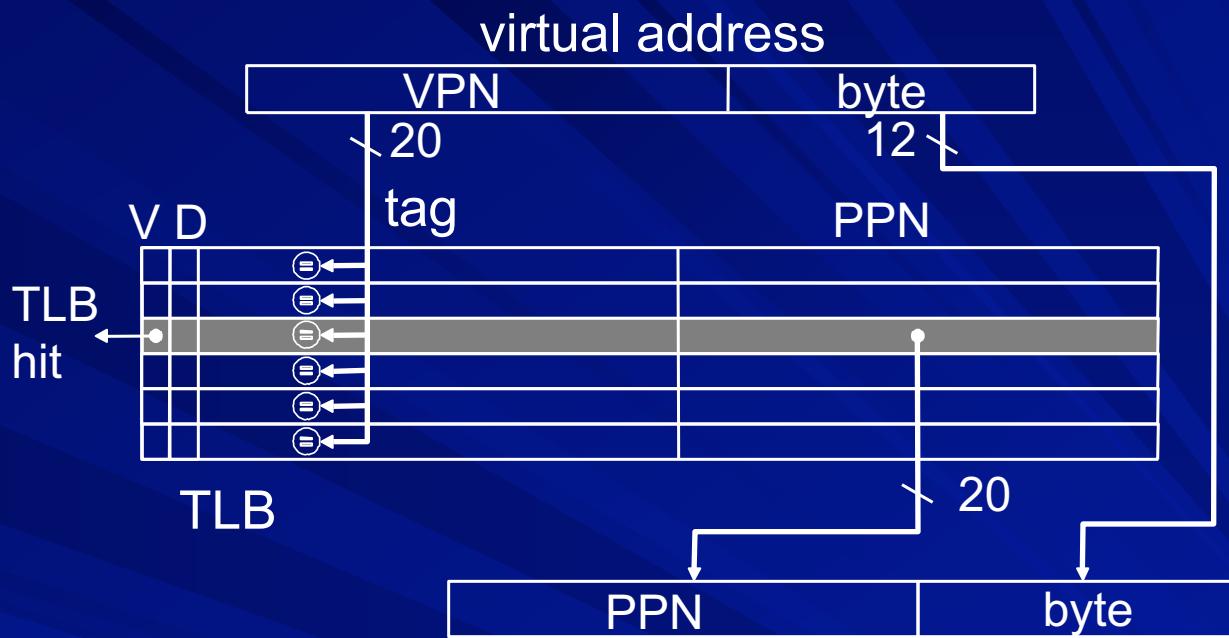
■ Hardware

- Memory Management Unit accesses Page Table
- Extra hardware, but efficient

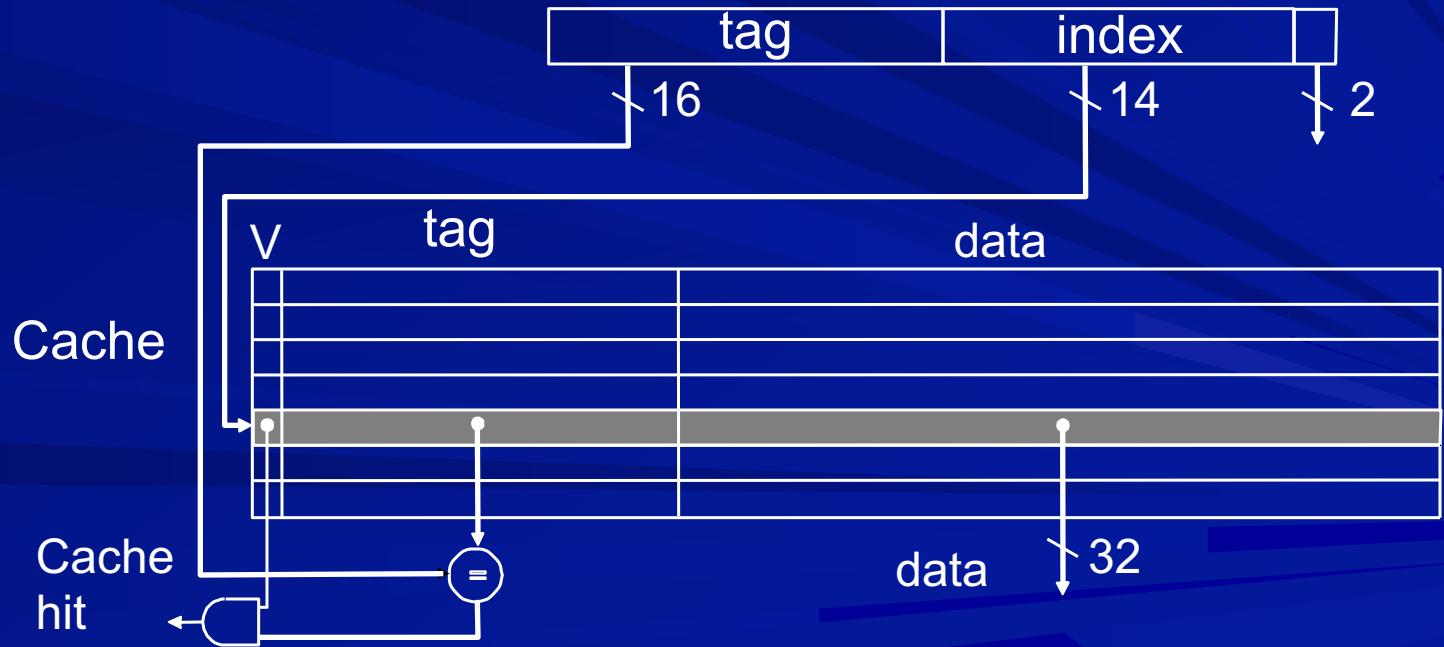
■ Software

- Exception is caused
- OS (exception handler) accesses Page Table
- Page Table structure defined by OS, not fixed by hardware

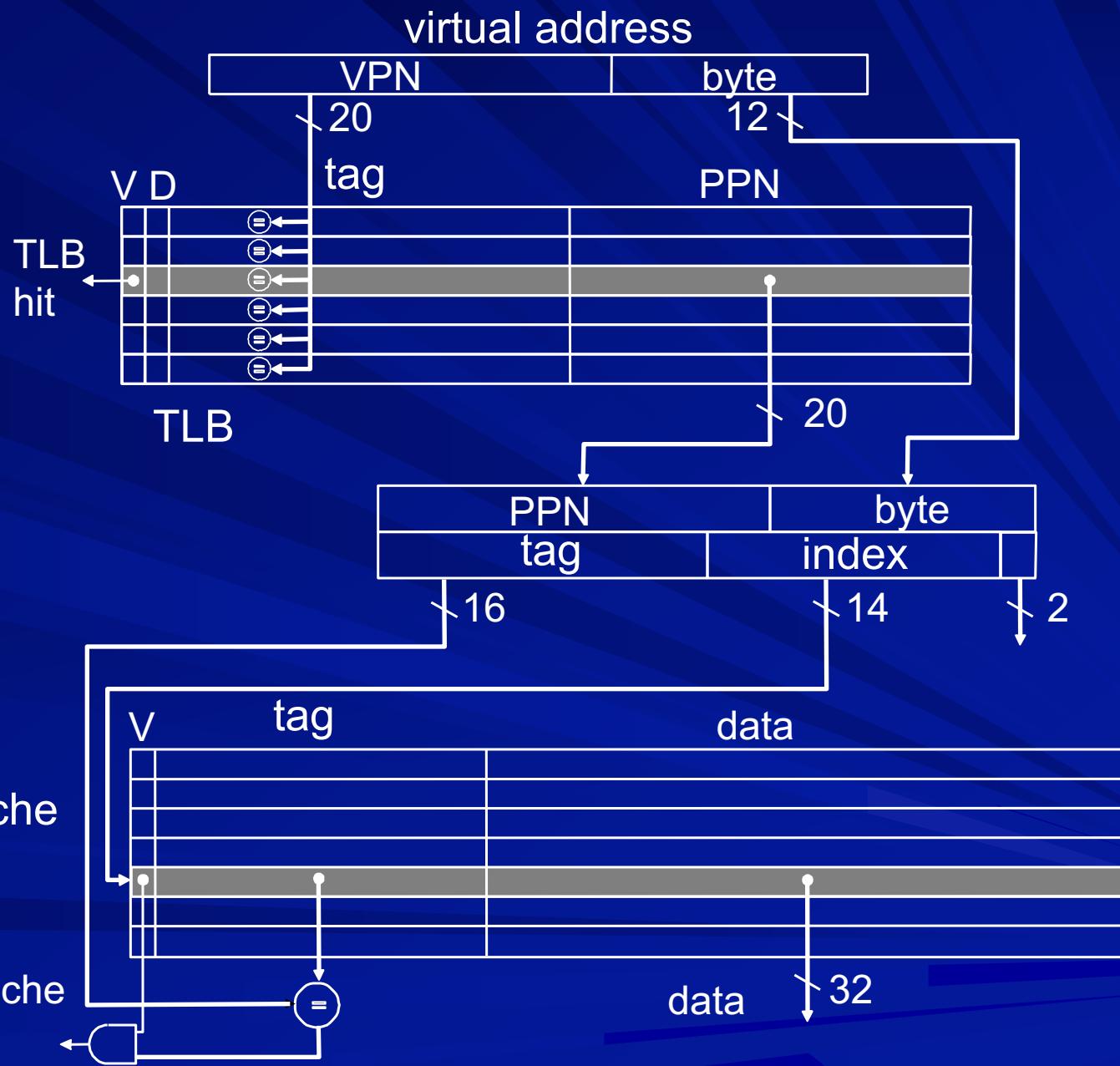
TLB with Cache



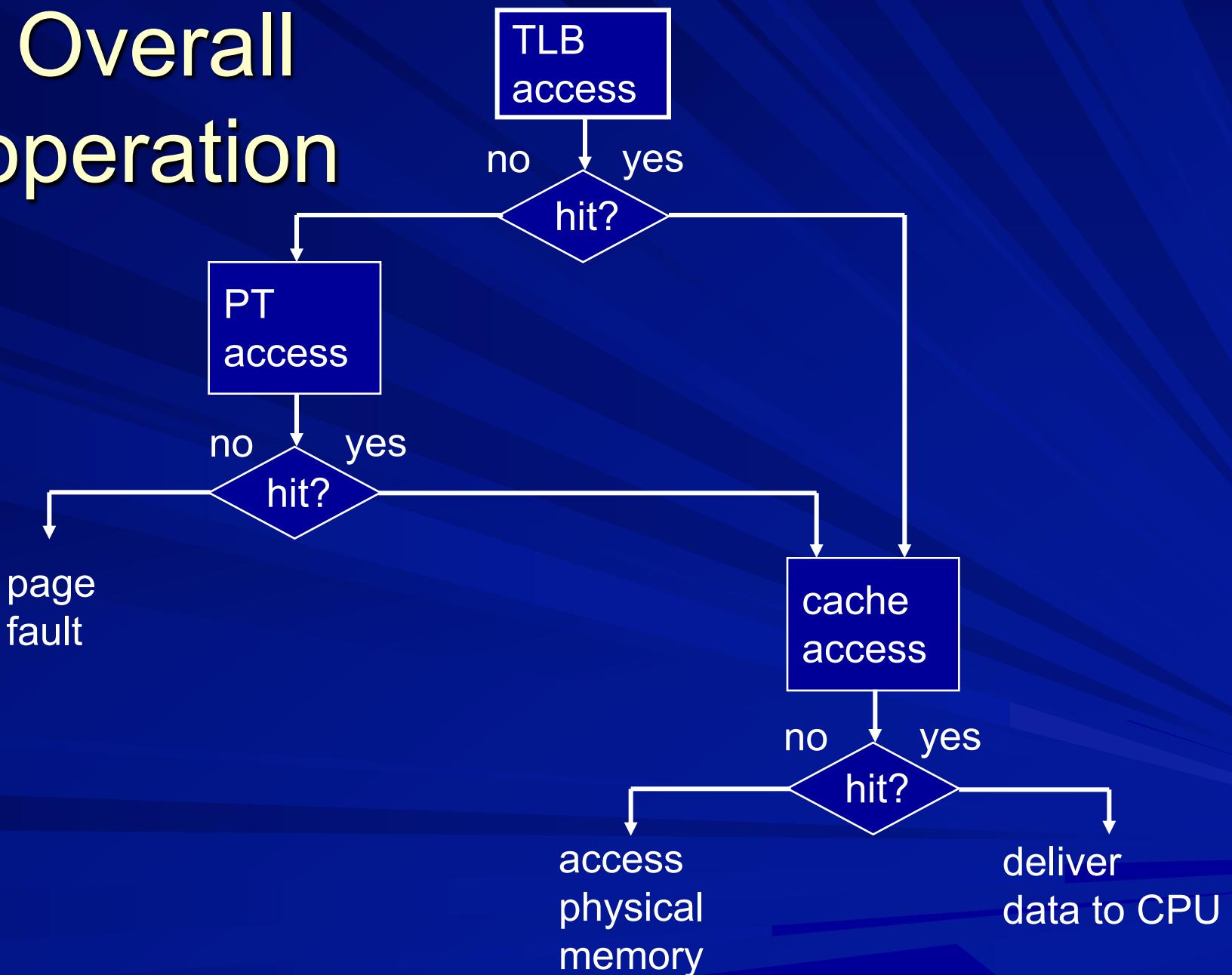
TLB with Cache



TLB with Cache



Overall operation



Virtually addressed cache

- Tags in the cache are from virtual addresses
- Cache access is made first
- TLB accessed only after cache miss
- Problem of aliasing - two copies of a shared object in physical memory
- Physically tagged and virtually indexed cache may be used

Indexing before address translation

- Virtually indexed, physically tagged cache
 - simple and effective approach
 - indexing overlaps with address translation, tag matching after address translation
 - valid only if index field does not change during address translation (possible for caches that are not too large)

Virtual Page No.	Offset in page
Tag	Index

COL216

Computer Architecture

Input/Output – 1

14th March 2022

Diversity of Input/Output devices

Communicate with

- Human
- Computer
- Physical systems
- Storage

Variations in terms of

- direction
- speed
- data type . . .



Communication with humans

- Key board, mouse, joy stick, tablet, touch, gesture
 - very slow, $10^1\text{-}10^2$ bytes per sec
- Documents: Scanners, printers, plotters
 - $8\text{-}16$ ppm, $300\text{-}1200$ dpi, $10^2\text{-}10^3$ KB/sec
- Sound (voice/music): mikes, speakers
 - $10^1\text{-}10^3$ KB/sec
- Graphic displays
 - mega pixels, refresh rate, $10^1\text{-}10^3$ MB/sec

Communication with machines

- Wired network controllers

- $10^1\text{-}10^3 \text{ MB/sec}$

- Wireless network controllers

- ethernet, blue tooth, zigbee etc

- $10^0\text{-}10^1 \text{ MB/sec}$

- Modems

- $10^1\text{-}10^3 \text{ KB/sec}$

Communication with physical systems

■ Sensors

- pressure, temperature, humidity, flow, level, position, speed, proximity, gas
- audio, visual

■ Actuators

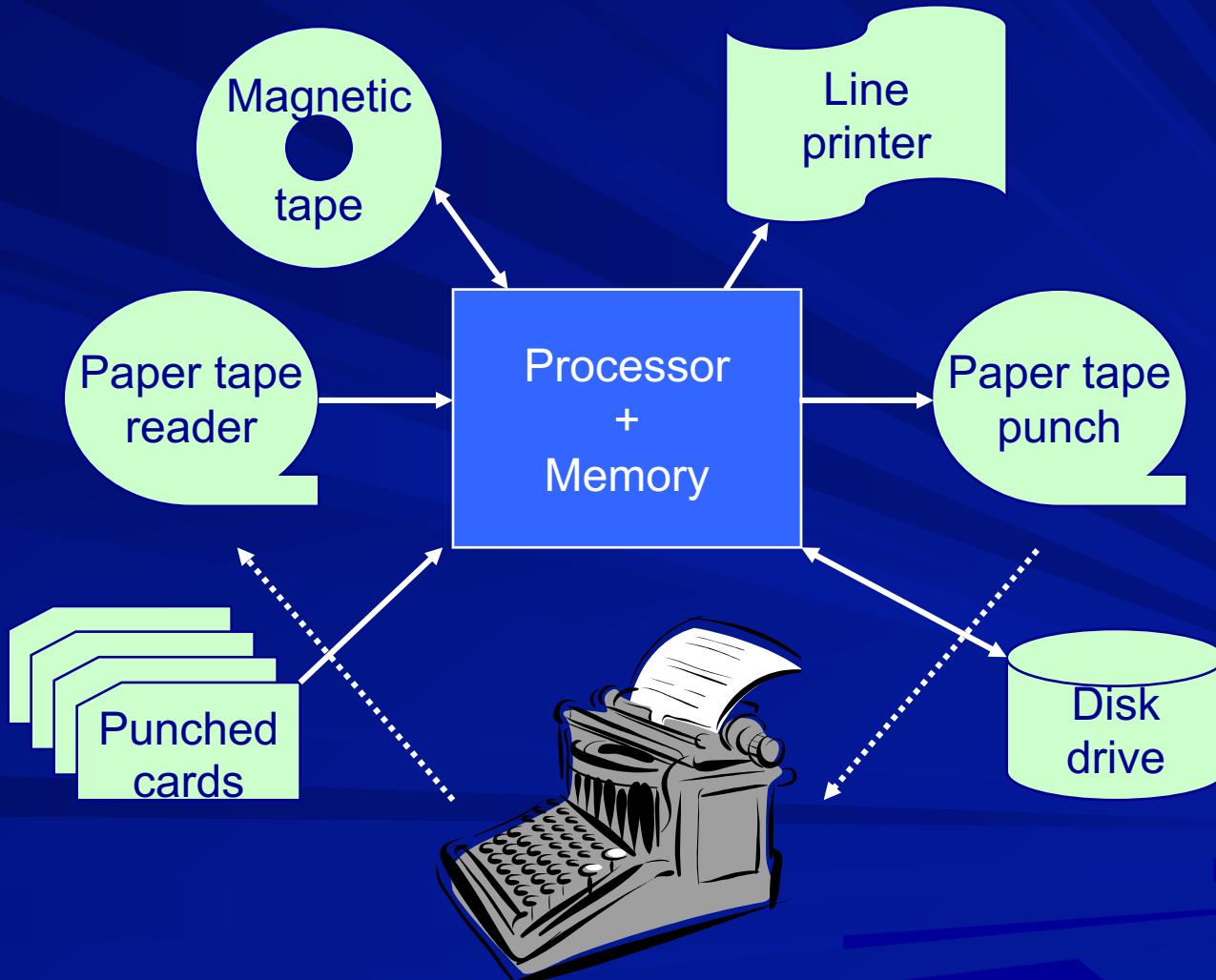
- switches, valves, solenoid, motors

Signal forms: on/off, analog, pulse train etc

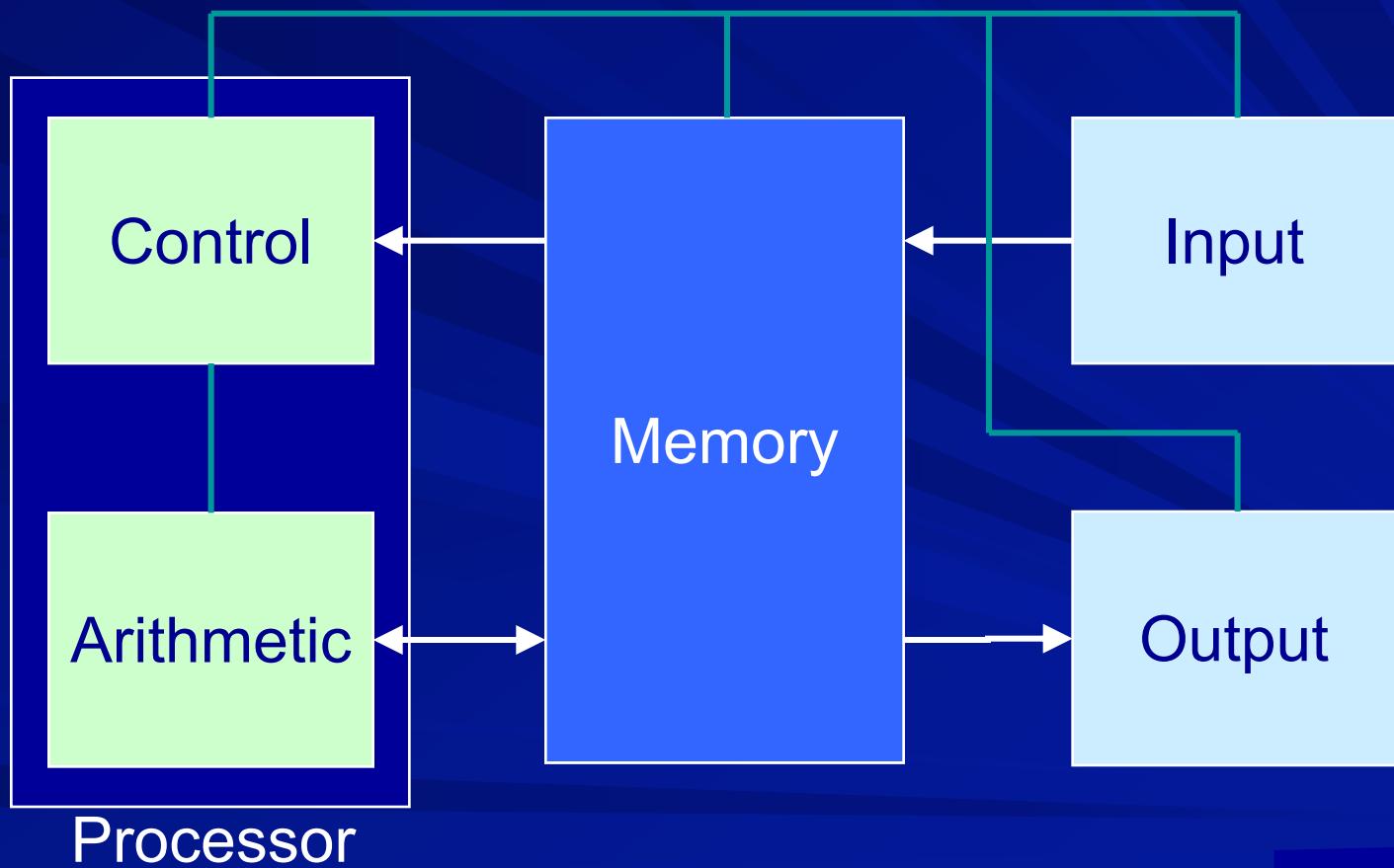
Storage devices

- Magnetic
 - disks, tapes
- Optical
 - CDROM, DVD, Blue Ray
- Semi-conductor
 - Nor Flash, Nand Flash

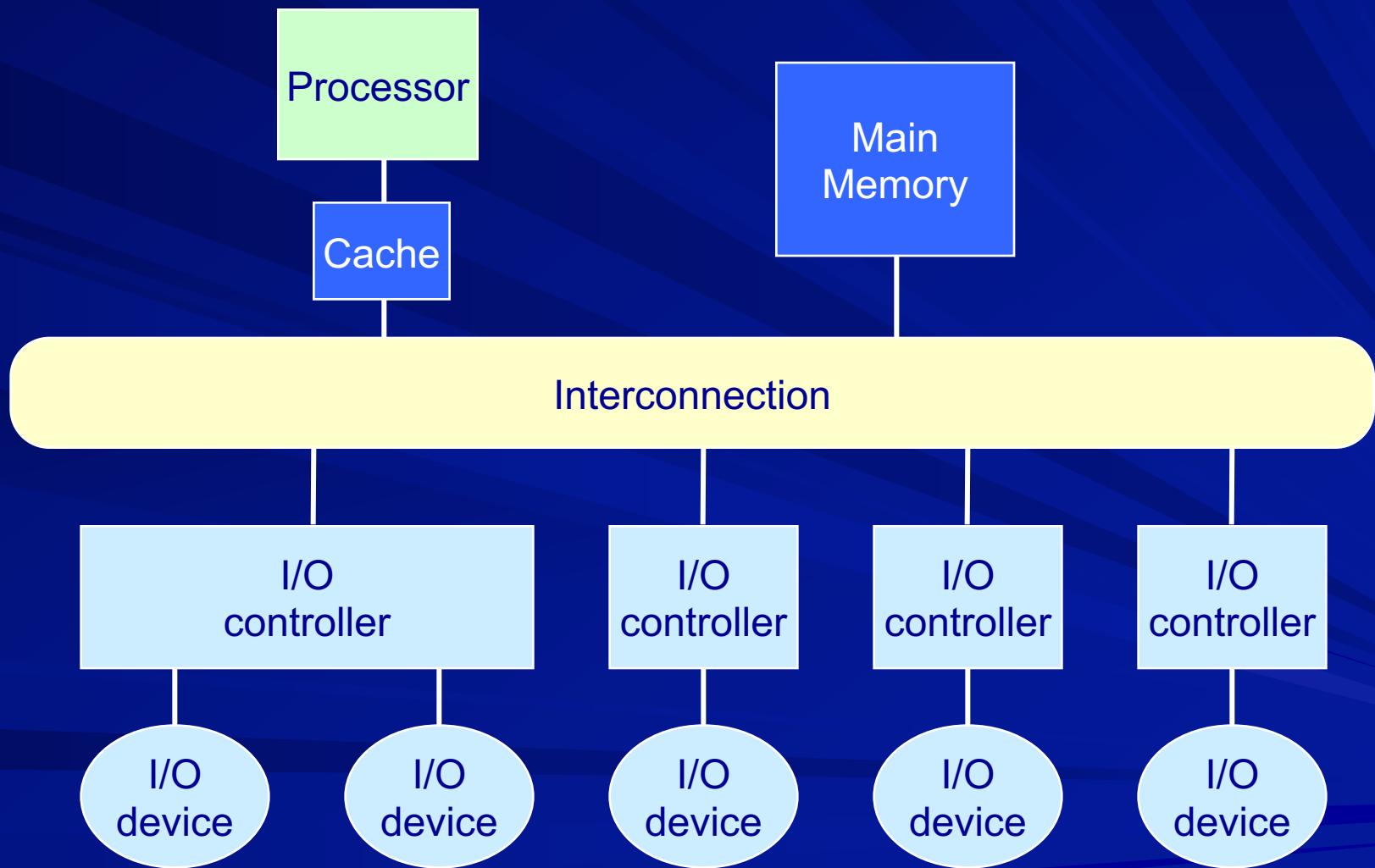
Peripherals circa 1970 : ICL 1909



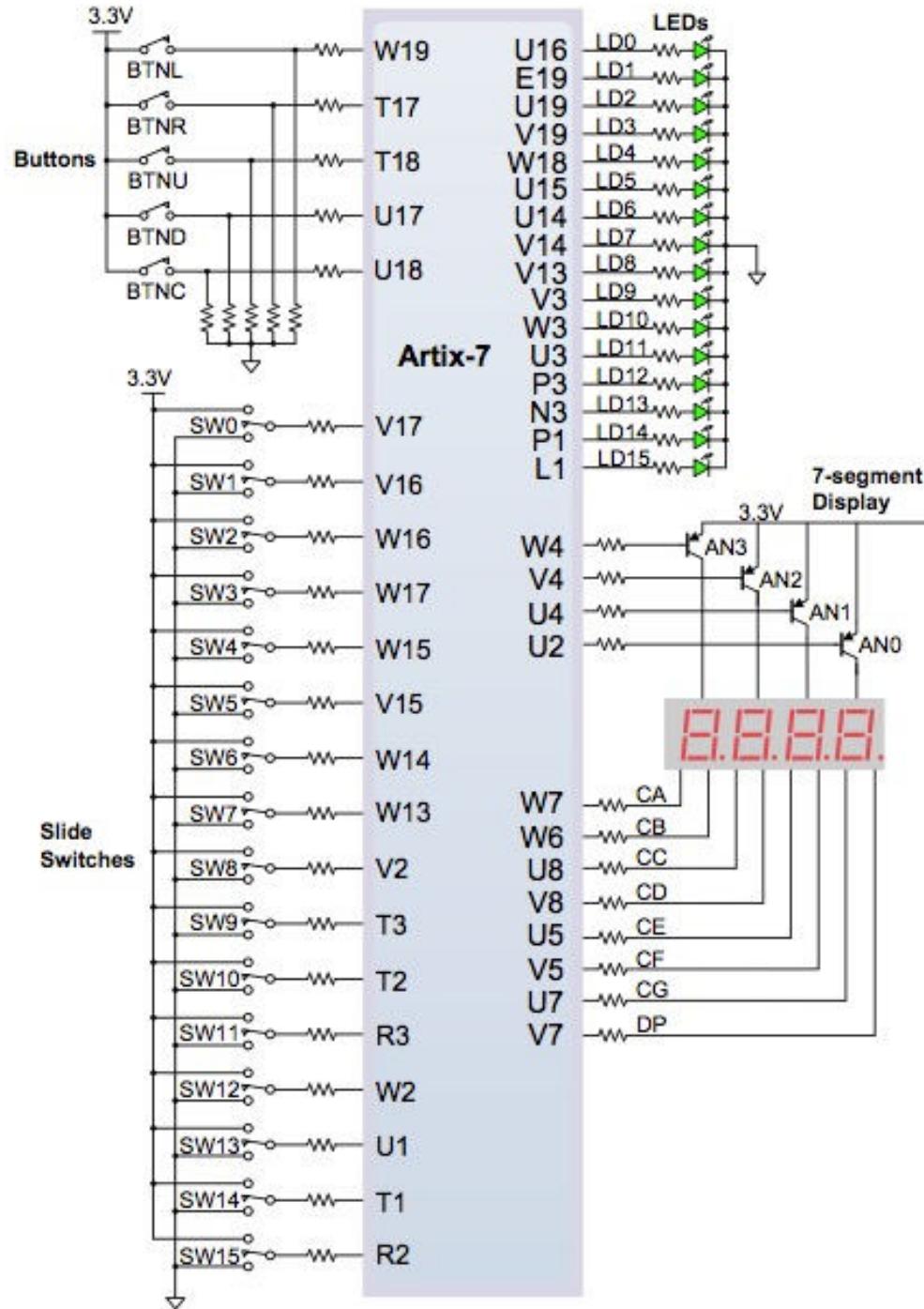
Computer System: Simplified Block Diagram



More realistic block diagram



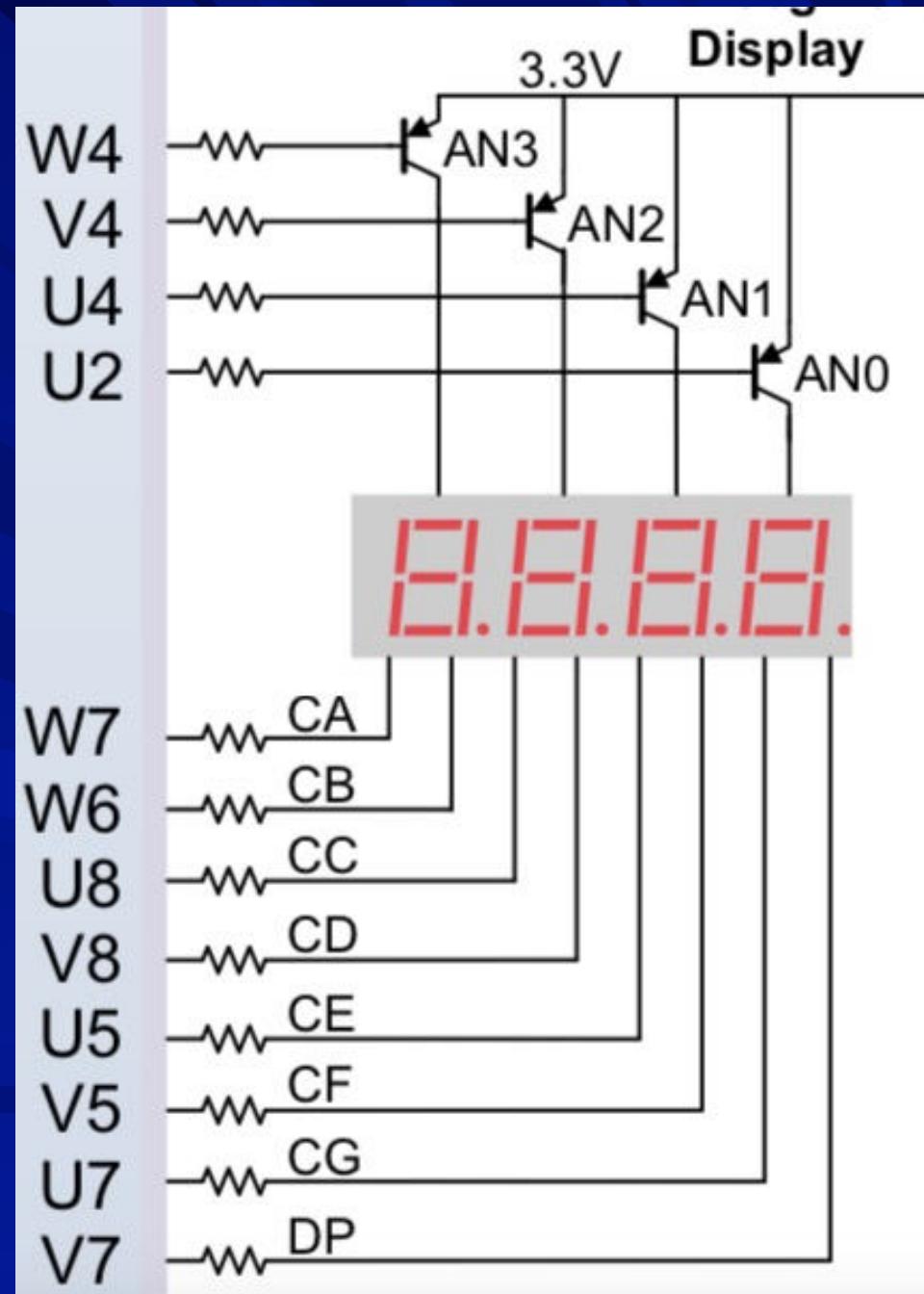
Switches and displays on BASYS3 board



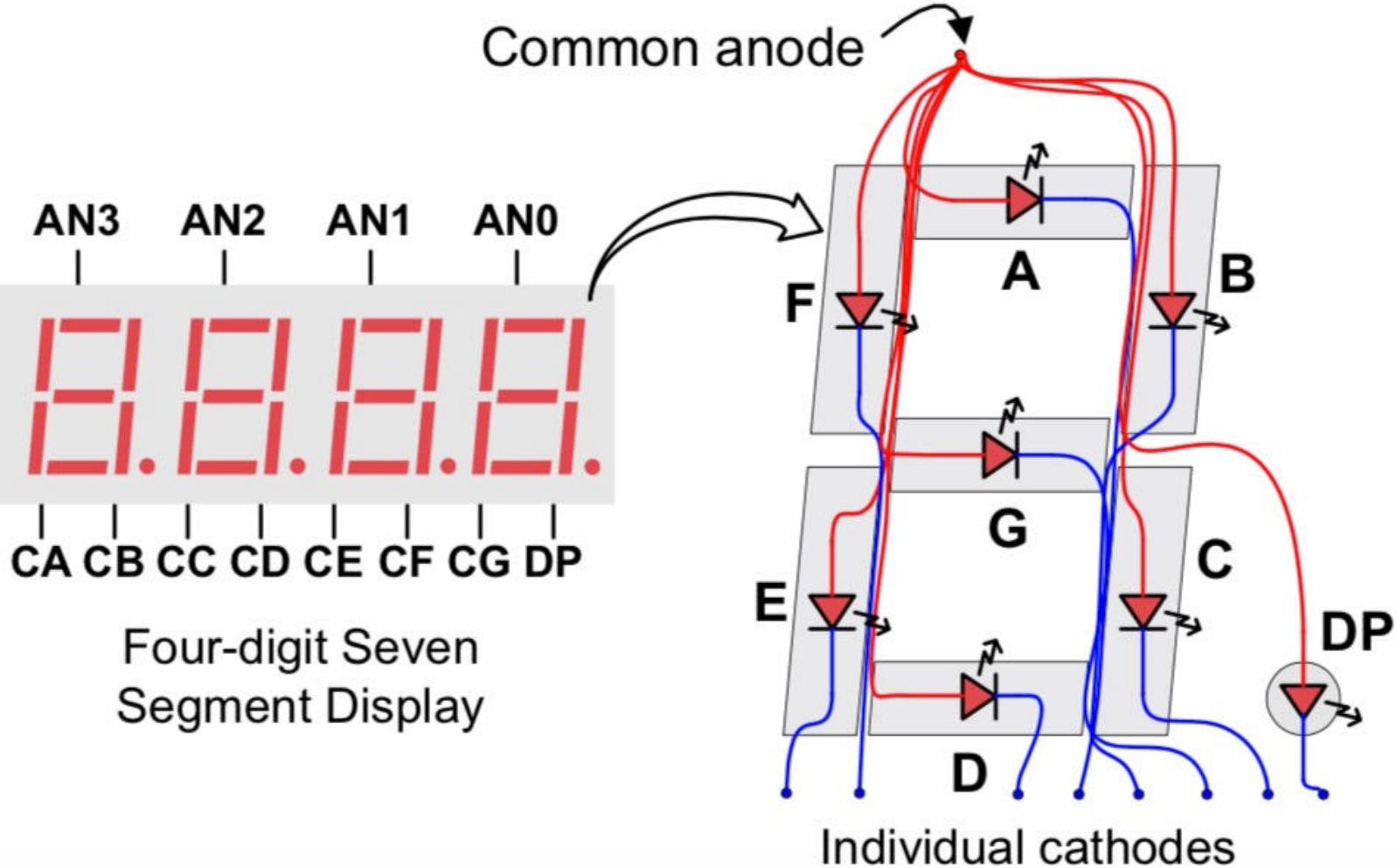
16 Button Key-pad



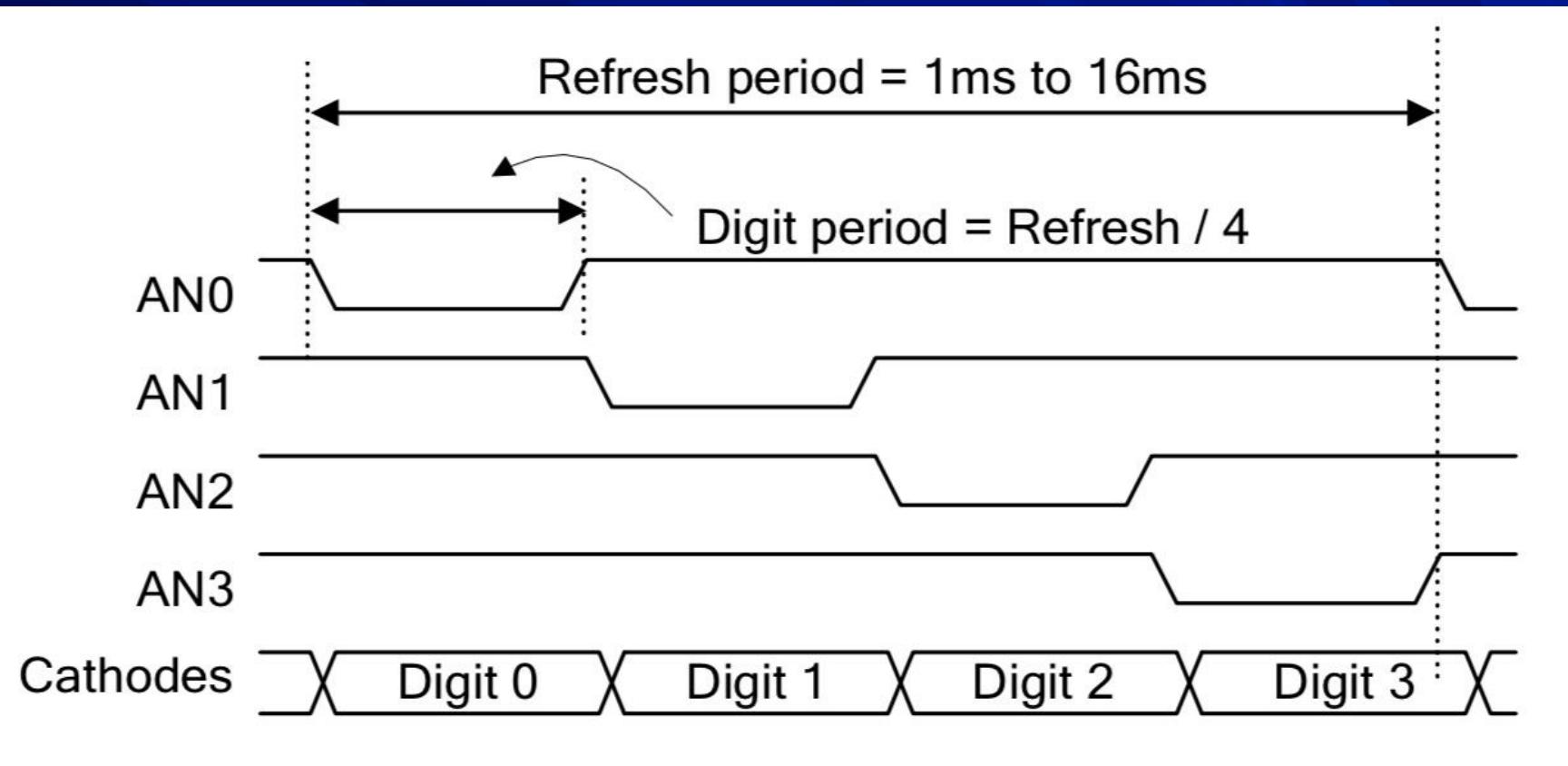
4-digit Display on BASYS 3 Board



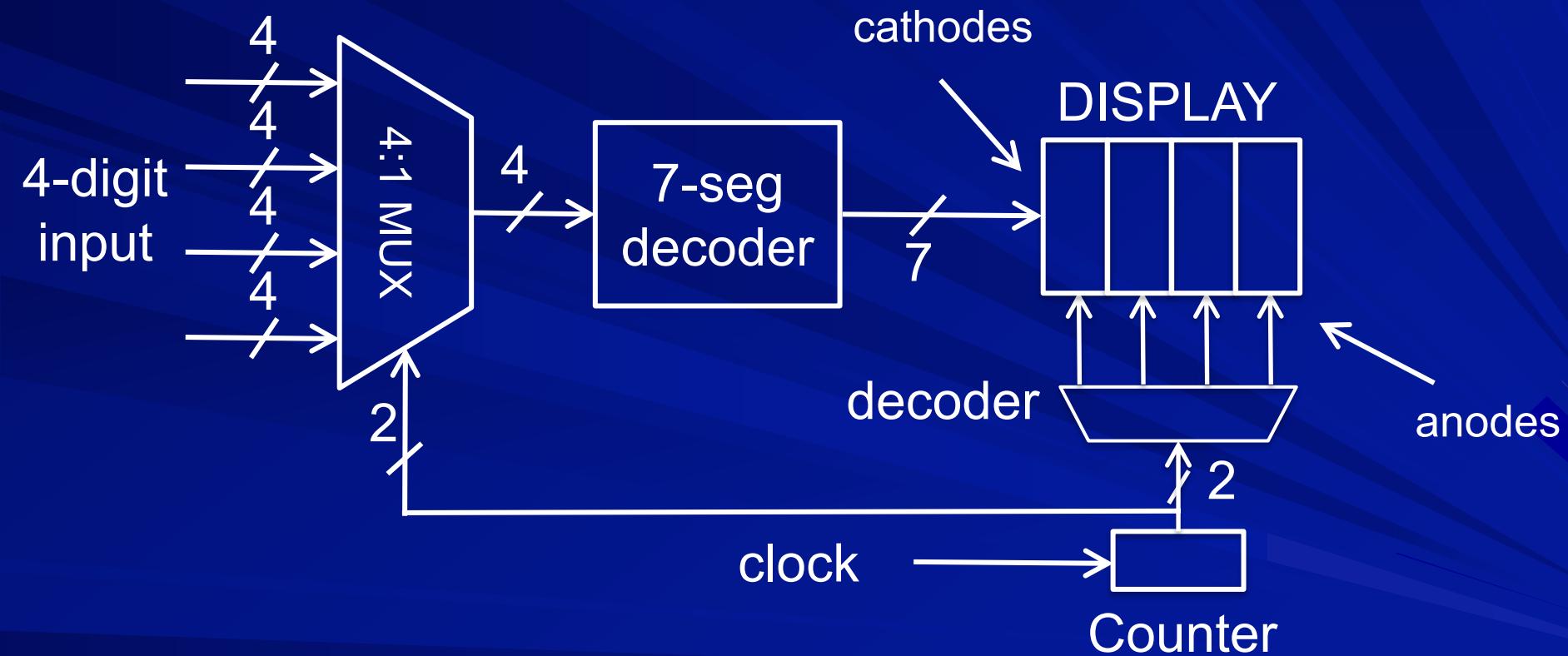
7-segment Display



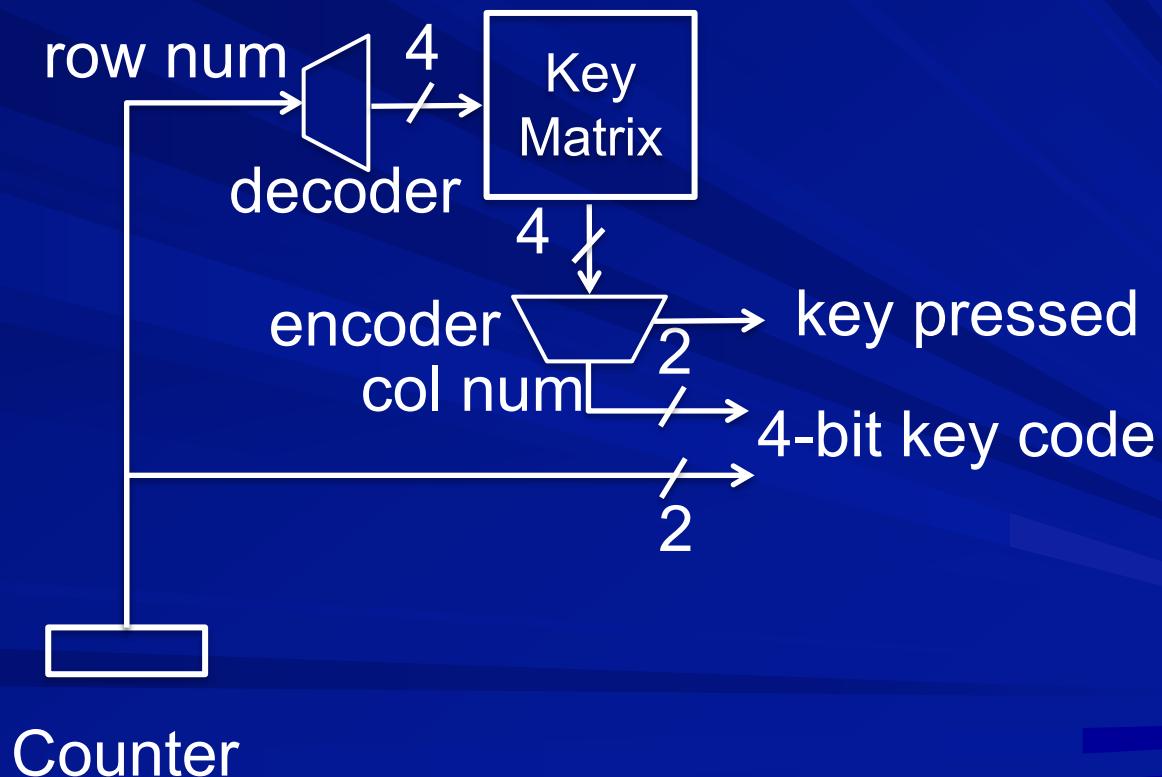
Refresh timings



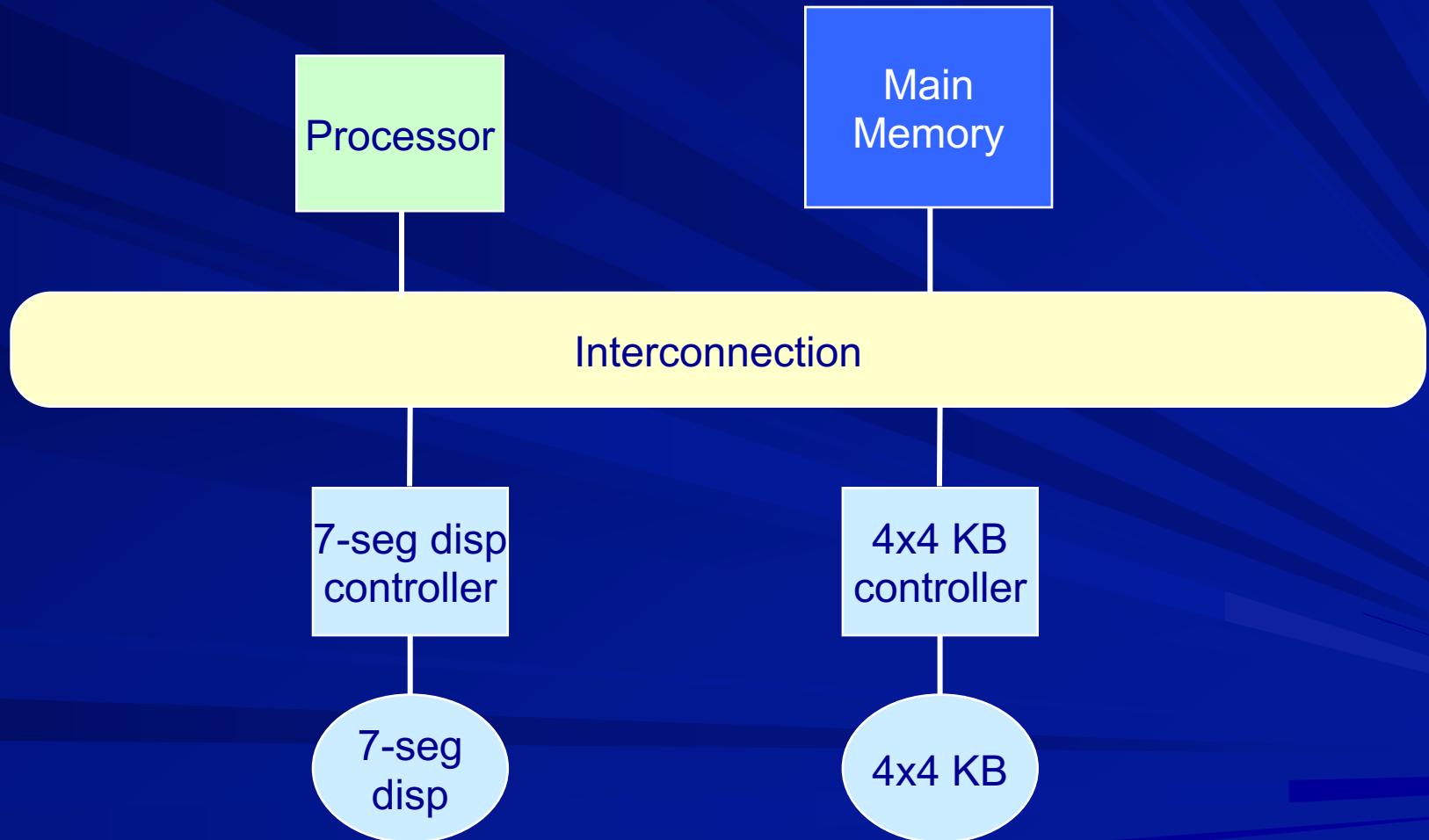
7-segment Display Controller



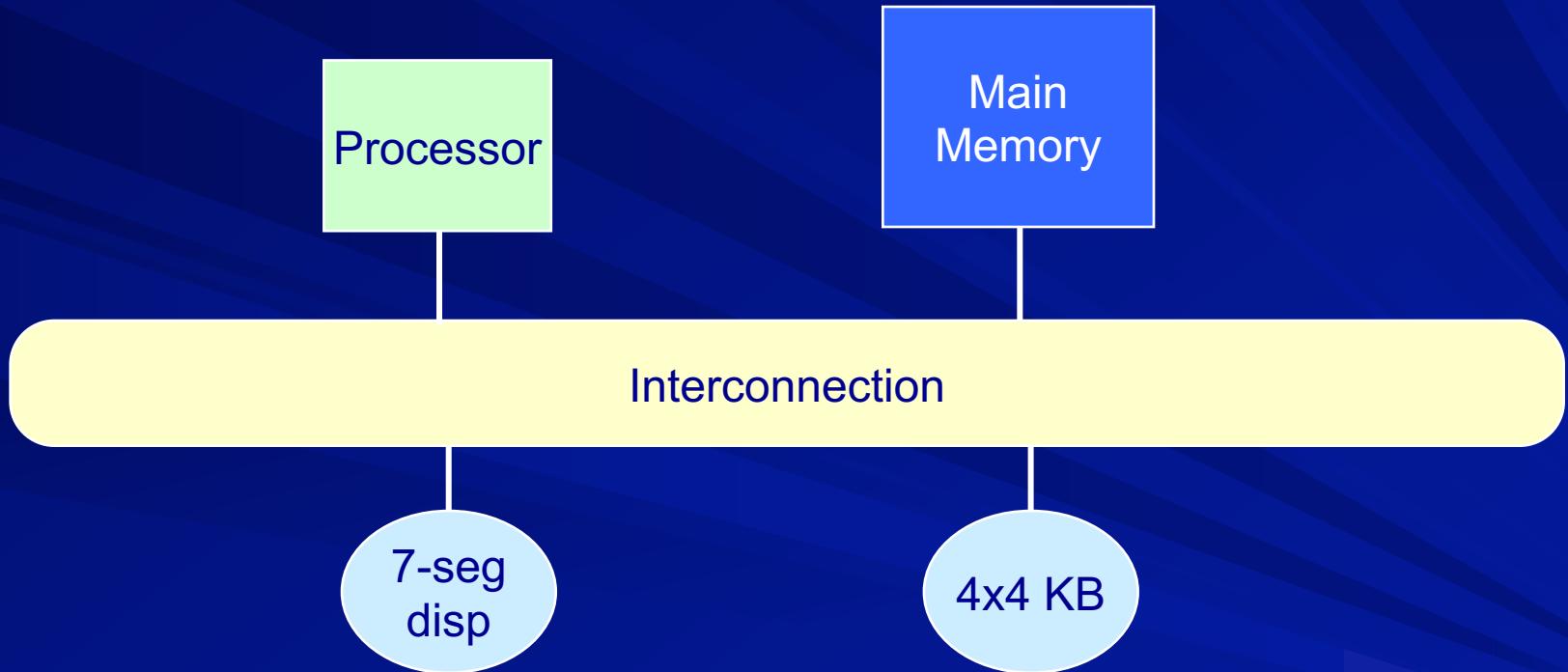
4 x 4 Keyboard Encoder



Block diagram



Block diagram



Low level control by software

Instructions for I/O

Use SWI instruction/system call

Require instruction for I/O inside handler

- Special input/output instructions

- specify device
 - specify register/memory location

=> Dedicated I/O

- Use LDR, STR instructions

- use devices as memory locations

=> Memory mapped I/O

I/O address space

■ Memory mapped I/O

- part of memory address space reserved for I/O
- usual load/store instructions are used for I/O
- memory ignores these addresses

■ Dedicated I/O

- separate address space for I/O - much smaller than the memory space
- control signal from processor indicates which space is being addressed
- separate instructions are required

Addressing I/O devices

Each I/O device (controller) viewed as a set of registers or ports by CPU

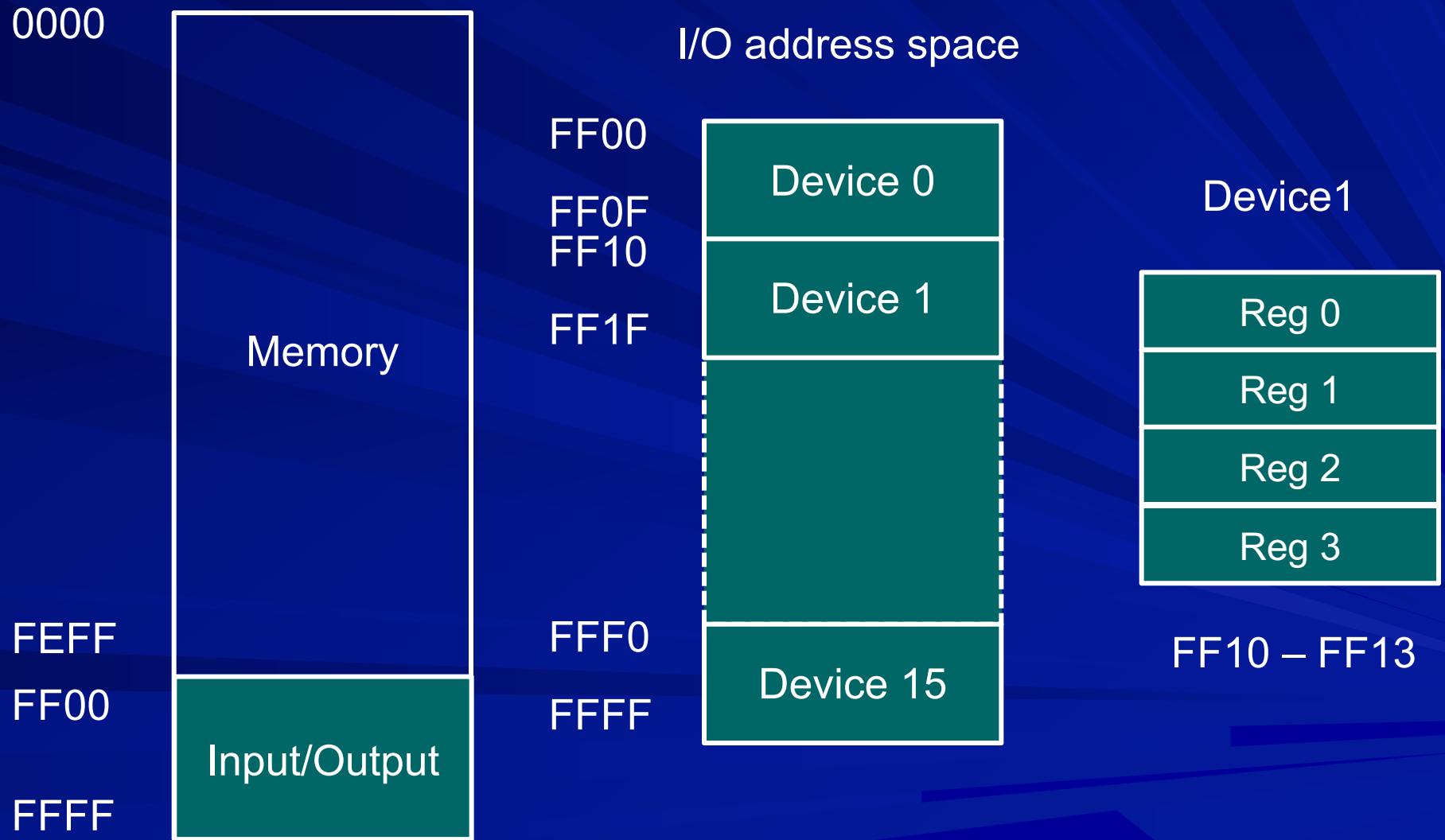
CPU can use these registers to

- write commands and parameters
- read status
- read/write data

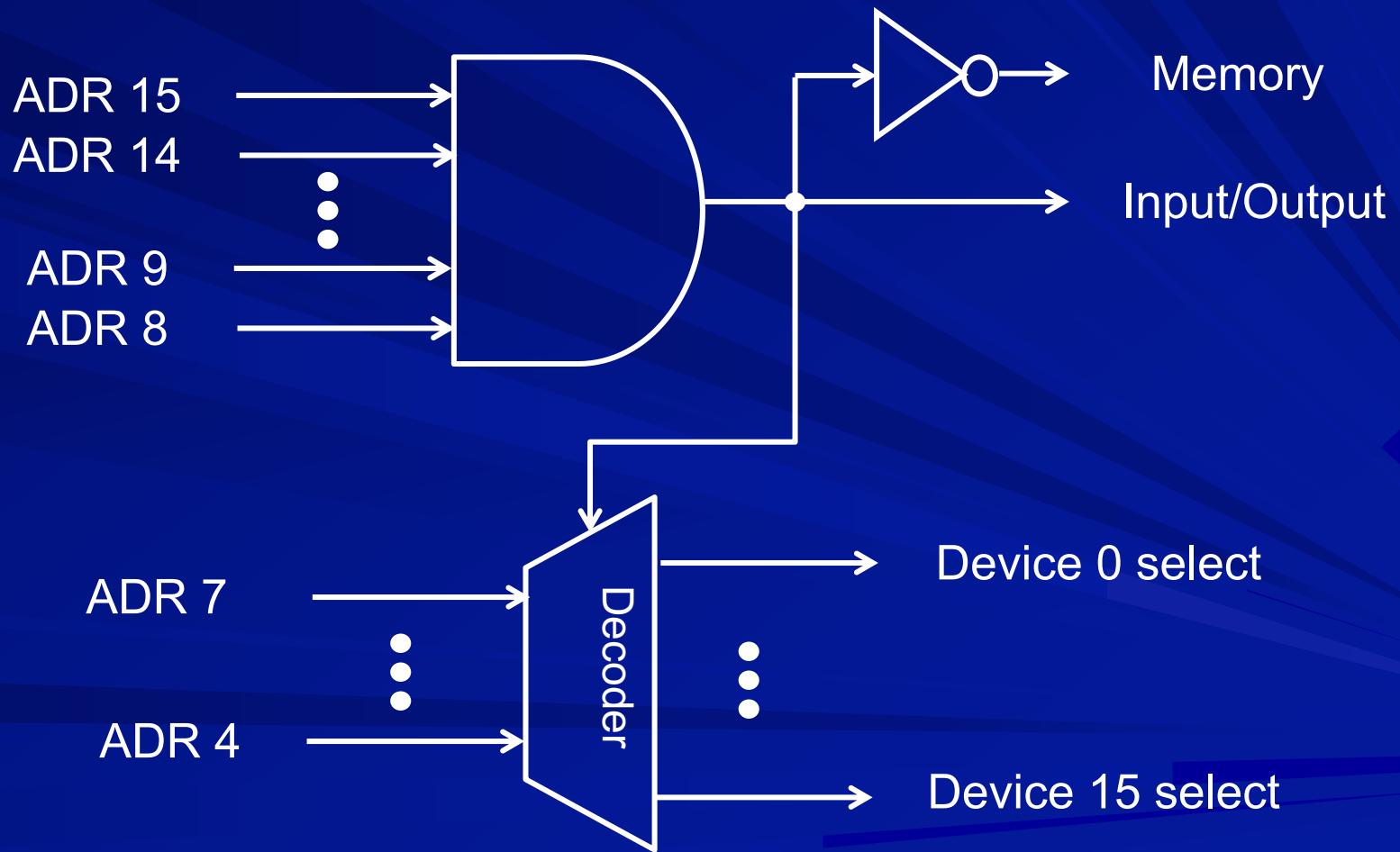
These registers are addressable

- Memory mapped I/O
- Dedicated I/O

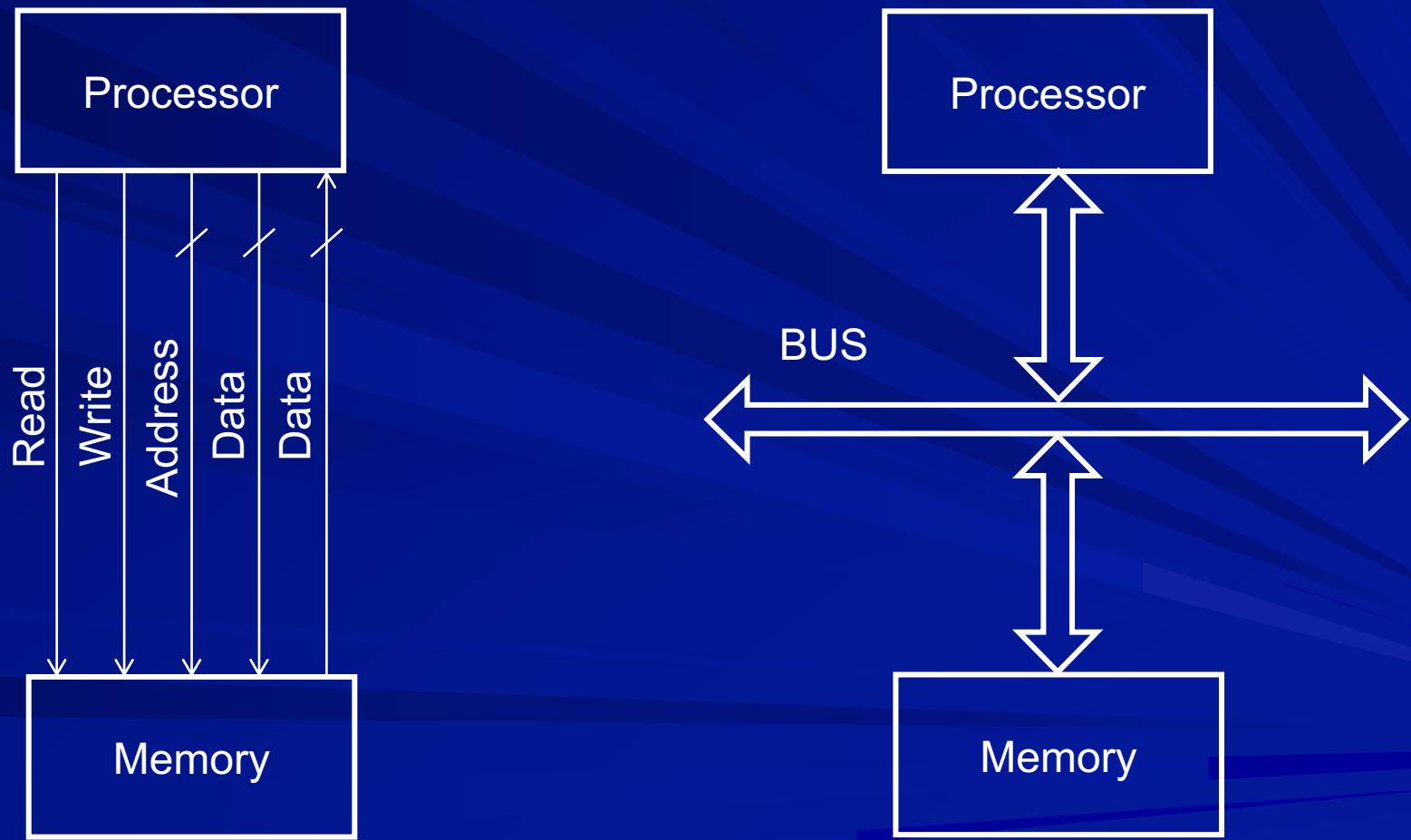
Address Map Example



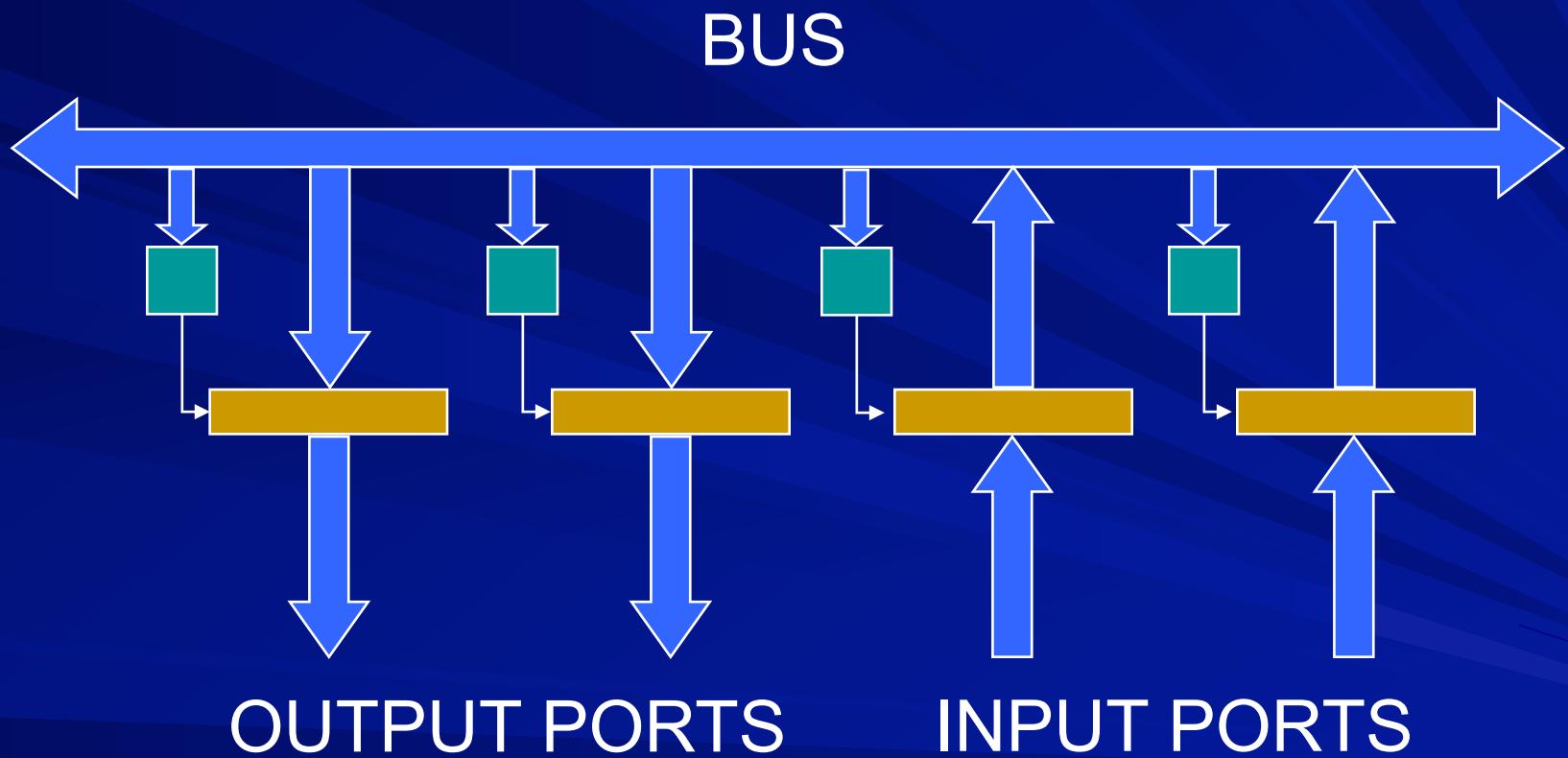
Decoding Address



Interconnection using a bus



INPUT / OUTPUT PORTS



Ports for Disp, KB controllers



Ports for raw Disp, KB

Display

out	anode pattern
out	cathode pattern

refresh required

Key Board

out	row pattern
in	column pattern

repeated scan required

COL216

Computer Architecture

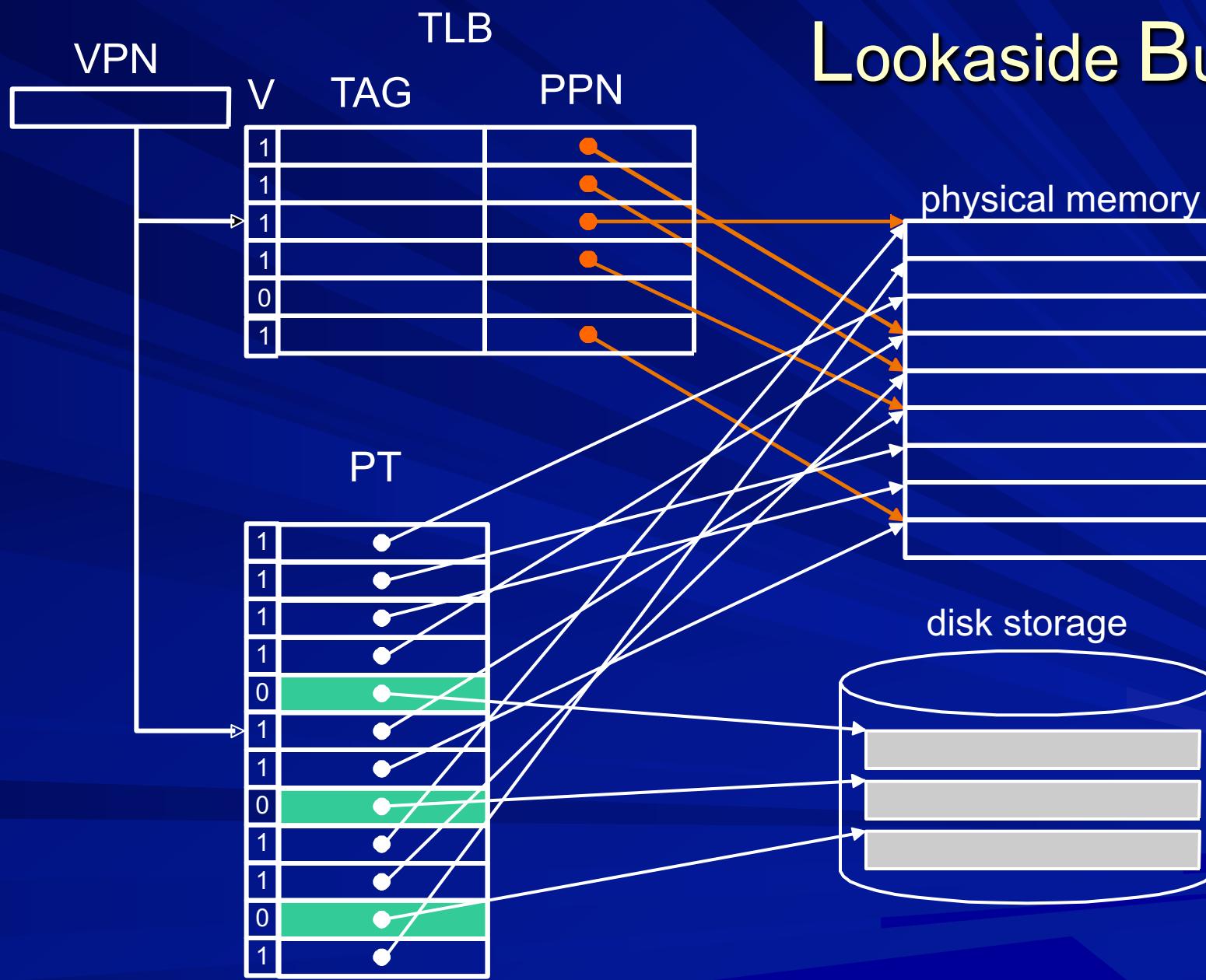
Virtual Memory
TLB illustration
14th March 2022

Virtual memory hit time

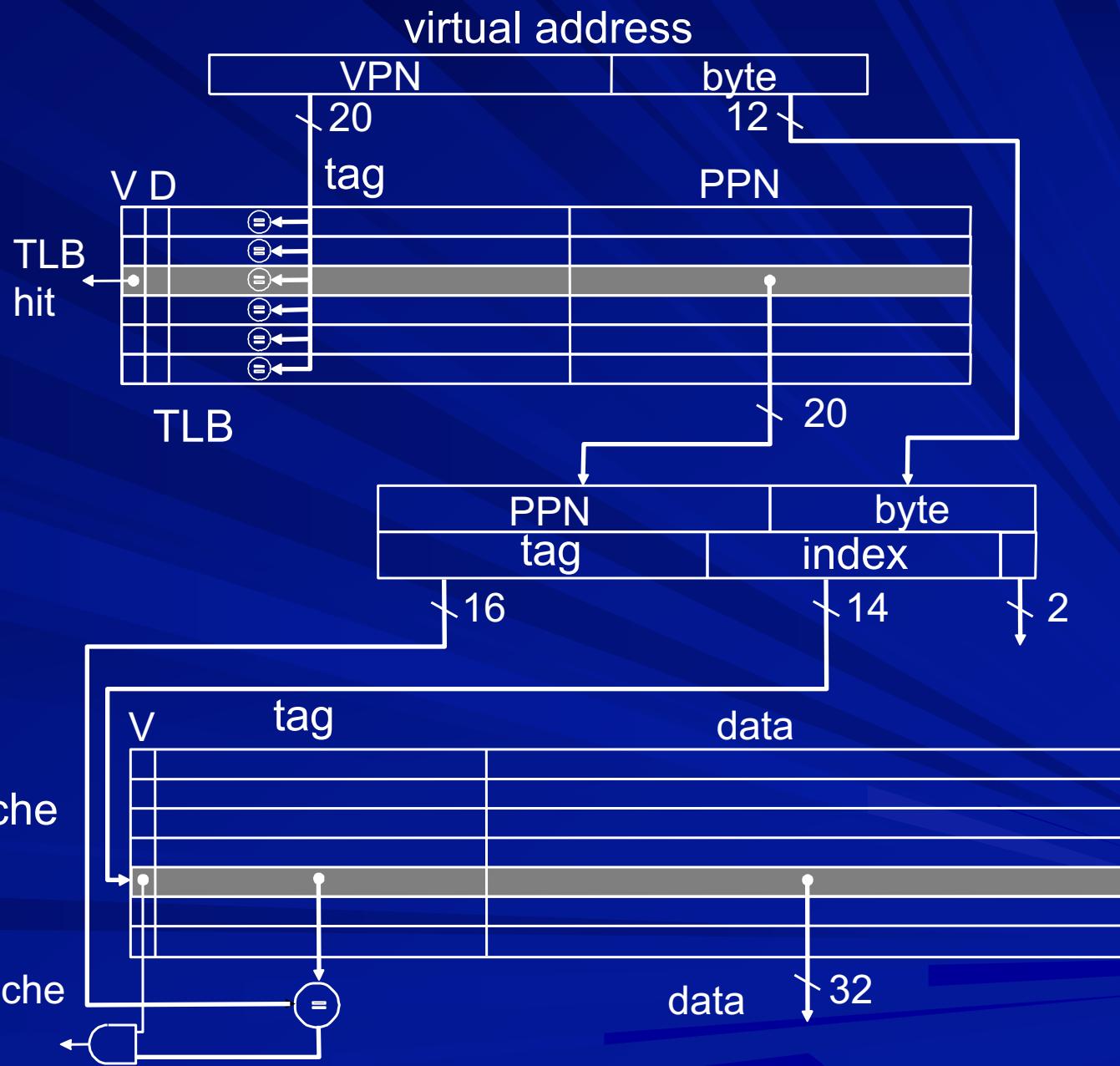
- simple page table :
 $2 \times$ physical memory access time
- 2 level page table :
 $3 \times$ physical memory access time
- paged page table :
 $3 \times$ physical memory access time

Can this be better??

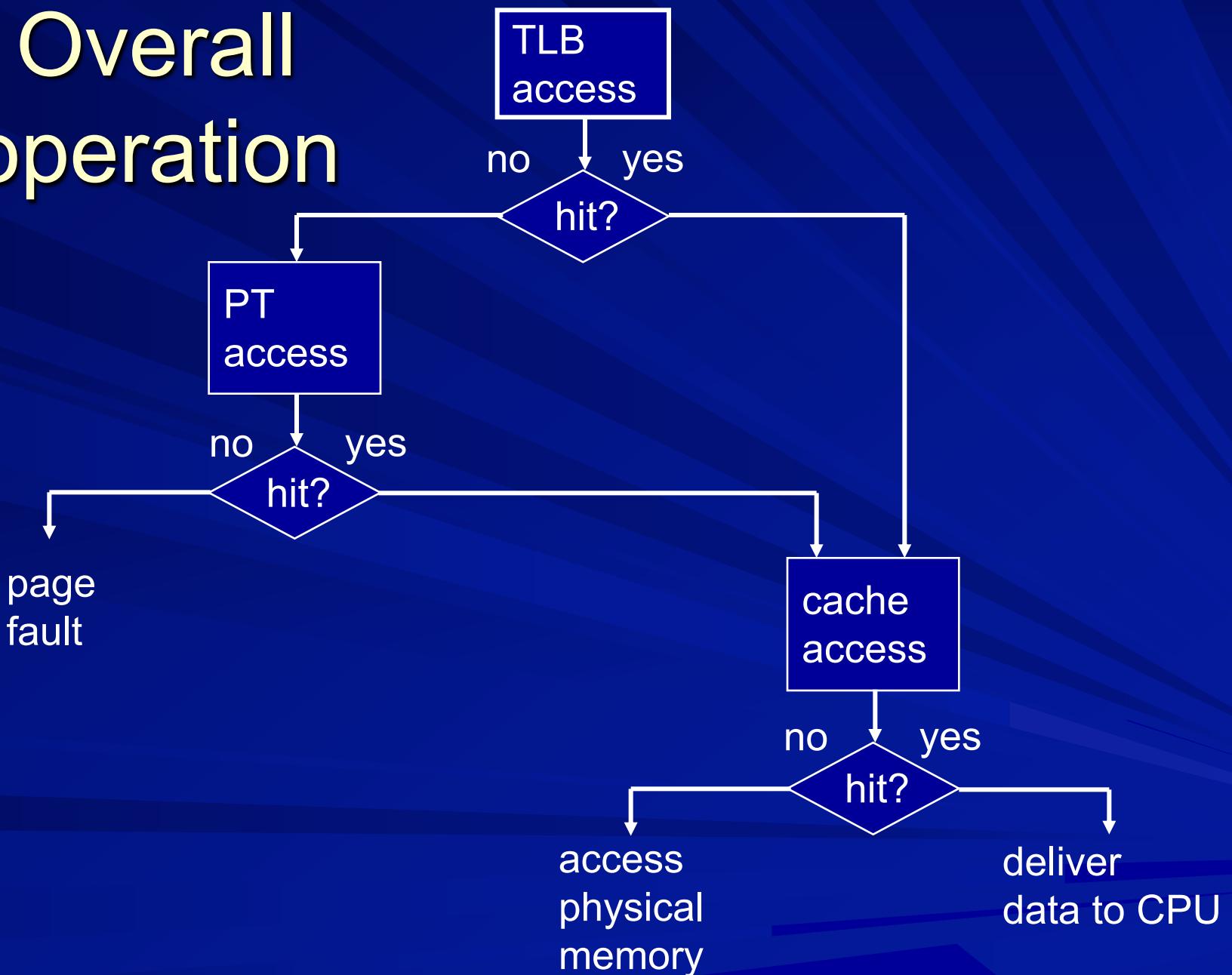
Translation Lookaside Buffer



TLB with Cache



Overall operation



Virtually addressed cache

- Tags in the cache are from virtual addresses
- Cache access is made first
- TLB accessed only after cache miss
- Problem of aliasing - two copies of a shared object in physical memory
- Physically tagged and virtually indexed cache may be used

Indexing before address translation

- Virtually indexed, physically tagged cache
 - simple and effective approach
 - indexing overlaps with address translation, tag matching after address translation
 - valid only if index field does not change during address translation (possible for caches that are not too large)

Virtual Page No.	Offset in page
Tag	Index

Example: TLB & PT operation (Ex 5.10)

PT

	V	PPN
0	1	5
1	0	Disk
2	0	Disk
3	1	6
4	1	9
5	1	11
6	0	Disk
7	1	4
8	0	Disk
9	0	Disk
10	1	3
11	1	12

TLB

	V	Tag(VPN)	PPN
	1	11	12
	1	7	4
	1	3	6
	0	4	9

fully associative

Page size = 4 KB

Example: TLB & PT operation (Ex 5.10)

PT

	V	PPN
0	1	5
1	0	Disk
2	0	Disk
3	1	6
4	1	9
5	1	11
6	0	Disk
7	1	4
8	0	Disk
9	0	Disk
10	1	3
11	1	12

TLB

	V	Tag(VPN)	PPN
	1	11	12
	1	7	4
	1	3	6
	0	4	9

fully associative

Page size = 4 KB

Virtual address stream:

4095, 31272, 15789, 15000, 7193, 4096, 8912

Example: TLB & PT operation (Ex 5.10)

PT

	V	PPN
0	1	5
1	0	Disk
2	0	Disk
3	1	6
4	1	9
5	1	11
6	0	Disk
7	1	4
8	0	Disk
9	0	Disk
10	1	3
11	1	12

TLB

	V	Tag(VPN)	PPN
	1	11	12
	1	7	4
	1	3	6
	0	4	9

fully associative

Page size = 4 KB

Vitual address stream:

4095, 31272, 15789, 15000, 7193, 4096, 8912

VPN:

0, 7, 3, 3, 1, 1, 2

Example: TLB & PT operation (Ex 5.10)

PT

	V	PPN
0	1	5
1	0	Disk
2	0	Disk
3	1	6
4	1	9
5	1	11
6	0	Disk
7	1	4
8	0	Disk
9	0	Disk
10	1	3
11	1	12

TLB

	V	Tag(VPN)	PPN
	1	11	12
	1	7	4
	1	3	6
	0	4	9

fully associative

Page size = 4 KB

Vitual address stream:

4095, 31272, 15789, 15000, 7193, 4096, 8912

VPN:

0, 7, 3, 3, 1, 1, 2

TLB hit/miss:

M

Page fault:

No

Example: TLB & PT operation (Ex 5.10)

PT

	V	PPN
0	1	5
1	0	Disk
2	0	Disk
3	1	6
4	1	9
5	1	11
6	0	Disk
7	1	4
8	0	Disk
9	0	Disk
10	1	3
11	1	12

TLB

V	Tag(VPN)	PPN
1	11	12
1	7	4
1	3	6
1	0	5

fully associative

Page size = 4 KB

Vitual address stream:

4095, 31272, 15789, 15000, 7193, 4096, 8912

VPN:

0, 7, 3, 3, 1, 1, 2

TLB hit/miss:

M

Page fault:

No

Example: TLB & PT operation (Ex 5.10)

PT

	V	PPN
0	1	5
1	0	Disk
2	0	Disk
3	1	6
4	1	9
5	1	11
6	0	Disk
7	1	4
8	0	Disk
9	0	Disk
10	1	3
11	1	12

TLB

V	Tag(VPN)	PPN
1	11	12
1	7	4
1	3	6
1	0	5

fully associative

Page size = 4 KB

Vitual address stream:

4095, 31272, 15789, 15000, 7193, 4096, 8912

VPN:

0, 7, 3, 3, 1, 1, 2

TLB hit/miss:

M H H H

Page fault:

No

Example: TLB & PT operation (Ex 5.10)

PT

	V	PPN
0	1	5
1	0	Disk
2	0	Disk
3	1	6
4	1	9
5	1	11
6	0	Disk
7	1	4
8	0	Disk
9	0	Disk
10	1	3
11	1	12

TLB

V	Tag(VPN)	PPN
1	11	12
1	7	4
1	3	6
1	0	5

fully associative

Page size = 4 KB

Vitual address stream:

4095, 31272, 15789, 15000, 7193, 4096, 8912

VPN:

0, 7, 3, 3, 1, 1, 2

TLB hit/miss:

M H H H M

Page fault:

No

Yes

Example: TLB & PT operation (Ex 5.10)

PT

	V	PPN
0	1	5
1	1	13
2	0	Disk
3	1	6
4	1	9
5	1	11
6	0	Disk
7	1	4
8	0	Disk
9	0	Disk
10	1	3
11	1	12

TLB

V	Tag(VPN)	PPN
1	11	12
1	7	4
1	3	6
1	0	5

fully associative

Page size = 4 KB

Vitual address stream:

4095, 31272, 15789, 15000, 7193, 4096, 8912

VPN:

0, 7, 3, 3, 1, 1, 2

TLB hit/miss:

M H H H M

Page fault:

No

Yes

Example: TLB & PT operation (Ex 5.10)

PT

	V	PPN
0	1	5
1	1	13
2	0	Disk
3	1	6
4	1	9
5	1	11
6	0	Disk
7	1	4
8	0	Disk
9	0	Disk
10	1	3
11	1	12

TLB

V	Tag(VPN)	PPN
1	1	13
1	7	4
1	3	6
1	0	5

fully associative

Page size = 4 KB

Vitual address stream:

4095, 31272, 15789, 15000, 7193, 4096, 8912

VPN:

0, 7, 3, 3, 1, 1, 2

TLB hit/miss:

M H H H M

Page fault:

No

Yes

Example: TLB & PT operation (Ex 5.10)

PT

	V	PPN
0	1	5
1	1	13
2	0	Disk
3	1	6
4	1	9
5	1	11
6	0	Disk
7	1	4
8	0	Disk
9	0	Disk
10	1	3
11	1	12

TLB

V	Tag(VPN)	PPN
1	1	13
1	7	4
1	3	6
1	0	5

fully associative

Page size = 4 KB

Vitual address stream:

4095, 31272, 15789, 15000, 7193, 4096, 8912

VPN:

0, 7, 3, 3, 1, 1, 2

TLB hit/miss:

M H H H M H M

Page fault:

No

Yes

Yes

Example: TLB & PT operation (Ex 5.10)

PT

	V	PPN
0	1	5
1	1	13
2	1	14
3	1	6
4	1	9
5	1	11
6	0	Disk
7	1	4
8	0	Disk
9	0	Disk
10	1	3
11	1	12

TLB

V	Tag(VPN)	PPN
1	1	13
1	7	4
1	3	6
1	0	5

fully associative

Page size = 4 KB

Vitual address stream:

4095, 31272, 15789, 15000, 7193, 4096, 8912

VPN:

0, 7, 3, 3, 1, 1, 2

TLB hit/miss:

M H H H M H M

Page fault:

No

Yes

Yes

Example: TLB & PT operation (Ex 5.10)

PT

	V	PPN
0	1	5
1	1	13
2	1	14
3	1	6
4	1	9
5	1	11
6	0	Disk
7	1	4
8	0	Disk
9	0	Disk
10	1	3
11	1	12

TLB

V	Tag(VPN)	PPN
1	1	13
1	7	4
1	3	6
1	2	14

fully associative

Page size = 4 KB

Vitual address stream:

4095, 31272, 15789, 15000, 7193, 4096, 8912

VPN:

0, 7, 3, 3, 1, 1, 2

TLB hit/miss:

M H H H M H M

Page fault:

No

Yes

Yes

Example: TLB & PT operation (Ex 5.10)

PT

	V	PPN
0	1	5
1	0	Disk
2	0	Disk
3	1	6
4	1	9
5	1	11
6	0	Disk
7	1	4
8	0	Disk
9	0	Disk
10	1	3
11	1	12

TLB

	V	Tag(VPN)	PPN
	1	11	12
	1	7	4
	1	10	3
	0	4	9

2-way set associative

Page size = 4 KB

Example: TLB & PT operation (Ex 5.10)

PT

	V	PPN
0	1	5
1	0	Disk
2	0	Disk
3	1	6
4	1	9
5	1	11
6	0	Disk
7	1	4
8	0	Disk
9	0	Disk
10	1	3
11	1	12

TLB

V	Tag(VPN)	PPN
1	11	12
1	7	4
1	10	3
0	4	9

2-way set associative

Page size = 4 KB

Vitual address stream:

4095, 31272, 15789, 15000, 7193, 4096, 8912

VPN:

0, 7, 3, 3, 1, 1, 2

Example: TLB & PT operation (Ex 5.10)

PT

	V	PPN
0	1	5
1	0	Disk
2	0	Disk
3	1	6
4	1	9
5	1	11
6	0	Disk
7	1	4
8	0	Disk
9	0	Disk
10	1	3
11	1	12

TLB

V	Tag(VPN)	PPN
1	11	12
1	7	4
1	10	3
0	4	9

2-way set associative

Page size = 4 KB

Vitual address stream:

4095, 31272, 15789, 15000, 7193, 4096, 8912

VPN:

0, 7, 3, 3, 1, 1, 2

TLB hit/miss:

M

Page fault:

No

Example: TLB & PT operation (Ex 5.10)

PT

	V	PPN
0	1	5
1	0	Disk
2	0	Disk
3	1	6
4	1	9
5	1	11
6	0	Disk
7	1	4
8	0	Disk
9	0	Disk
10	1	3
11	1	12

TLB

V	Tag(VPN)	PPN
1	11	12
1	7	4
1	10	3
1	0	5

2-way set associative

Page size = 4 KB

Vitual address stream:

4095, 31272, 15789, 15000, 7193, 4096, 8912

VPN:

0, 7, 3, 3, 1, 1, 2

TLB hit/miss:

M

Page fault:

No

Example: TLB & PT operation (Ex 5.10)

PT

	V	PPN
0	1	5
1	0	Disk
2	0	Disk
3	1	6
4	1	9
5	1	11
6	0	Disk
7	1	4
8	0	Disk
9	0	Disk
10	1	3
11	1	12

TLB

V	Tag(VPN)	PPN
1	11	12
1	7	4
1	10	3
1	0	5

2-way set associative

Page size = 4 KB

Vitual address stream:

4095, 31272, 15789, 15000, 7193, 4096, 8912

VPN:

0, 7, 3, 3, 1, 1, 2

TLB hit/miss:

M H M

Page fault:

No No

Example: TLB & PT operation (Ex 5.10)

PT

	V	PPN
0	1	5
1	0	Disk
2	0	Disk
3	1	6
4	1	9
5	1	11
6	0	Disk
7	1	4
8	0	Disk
9	0	Disk
10	1	3
11	1	12

TLB

V	Tag(VPN)	PPN
1	3	6
1	7	4
1	10	3
1	0	5

2-way set associative

Page size = 4 KB

Vitual address stream:

4095, 31272, 15789, 15000, 7193, 4096, 8912

VPN:

0, 7, 3, 3, 1, 1, 2

TLB hit/miss:

M H M

Page fault:

No No

Example: TLB & PT operation (Ex 5.10)

PT

	V	PPN
0	1	5
1	0	Disk
2	0	Disk
3	1	6
4	1	9
5	1	11
6	0	Disk
7	1	4
8	0	Disk
9	0	Disk
10	1	3
11	1	12

TLB

V	Tag(VPN)	PPN
1	3	6
1	7	4
1	10	3
1	0	5

2-way set associative

Page size = 4 KB

Vitual address stream:

4095, 31272, 15789, 15000, 7193, 4096, 8912

VPN:

0, 7, 3, 3, 1, 1, 2

TLB hit/miss:

M H M H M

Page fault:

No No

Yes

Example: TLB & PT operation (Ex 5.10)

PT

	V	PPN
0	1	5
1	1	13
2	0	Disk
3	1	6
4	1	9
5	1	11
6	0	Disk
7	1	4
8	0	Disk
9	0	Disk
10	1	3
11	1	12

TLB

V	Tag(VPN)	PPN
1	3	6
1	7	4
1	10	3
1	0	5

2-way set associative

Page size = 4 KB

Vitual address stream:

4095, 31272, 15789, 15000, 7193, 4096, 8912

VPN:

0, 7, 3, 3, 1, 1, 2

TLB hit/miss:

M H M H M

Page fault:

No No

Yes

Example: TLB & PT operation (Ex 5.10)

PT

	V	PPN
0	1	5
1	1	13
2	0	Disk
3	1	6
4	1	9
5	1	11
6	0	Disk
7	1	4
8	0	Disk
9	0	Disk
10	1	3
11	1	12

TLB

V	Tag(VPN)	PPN
1	3	6
1	1	13
1	10	3
1	0	5

2-way set associative

Page size = 4 KB

Vitual address stream:

4095, 31272, 15789, 15000, 7193, 4096, 8912

VPN:

0, 7, 3, 3, 1, 1, 2

TLB hit/miss:

M H M H M

Page fault:

No No Yes

Example: TLB & PT operation (Ex 5.10)

PT

	V	PPN
0	1	5
1	1	13
2	0	Disk
3	1	6
4	1	9
5	1	11
6	0	Disk
7	1	4
8	0	Disk
9	0	Disk
10	1	3
11	1	12

TLB

V	Tag(VPN)	PPN
1	3	6
1	1	13
1	10	3
1	0	5

2-way set associative

Page size = 4 KB

Vitual address stream:

4095, 31272, 15789, 15000, 7193, 4096, 8912

VPN:

0, 7, 3, 3, 1, 1, 2

TLB hit/miss:

M H M H M H M

Page fault:

No No Yes Yes

Example: TLB & PT operation (Ex 5.10)

PT

	V	PPN
0	1	5
1	1	13
2	1	14
3	1	6
4	1	9
5	1	11
6	0	Disk
7	1	4
8	0	Disk
9	0	Disk
10	1	3
11	1	12

TLB

V	Tag(VPN)	PPN
1	3	6
1	1	13
1	10	3
1	0	5

2-way set associative

Page size = 4 KB

Vitual address stream:

4095, 31272, 15789, 15000, 7193, 4096, 8912

VPN:

0, 7, 3, 3, 1, 1, 2

TLB hit/miss:

M H M H M H M

Page fault:

No No Yes Yes

Example: TLB & PT operation (Ex 5.10)

PT

	V	PPN
0	1	5
1	1	13
2	1	14
3	1	6
4	1	9
5	1	11
6	0	Disk
7	1	4
8	0	Disk
9	0	Disk
10	1	3
11	1	12

TLB

V	Tag(VPN)	PPN
1	3	6
1	1	13
1	2	14
1	0	5

2-way set associative

Page size = 4 KB

Vitual address stream:

4095, 31272, 15789, 15000, 7193, 4096, 8912

VPN:

0, 7, 3, 3, 1, 1, 2

TLB hit/miss:

M H M H M H M

Page fault:

No No Yes Yes

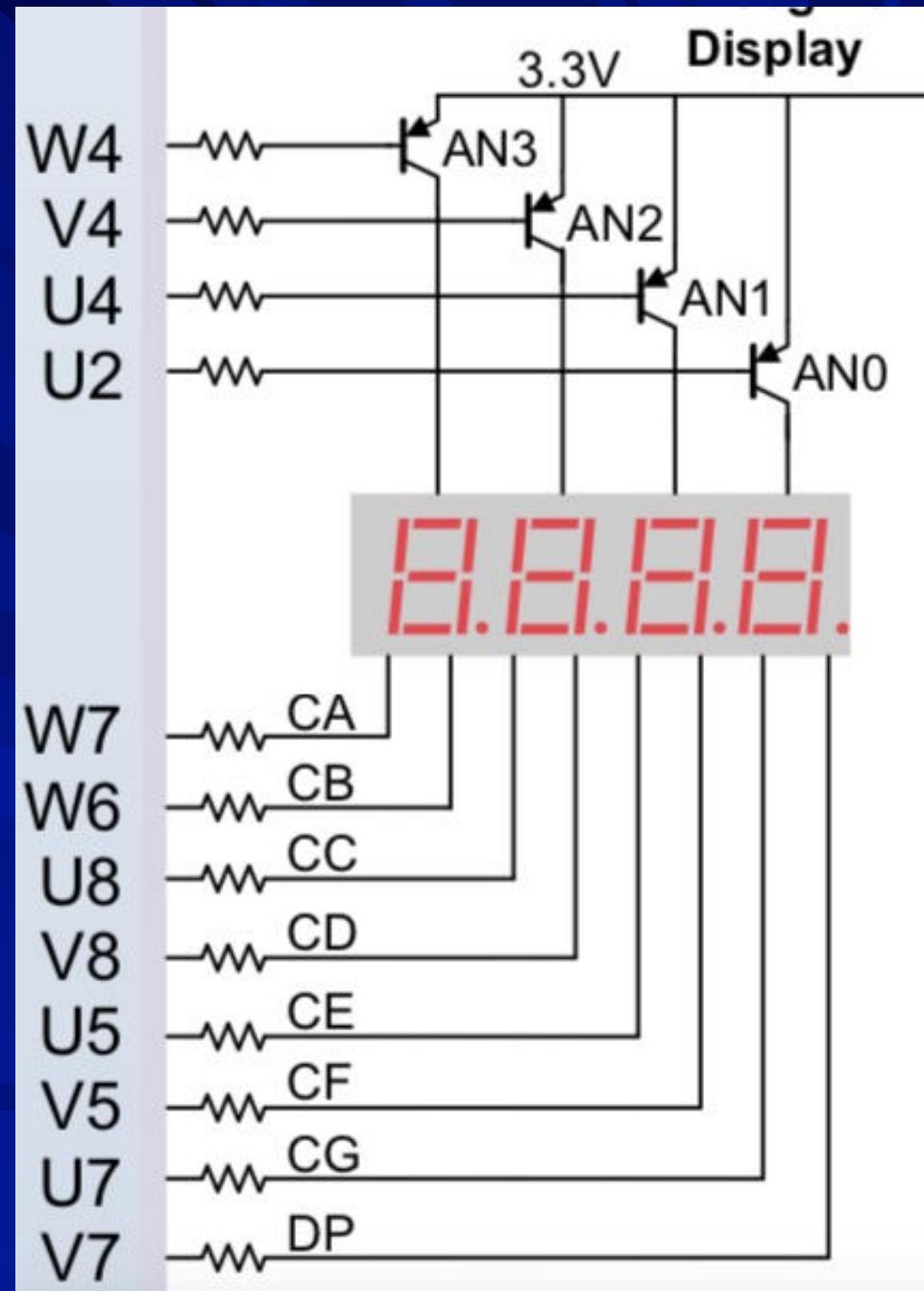
COL216

Computer Architecture

Input/Output – 2

17th March 2022

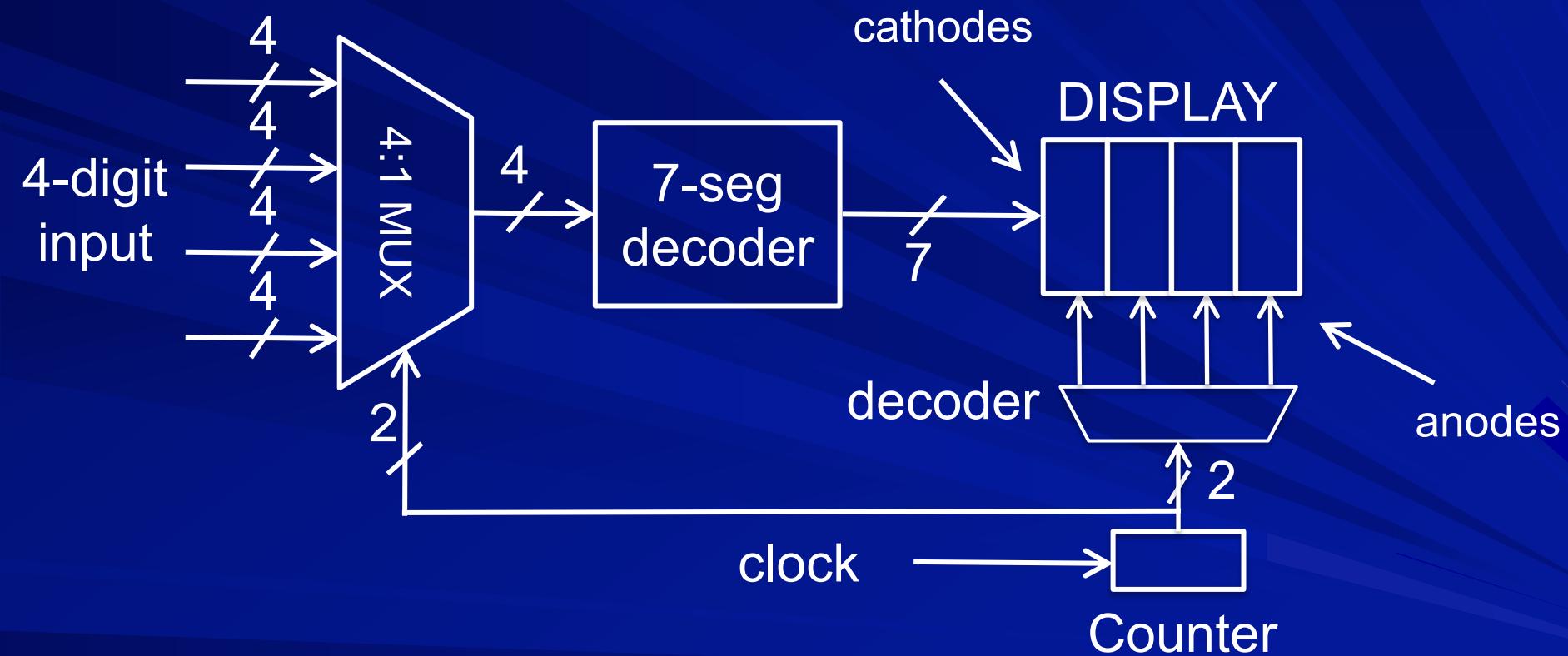
4-digit Display on BASYS 3 Board



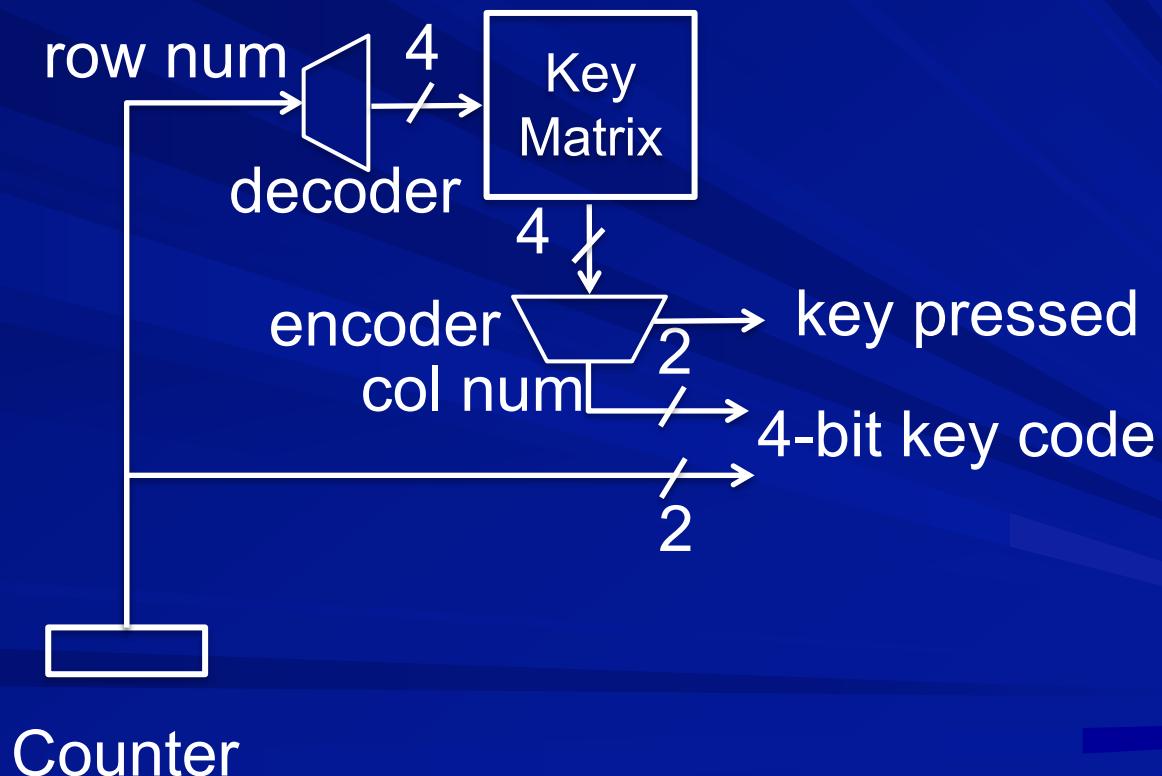
16 Button Key-pad



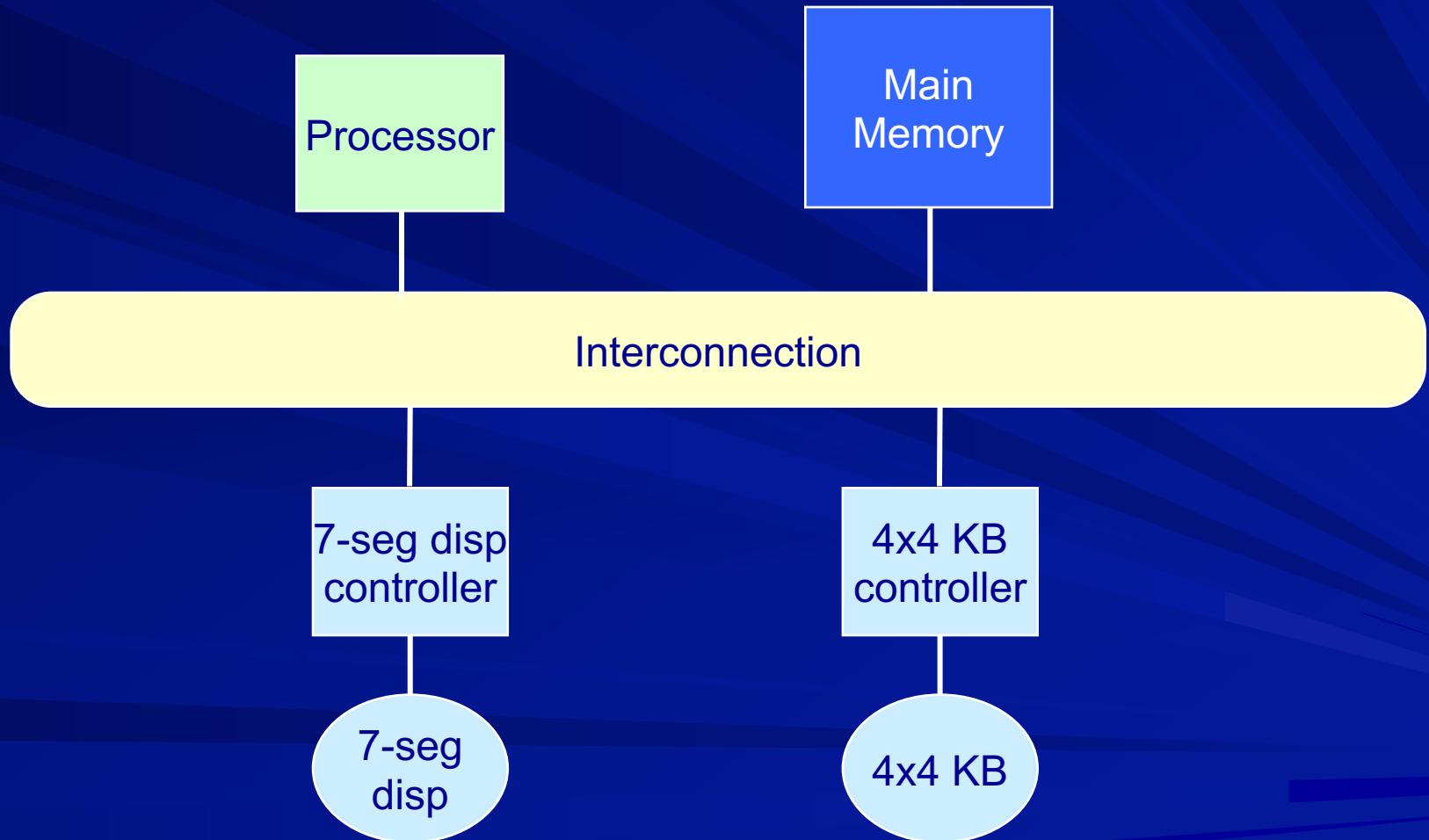
7-segment Display Controller



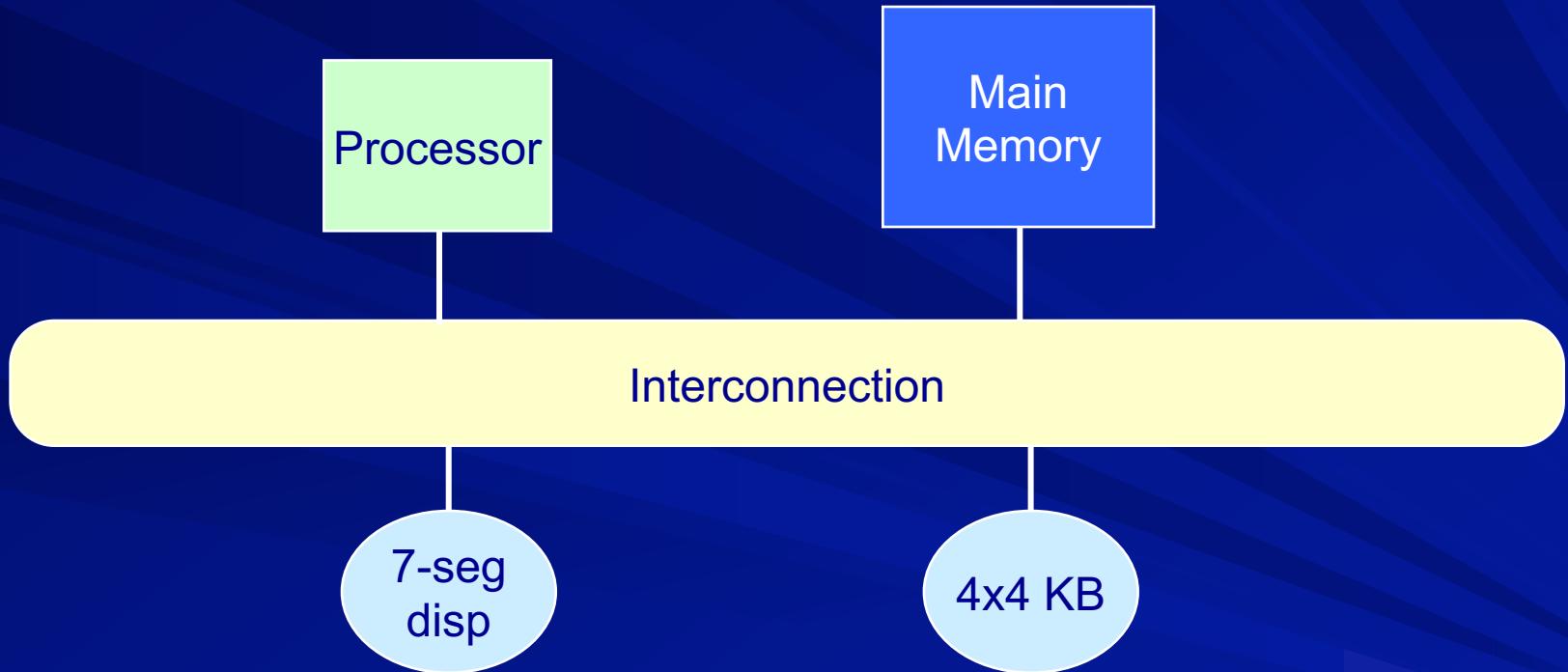
4 x 4 Keyboard Encoder



Block diagram

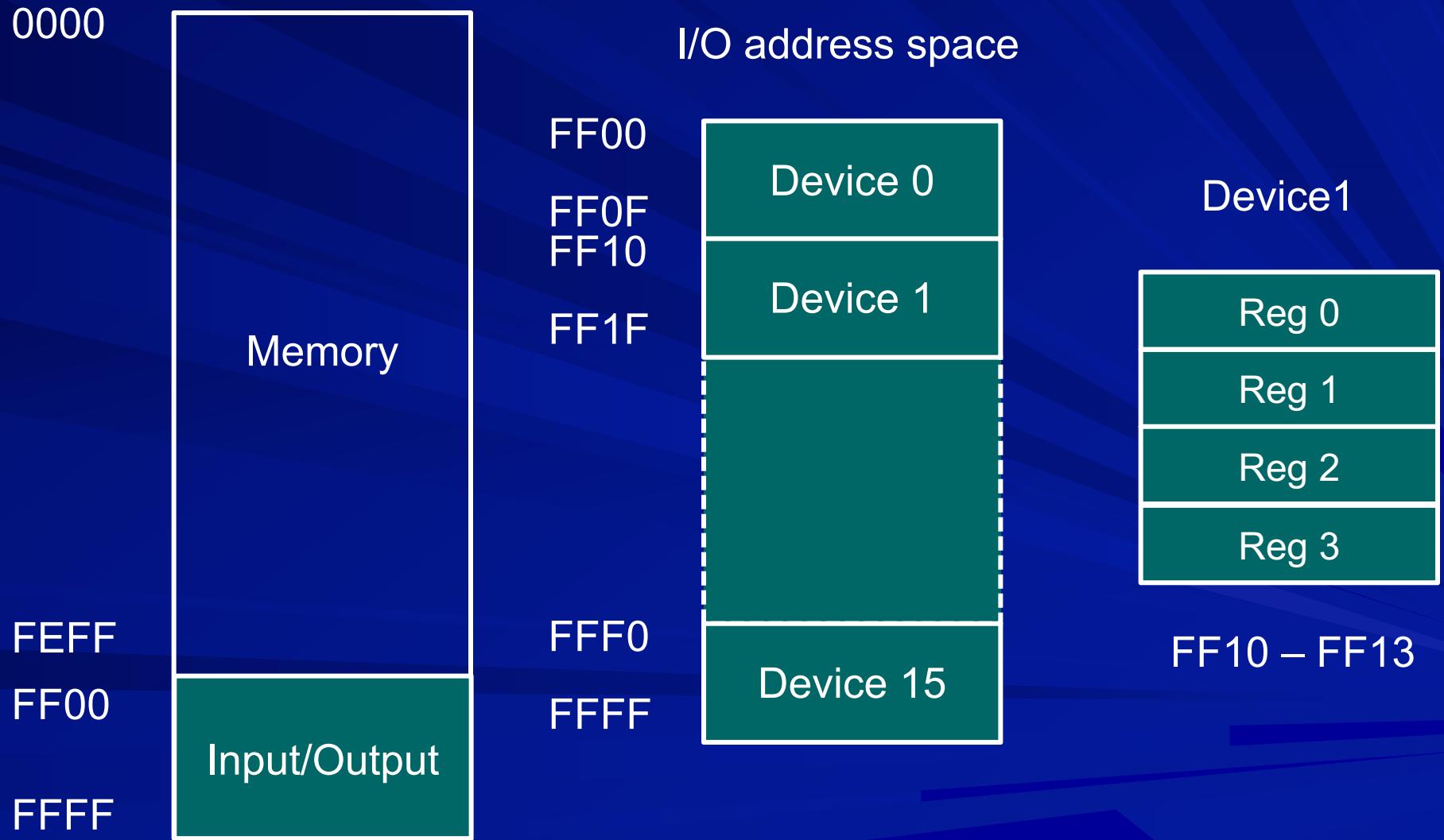


Block diagram

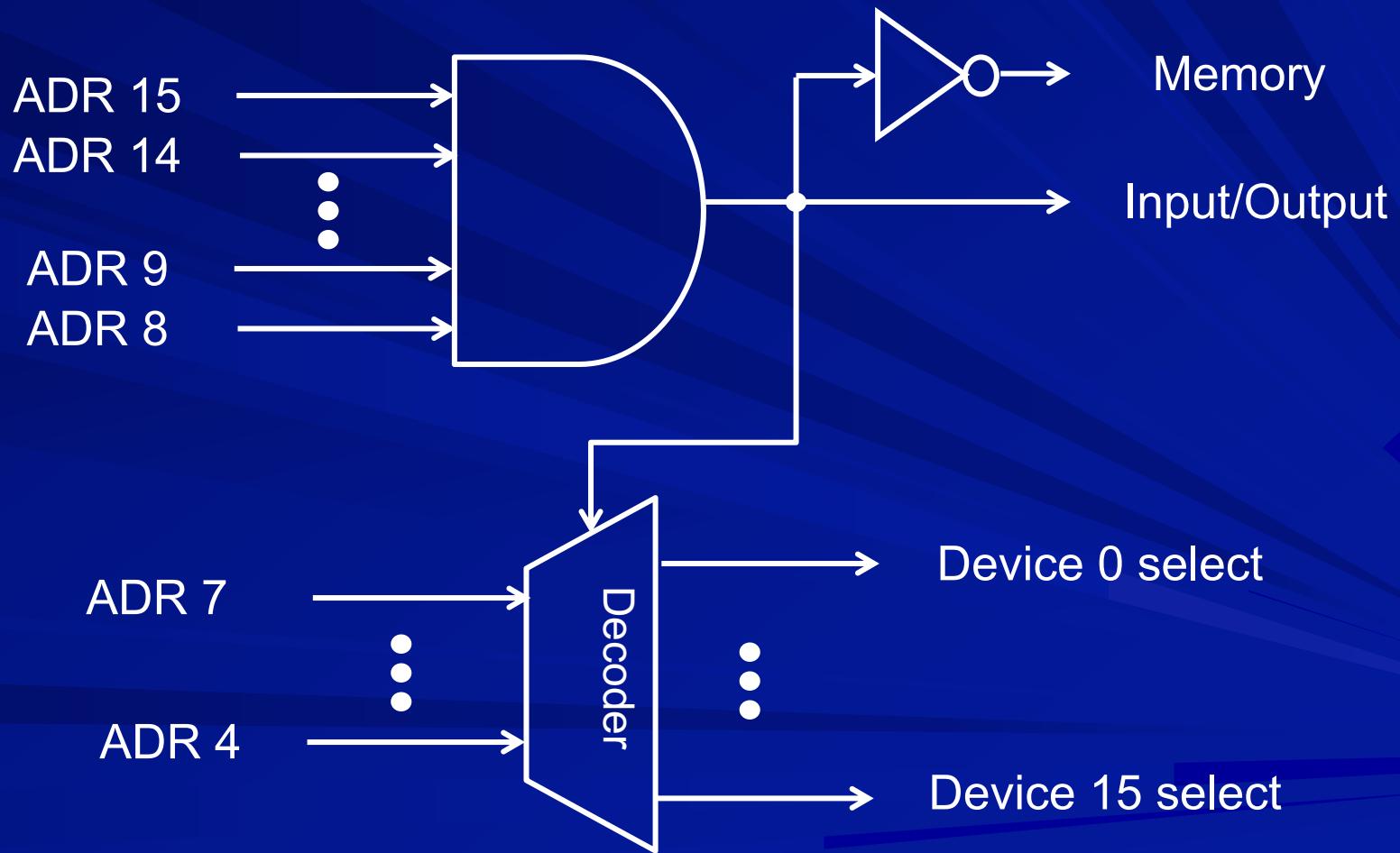


Low level control by software

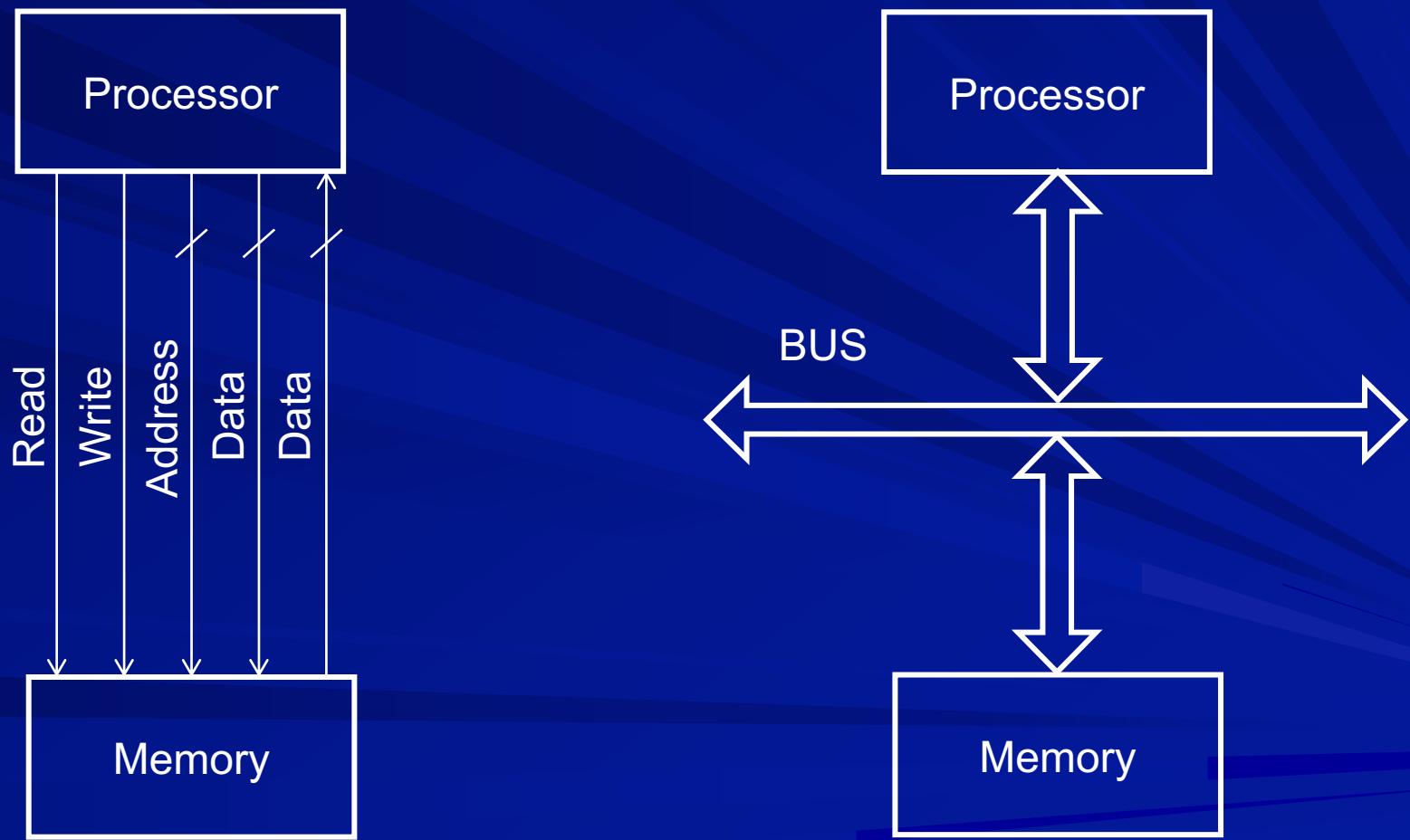
Address Map Example



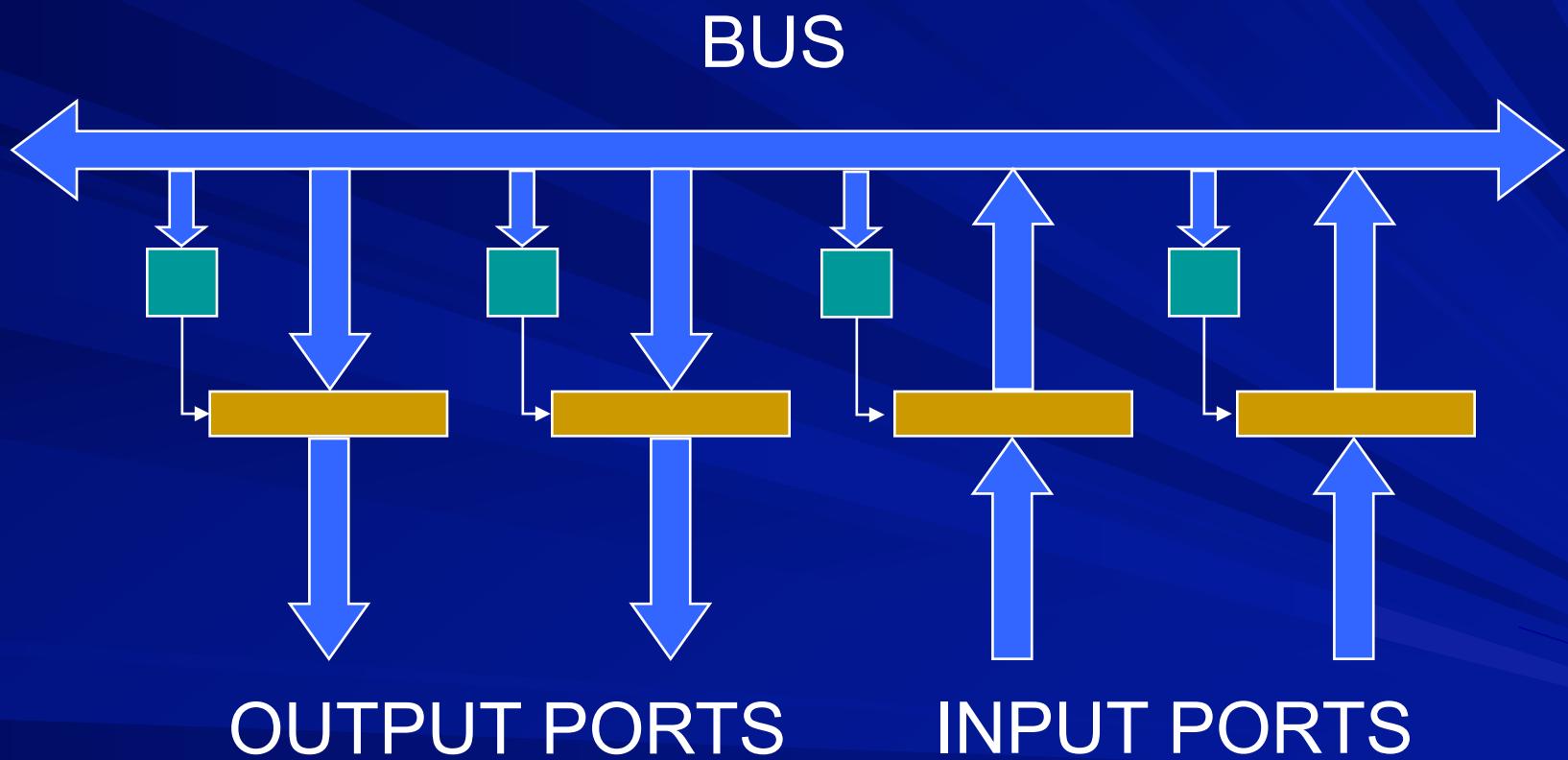
Decoding Address



Interconnection using a bus



INPUT / OUTPUT PORTS



Ports for Disp, KB controllers



Ports for raw Disp, KB

Display

out	anode pattern
out	cathode pattern

refresh required

Key Board

out	row pattern
in	column pattern

repeated scan required

Taking care of device timings

Devices may have latency of hundreds or thousands of cycles.

Can the instruction execution be stretched for so long?

How CPU checks the device status

■ Polling

- periodically read the status register and figure out whether the device is ready or not
- should be frequent enough so that no events are missed out

■ Interrupt

- keep busy with some other task and allow the device to intimate its readiness by sending a signal

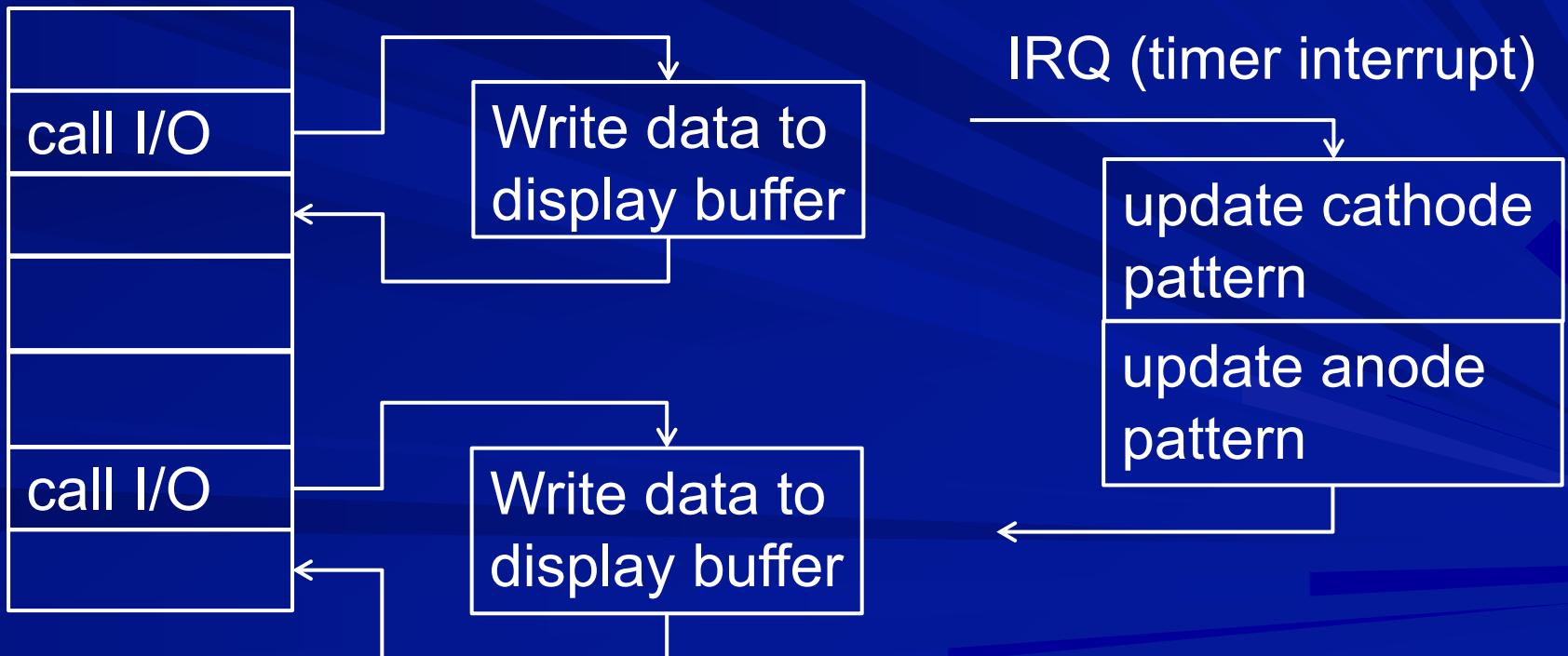
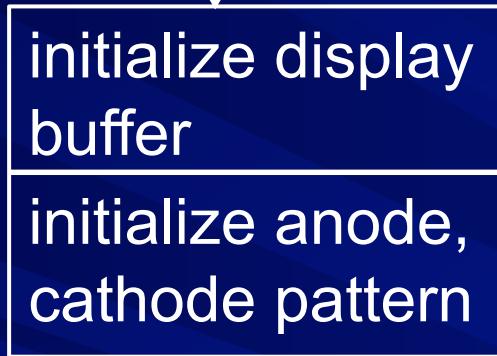
Polling vs interrupt

status

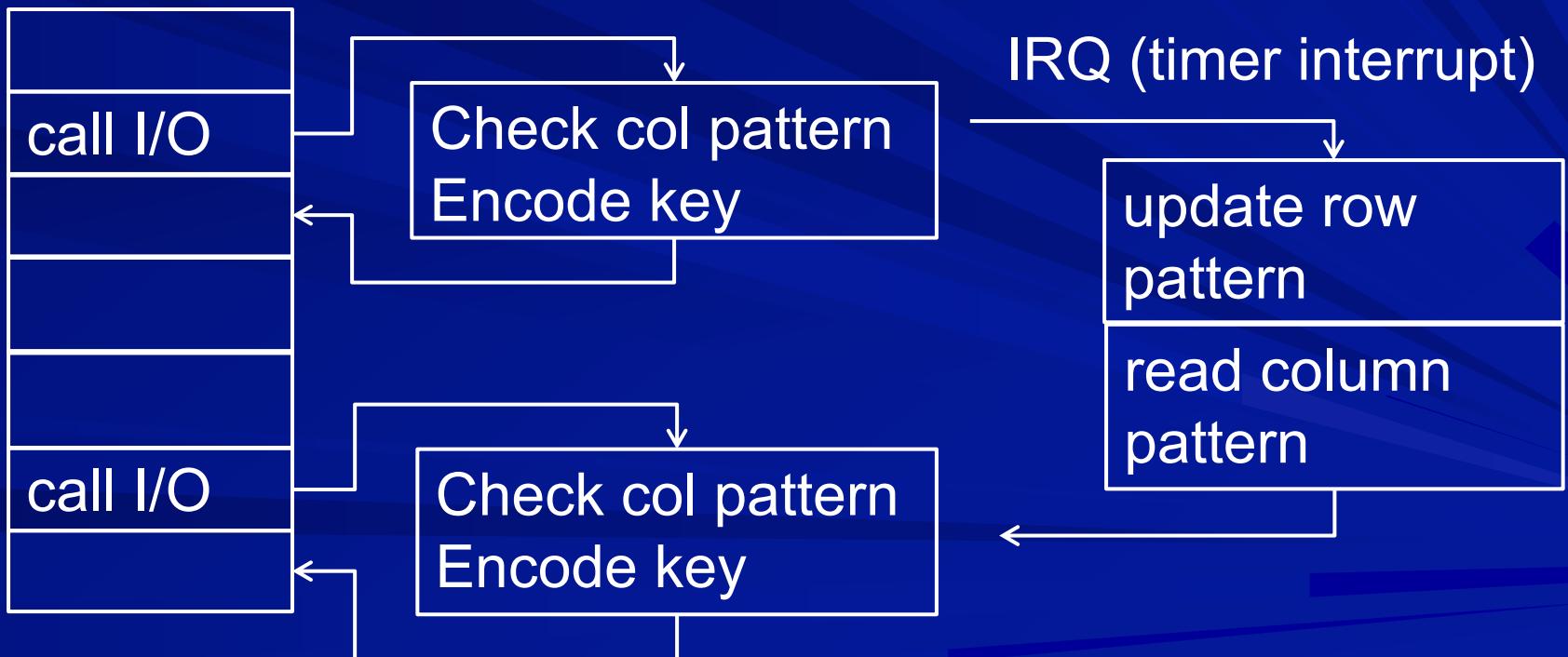
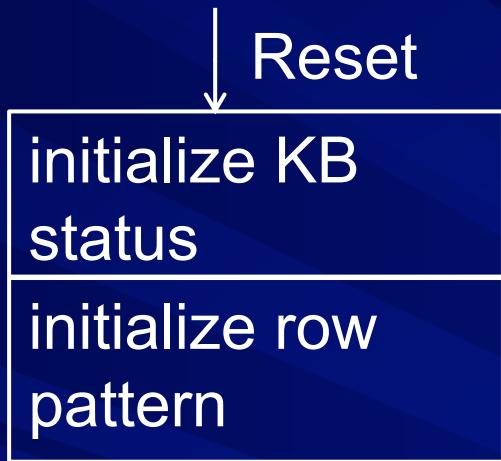


interrupt

Display refresh using interrupt



Keyboard scan using interrupt

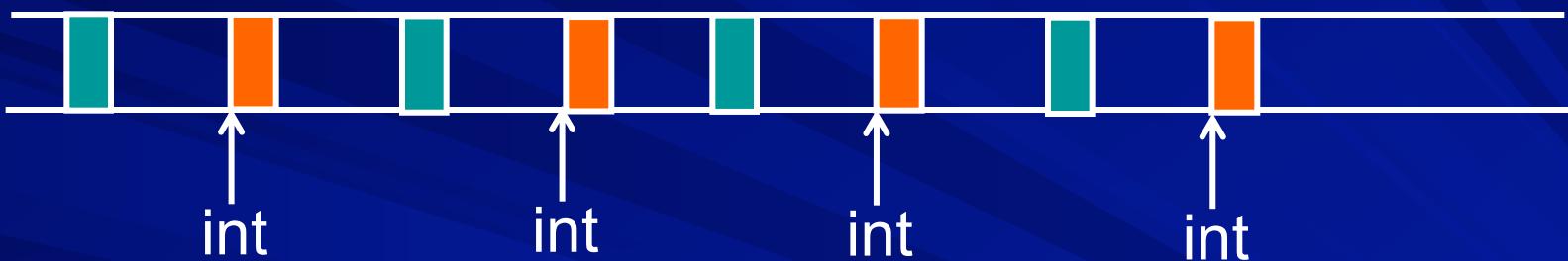


Direct memory access

- Allow direct data transfer between the device and the memory
- The process is initiated by the processor
- When the entire job is done, the device informs the processor by an interrupt
- A specialized controller called DMA controller is used

Benefit of DMA

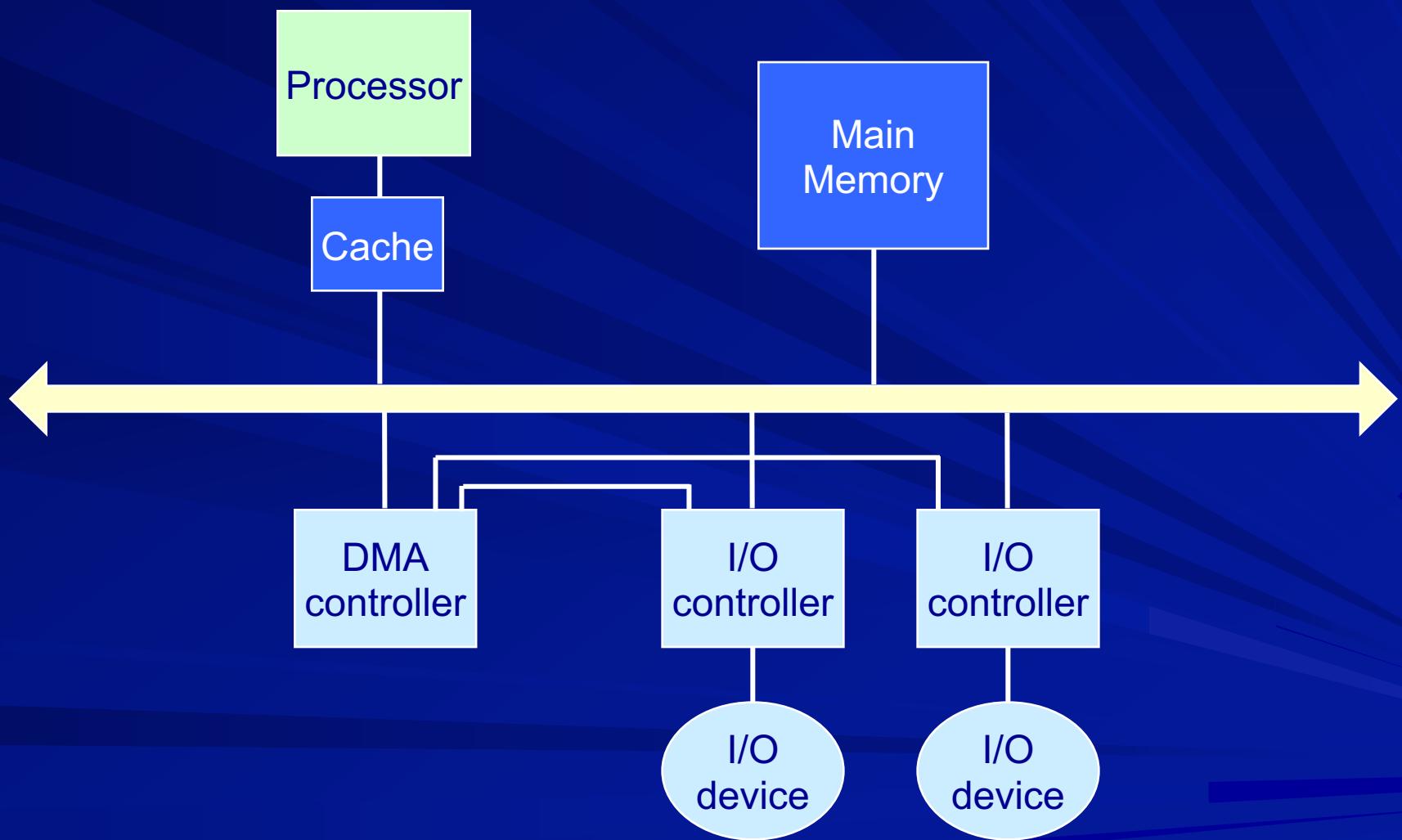
Without DMA



With DMA



Direct memory access



COL216

Computer Architecture

Input/Output – 3

21st March 2022

Interrupts/Exceptions

What are exceptions

- Unusual events/conditions

- Source: internal or external
 - Synchronous or asynchronous
 - Intentional or unintentional

- Require change in flow of control

- Mechanism to alter the control flow
 - Event identification and response to it
 - Halt or resume original control flow

Hardware and Software Interrupts

- Software interrupts
 - caused by specific instructions to get some system service
 - alternative terms : system calls, traps, exceptions
- Hardware interrupts are caused by events/conditions detected by hardware
 - intentional : e.g., I/O event
 - unintentional : e.g., hardware fault, power outage, arithmetic overflow

Terminology

Exceptions vs. Interrupts

- Use both the terms interchangeably
- MIPS, ARM: exception - internal as well as external, interrupt - external
- Intel 80x86: interrupt - internal as well as external
- PowerPC: exception - unusual event, interrupt - change in flow of control

Examples of exceptions

◆ synchronous

◆ asynchronous

	Intentional	Unintentional
Internal	Invoke OS function, Trace/debug	Access to privileged inst, Overflow/underflow, Undefined instruction, Hardware malfunction
External	I/O device request	Mem access exception, Alignment error, Timeouts, Power down, Hardware malfunction

Interrupt identification and response

- S/w interrupt identified during instruction decoding
- H/w interrupt identified by checking specific signals
- Response mechanism in both cases (h/w and s/w interrupts) is same
 - execution of some code : interrupt handler or interrupt service routine (**ISR**)
 - after serving interrupt, terminate or resume

Checking for exceptions



undefined instruction
checked here

overflow
checked here



hardware interrupts can be checked any time

ISR vs normal subroutine

- Processors have two or more modes
 - normal mode / user mode
 - privileged mode / kernel mode / supervisor mode
- Application program executes in user mode
- To do certain privileged tasks, execution of some code in OS kernel is required
- ISR executes in privileged mode, provides controlled access to kernel functions

How exceptions are handled

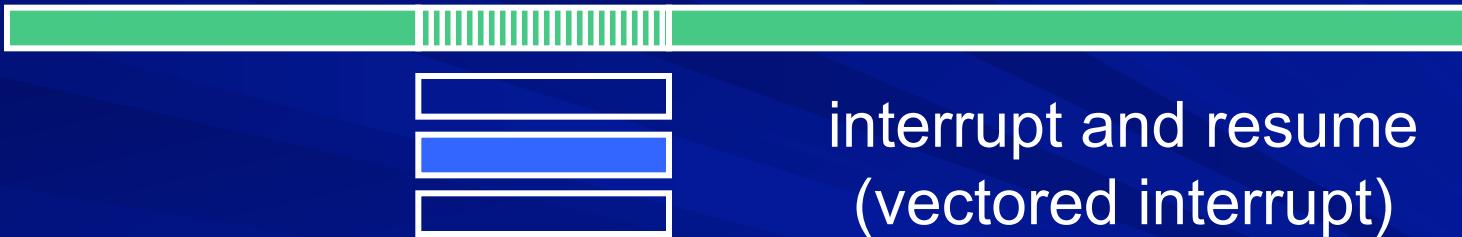
uninterrupted execution



interrupt and halt



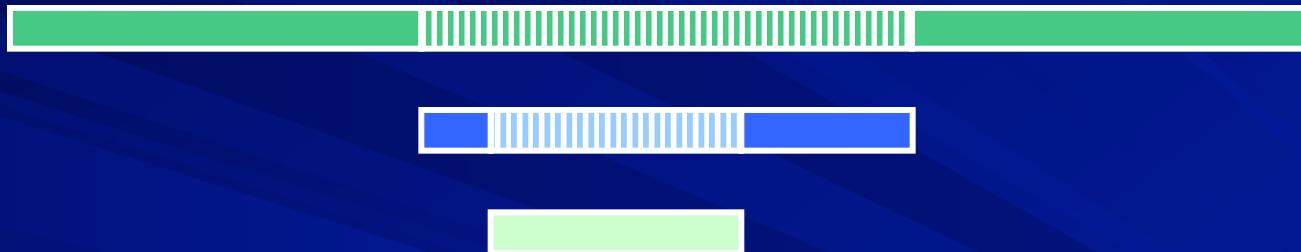
interrupt and resume
(vectored interrupt)



interrupt and resume
(non-vectored)



Nested interrupts



Exceptions in ARM

Exception type	Mode	Address
Reset	Supervisor	0x00000000
Undef instr	Undefined	0x00000004
S/W interrupt	Supervisor	0x00000008
Prefetch Abort	Abort	0x0000000C
Data Abort	Abort	0x00000010
Interrupt	IRQ	0x00000018
Fast interrupt	FIQ	0x0000001C

SWI software interrupt

cond	1111	comment
4	4	24

- Control transferred to address 0x00000008
- Mode changes to ‘supervisor’ mode
- CPSR (Current Processor Status Register) is saved in SPSR_{svc} (Saved Processor Status Register)
- PC is saved in R14_{svc}
- Comment field may specify a particular service/function

Processor status register



ARM register set

Mode	Registers			SP	LR	PC
	R0-R7	R8-R12	R13			
User:			R14	R15	CPSR	
System:						
Supervisor:			R13	R14		SPSR
Abort:			R13	R14		SPSR
Undefined:			R13	R14		SPSR
Interrupt:			R13	R14		SPSR
Fast int:	R8-R12		R13	R14		SPSR

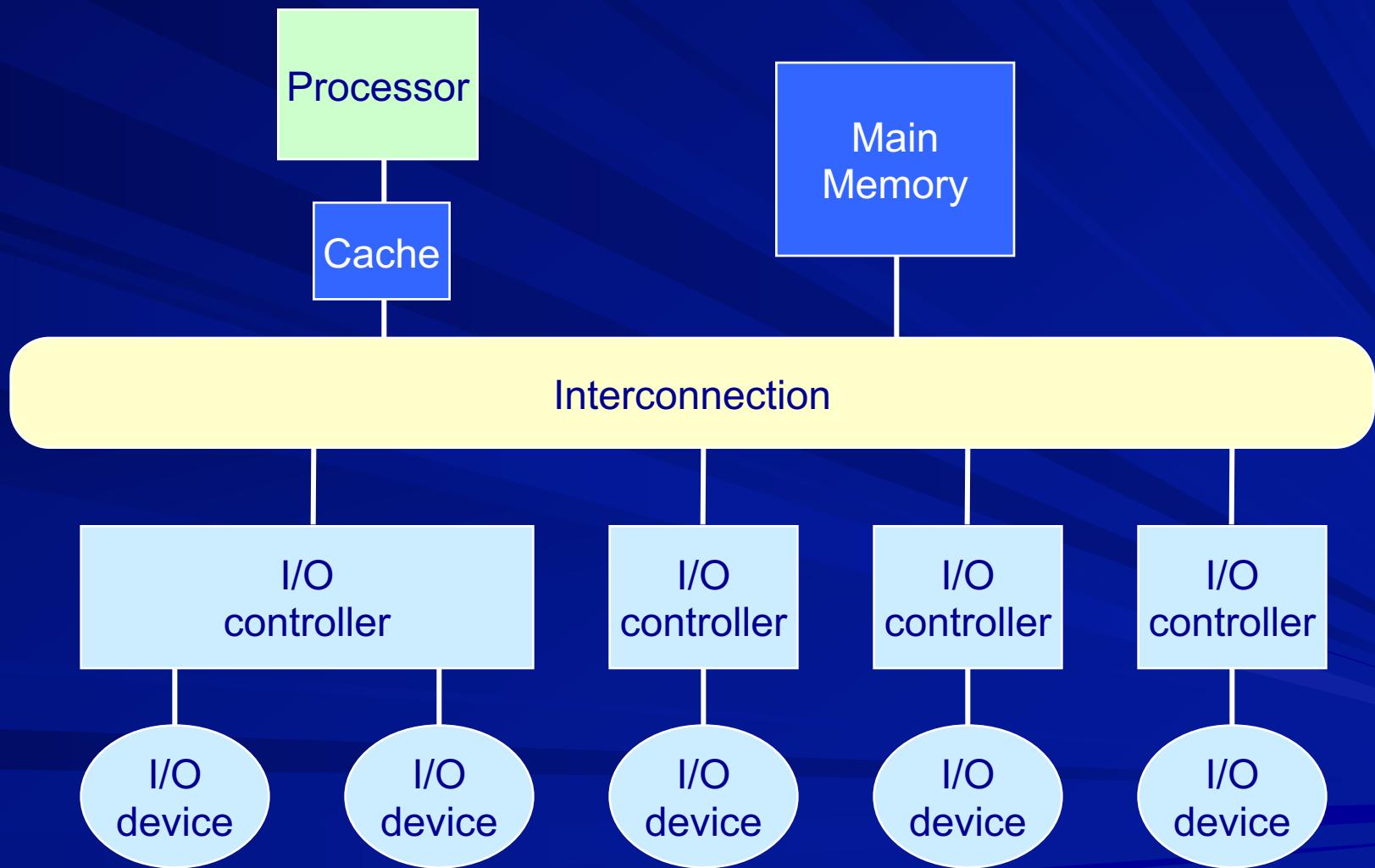
Total 37

Exception handling in pipelined design

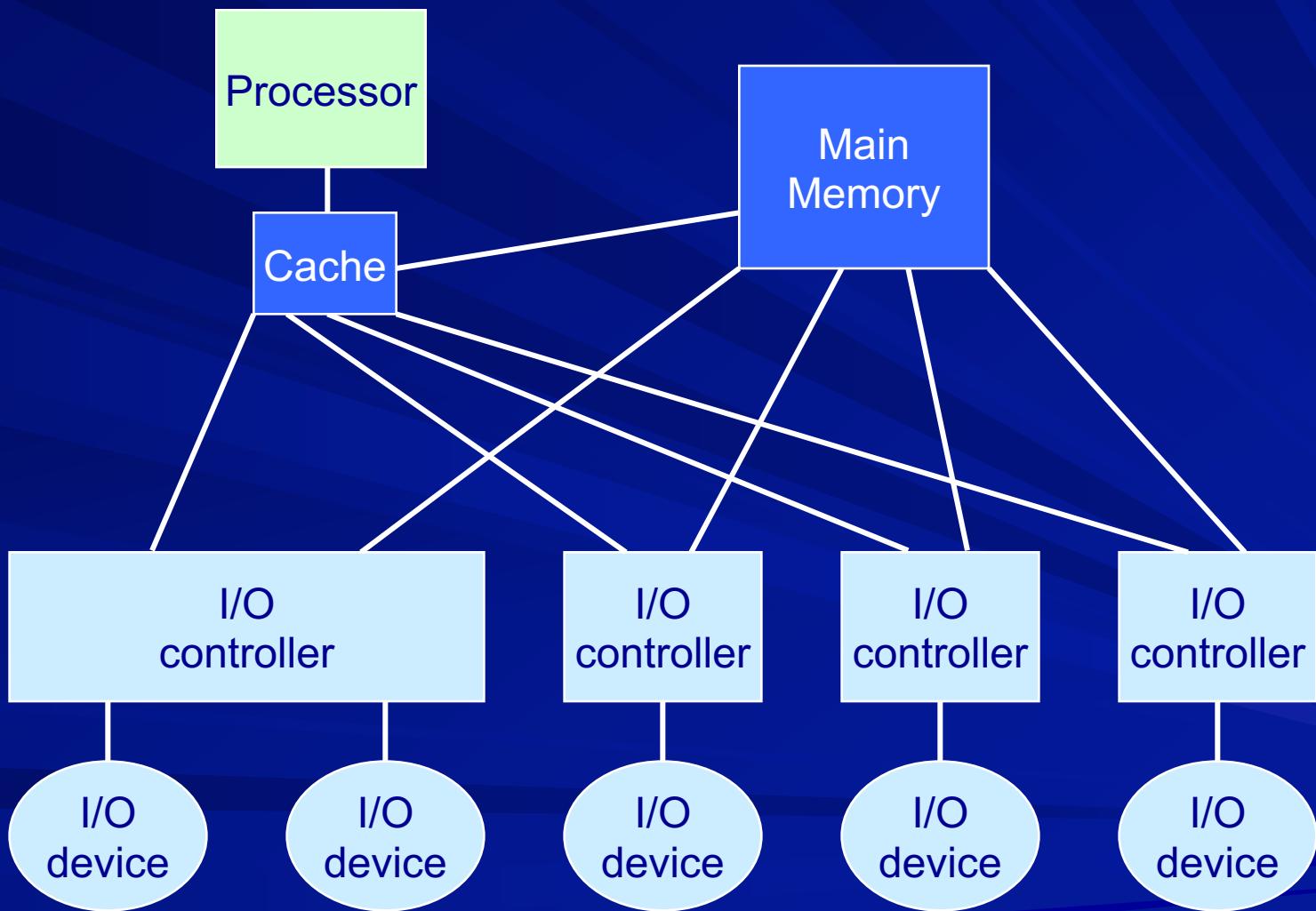
- Difficulty in status saving
 - multiple instructions under execution
- Possibility of
 - multiple exceptions in same cycle
 - change in order of exceptions
- Handling approaches
 - precise
 - imprecise

Interconnecting Subsystems

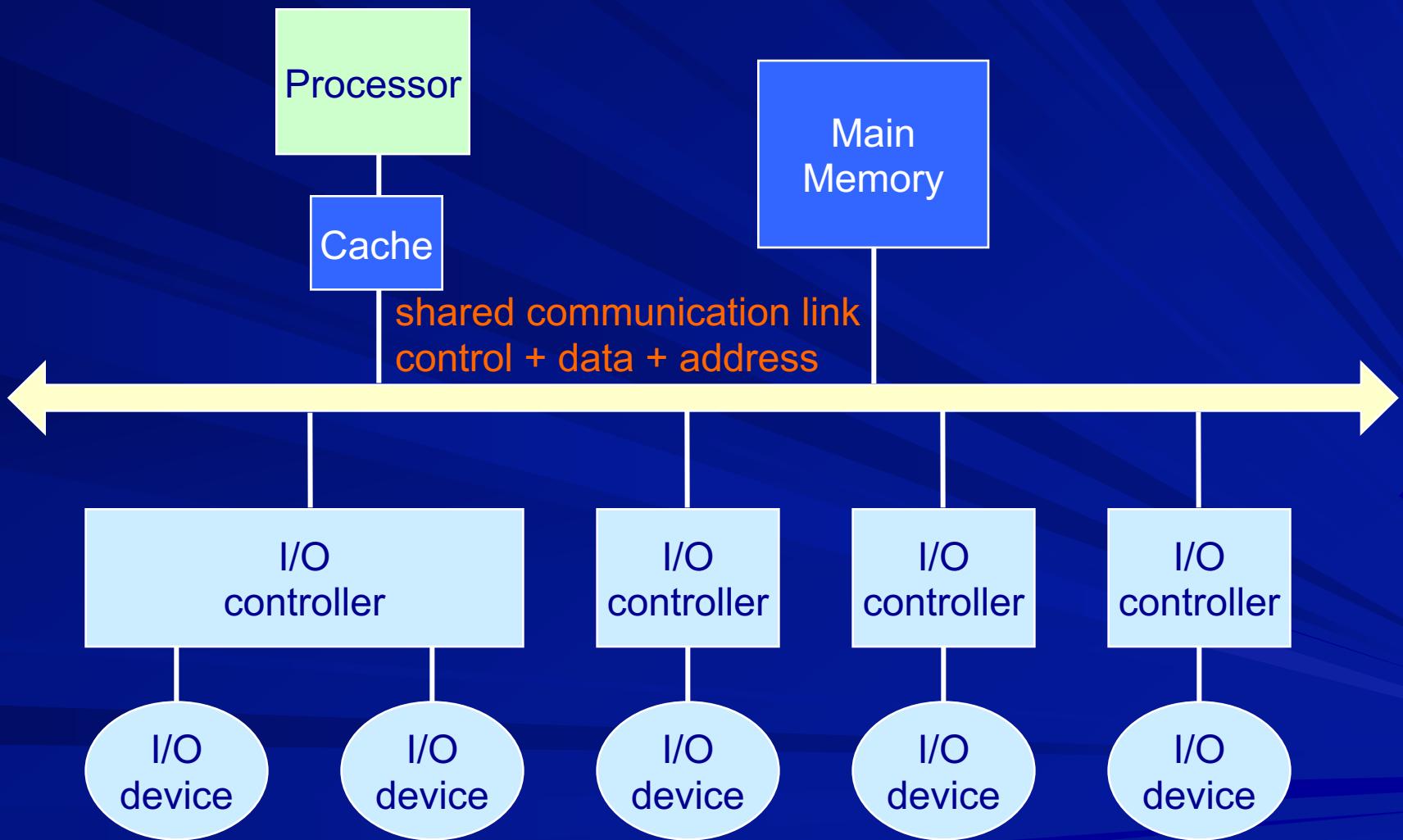
Interconnecting Subsystems



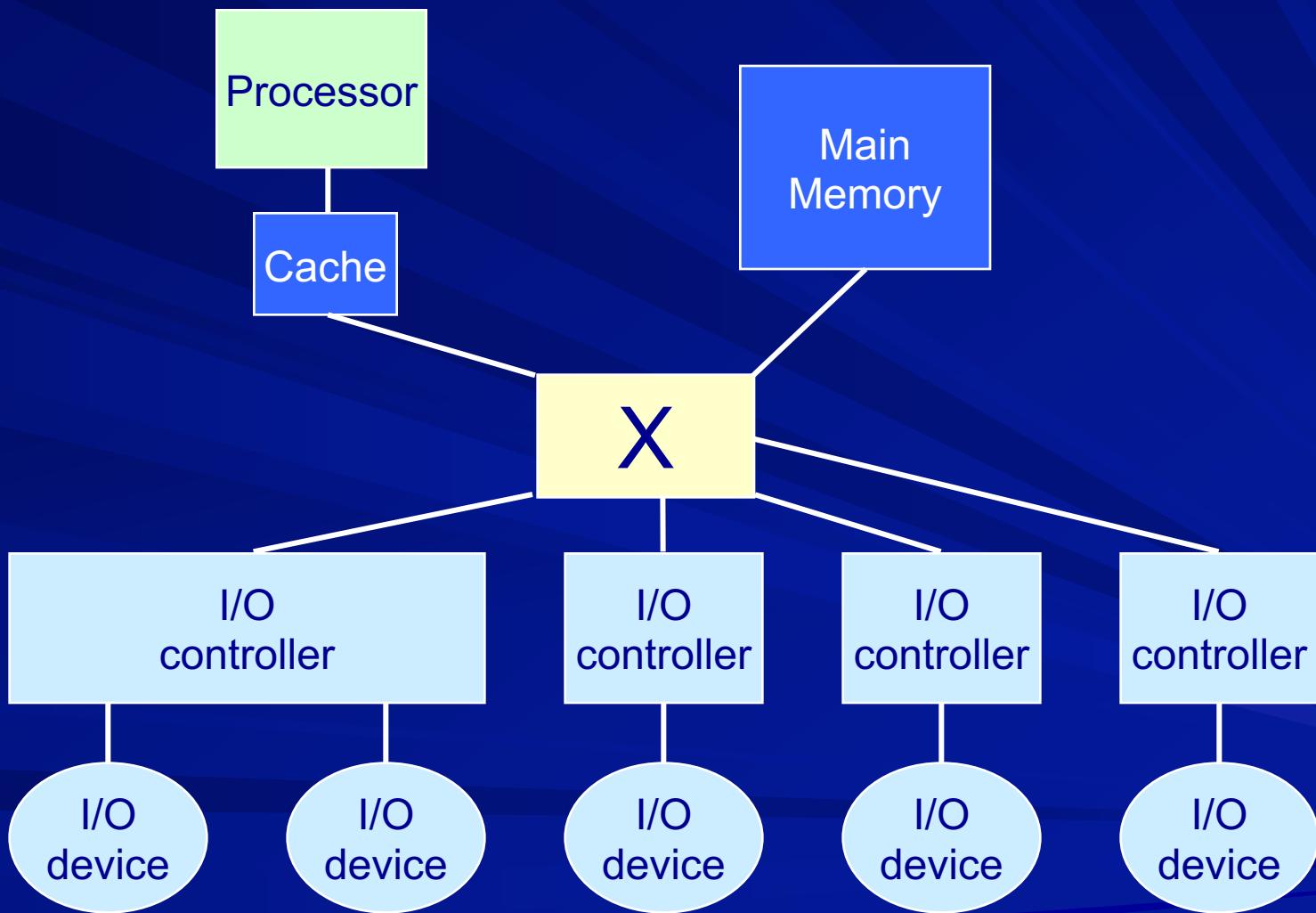
Point to point connections



Single bus connecting all



Cross bar switch



Comparison

	<u>P to P</u>	<u>Bus</u>	<u>X-bar</u>
Cost	high	low	high
Throughput	high	low	high
Ports	multi	single	single
Expansion	difficult	easy	difficult

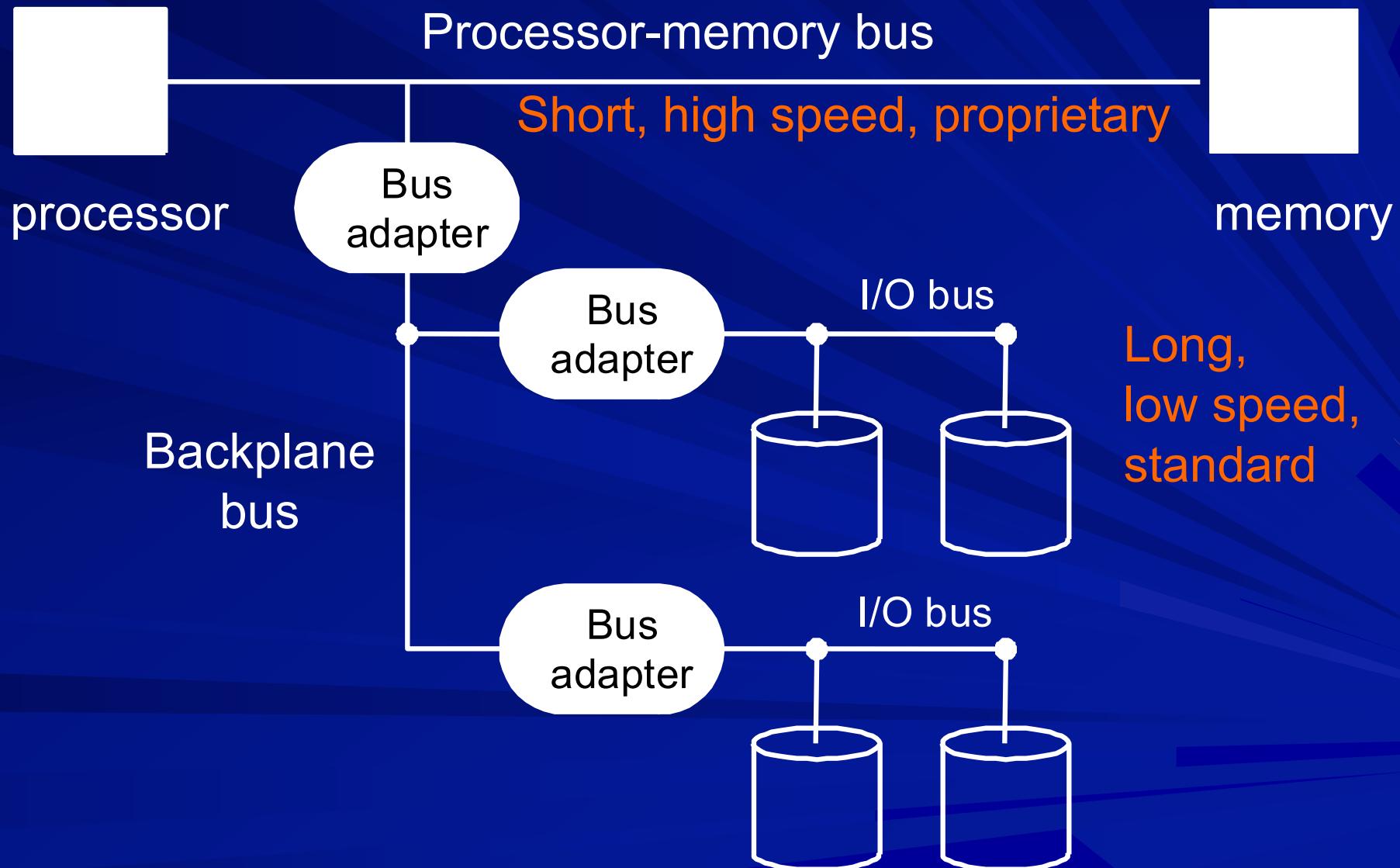
Multi-buses: More performance, more cost

Multi-switches: Lower cost, same performance

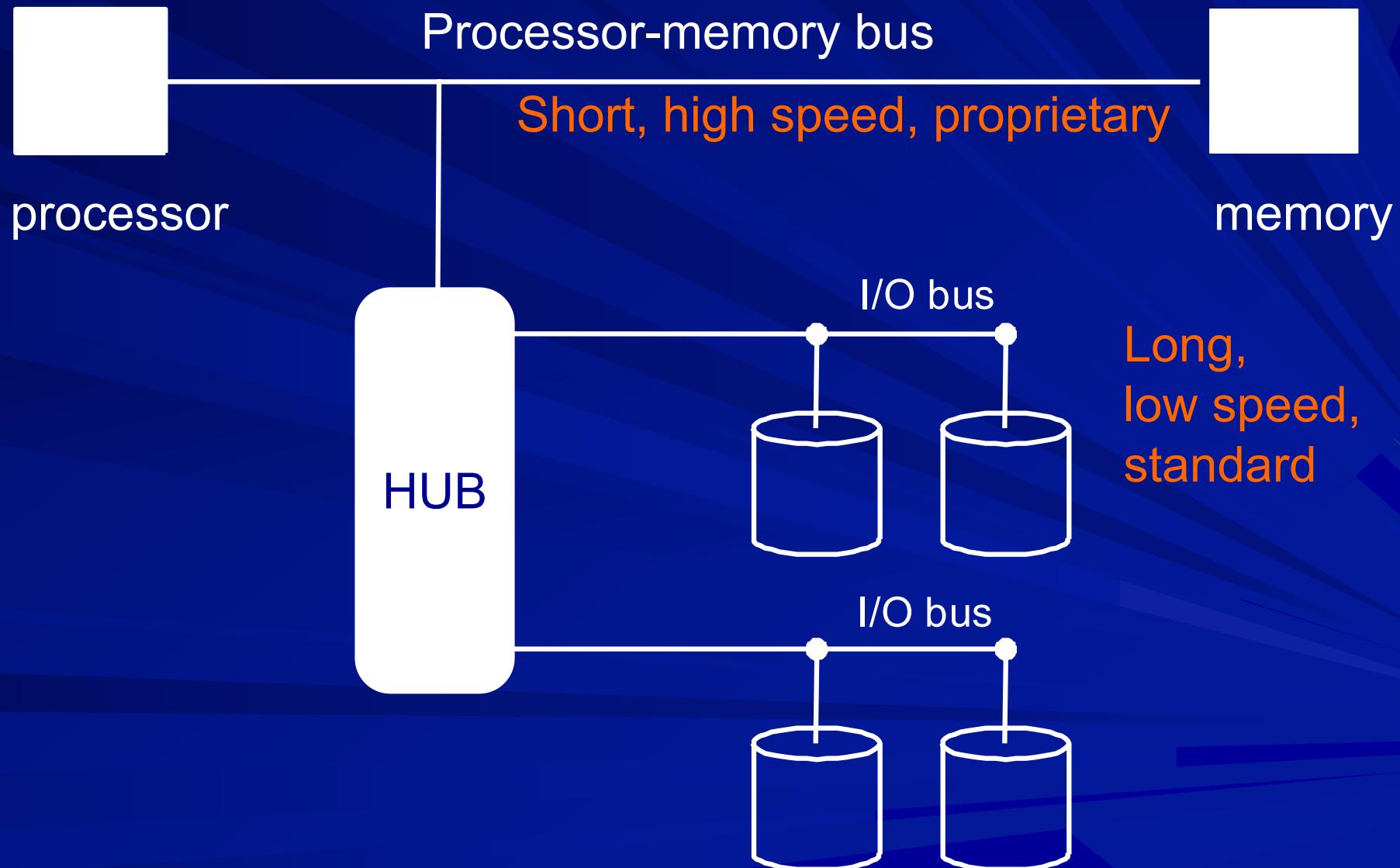
Backplane bus



System with multiple buses



System with multiple buses



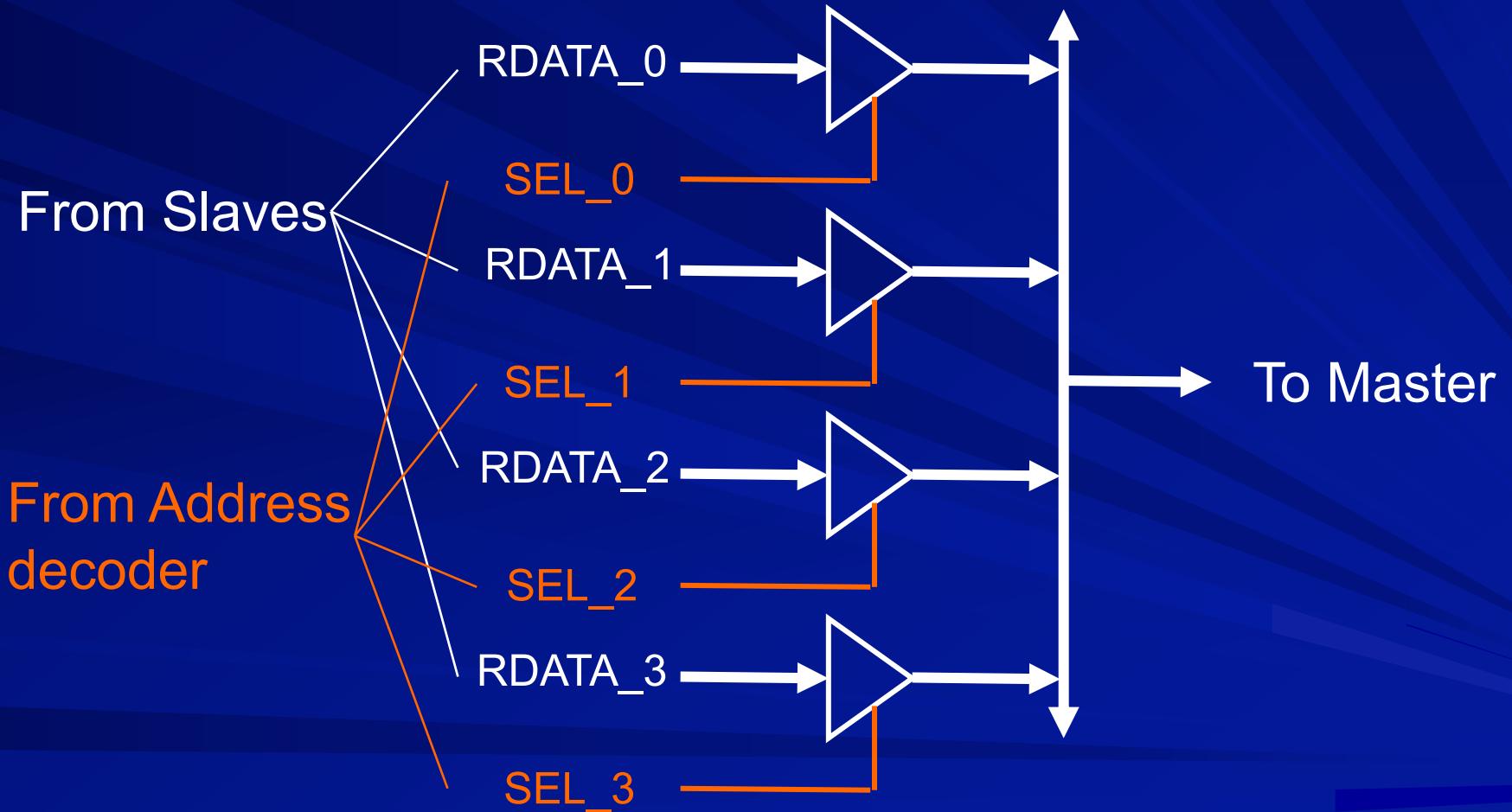
Bus Standards

- Physical / mechanical
 - Pins, connectors, cables
- Electrical
 - Voltage / current levels, impedances
- Logical
 - Definition of signals
 - Timings and protocols

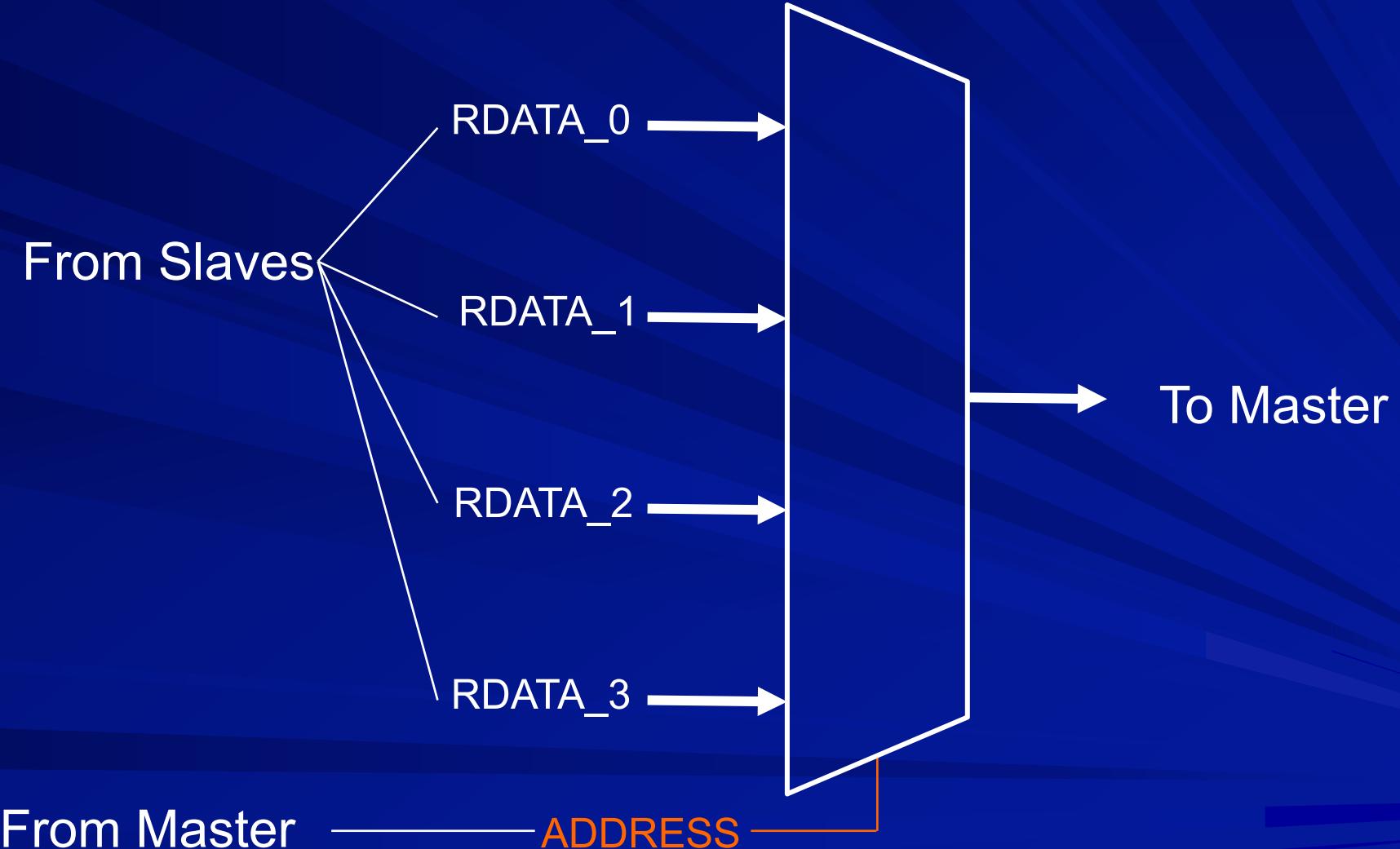
Subsystems on the buses

- Master
 - Initiates transfer
- Slave
 - Responds to master
- Bridge /
bus adapter
 - Connects two buses
- Hub
 - Connects many buses
- Arbiter
 - Resolves master requests
- Decoder
 - Selects slaves

Tri-state Bus



Multiplexed Bus



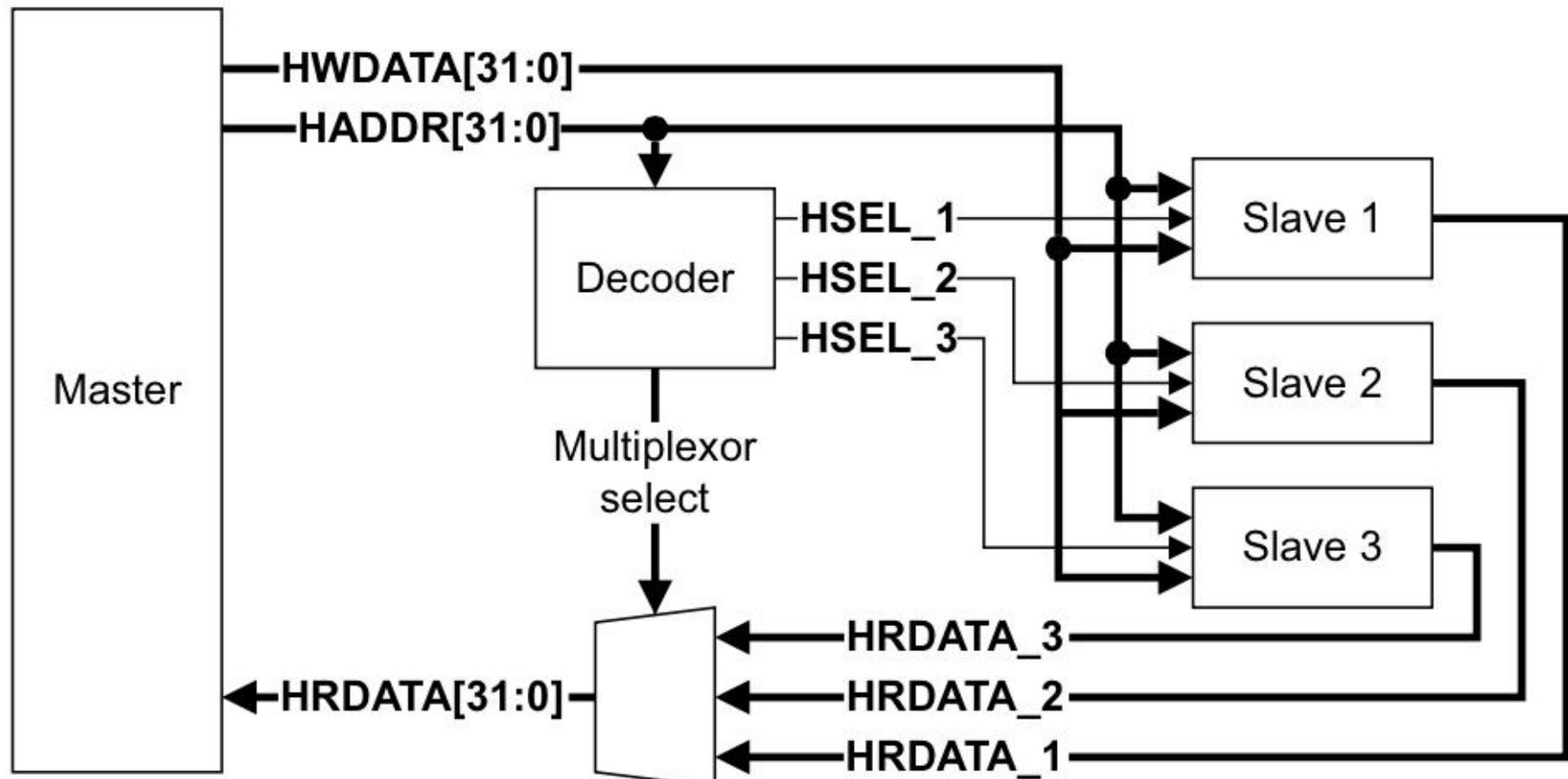
AMBA : Advanced Microcontroller Bus Architecture

- ARM open standard for on-chip buses
- For System-on-Chip (SoC)
- Variants
 - AHB : Advanced High speed Bus
 - APB : Advanced Peripheral Bus
 - AXI : Advanced eXtensible Interface
 - AHB-Lite : subset of AHB

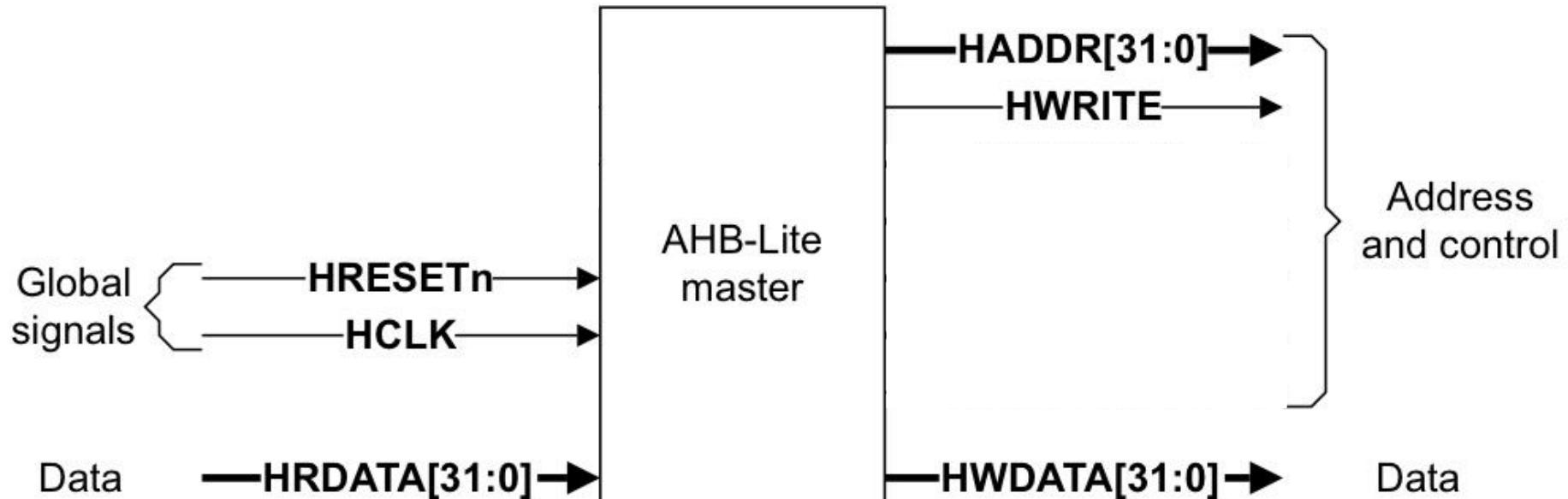
AHB-Lite

- Single master, multiple slaves
- Parallel
- Multiplexed (not tri-state)
- Separate data, address, control
- Synchronous
- Pipelined
- Burst transfer supported
- Split transaction not supported

AHB-Lite Block Diagram

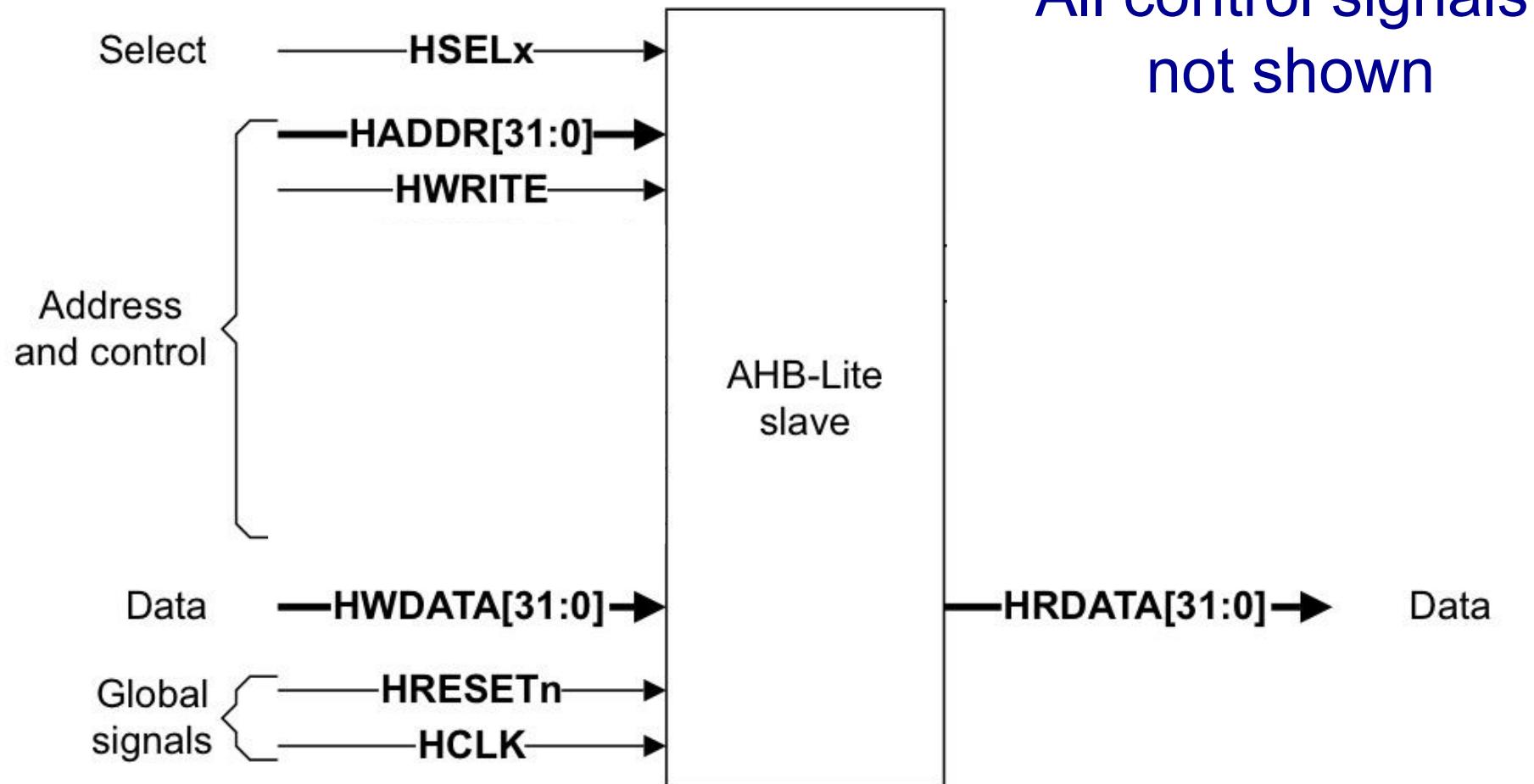


Master Interface



All control signals not shown

Slave Interface



THANKS

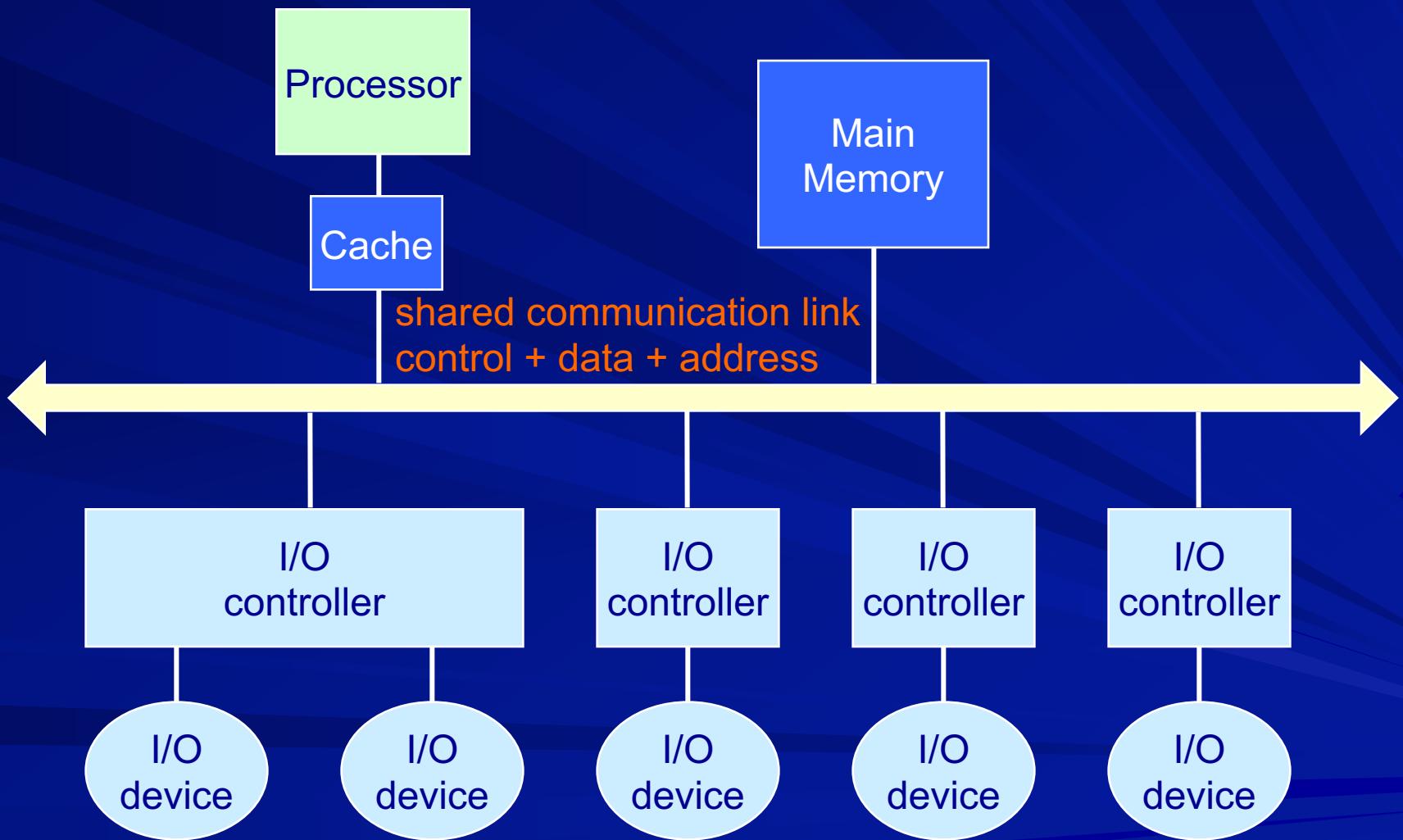
COL216

Computer Architecture

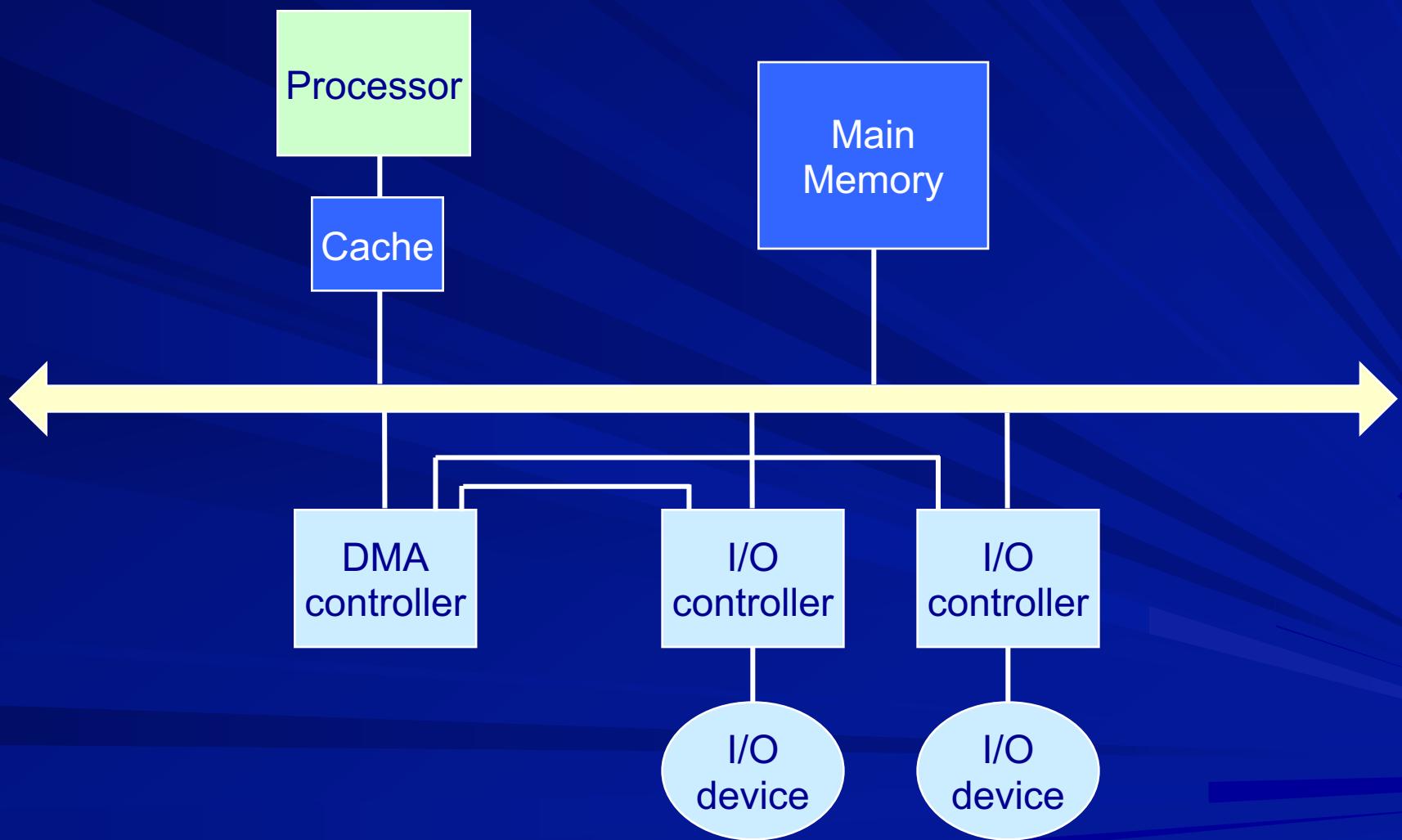
Input/Output – 4

24th March 2022

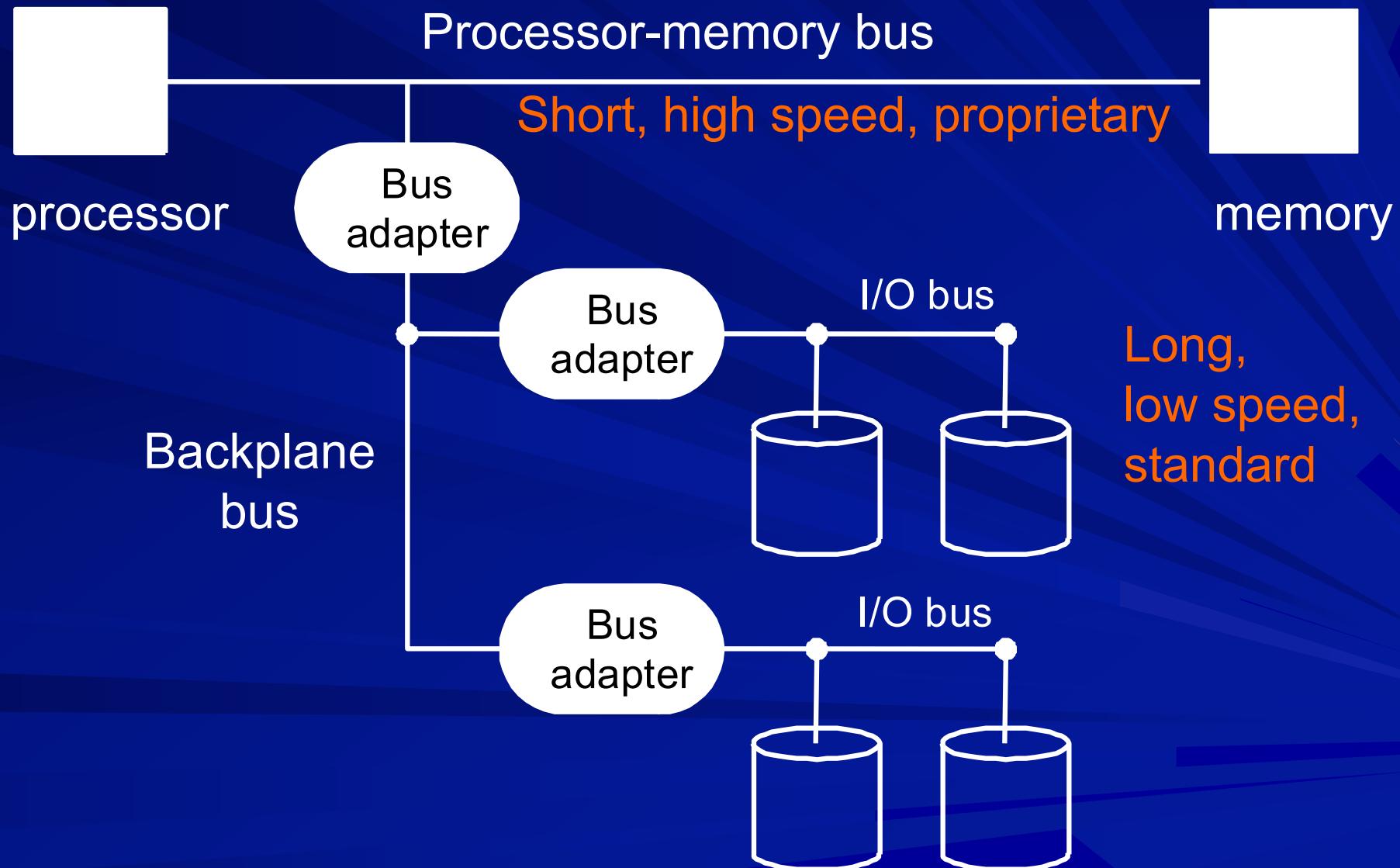
Single bus connecting all



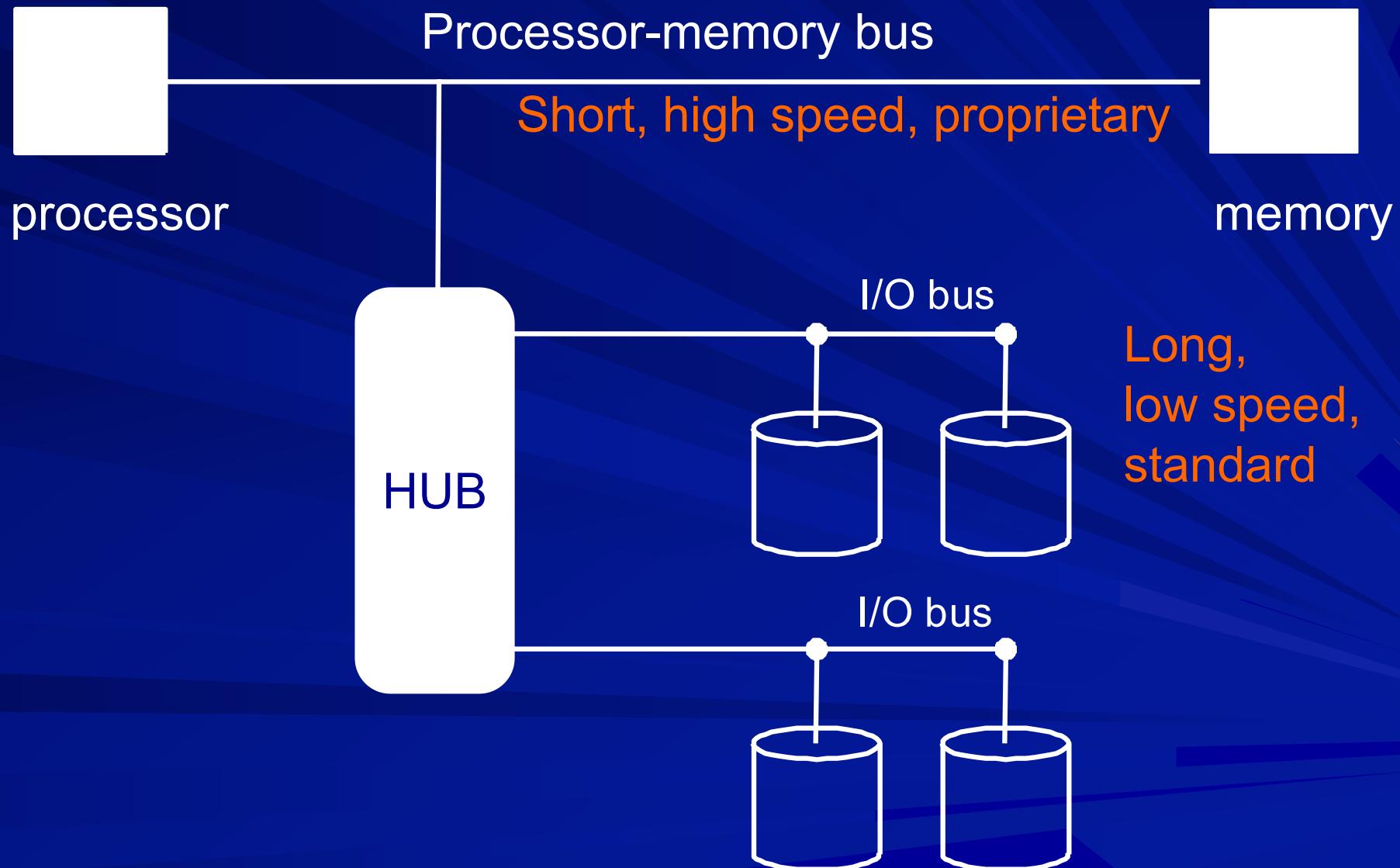
Direct memory access



System with multiple buses



System with multiple buses



Bus Standards

- Physical / mechanical
 - Pins, connectors, cables
- Electrical
 - Voltage / current levels, impedances
- Logical
 - Definition of signals
 - Timings and protocols

Standard buses/ports/interfaces

- Standardization required for subsystems from multiple sources to work together
- Technological developments lead to continuous improvement of specs, standardization implies freezing the specs
- Standards need to be revised and improved periodically
- Standardization is done by consortium of several organizations or professional bodies

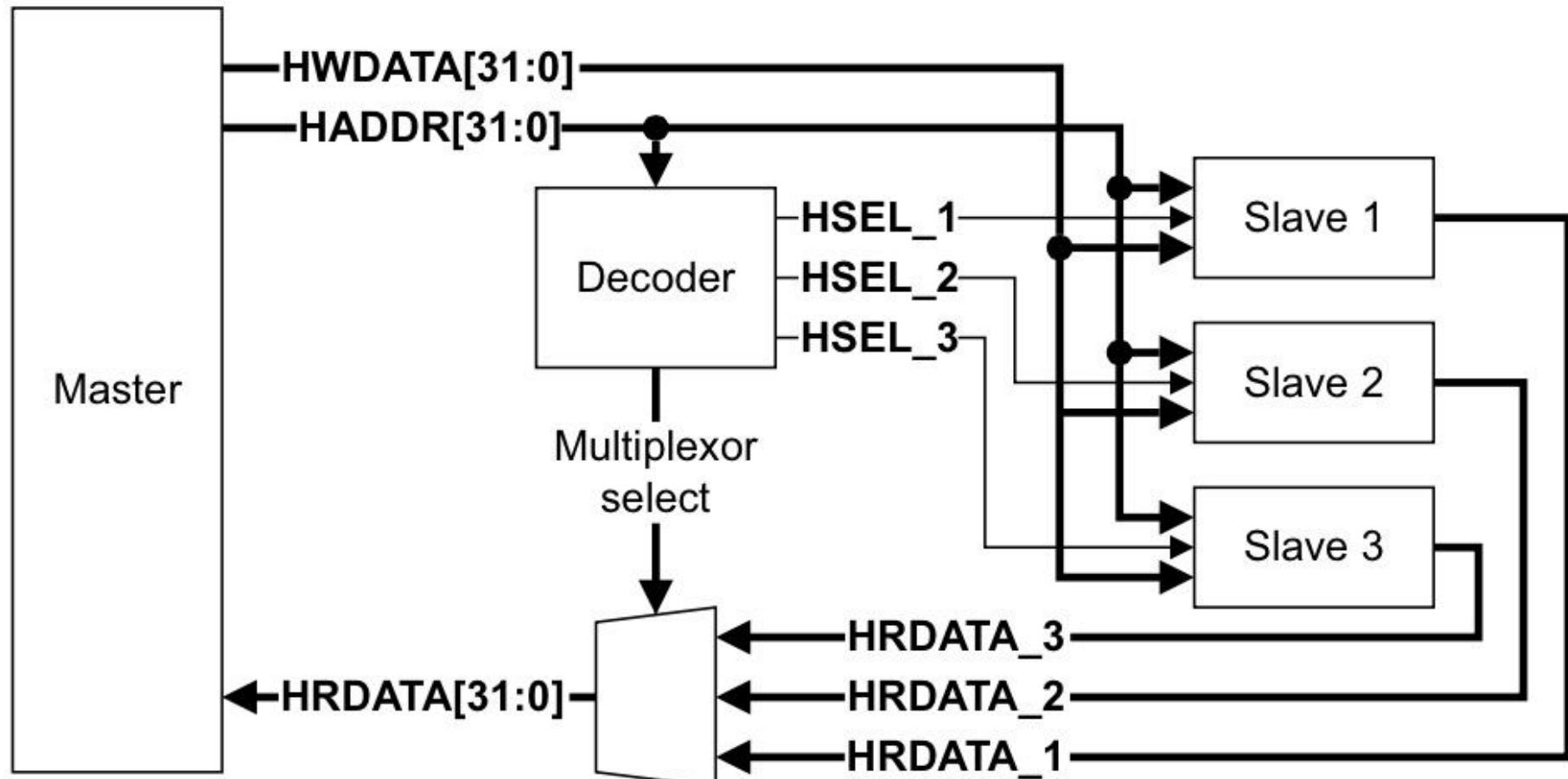
AMBA : Advanced Microcontroller Bus Architecture

- ARM open standard for on-chip buses
- For System-on-Chip (SoC)
- Variants
 - AHB : Advanced High speed Bus
 - APB : Advanced Peripheral Bus
 - AXI : Advanced eXtensible Interface
 - AHB-Lite : subset of AHB

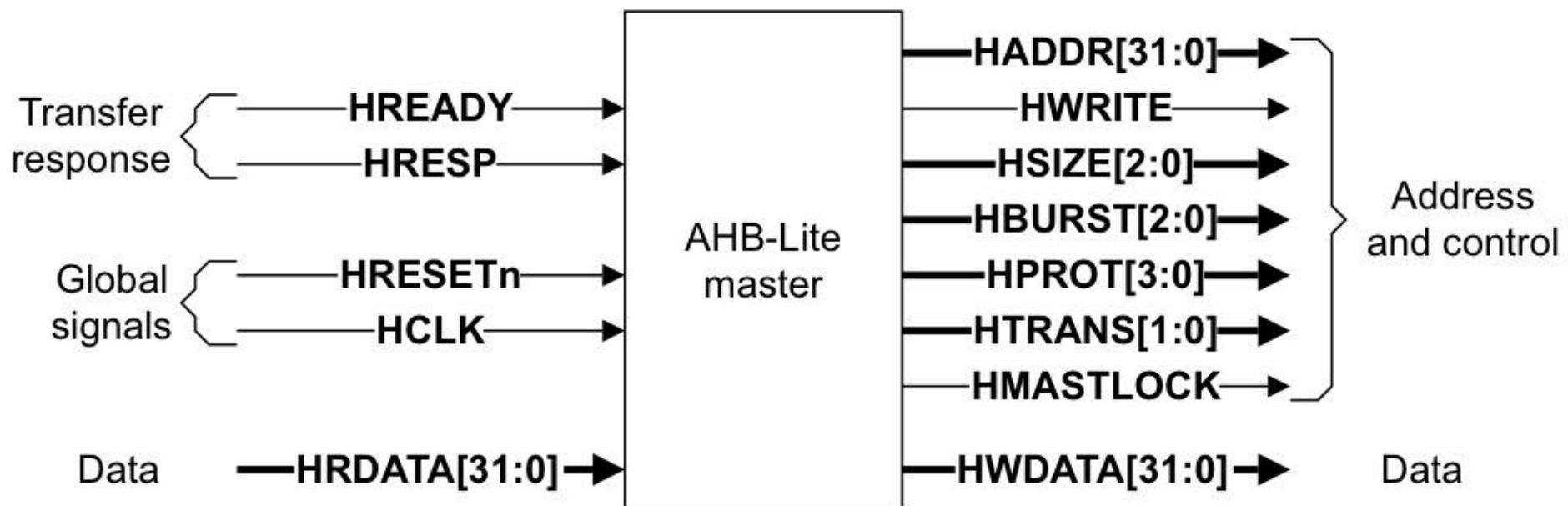
AHB-Lite

- Single master, multiple slaves
- Parallel
- Multiplexed (not tri-state)
- Separate data, address, control
- Synchronous
- Pipelined
- Burst transfer supported
- Split transaction not supported

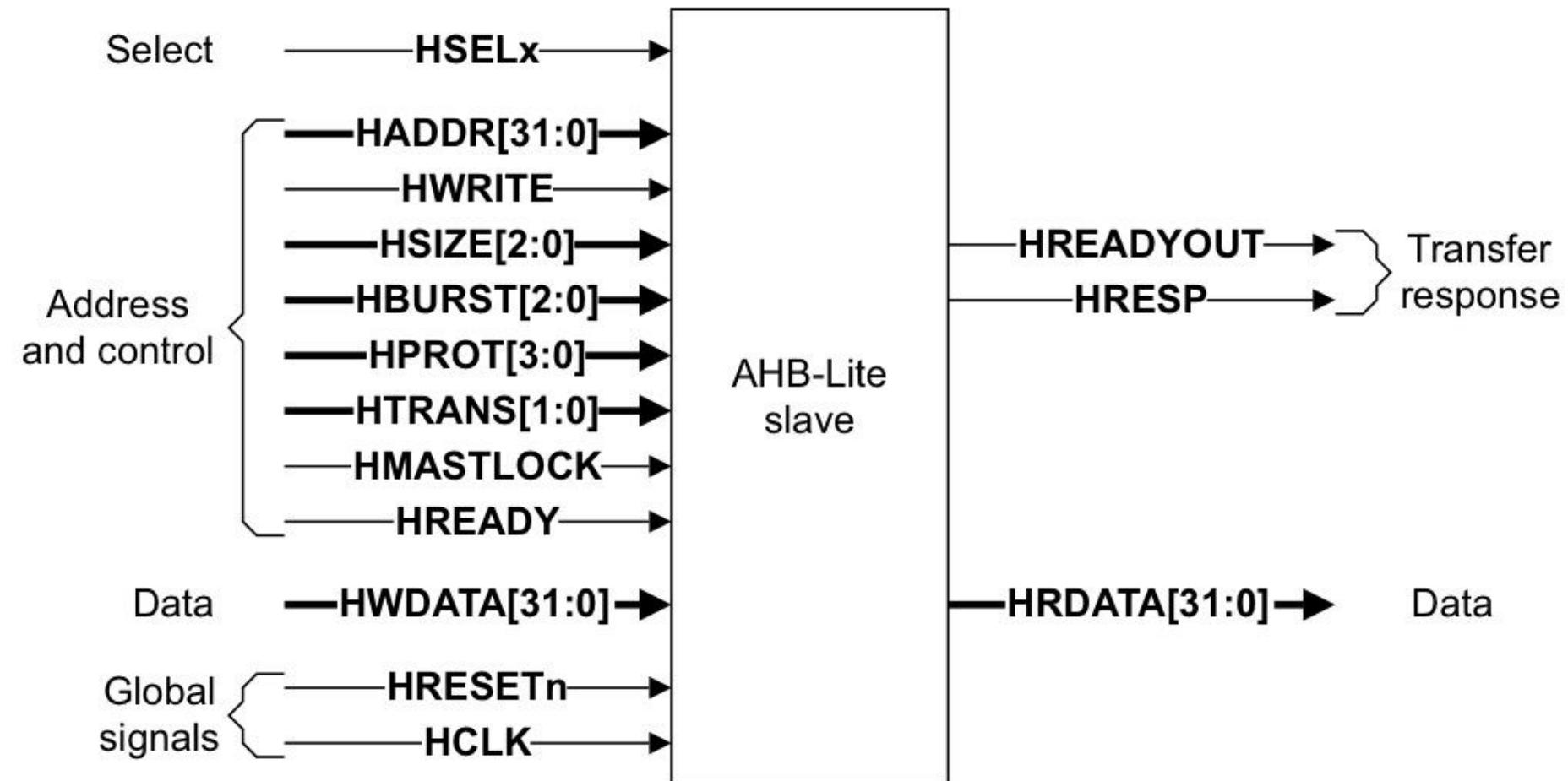
AHB-Lite Block Diagram



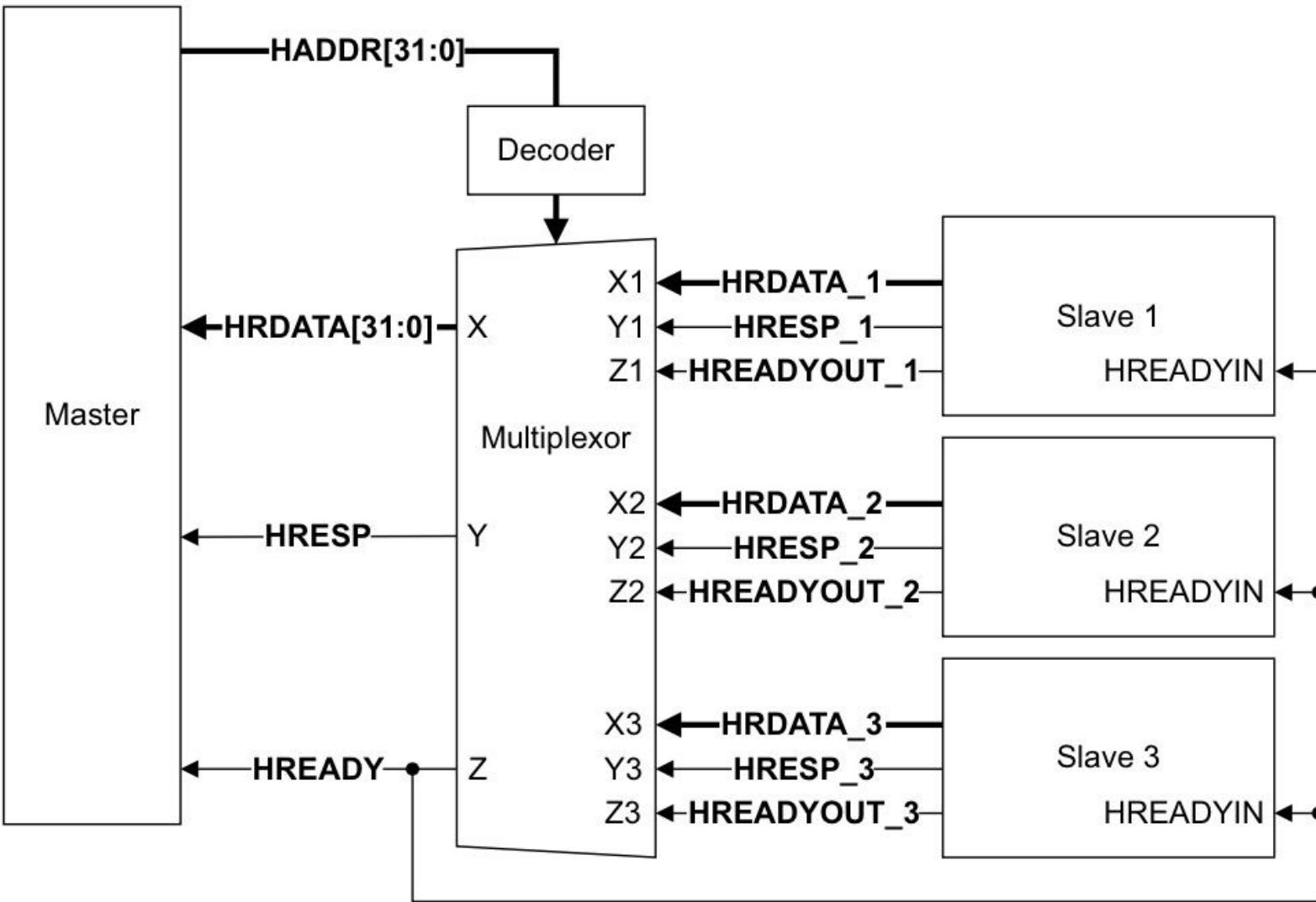
Master Interface



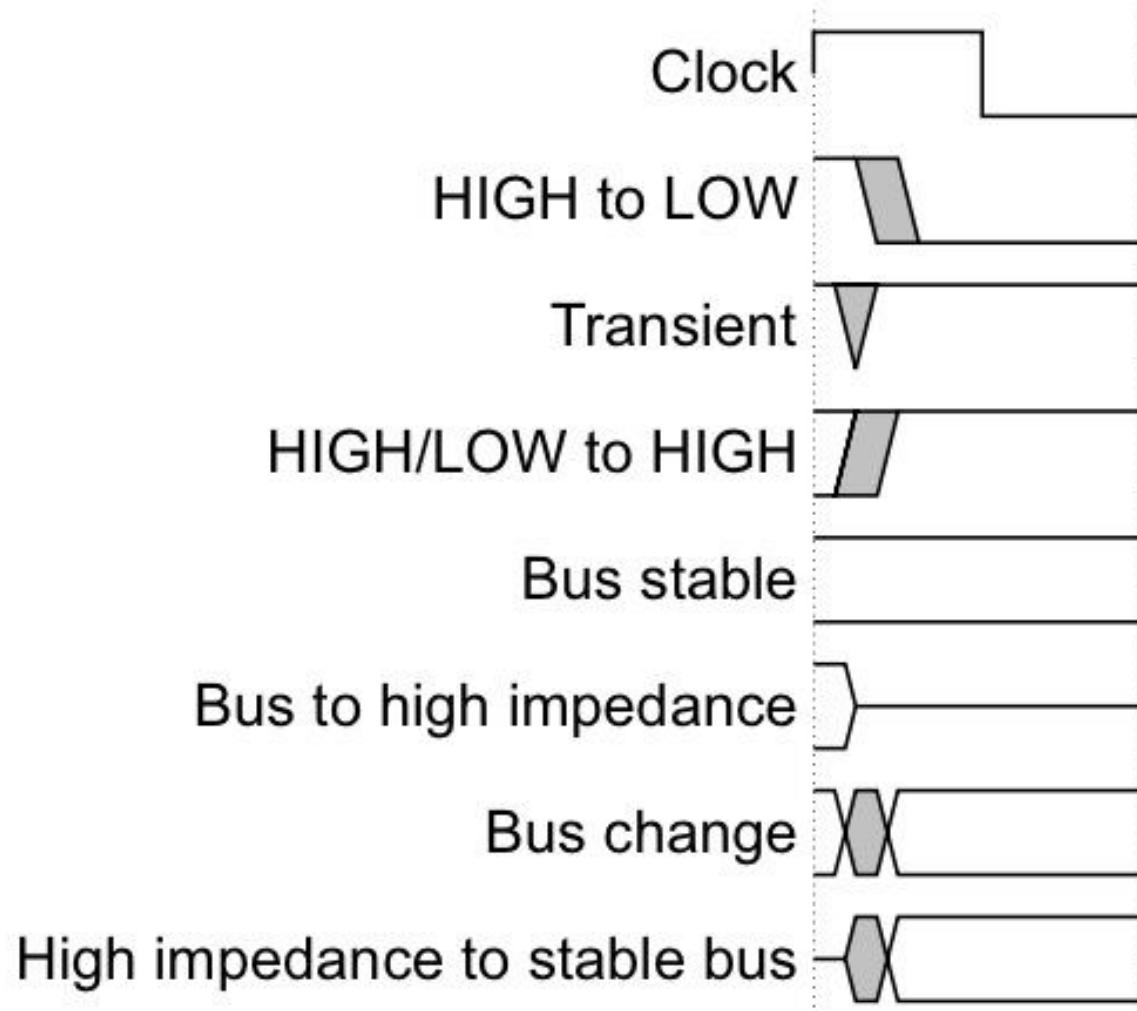
Slave Interface



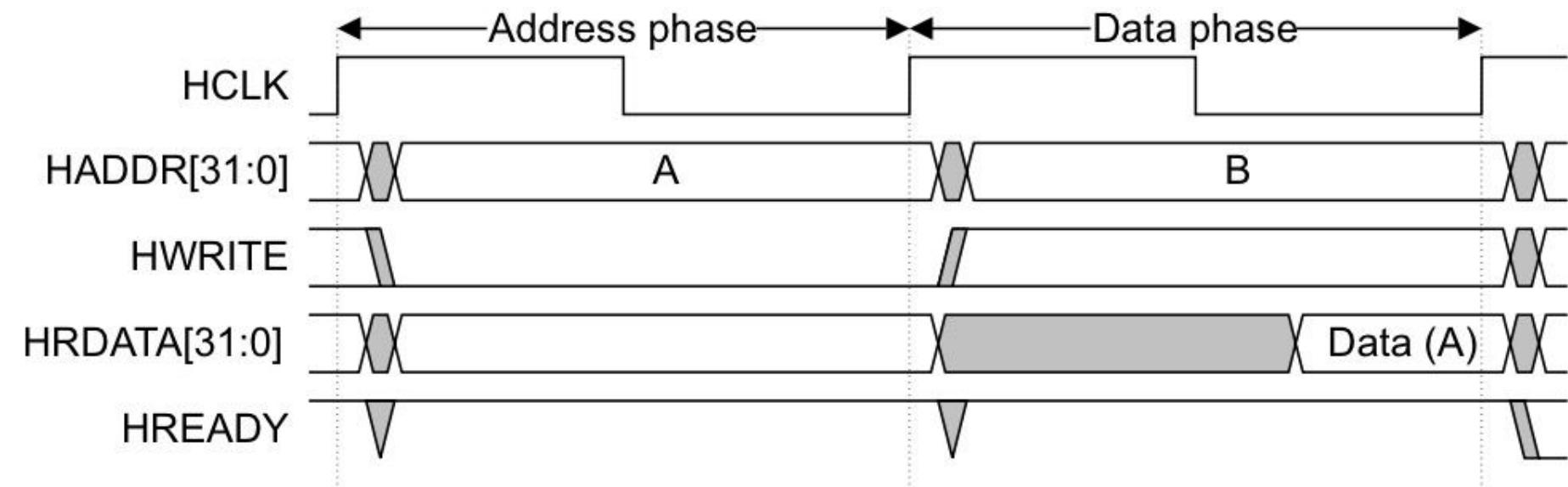
Multiplexor Interconnection



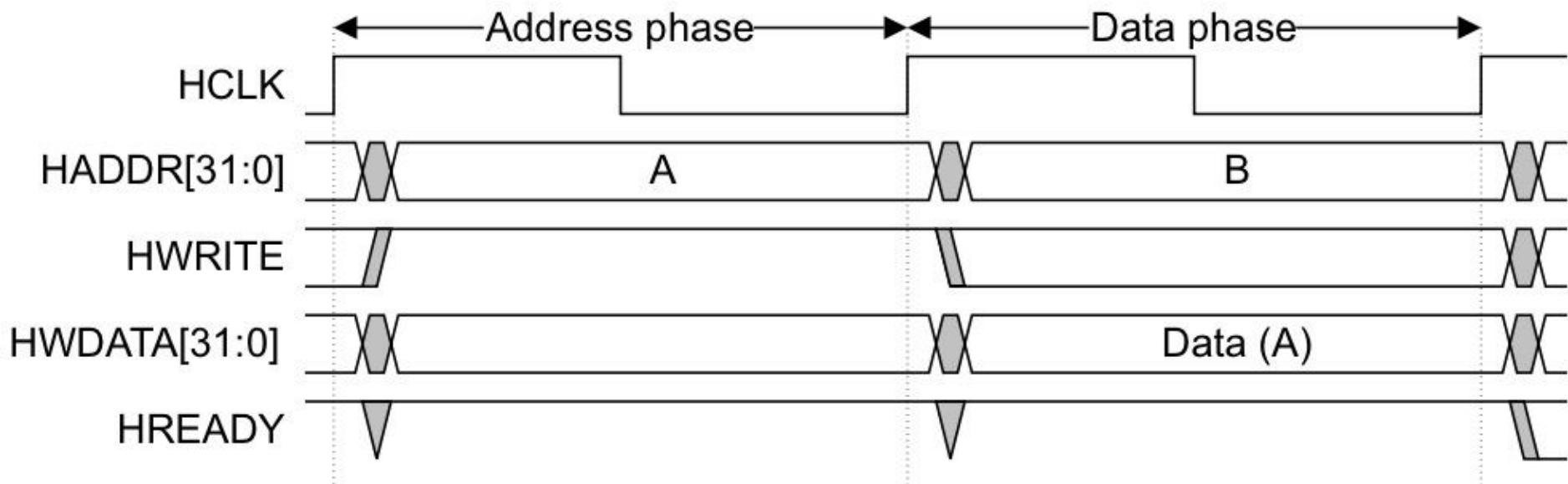
Timing Diagram Conventions



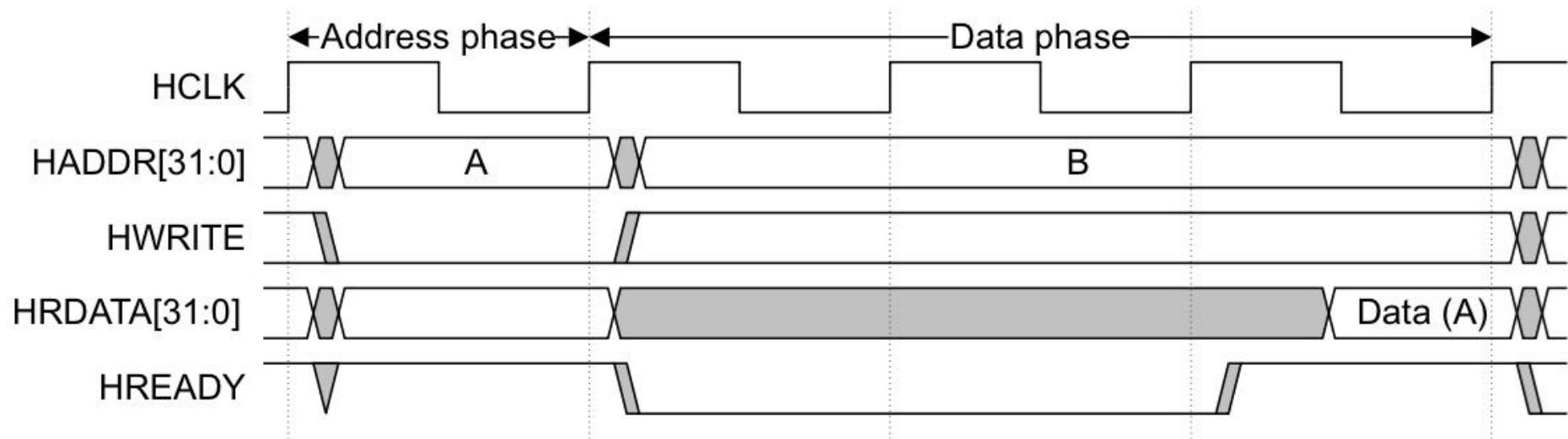
Read Transfer



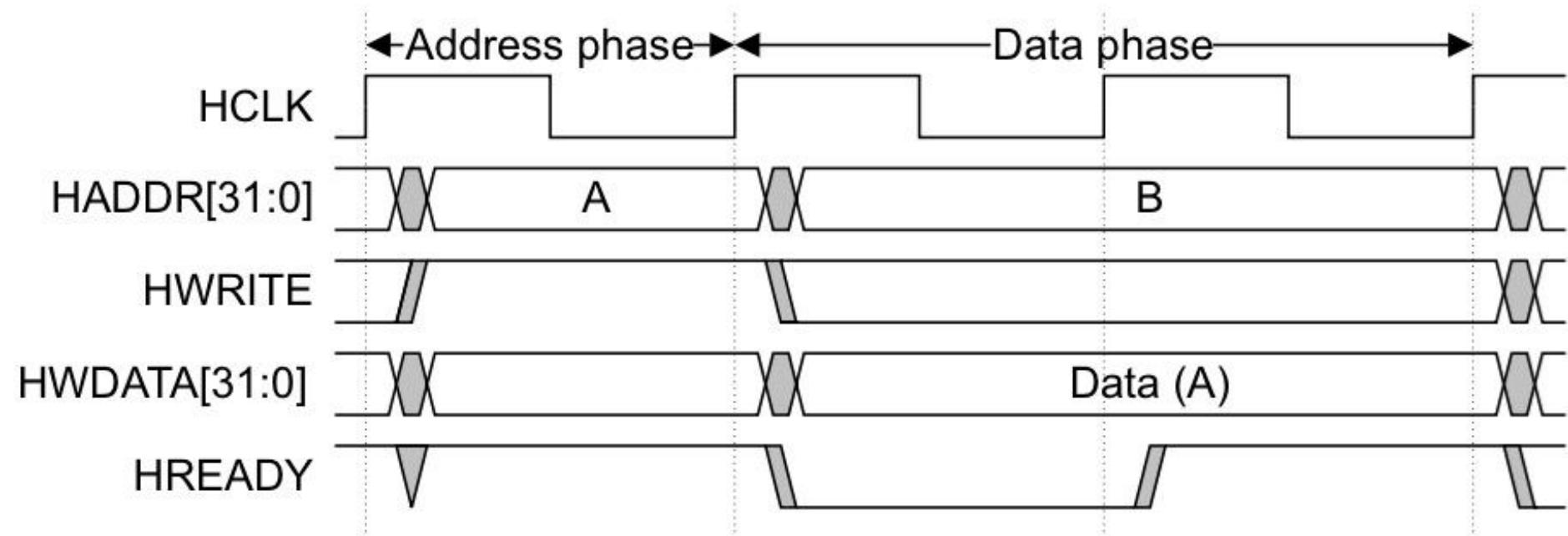
Write Transfer



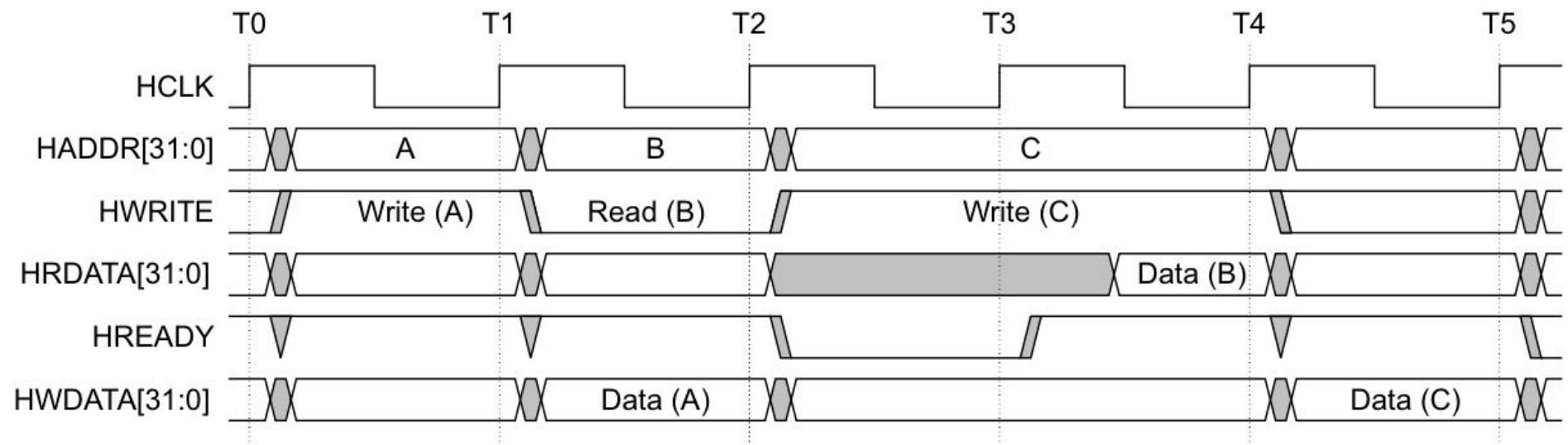
Read Transfer with Wait States



Write Transfer with Wait States



Multiple Transfers



Transfer Types

HTRANS[1:0] Type

00 IDLE

Master does not want transfer

01 BUSY

Master inserts wait cycle

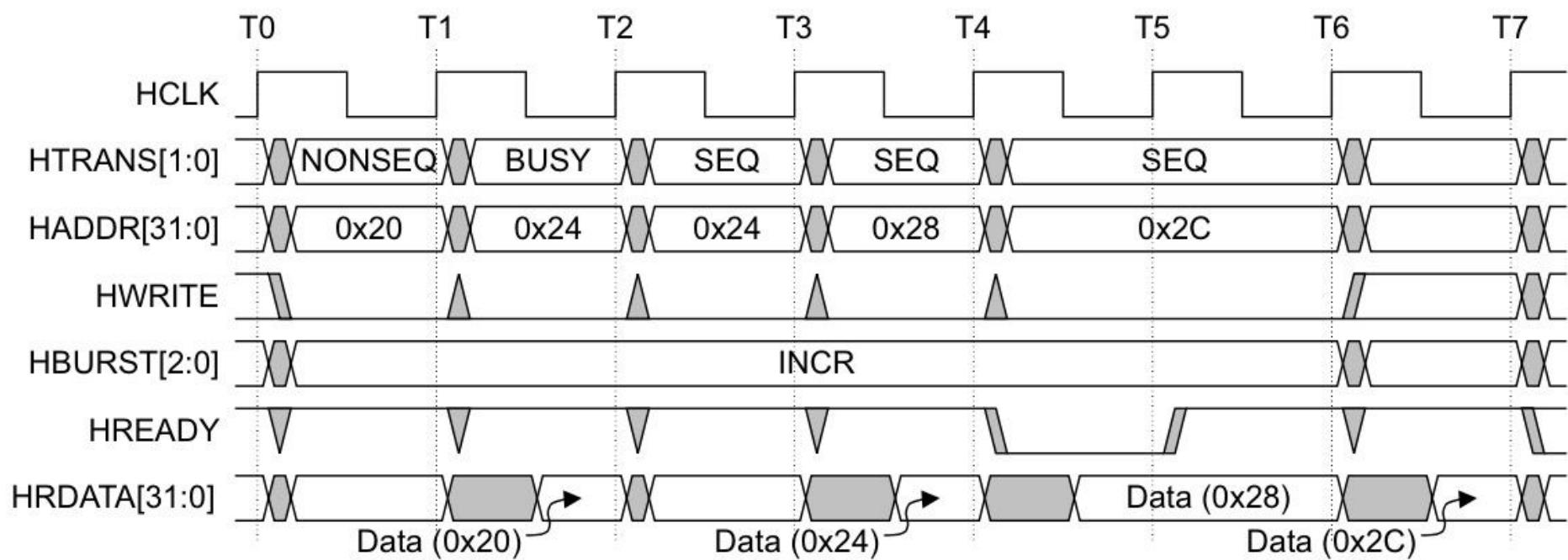
10 NONSEQ

Single transfer or first transfer of
a burst

11 SEQ

Remaining transfers of a burst

Transfer Type Examples



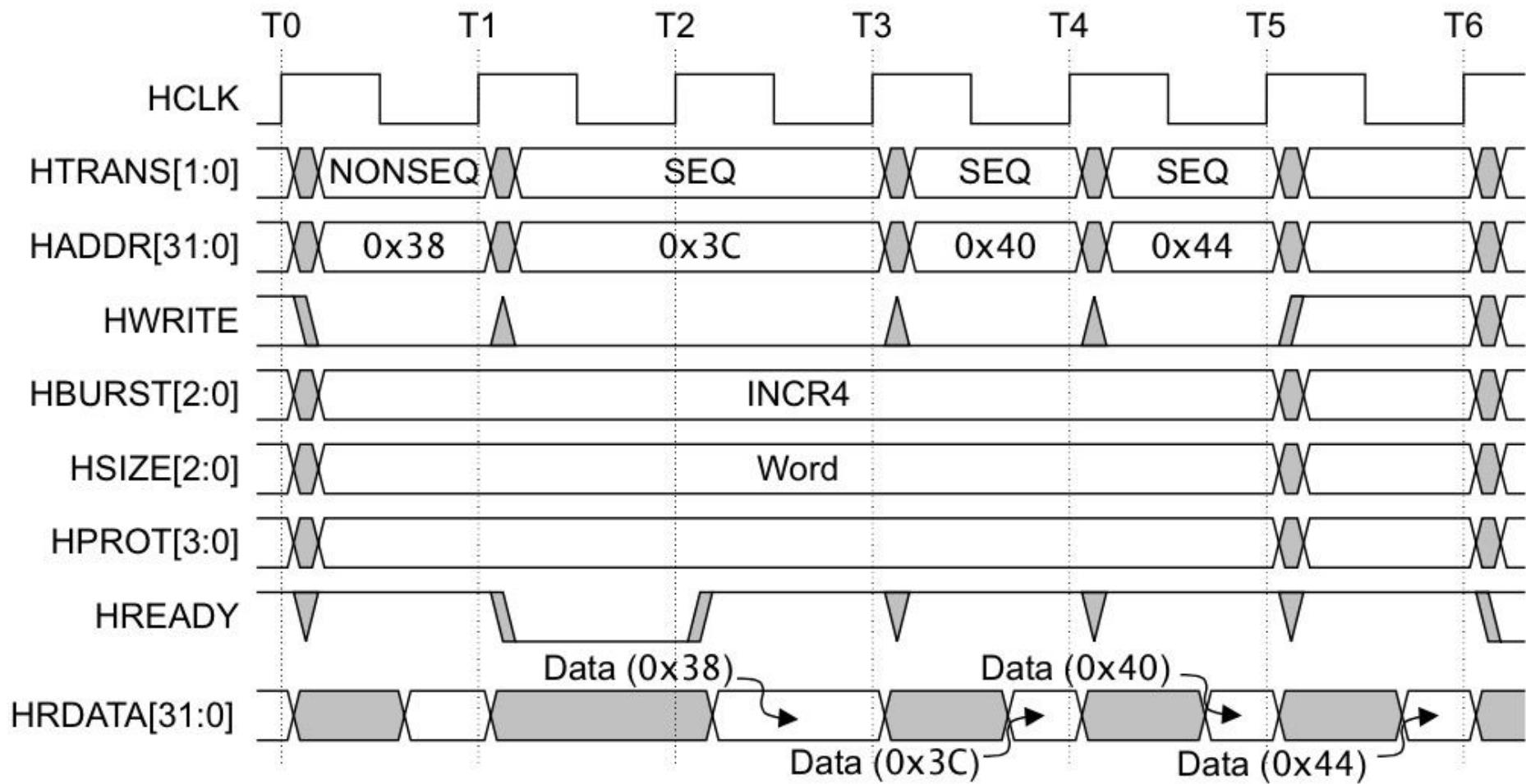
Transfer Size

HSIZE[2:0]	Size (bits)	Size (bytes)
000	8	1
001	16	2
010	32	4
011	64	8
100	128	16
101	256	32
110	512	64
111	1024	128

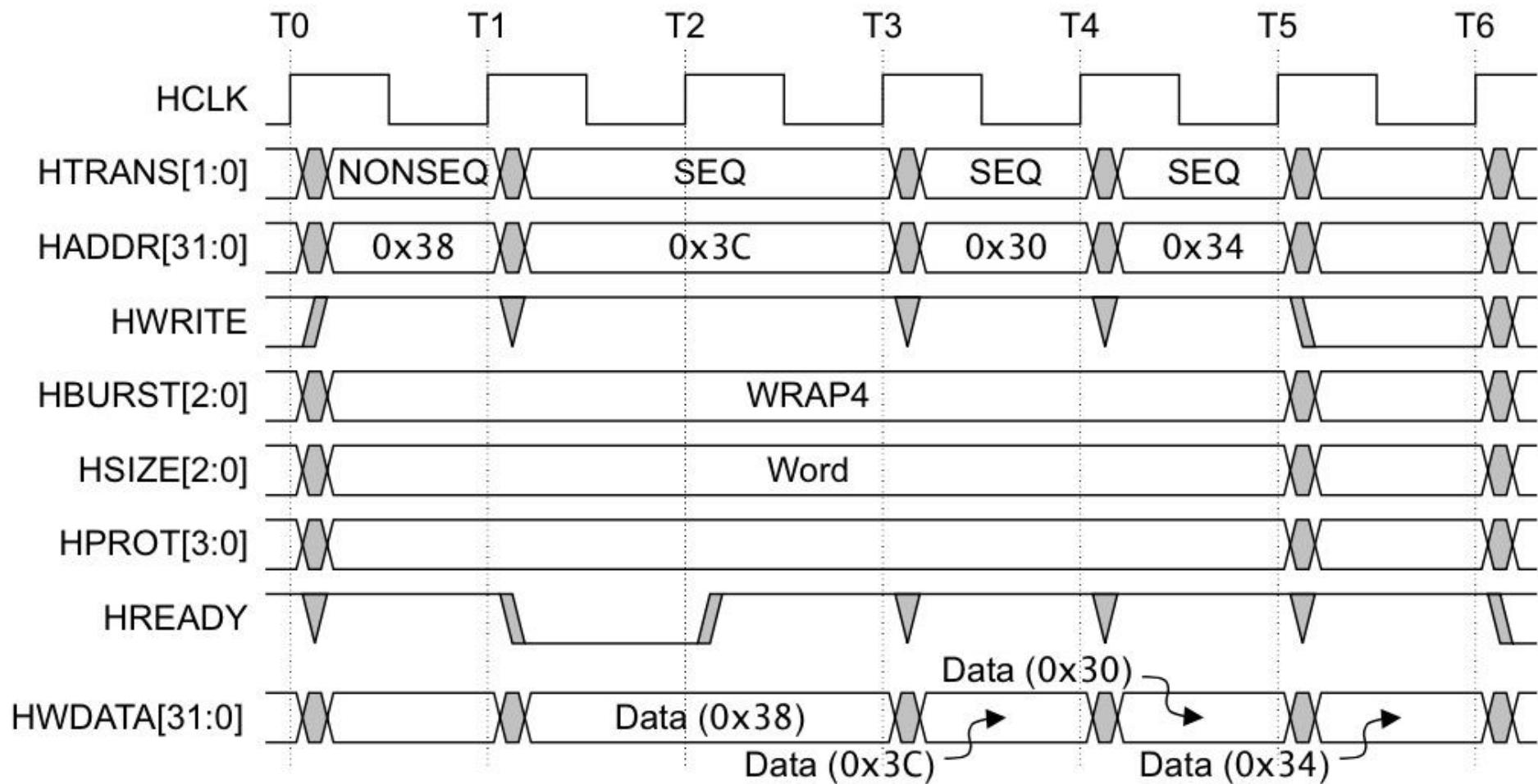
Burst type and size

HBURST[2:0]	Type and size
000	SINGLE
001	INCR
010	WRAP4
011	INCR4
100	WRAP8
101	INCR8
110	WRAP16
111	INCR16

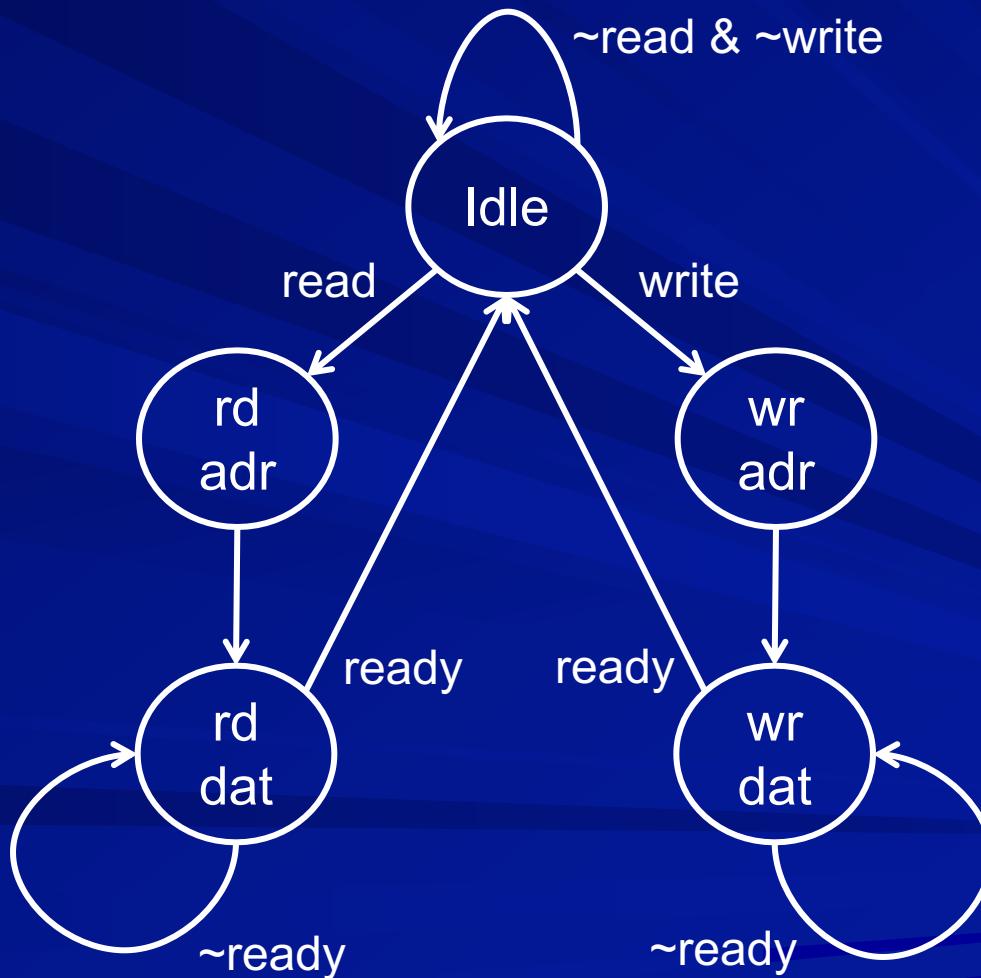
Four-Beat Incrementing Burst



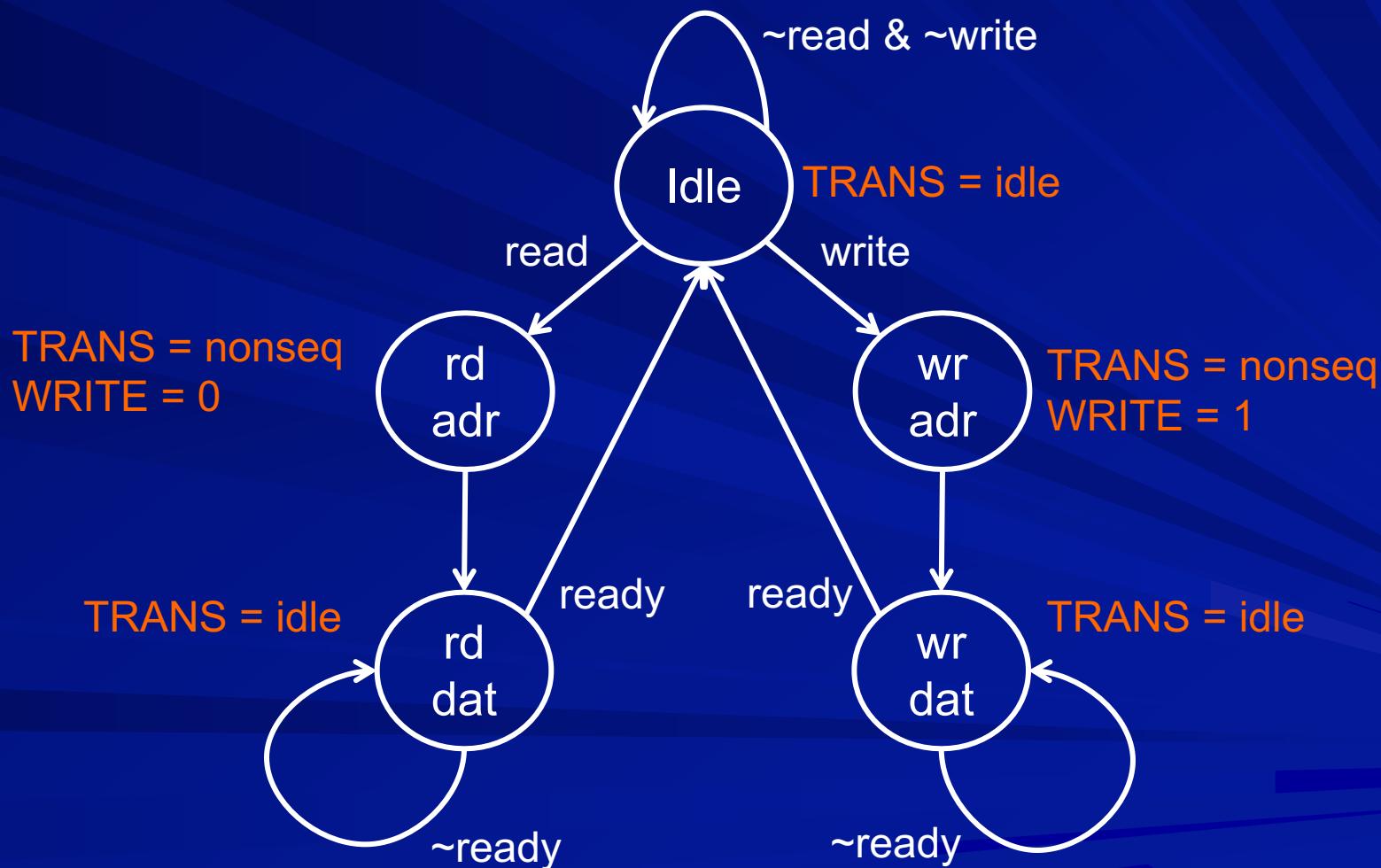
Four-Beat Wrapping Burst



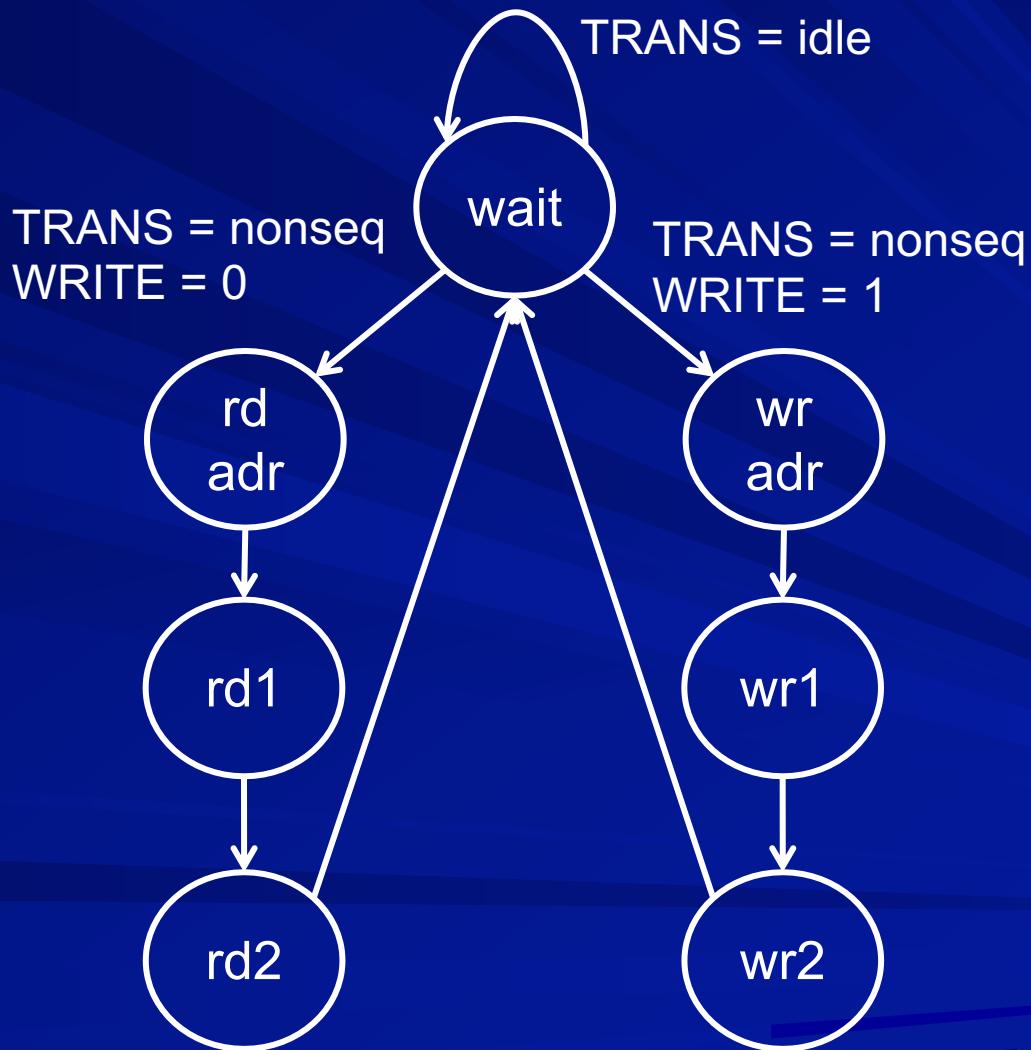
Master interface



Master interface



Slave interface



Obtaining access to a bus

Masters and slaves on a bus

masters (processors/peripherals) initiate transactions

slaves (peripherals/memories) respond to the masters

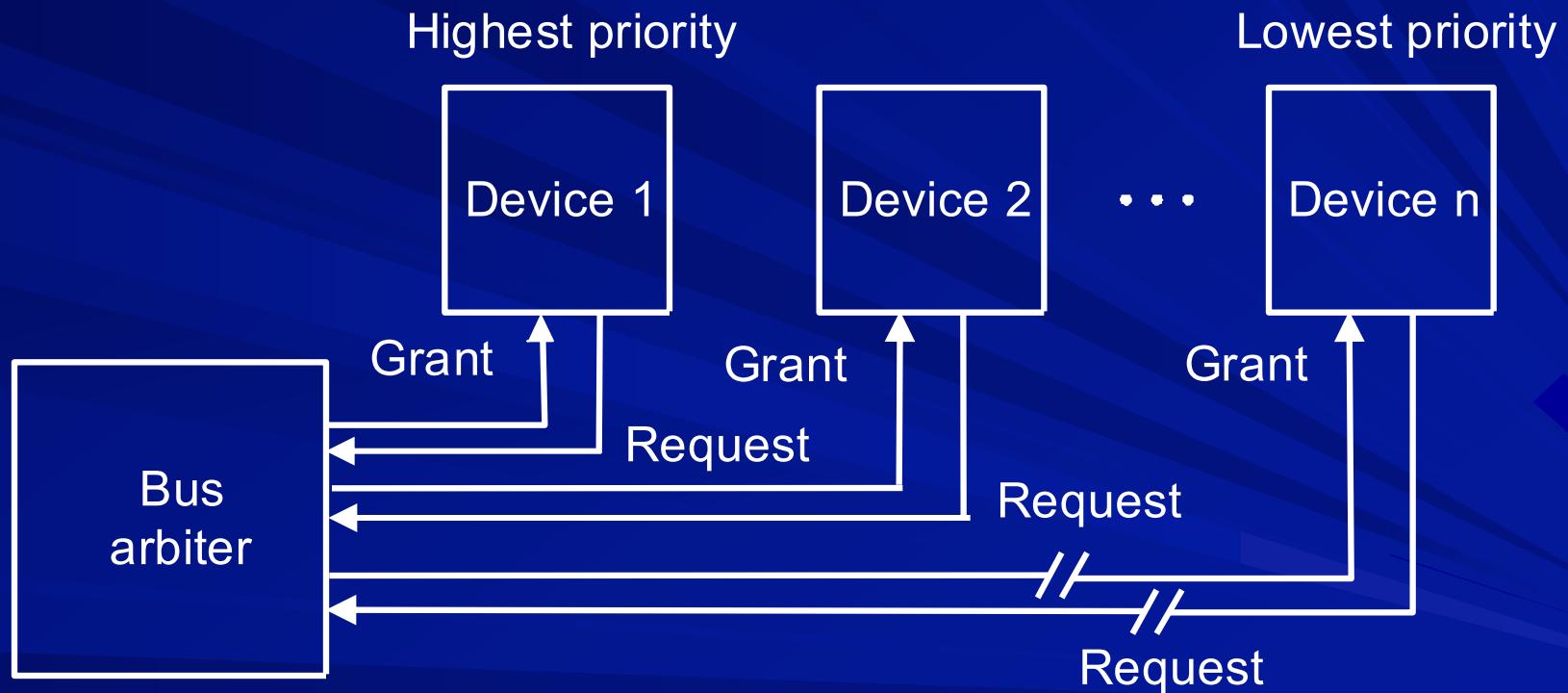
How does a master get control of a bus?

Priorities may be assigned. Fairness is essential.

- Daisy chaining
- Centralized parallel arbitration
- Distributed arbitration by self selection
- Distributed arbitration by collision detection

Centralized parallel arbitration

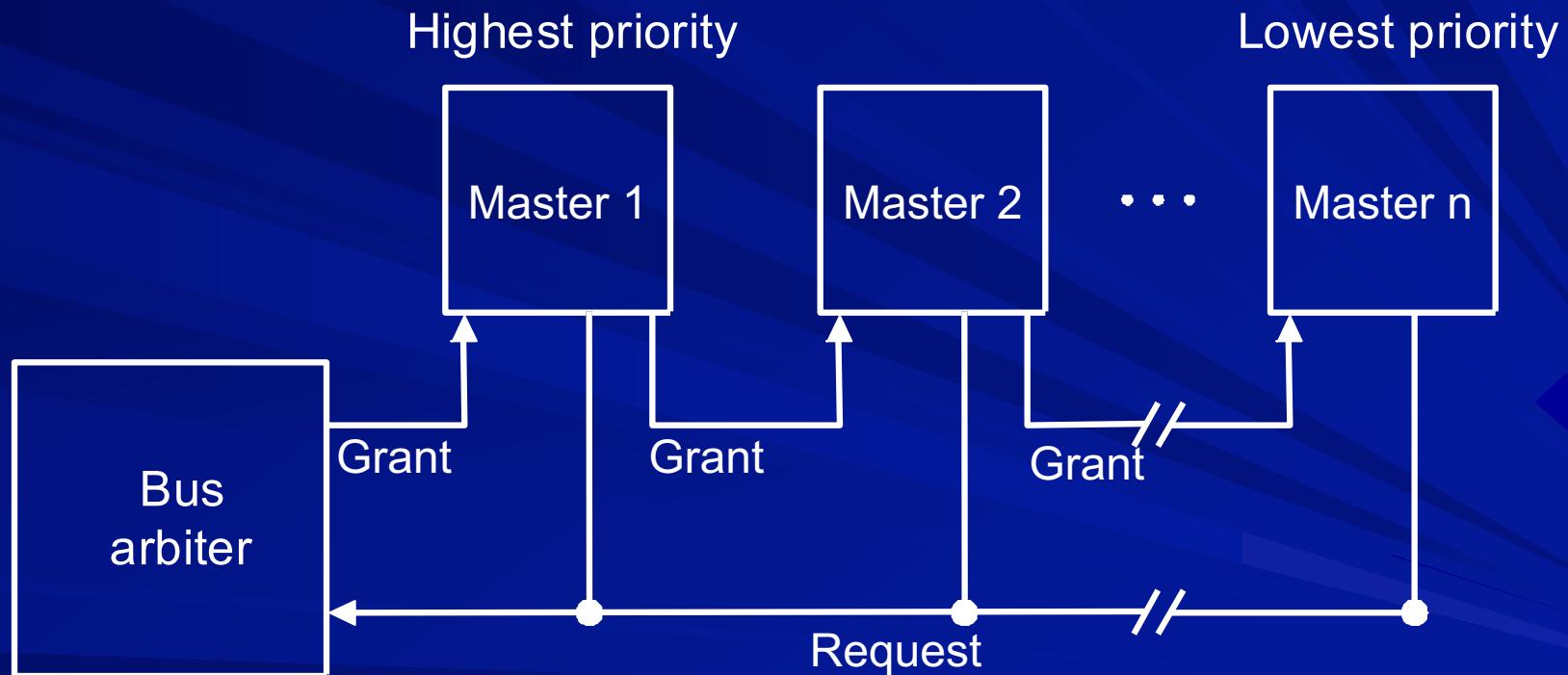
Example: PCI bus (a backplane bus)



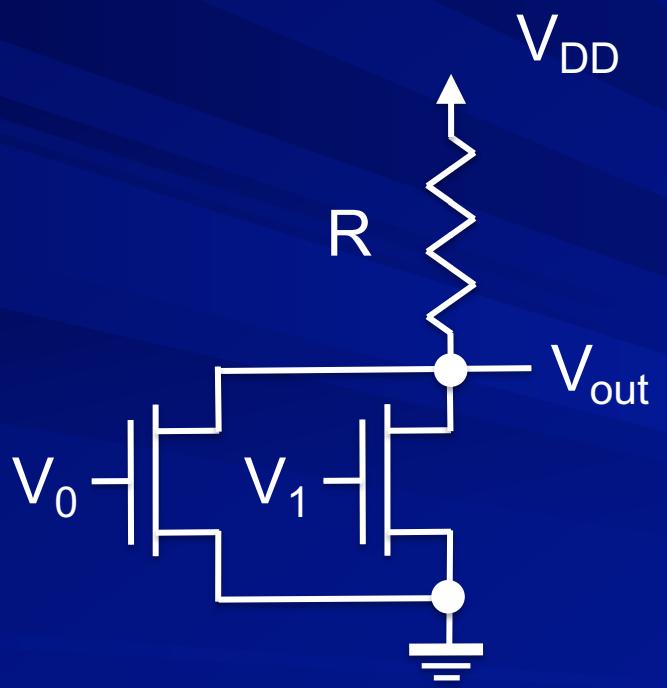
Request and Grant signals



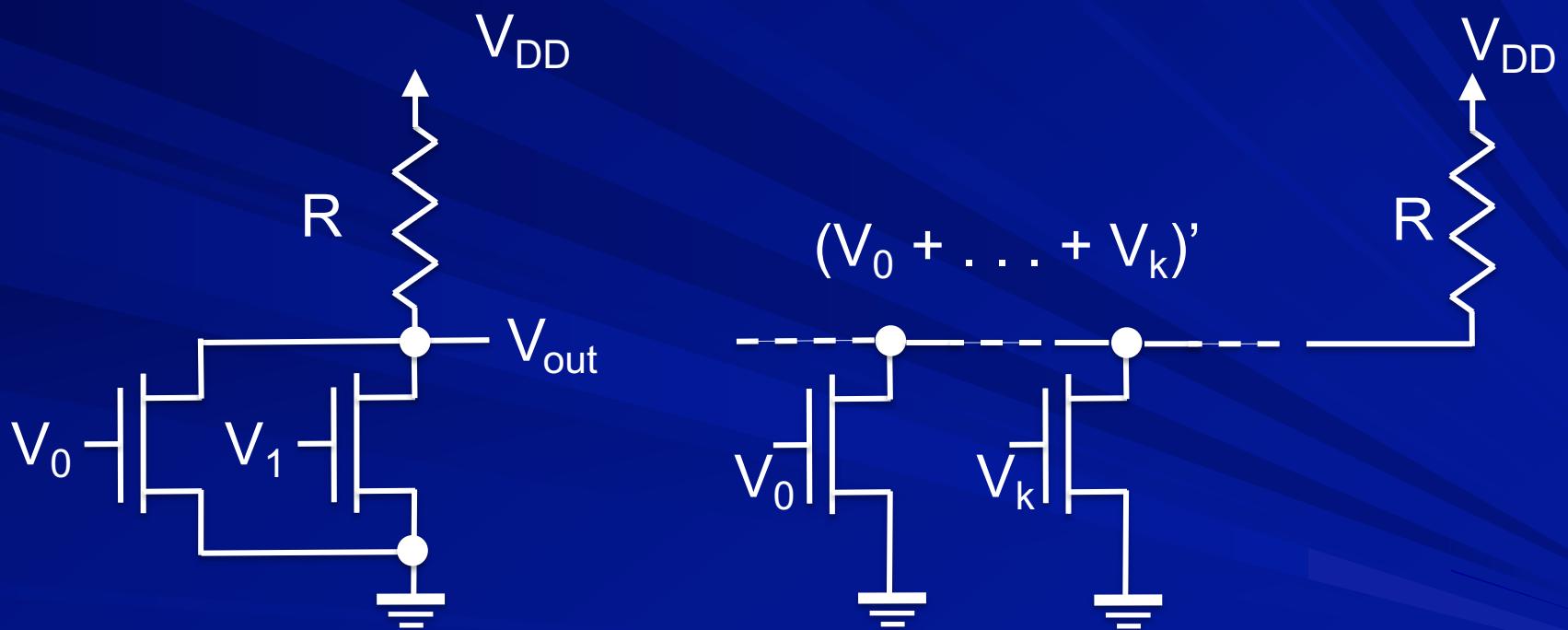
Combining/chaining signals



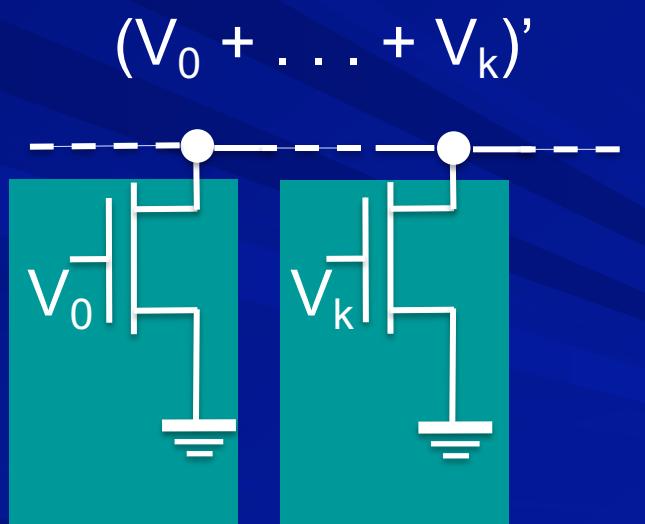
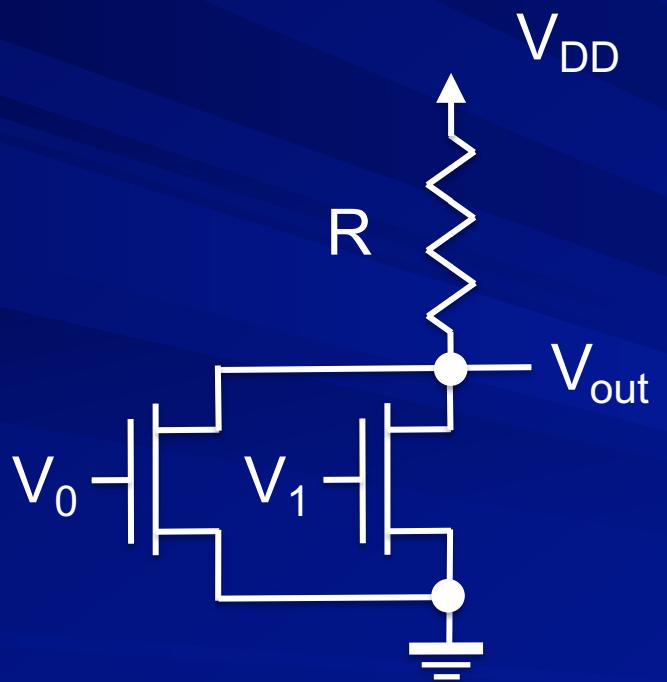
Wired OR



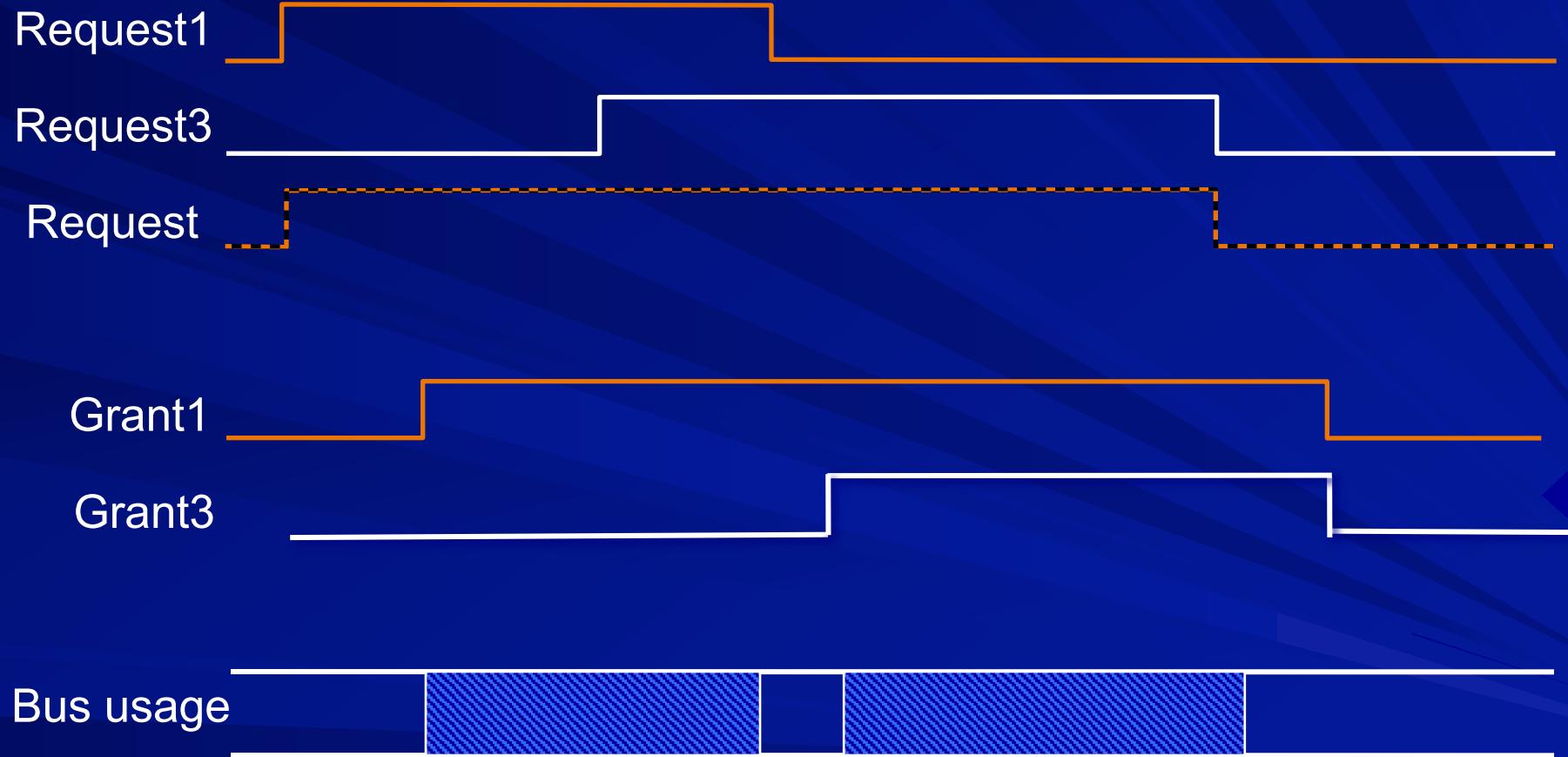
Wired OR



Wired OR

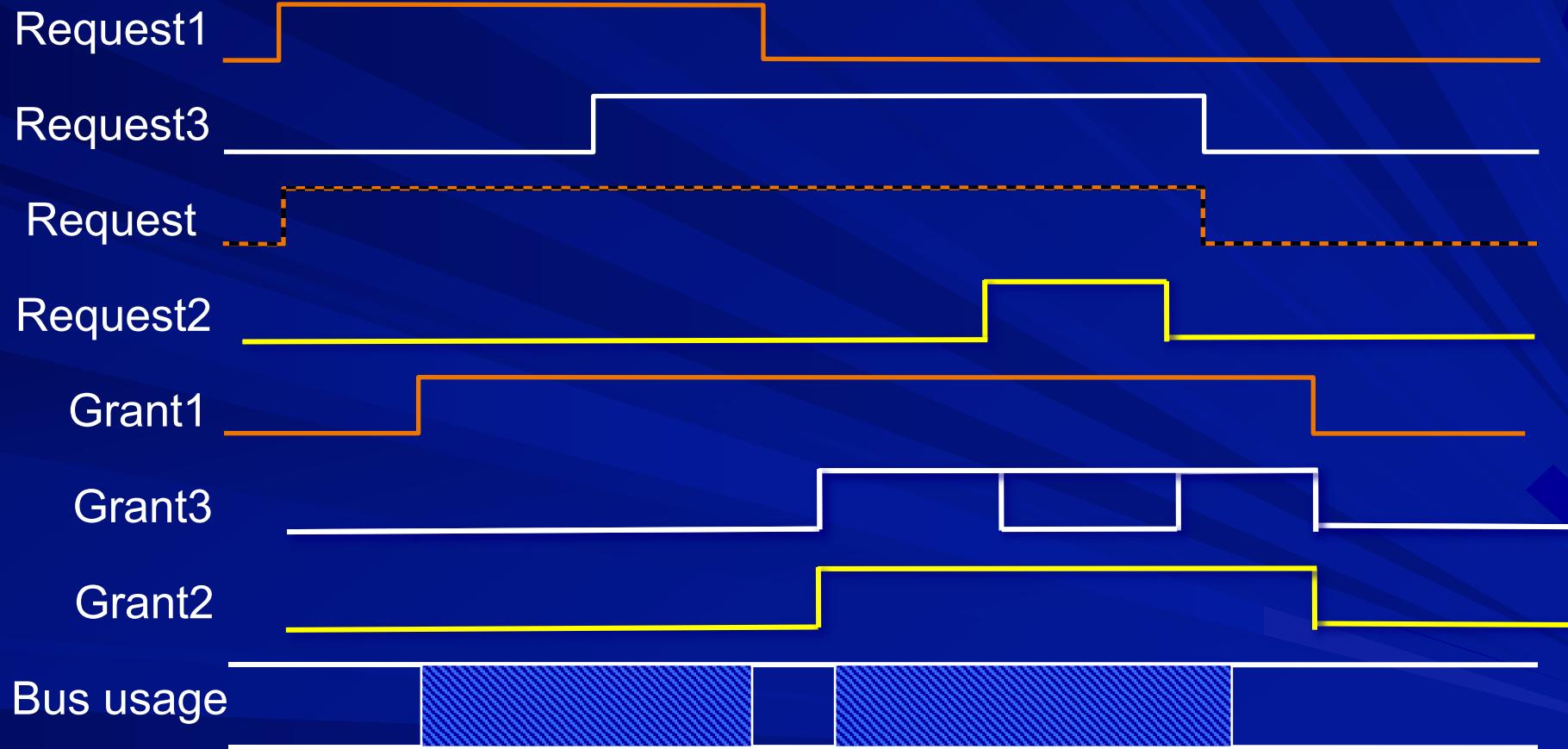


Request and Grant signals



Problem: High priority device can usurp the grant signal

Request and Grant signals



Problem: High priority device can usurp the grant signal
Solution: Daisy chain – request, grant, release

Daisy chain

Arbiter

Device Request ↑

Grant ↑

Request ↓

Use bus

Release ↑

Grant ↓

Release ↓



Highest priority

Device 1



Lowest priority

Device n

Example: VME bus

Grant

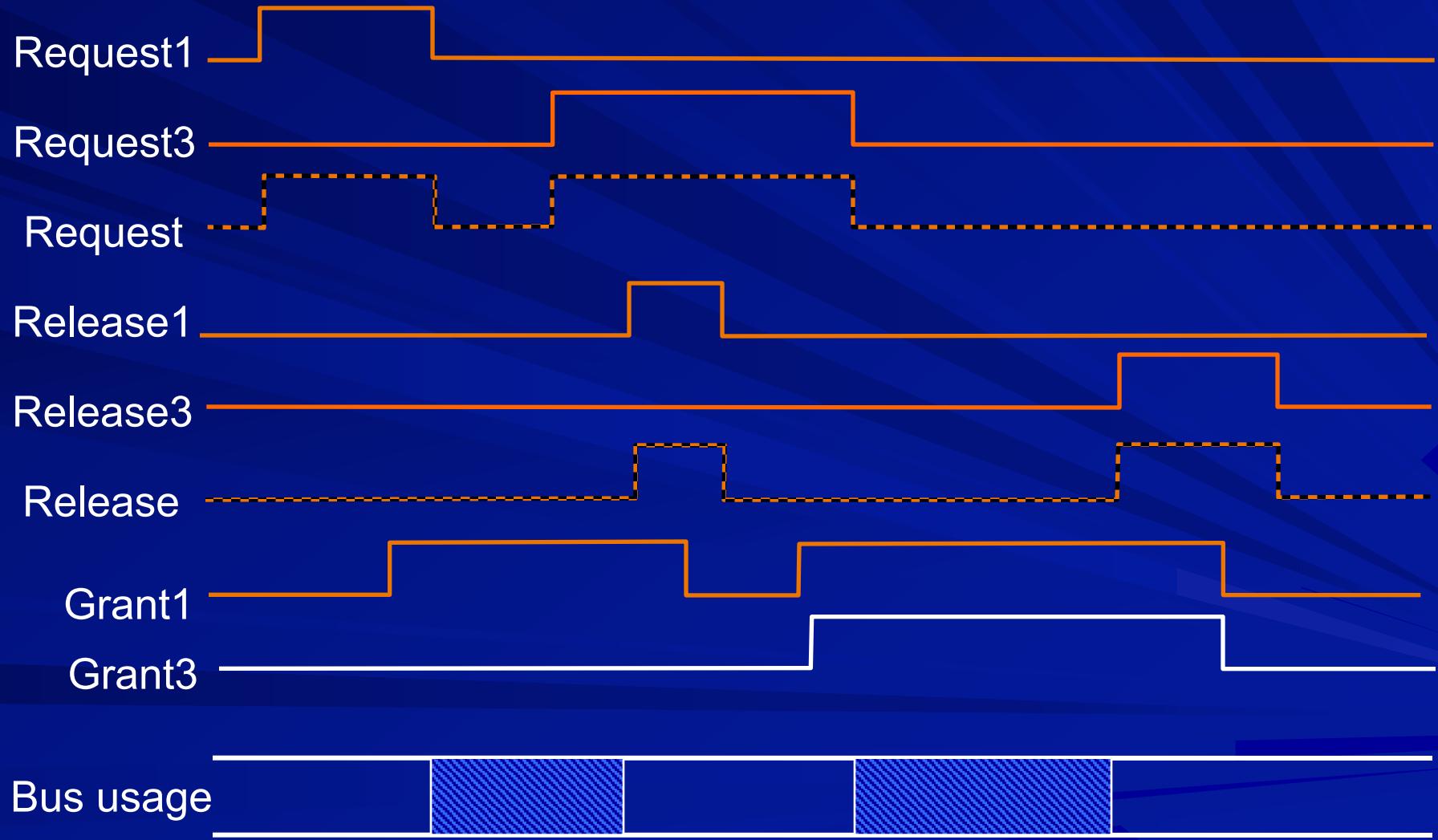
Grant

Release

Request

Note: Rising edge on Grant signal is significant, not level

Request, release and grant signals



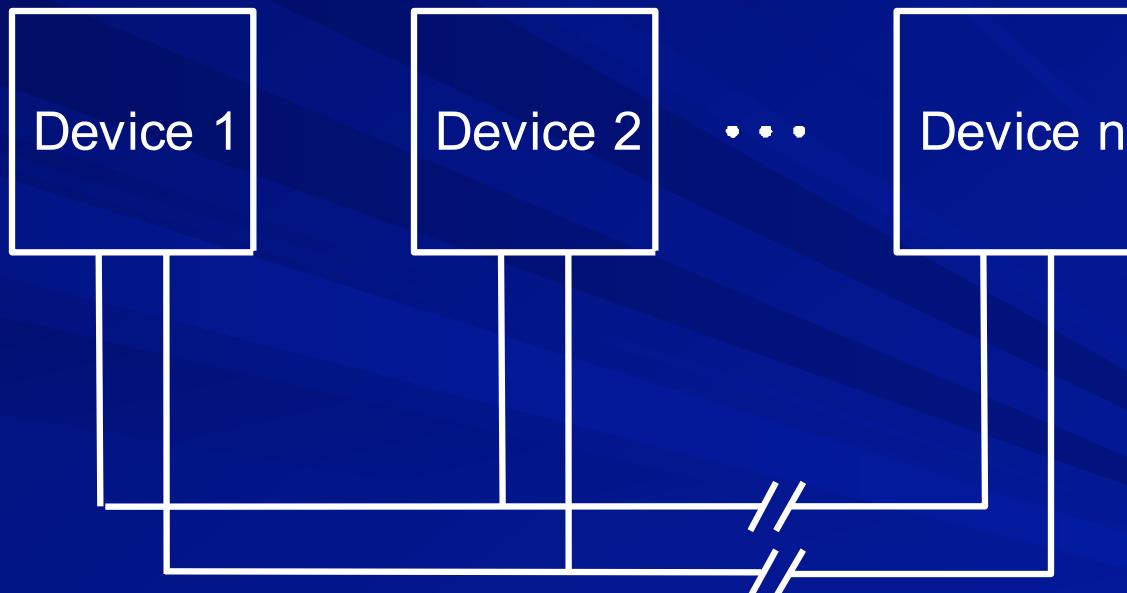
Daisy chain characteristics

- Simple and inexpensive
- Speed is limited due to chaining of grant signal
- Starvation of low priority devices may not be prevented

Rule to improve fairness: A device that has just used the bus cannot reacquire it until it sees the request line go low

Distributed arbitration

Example: NuBus (a backplane bus) in Apple Macintosh



Request lines and identity codes

Distributed arbitration in NuBus

- $\text{req} = \text{req}^0 + \text{req}^1 + \dots + \text{req}^{n-1}$ (wired OR)
- $\text{ID}_3^i \dots \text{ID}_0^i = \text{ID}$ of i^{th} requester
- $\text{arb}_3^i \dots \text{arb}_0^i = \text{arb}$ output of i^{th} requester
- $\text{arb}_{3..0} = \text{arb}_{3..0}^0 + \dots + \text{arb}_{3..0}^{n-1}$ (wired OR)
- $\text{arb}_3^i = \text{ID}_3^i \cdot \text{req}^i$
- $\text{arb}_2^i = \text{ID}_2^i \cdot \text{req}^i \cdot (\text{ID}_3^i + \text{arb}_3^i)$
- $\text{arb}_1^i = \text{ID}_1^i \cdot \text{req}^i \cdot (\text{ID}_3^i + \text{arb}_3^i) \cdot (\text{ID}_2^i + \text{arb}_2^i)$
- $\text{arb}_0^i = \text{ID}_0^i \cdot \text{req}^i \cdot (\text{ID}_3^i + \text{arb}_3^i) \cdot (\text{ID}_2^i + \text{arb}_2^i) \cdot (\text{ID}_1^i + \text{arb}_1^i)$

NuBus arbitration example-1

k	ID^k_3	ID^k_2	ID^k_1	ID^k_0	arb^k_3	arb^k_2	arb^k_1	arb^k_0
1	1	0	0	1				
2	0	1	1	0				
3	1	1	0	0				
4	0	0	1	1				

NuBus arbitration example-1

k	ID^k_3	ID^k_2	ID^k_1	ID^k_0	arb^k_3	arb^k_2	arb^k_1	arb^k_0
1	1	0	0	1	1	0	0	1
2	0	1	1	0	0	1	1	0
3	1	1	0	0	1	1	0	0
4	0	0	1	1	0	0	1	1
<hr/>								
OR					1	1	1	1

NuBus arbitration example-1

k	ID^k_3	ID^k_2	ID^k_1	ID^k_0	arb^k_3	arb^k_2	arb^k_1	arb^k_0
1	1	0	0	1	1			
2	0	1	1	0	0			
3	1	1	0	0	1			
4	0	0	1	1	0			

NuBus arbitration example-1

k	ID^k_3	ID^k_2	ID^k_1	ID^k_0	arb^k_3	arb^k_2	arb^k_1	arb^k_0
1	1	0	0	1	1	0		
2	0	1	1	0	0	x		
3	1	1	0	0	1	1		
4	0	0	1	1	0	x		
					1	1		

NuBus arbitration example-1

k	ID^k_3	ID^k_2	ID^k_1	ID^k_0	arb^k_3	arb^k_2	arb^k_1	arb^k_0
1	1	0	0	1	1	0	x	
2	0	1	1	0	0	x	x	
3	1	1	0	0	1	1	0	
4	0	0	1	1	0	x	x	
					1	1	0	

NuBus arbitration example-1

k	ID^k_3	ID^k_2	ID^k_1	ID^k_0	arb^k_3	arb^k_2	arb^k_1	arb^k_0
1	1	0	0	1	1	0	x	x
2	0	1	1	0	0	x	x	x
3	1	1	0	0	1	1	0	0
4	0	0	1	1	0	x	x	x
<hr/>								
					1	1	0	0

NuBus arbitration example-2

k	ID^k_3	ID^k_2	ID^k_1	ID^k_0	arb^k_3	arb^k_2	arb^k_1	arb^k_0
1	1	0	0	1				
2	0	1	1	0				
3	1	0	1	0				
4	0	0	1	1				

NuBus arbitration example-2

k	ID^k_3	ID^k_2	ID^k_1	ID^k_0	arb^k_3	arb^k_2	arb^k_1	arb^k_0
1	1	0	0	1	1	0	0	1
2	0	1	1	0	0	1	1	0
3	1	0	1	0	1	0	1	0
4	0	0	1	1	0	0	1	1

OR

1 1 1 1

NuBus arbitration example-2

k	ID^k_3	ID^k_2	ID^k_1	ID^k_0	arb^k_3	arb^k_2	arb^k_1	arb^k_0
1	1	0	0	1	1			
2	0	1	1	0	0			
3	1	0	1	0	1			
4	0	0	1	1	0			

NuBus arbitration example-2

k	ID^k_3	ID^k_2	ID^k_1	ID^k_0	arb^k_3	arb^k_2	arb^k_1	arb^k_0
1	1	0	0	1	1	0		
2	0	1	1	0	0	x		
3	1	0	1	0	1	0		
4	0	0	1	1	0	x		
					1	0		

NuBus arbitration example-2

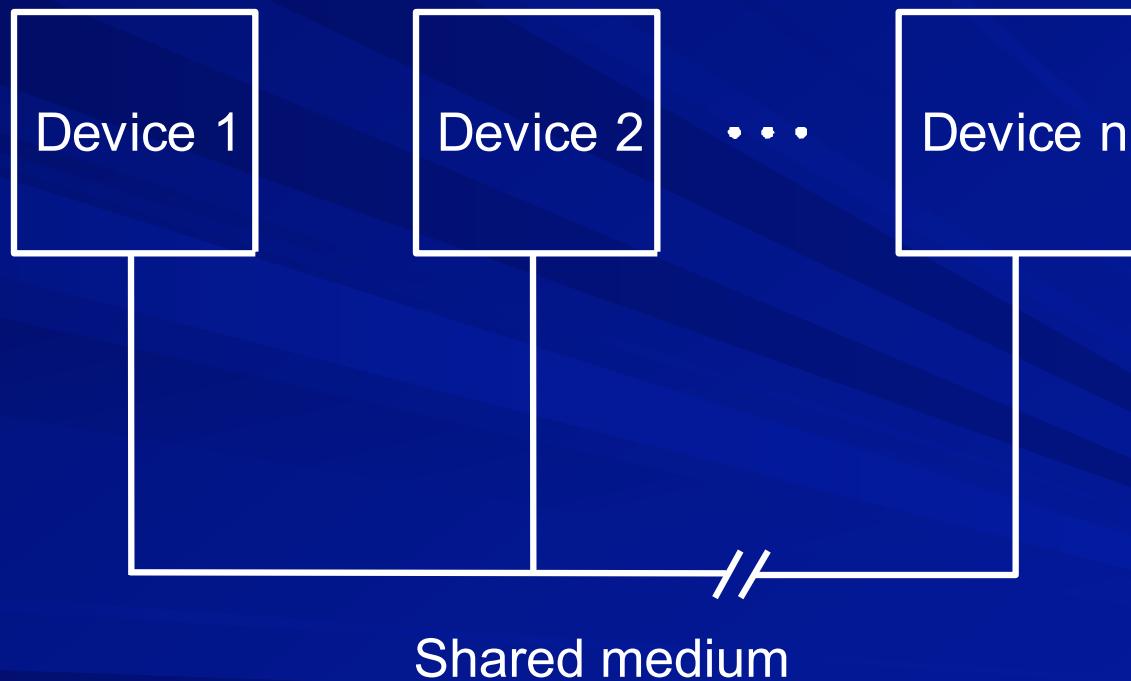
k	ID^k_3	ID^k_2	ID^k_1	ID^k_0	arb^k_3	arb^k_2	arb^k_1	arb^k_0
1	1	0	0	1	1	0	0	
2	0	1	1	0	0	x	x	
3	1	0	1	0	1	0	1	
4	0	0	1	1	0	x	x	
					1	0	1	

NuBus arbitration example-2

k	ID^k_3	ID^k_2	ID^k_1	ID^k_0	arb^k_3	arb^k_2	arb^k_1	arb^k_0
1	1	0	0	1	1	0	0	x
2	0	1	1	0	0	x	x	x
3	1	0	1	0	1	0	1	0
4	0	0	1	1	0	x	x	x
<hr/>								
					1	0	1	0

Arbitration by collision detection

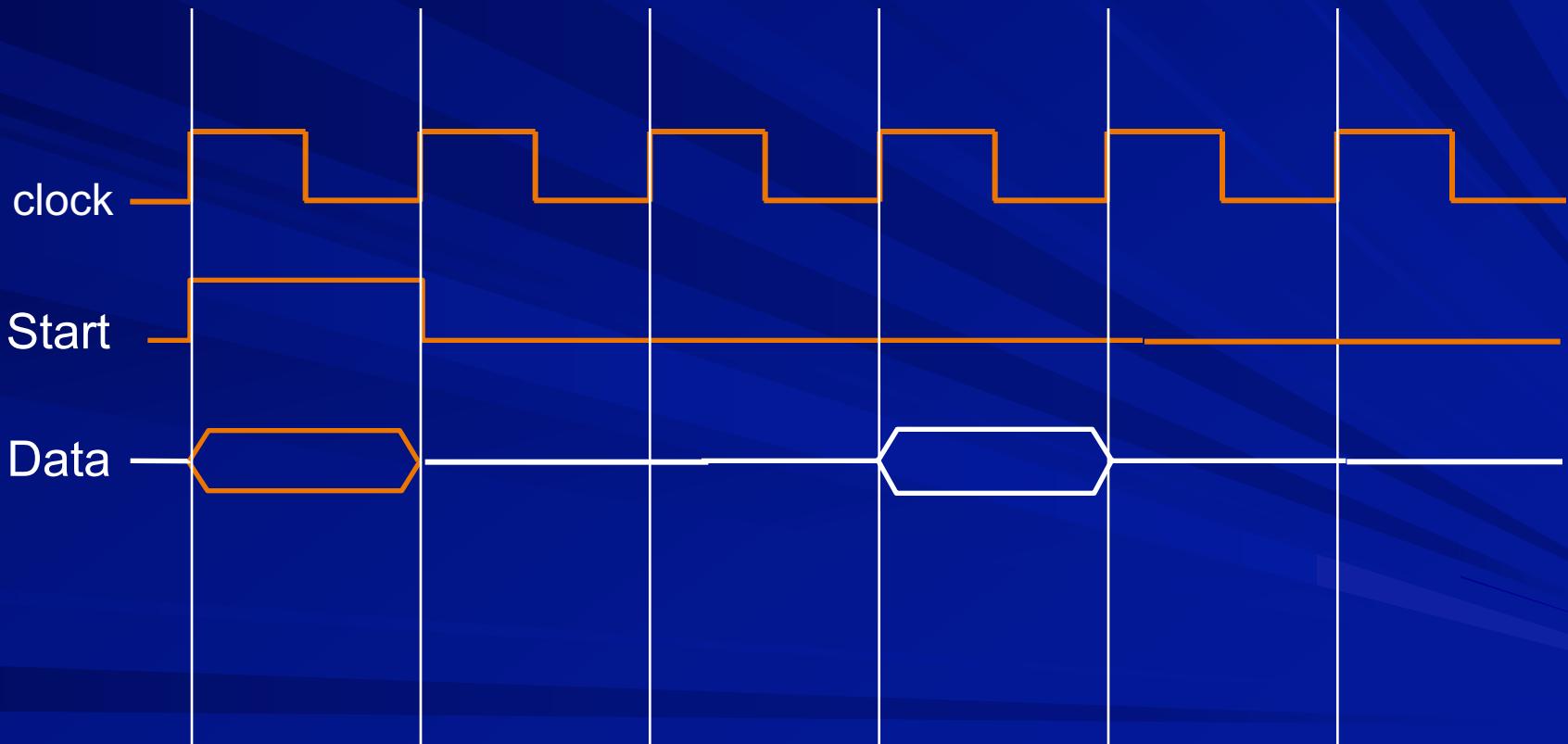
Example: Ethernet



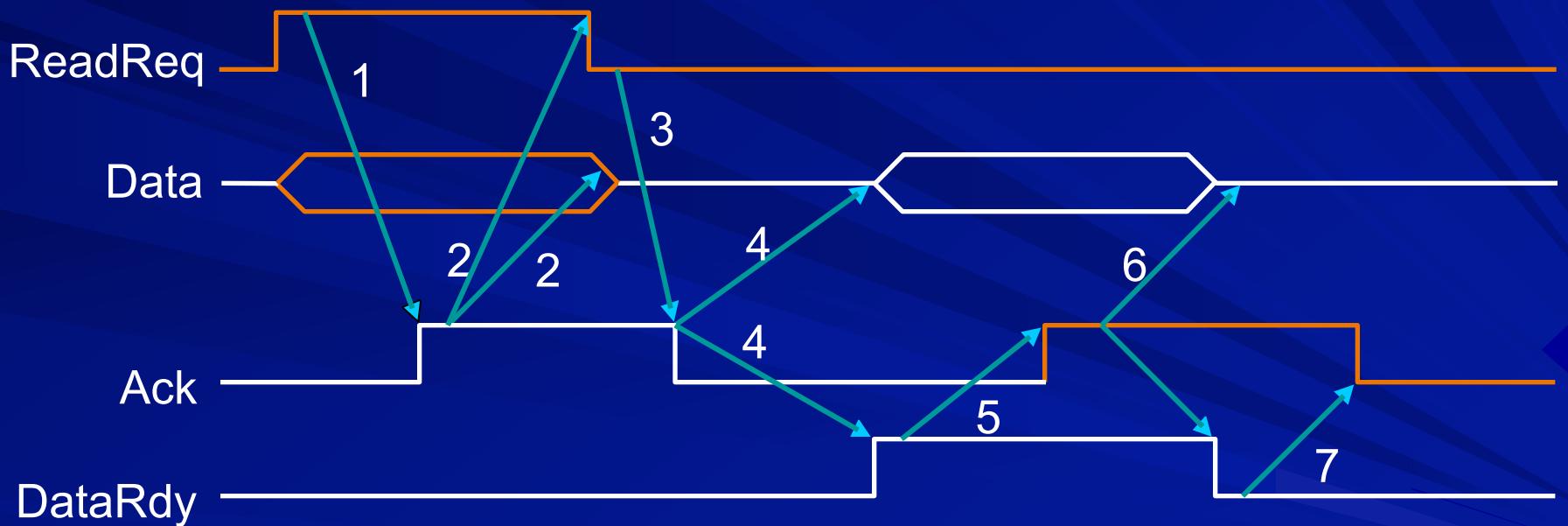
Increasing the bus bandwidth

- increase bus width
- separate data and address lines
- pipelining
- multiple word blocks
- synchronous
- split transaction

Synchronous transfer



Asynchronous handshaking



THANKS

COL216

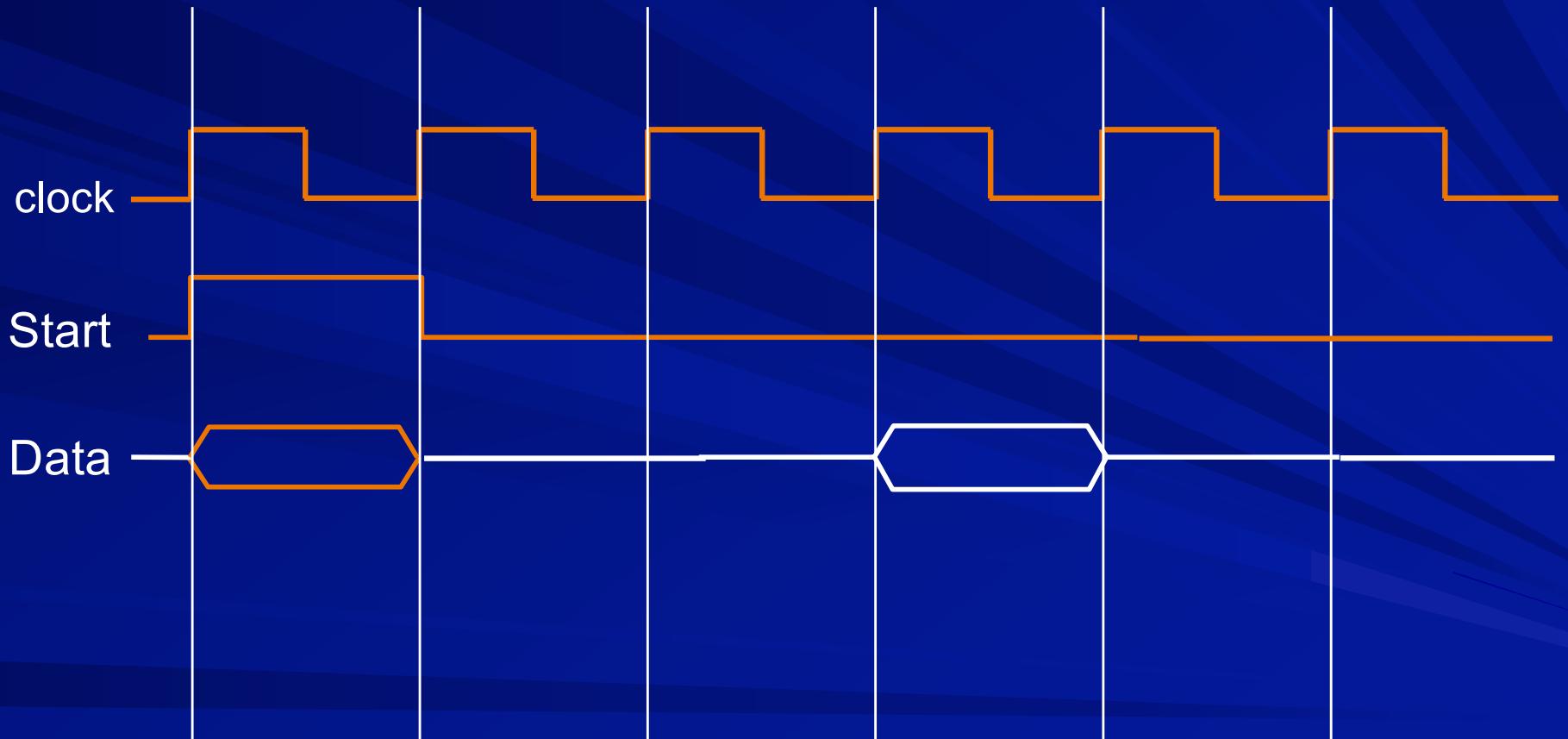
Computer Architecture

Input/Output – 5
Parallel and Serial Buses
28th March 2022

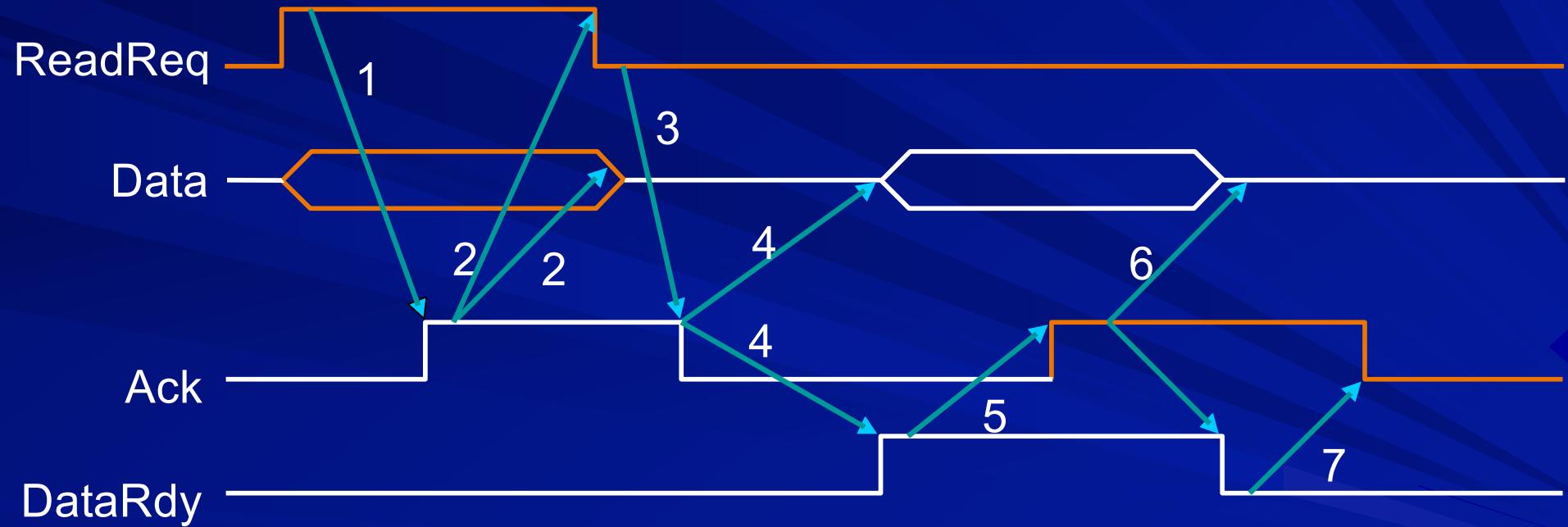
Increasing the bus bandwidth

- increase bus width
- separate data and address lines
- pipelining
- multiple word blocks
- synchronous
- split transaction

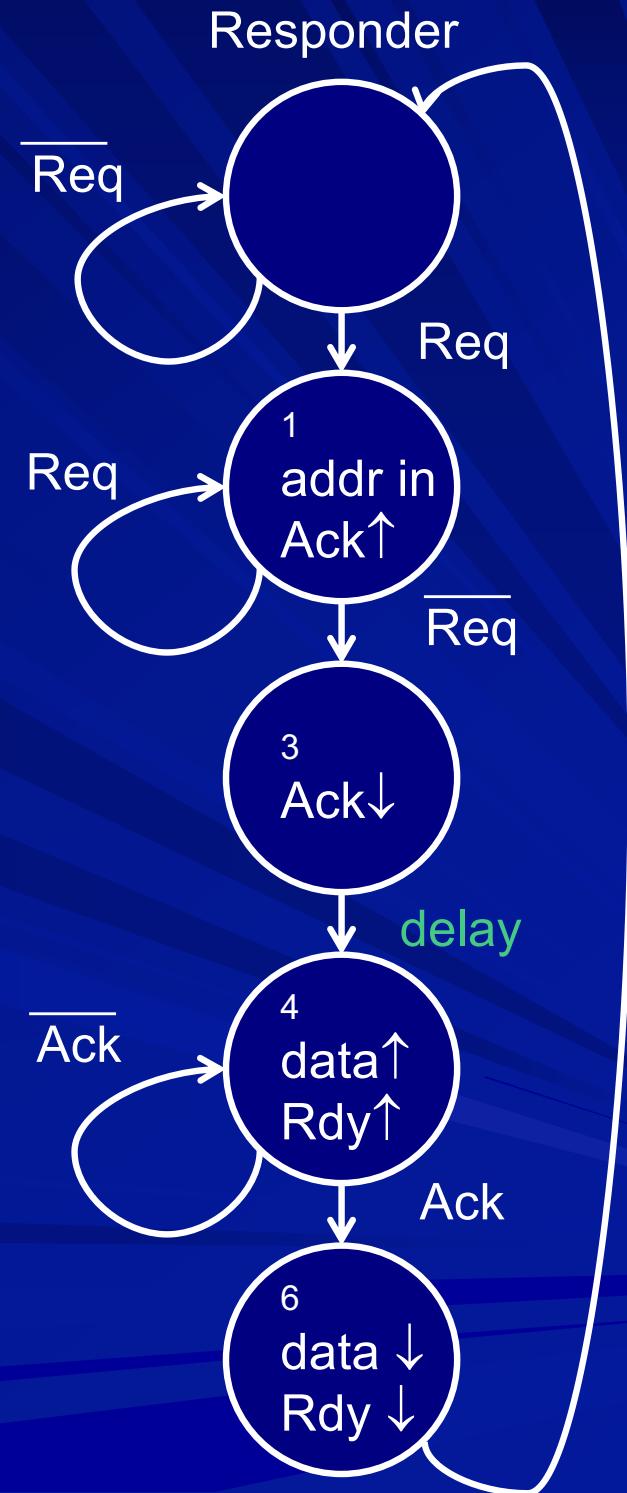
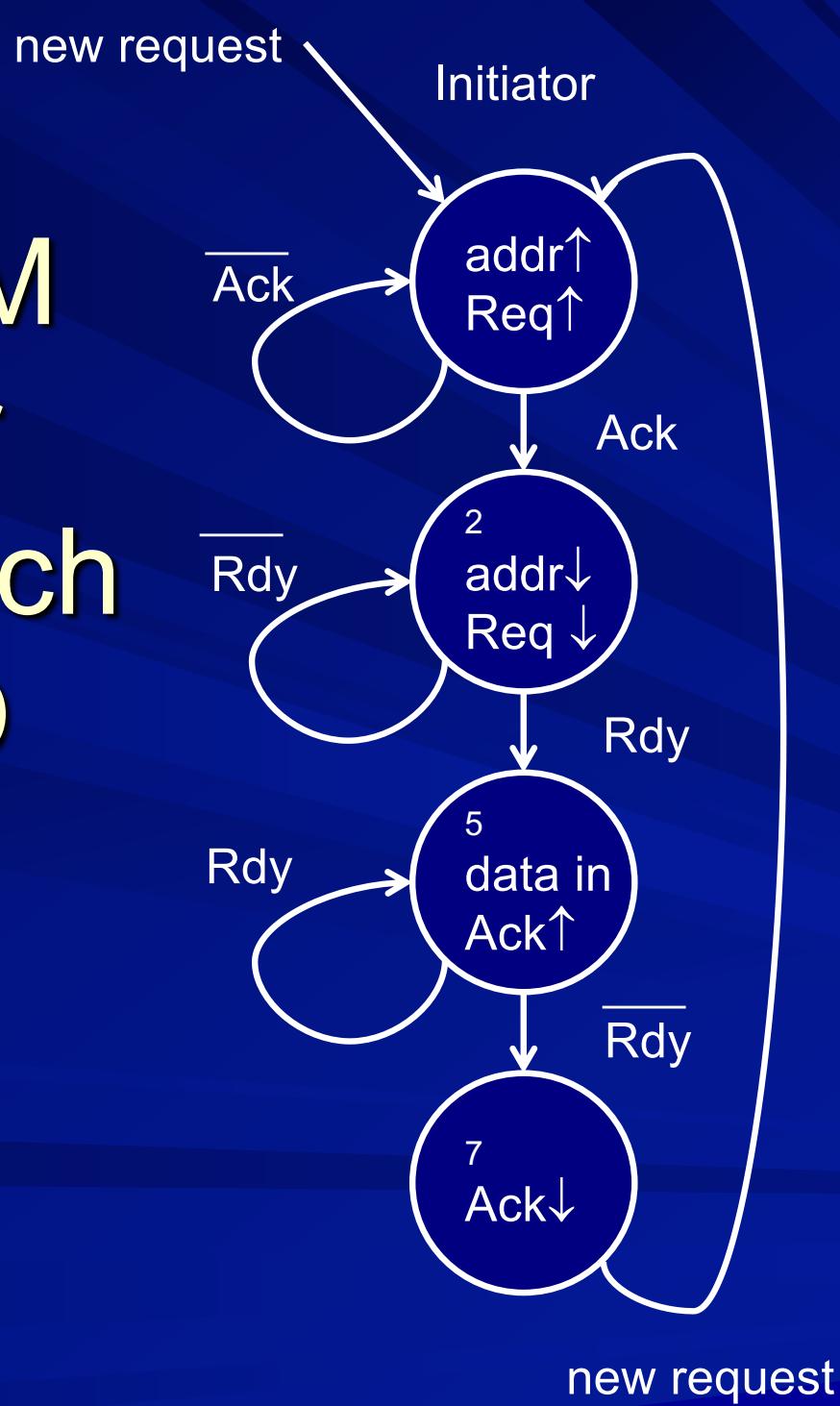
Synchronous transfer



Asynchronous handshaking



FSM for asynch I/O



Split transactions



Parallel vs Serial

■ Parallel

- multiple wires carry bits in parallel : 8-bit, 16-bit, 32-bit, 64-bit . . .
- address, read data, write data may use same wires or separate wires

■ Serial

- only one bit at a time
- read data, write data may use same wires or separate wires, no separate address wires

Series/parallel bus examples

Parallel

- ISA (Industry Standard Architecture)
- EISA (Extended ISA)
- VLB (VESA (Video Electronics Standards Association) Local Bus)
- PCI (Peripheral Component Interconnect)
- AGP (Accelerated Graphics Port)

Serial

- USB (Universal Serial Bus)
- Fire wire
- Fibre channel
- PCIe (Peripheral Component Interconnect express)
- SATA (Serial Advanced Technology Attachment)

Why serial buses?

- Serial buses are less expensive
- At high speed, keeping all the bits synchronized in parallel buses is very difficult
- Longer the signals travel, more skew is likely among the bits of parallel buses

Move from parallel to serial

PCI Bus

- PCI 1.0: 33MHz, 32-bit, peak BW 133MB/s
- PCI 2.2: 66 MHz, 64-bit, BW 533 MB/s
- PCI-X: 133MHz, 64-bit, BW 1066 MB/s
- PCI-X 2.0: 266MHz, BW 2133 MB/s

PCI-Express (serial bus), 1-16 lanes

- PCIe 1.0: 250 MB/s per lane
- PCIe 2.0: 500 MB/s per lane
- PCIe 3.0: 985 MB/s per lane

Move from parallel to serial

Parallel ATA

33 MB/s =>

66 MB/s =>

100 MB/s =>

133 MB/s

Serial ATA

150 MB/s =>

300 MB/s =>

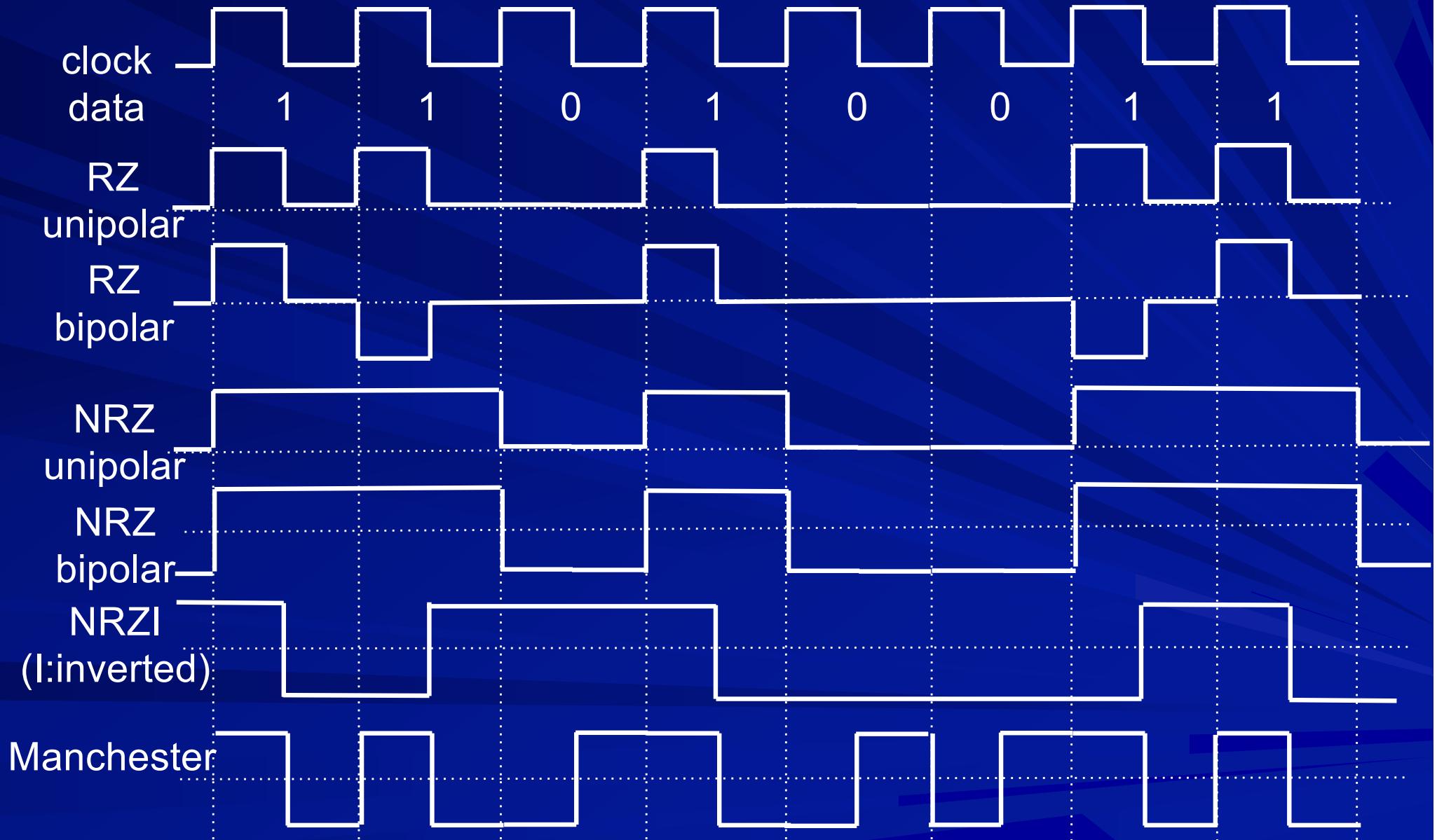
600 MB/s =>

1969 MB/s

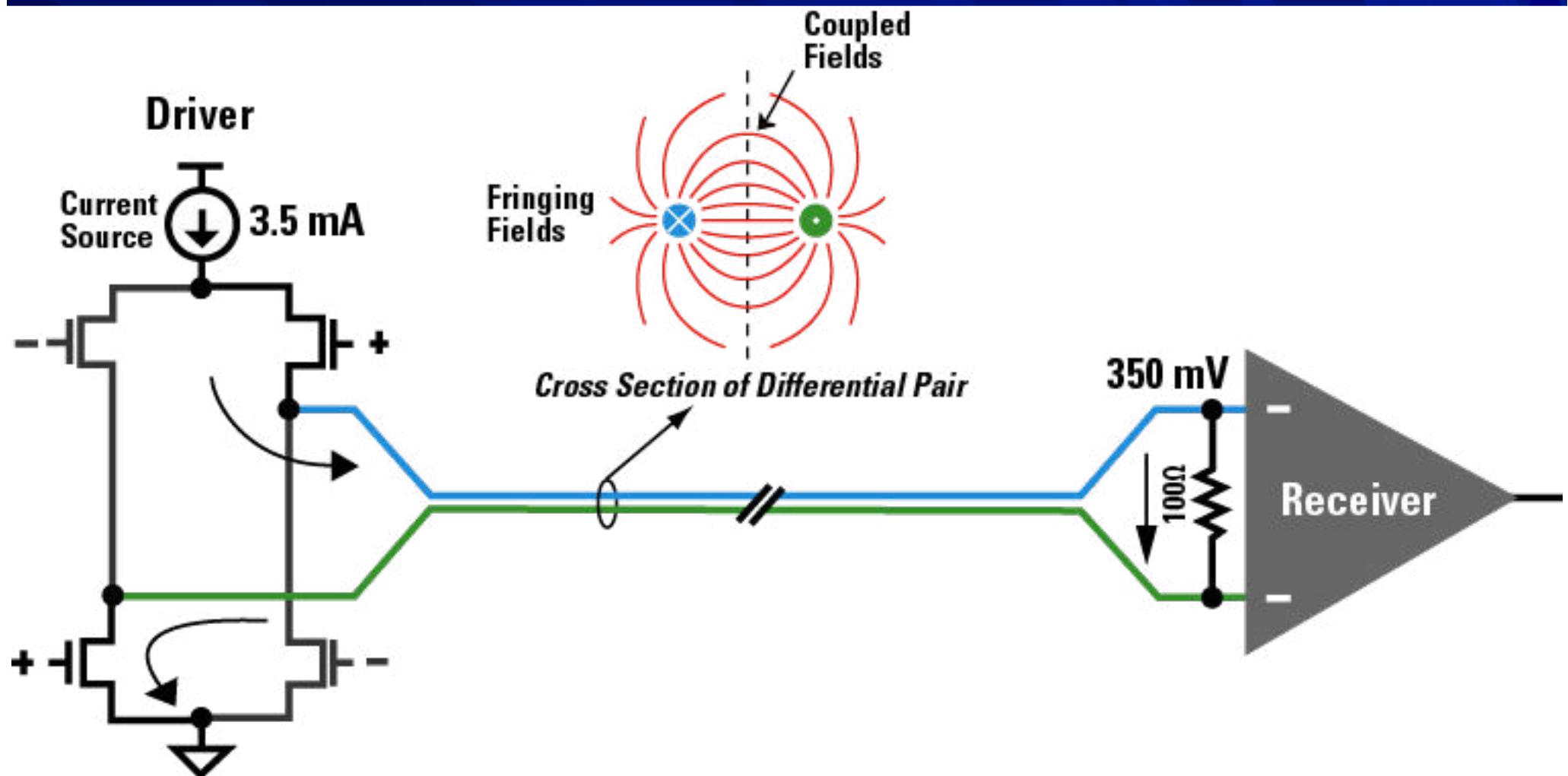
Representing bits

- Unipolar / bipolar
- Active high / active low
- Return to zero (RZ) / Non return to zero (NRZ)
- Encode using levels or transitions
- One wire / two wire (differential)

Representing bits



Low voltage differential signalling (LVDS)

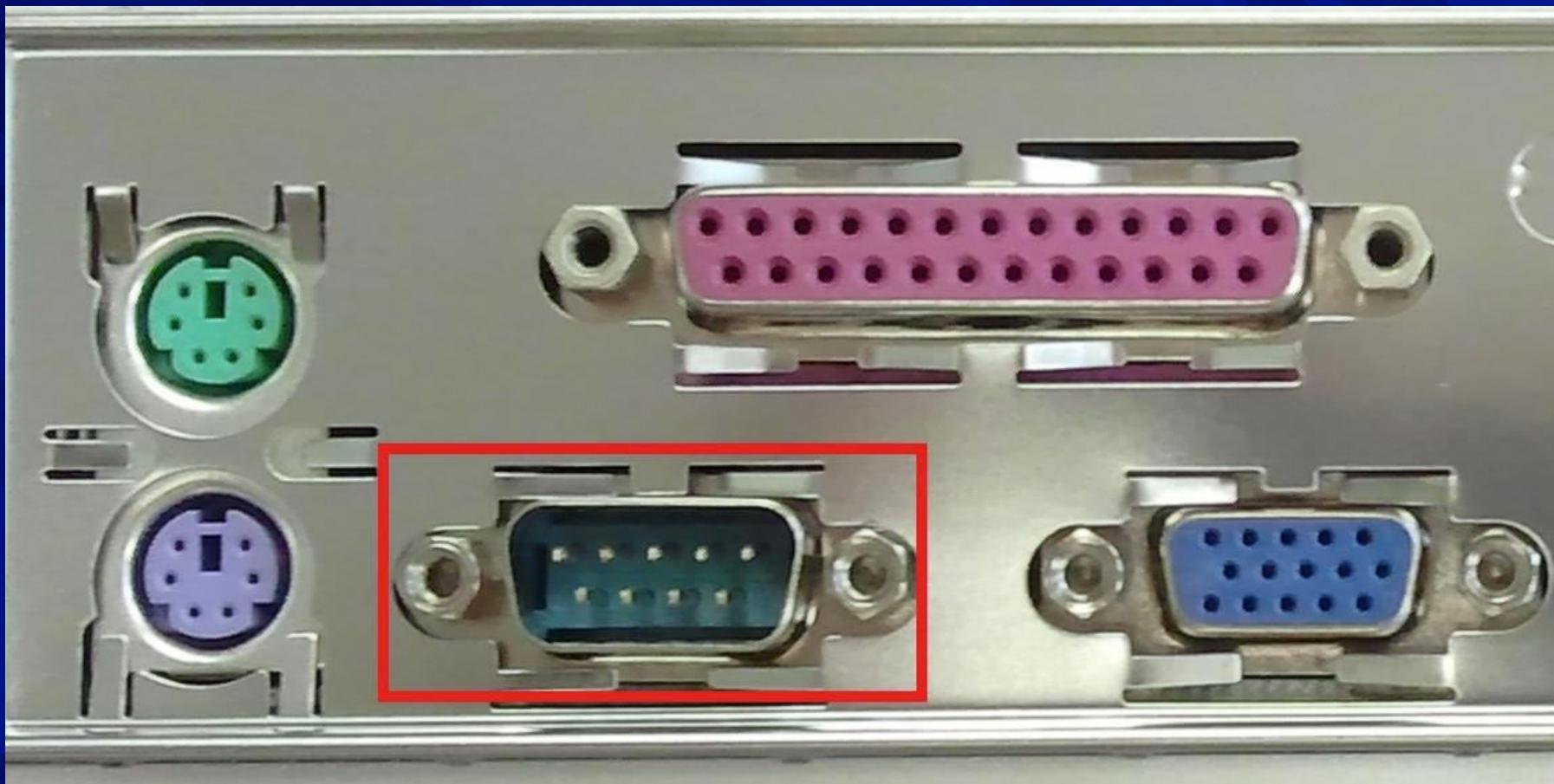


from wikipedia

Timing in serial transfer

- Synchronous
 - shared clock
 - same frequency and phase
- Mesochronous
 - transfer clock signal, separately or with data
 - same frequency but not phase
- Plesiochronous
 - locally generated clock
 - nearly same frequency, changing phase

Serial port / COM port



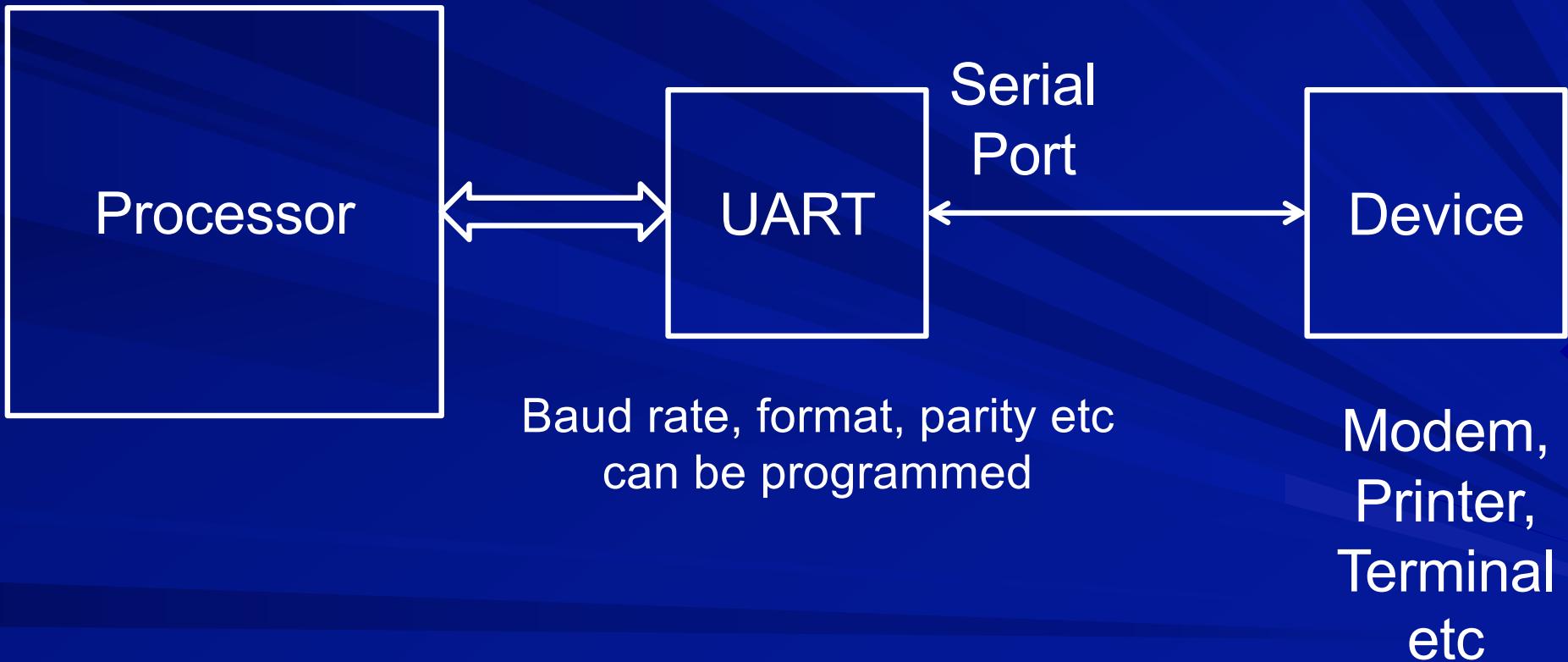
RS232 has been a common standard
75 to 115.2 K bits/s

RS232 Interface Standard



UART

(Universal Asynchronous
Receiver Transmitter)



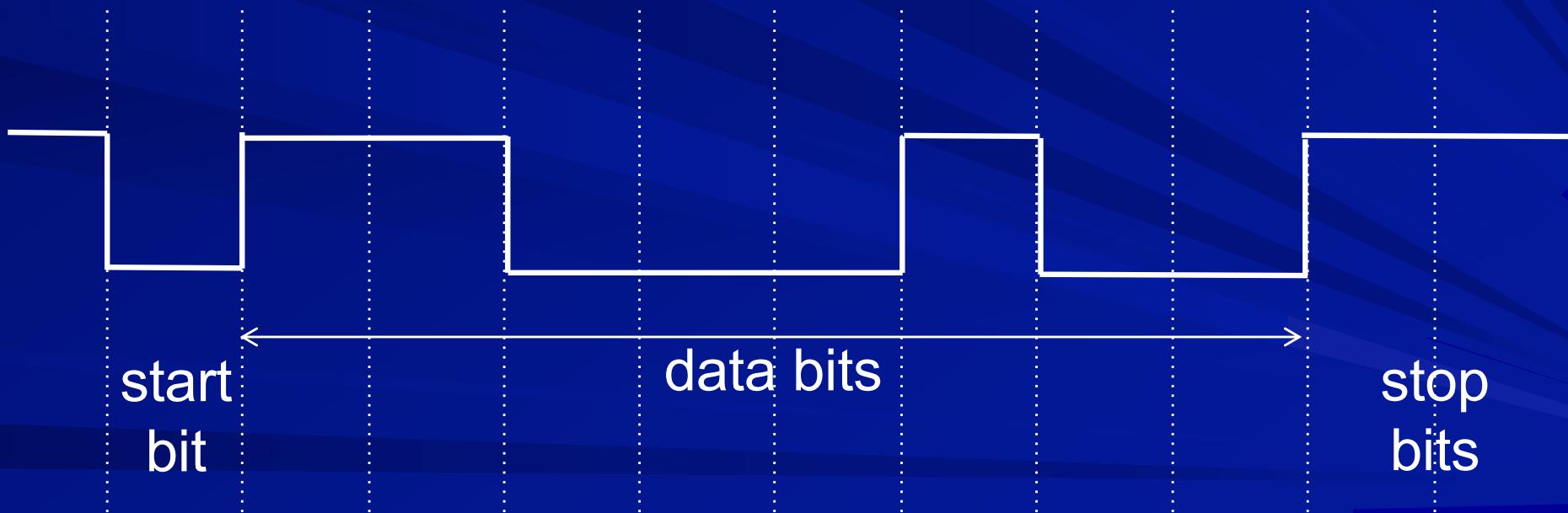
UART



Serial port: data waveform

- Plesiochronous

- Synchronous at bit level
- Asynchronous at frame level



Data Framing



Data bits : 5 to 8

Parity : even / odd / none

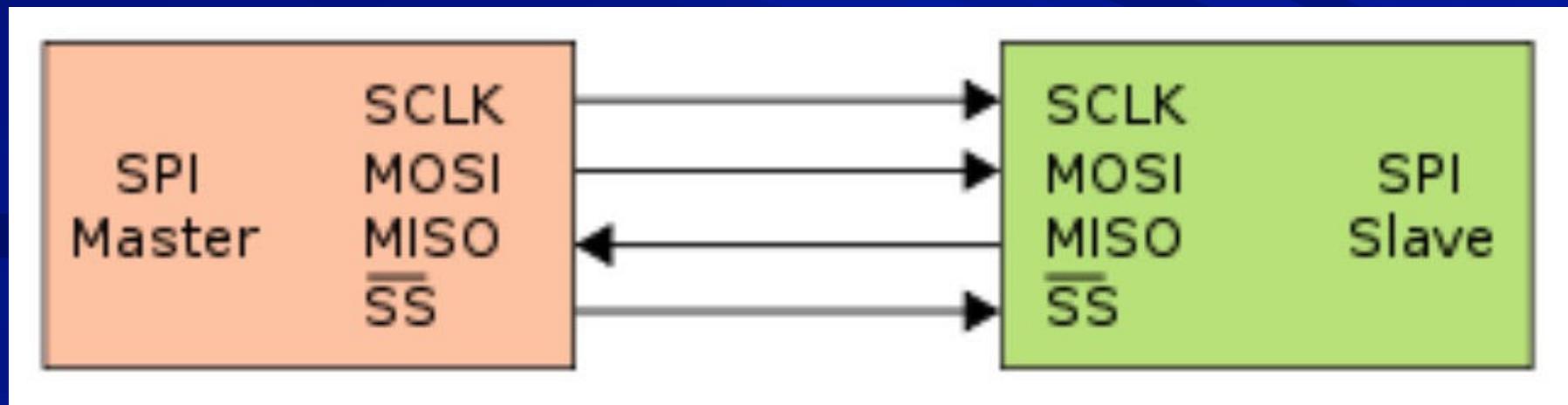
Start bits : 1

Stop bits : 1 or 2

Data rate : 300 / 600 / 1200 / 2400 / 4800 / 9600 . . .
bits/sec (bauds)

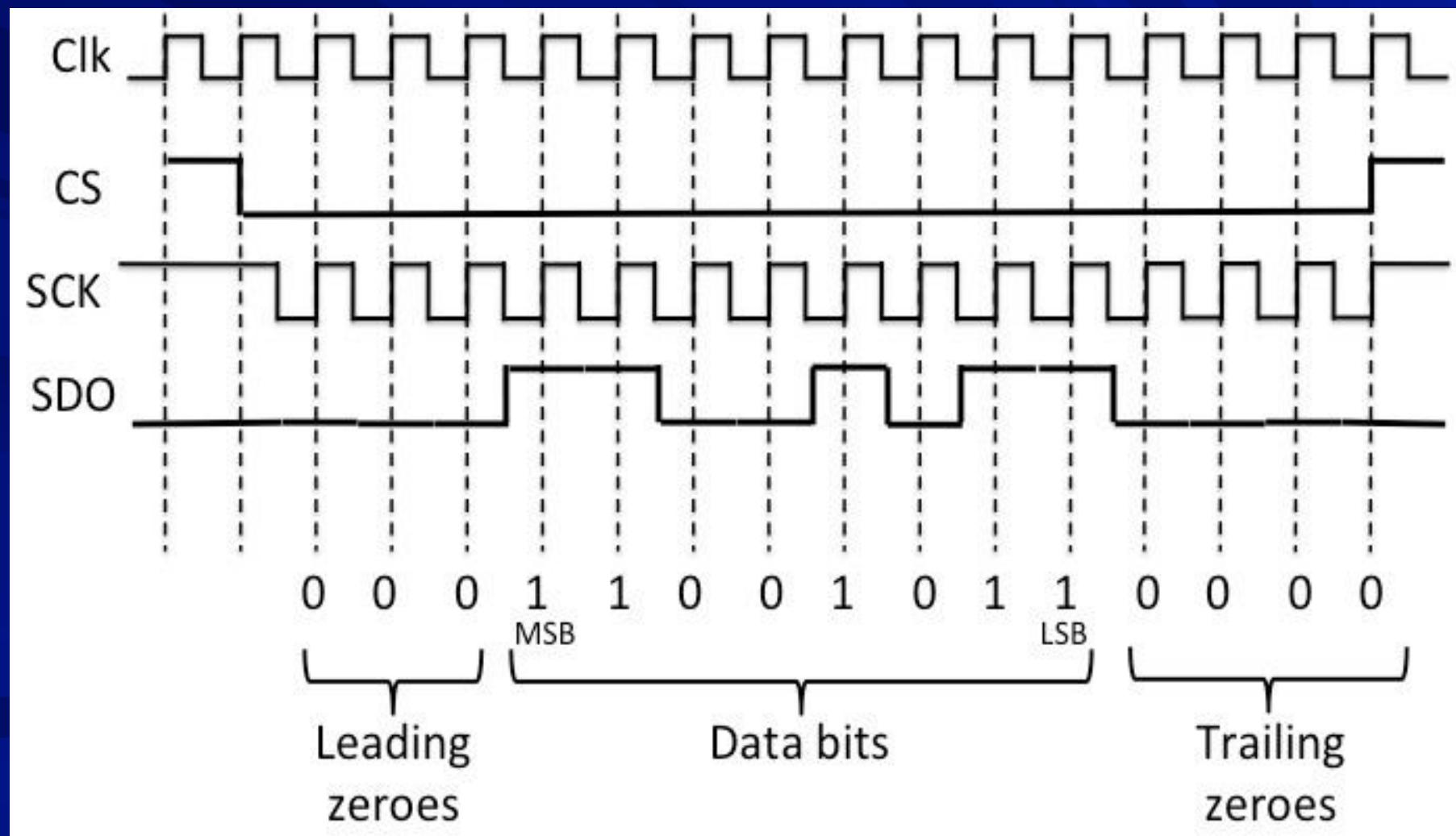
SPI : A simple serial interface

- SPI (Serial Peripheral Interface) is a synchronous serial protocol.
- Used for connecting some Pmods with BASYS-3.



Ref: Wikipedia

SPI Signal Waveforms



COL216

Computer Architecture

Input/Output – 5

USB

28th March 2022

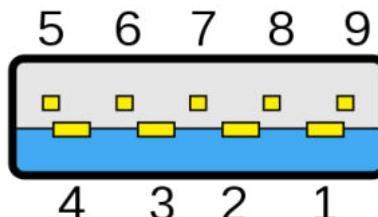
USB : Universal Serial Bus

- Introduced in mid-90's
- Used for a huge variety of devices
- Replaced earlier serial and parallel ports
- Much higher speed than RS232 etc
- Packet oriented, not character oriented
- Clock is recovered from data
- Differential signalling (D+ and D-)
- Up to 127 devices through hubs, each with up to 16 IN and 16 OUT end points

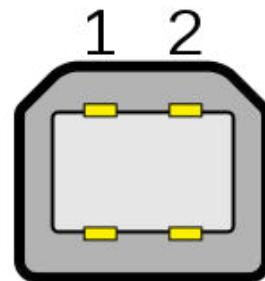
USB connectors



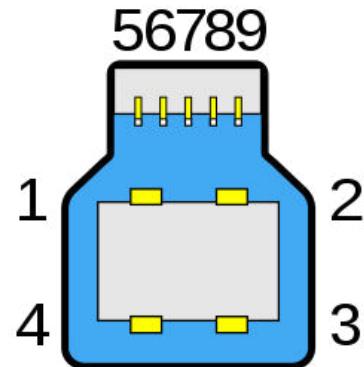
Type A



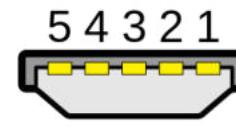
Type A
SuperSpeed



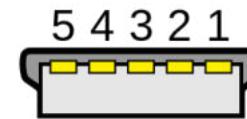
Type B



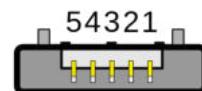
Type B
SuperSpeed



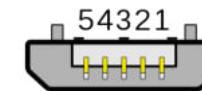
Mini-A



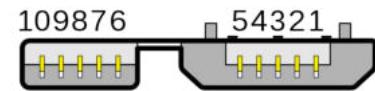
Mini-B



Micro-A



Micro-B



Micro-B SuperSpeed



Type-C

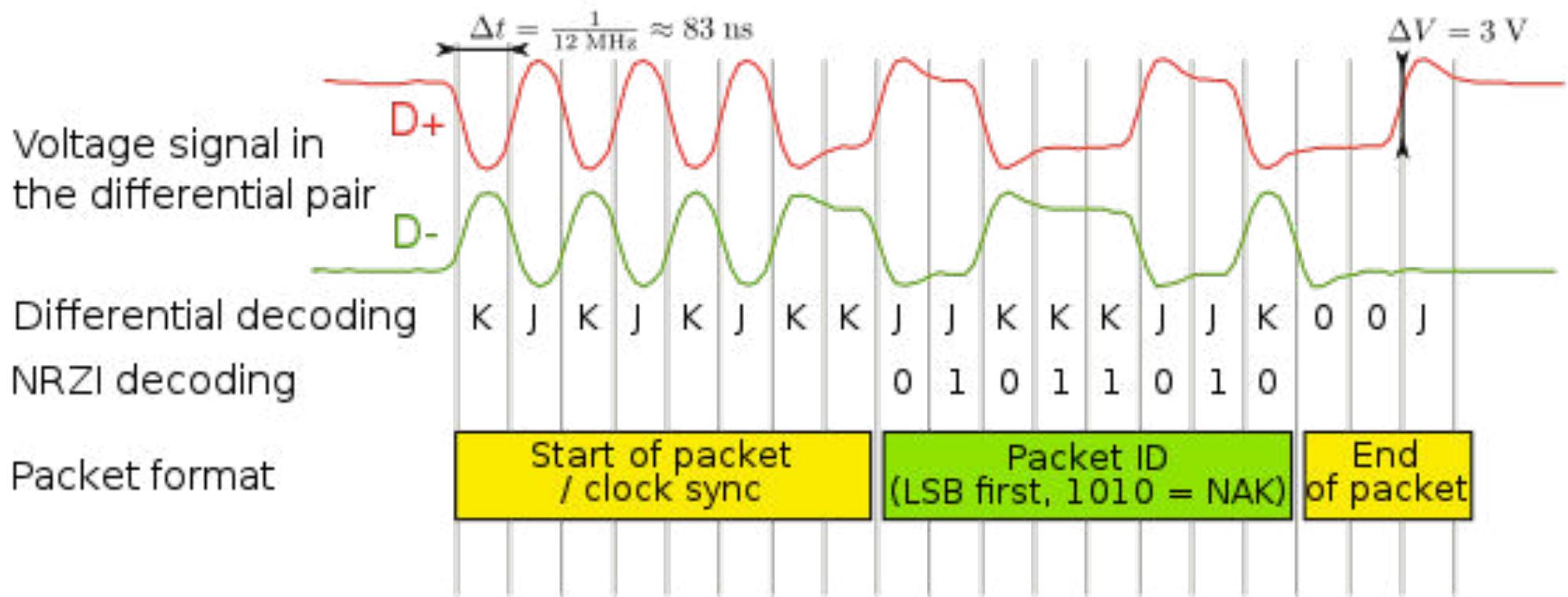
USB speed, levels

■ USB 1.0	Low Speed	1.5 Mb/s
■ USB 1.0	Full Speed	12 Mb/s
■ USB 2.0	High Speed	480 Mb/s
■ USB 3.0	Super Speed	5 Gb/s
■ USB 3.1	Super Speed+	10 Gb/s
■ USB 3.2	Super Speed+	20 Gb/s
■ USB 4	forthcoming	
LS, FS :	0.0 to 0.3 V	2.8 to 3.6 V
HS, SS :	-10 to 10 mV	-360 to 440mV

What D+,D- patterns can indicate

- Device found detached
- Device wakes up host
- Host and device idling
- Start of packet
- End of Packet
- Reset device to a known state
- Power down device
- Host wakes up device
- Host wants device to stay awake
- Device wants to wake up

USB NAK packet



source : wikipedia

Packet IDs

■ Token packets

- SPLIT: Split transaction
- PING
- OUT, IN: Address for data transfer
- SOF: start of frame marker
- SETUP: Device management

■ Handshake packets

- ACK: Packet accepted
- NAK: Packet not accepted, retransmit
- NYET: Data not ready yet
- STALL: Transfer impossible, do recovery

■ DATA0, DATA1 etc: data packet

COL216

Computer Architecture

Input/Output – 5
Secondary Storage
28th March 2022

Hard Disk Drive Example

Capacity: 14TB

Size: 3.5"

RPM: 7200

Data rate: 250 MB/s

Disks: 8 (16 heads)

Sector: 512B (logical)

4096B (physical)

Density: 2426 Kb/in

436 Ktracks/in

Cache: 256 MB

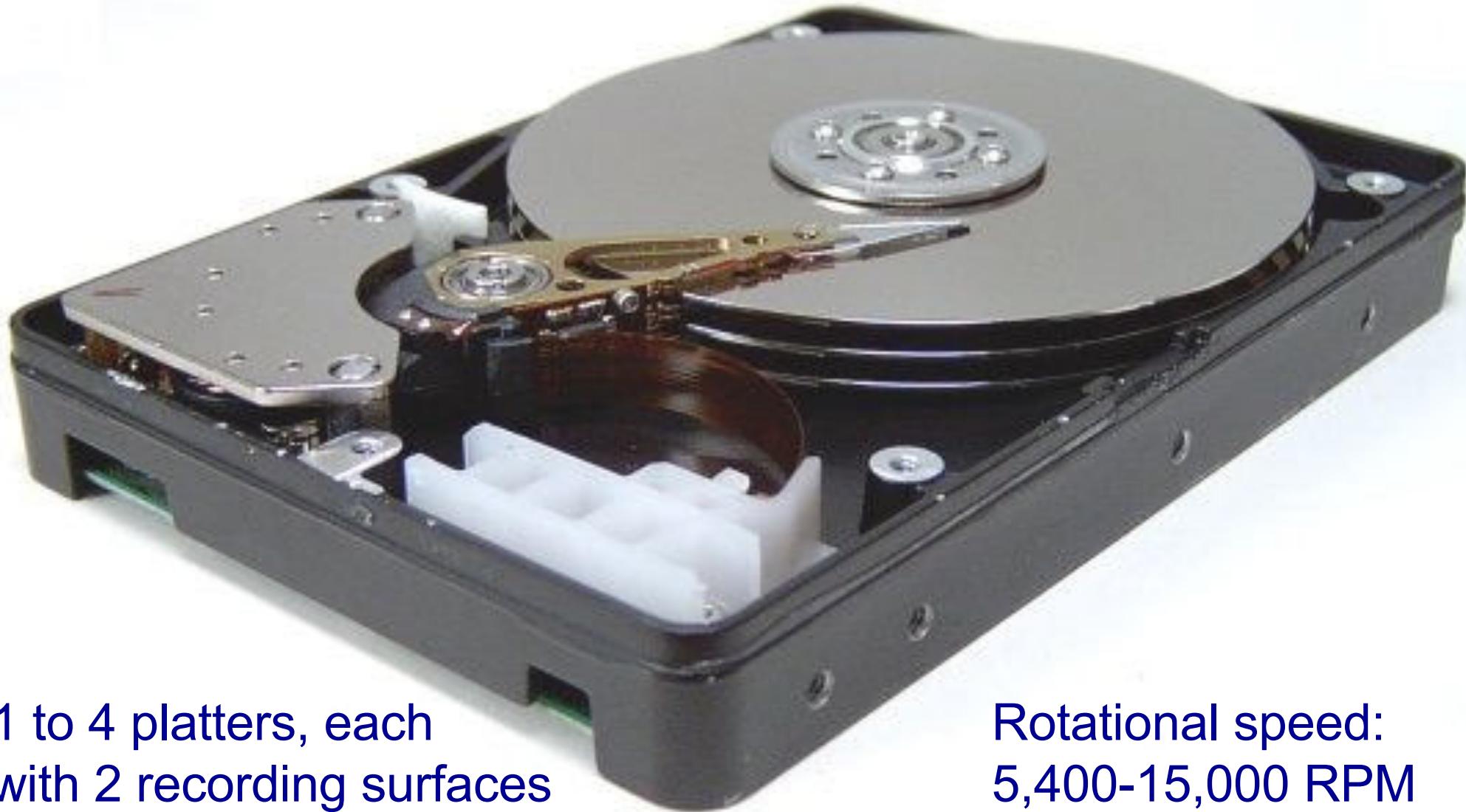


HDD comparison – then and now

Parameter	1957	2018-20	Improvement
Capacity	3.75 MB	18 TB	4.8×10^6
Volume	1.9 m ³	34 cm ³	5.6×10^4
Weight	910 kg	62 g	1.5×10^4
Access Time	600 ms	2.5 – 10 ms	2×10^2
Price	USD 9.2 / KB	USD 24 / TB	3×10^6
Density	2K b / in ²	1.3 TB / in ²	6.5×10^8
MTBF	2000 hrs	285 yrs	1.25×10^3

Source: Wikipedia

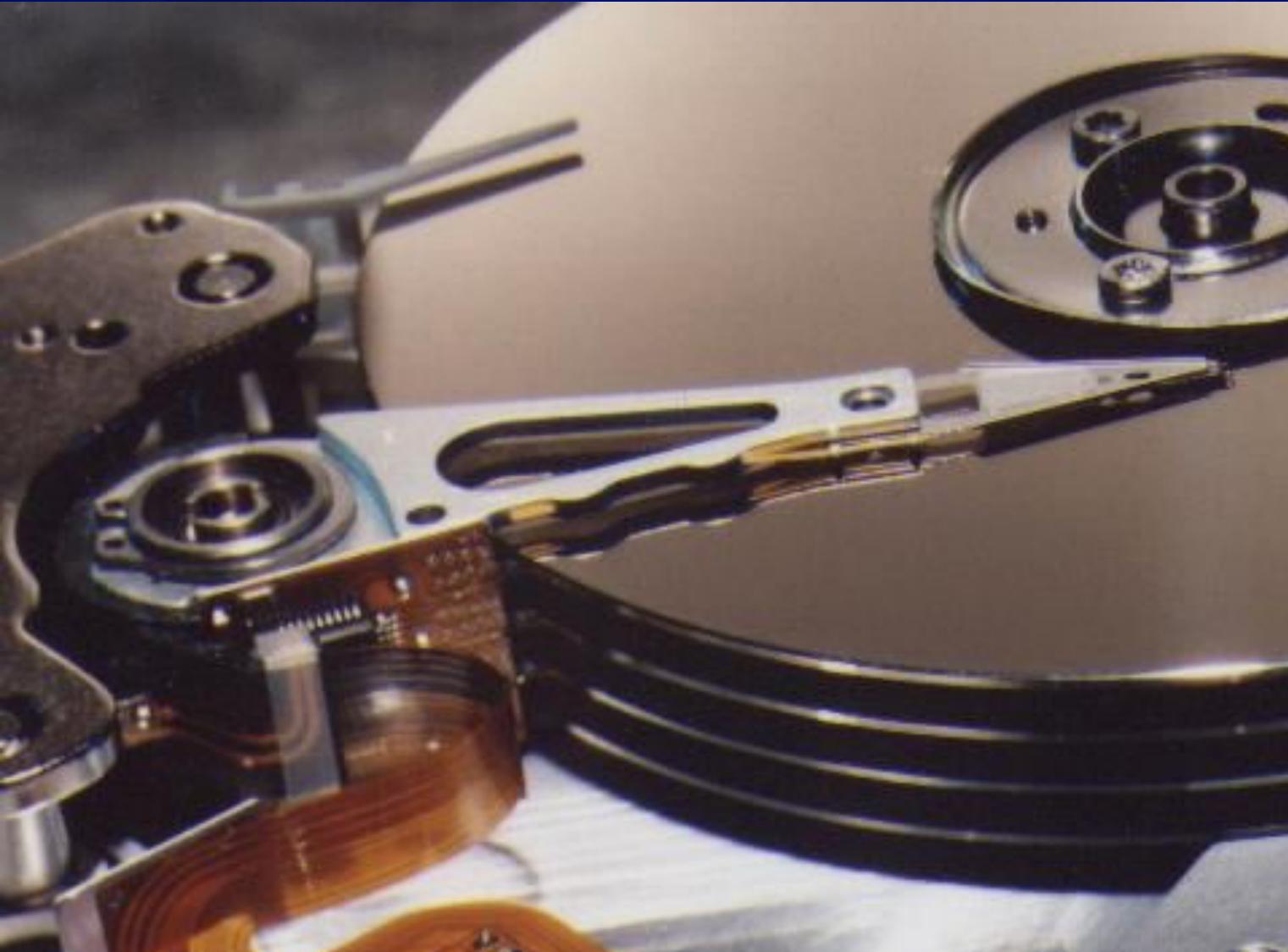
Hard Disk Drive



1 to 4 platters, each
with 2 recording surfaces

Rotational speed:
5,400-15,000 RPM

Hard disk drive close-up



Tracks:
10-50 K
Sector/track
100-500
Bytes/sector
512
Seek time:

- Track to track
.5-2 ms
- Average
~10 ms

Disk Latency

Disk Latency =

Seek time + Rotational Latency + data
transfer time + controller overhead

Average rotational latency =

time for $\frac{1}{2}$ rotation =

$(.5 \times 60 \times 1000 / \text{rpm}) \text{ msec}$

Transfer rate = $10^1 - 10^2 \text{ MB/sec}$

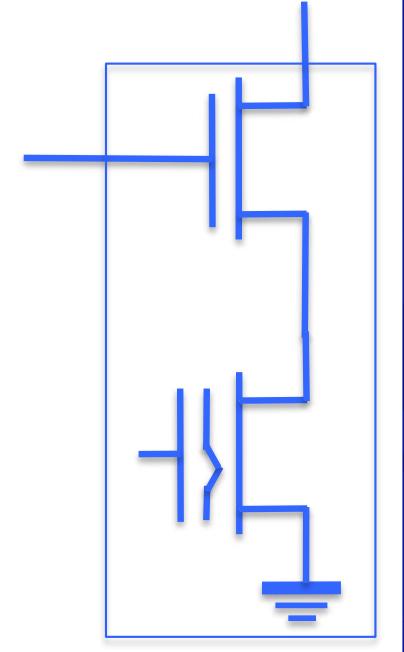
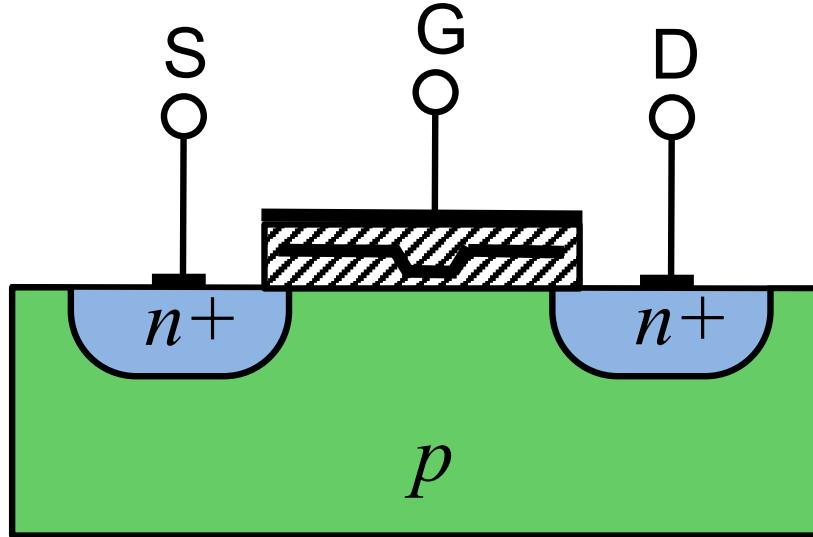
Bulk storage without moving parts

- Flash: semiconductor memory like SRAM, DRAM, but non-volatile like magnetic disks
- Latency is 100–1000 times less than disk
- Smaller, lighter, more power efficient, and more shock resistant than disk
- Has caught up with HDD in capacity, but more expensive
- SSD wears out faster than HDD

Flash Memory

- Semiconductor memory made up of “Floating gate” transistors – these store information in a non-volatile manner
- Reading is fast, but writing is slow as it requires large current
- Writing involves one way state change (e.g. 0 to 1), therefore needs erasing first (make all bits 0 first)
- Block erase, unlike byte erase of EEPROM

Floating gate transistor



Programming: pass high current through channel
Electrons are trapped on the floating gate

Before trapping: like a normal NMOS transistor
After trapping: transistor remains OFF

NOR and NAND flash

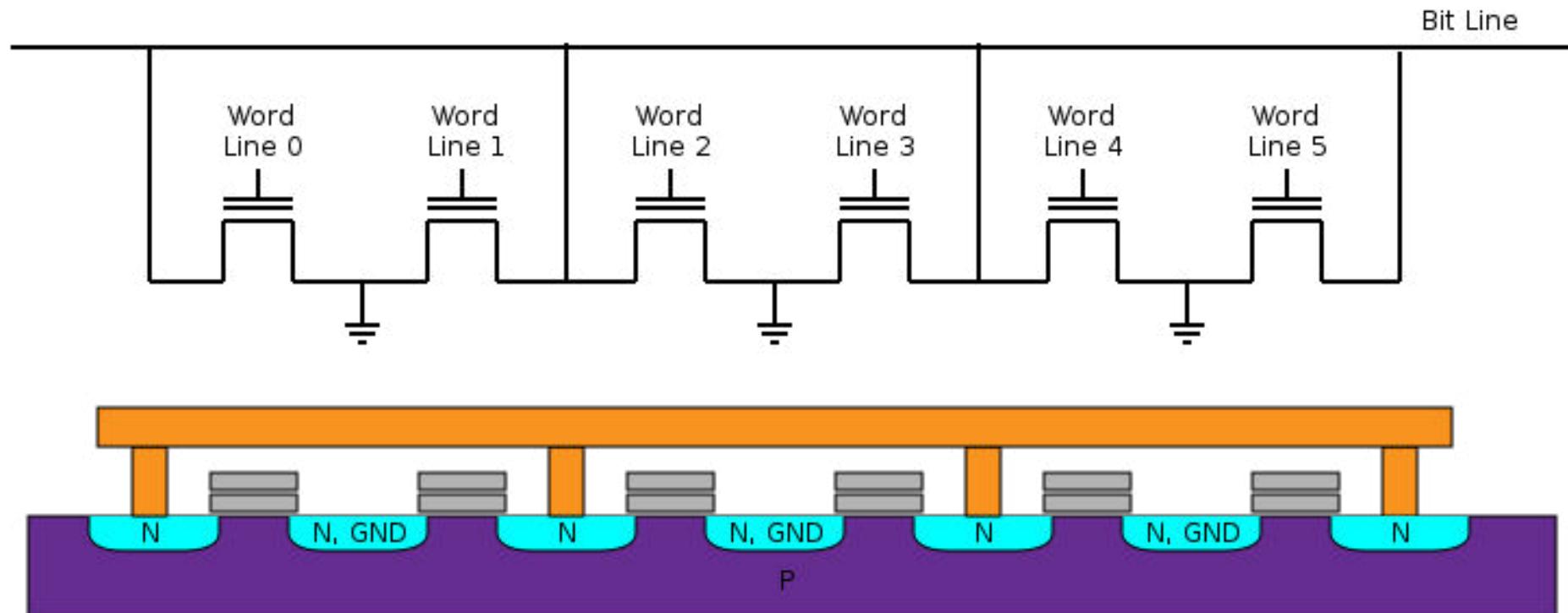
NOR flash

- Structure like NOR gate
- Random access
- Lower density
- More expensive
- Used for BIOS memory

NAND flash

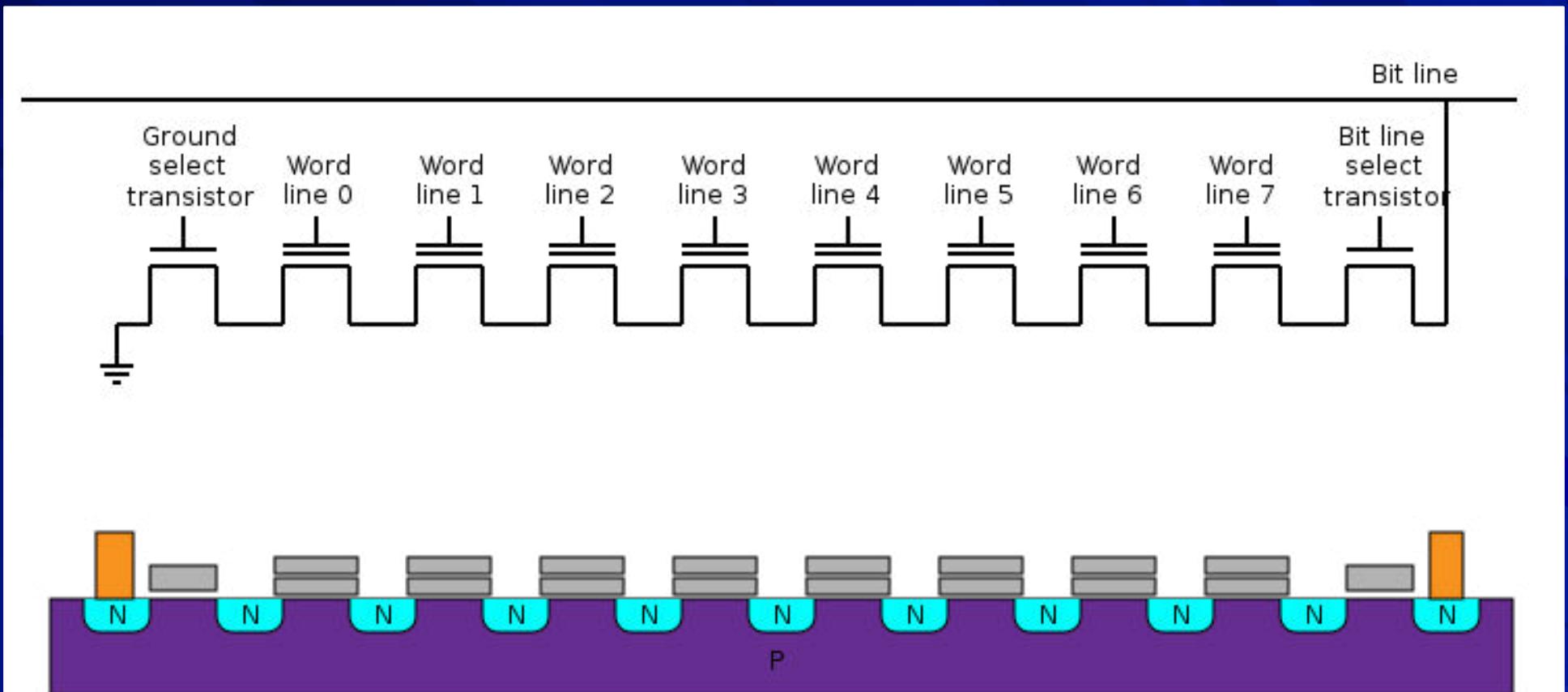
- Structure like NAND gate
- Random plus sequential access
- Higher density
- Less expensive
- Used for pen drive, SD card, SSD
- 32 Mb on BASYS 3

NOR Flash



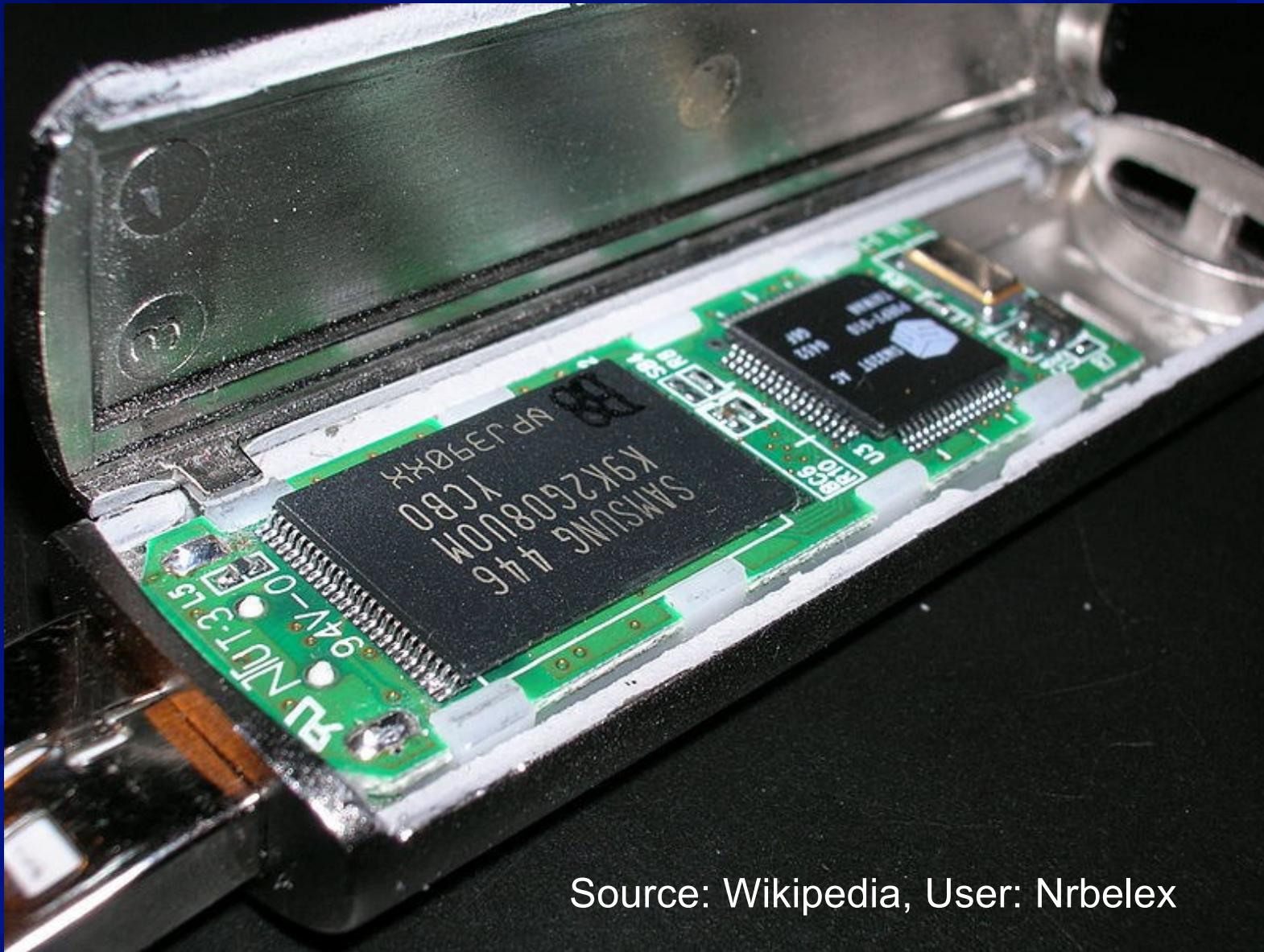
Source: Wikipedia, Author: Cyferz

NAND Flash



Source: Wikipedia, Author: Cyferz

A USB Flash Drive



Source: Wikipedia, User: Nrbelex

COL216

Computer Architecture

Input/Output – 5
I/O Performance
28th March 2022

I/O Performance

- It often gets neglected
- Suppose a benchmark executes in 100 sec
 - CPU 90%
 - I/O 10%
- CPU performance improves by 50% every year for 5 years
 - $90 \rightarrow 60 \rightarrow 40 \rightarrow 27 \rightarrow 18 \rightarrow 12$ (total = 7.5 times)
- I/O performance unchanged
- Overall performance
 - $100 \rightarrow 70 \rightarrow 50 \rightarrow 37 \rightarrow 28 \rightarrow 22$ (total = 4.5 times)
- If program is I/O bound, improvement is even smaller

I/O performance definitions

- Throughput

- Amount of data transfer in unit time (KB/s, MB/s)
 - Number of I/O operations in unit time

- Response time

- Relevant in interactive environment, embedded systems

- Both important in some cases

Examples

- Supercomputers
 - data throughput (KB or MB/s), more output than input
- Transaction processing
 - transactions per sec, response time
- File server
 - file operations per sec

Discrepancy in units

- Memory size

- $1 \text{ KB} = 1024 \text{ B} (2^{10})$

- $1 \text{ MB} = 1024 \text{ KB} (2^{10}) = 1048576 \text{ B} (2^{20})$

- I/O rate

- $1 \text{ KB/s} = 1000 \text{ B/s} (10^3)$

- $1 \text{ MB/s} = 1000 \text{ KB/s} (10^3) = 1000000 \text{ B/s} (10^6)$

I/O system performance example

Given :

- Server configuration
- I/O intensive application characteristics

Required :

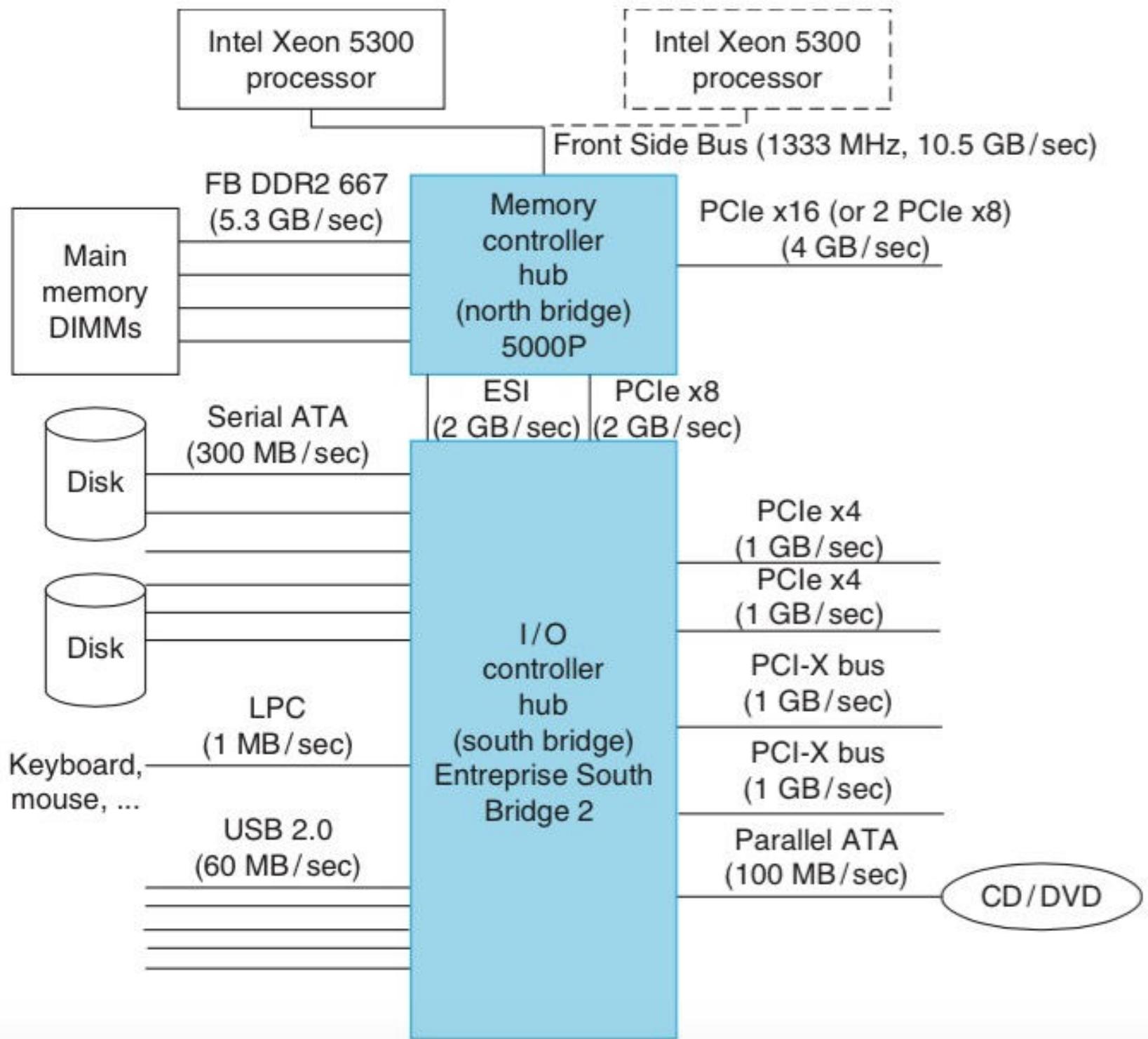
- Maximum sustainable performance
(in terms of rate of I/O operations)

I/O example: server configuration

Sun Fire x4150 server

- 8 processors, across two sockets (Xeon 5345)
- 64 GB DRAM (DDR2 667), across 16 FB DIMMs
- 8 HDDs, 2.5 inch, 15,000 RPM 73 GB SAS
- 4 Ethernet ports 10/100/1000 Gbps
- 3 PCI Express x8 ports
- 4 external and 1 internal USB 2.0 ports

Server Architecture



I/O example: application characteristics

- User program uses 200,000 instructions per I/O operation
- Operating system averages 100,000 instructions per I/O operation
- Workload consists of 64 KB reads
- Each processor sustains 1 billion instructions per second

I/O example: requirement

- Find the maximum sustainable I/O rate for a fully loaded Sun Fire x4150
 - for random reads
 - sequential reads
- Assume that the reads can always be done on an idle disk if one exists (i.e., ignore disk conflicts)

IOPS for processors

Maximum I/O rate of 1 processor =

Instruction execution rate / Instructions per I/O

$$= 1 \times 10^9 / (200 + 100) \times 10^3 = 3,333 \text{ IOPS}$$

One socket has four processors

$$\Rightarrow 4 \times 3,333 = 13,333 \text{ IOPS}$$

Two sockets with eight processors

$$\Rightarrow 2 \times 13,333 = 26,666 \text{ IOPS.}$$

IOPS for disks

Time per I/O at disk (for random access)

$$= \text{seek} + \text{rotational time} + \text{transfer time}$$

$$= .725 \text{ ms} + 2 \text{ ms} + 64 \text{ KB} / 112 \text{ MB/s}$$

$$= 3.3 \text{ ms}$$

Each disk can do $1000/3.3 = 303$ IOPS

8 disks can do $8 \times 303 = 2,424$ IOPS

For sequential access,

$$\text{time} = 64 \text{ KB} / 112 \text{ MB/s} \Rightarrow 1,750 \text{ IOPS}$$

i.e., 14,000 IOPS for 8 disks

IOPS for PCI express

Bandwidth of a PCIe lane = 250 MB/s

Bandwidth of 8 PCIe lanes = $8 \times 250 \text{ MB/s}$
= 2,000 MB/s

IOPS for 8 PCIe lanes = $2,000 \text{ MB/s} / 64 \text{ KB}$
= 31,250

IOPS for DRAM

Bandwidth of 667 MHz FBDIMM

$$= 8 \times 667 = 5,336 \text{ MB/s}$$

IOPS for one FBDIMM = $(5,336 / 64) \times 1000$

$$= 83,375$$

Max no. of DIMMs = 16

FSB limit

FSB peak bandwidth = 10.6 GB/s

Sustained bandwidth = 5.3 GB/s

IOPS for each FSB = $5.3 \times 10^6 / 64 = 81,540$

For 2 FSBs = $> 2 \times 81,540 = 163,080$ IOPS

IOPS at a glance

■ 8 Processors :	26,667
■ 8 Disks :	2,424 (random) 14,000 (sequential)
■ PCIe 8x :	31,250
■ 1 FBDIMM :	83,375
■ 1 FSB :	81,540

THANKS

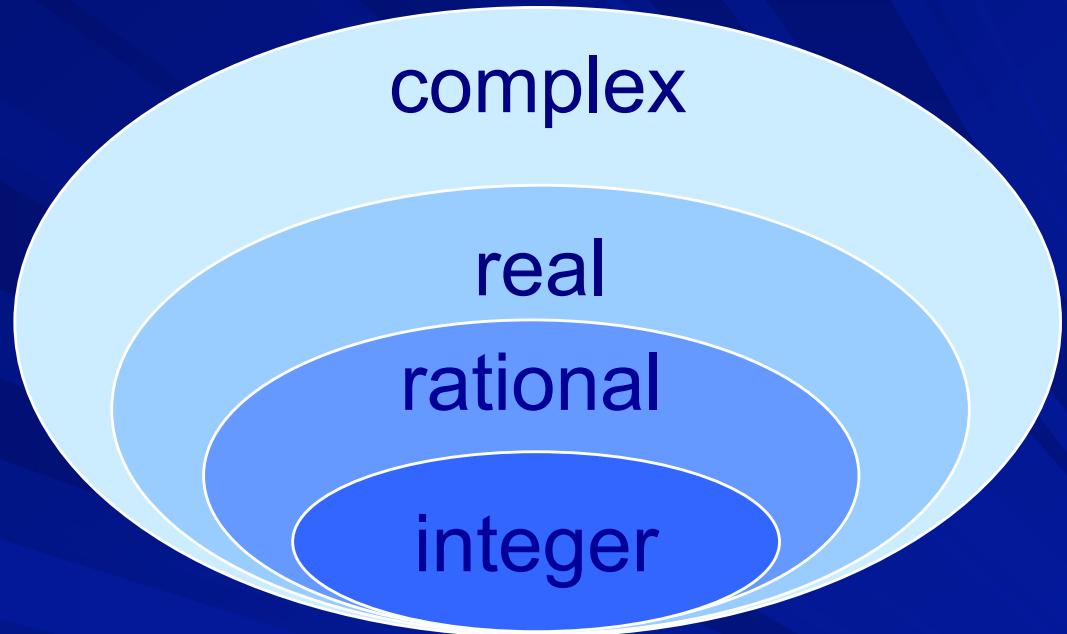
COL216

Computer Architecture

Concluding Remarks:
Floating point numbers and arithmetic
31st March 2022

Need to go beyond integers

- integer 7
- rational $\frac{5}{8}$
- real $\sqrt{3}$
- complex $2 - 3i$



Extremely large and small values:

- distance pluto - sun = 5.9×10^{12} m
- mass of electron = 9.1×10^{-31} gm

Representing fractions

- Integer pairs (for rational numbers)

5 8 = 5/8

- Strings with explicit decimal point

- 2 4 7 . 0 9

- Implicit point at a **fixed** position

0 1 0 0 1 1 0 1 0 1 1 0 0 0 1 0 1 1

↑ implicit point

- Floating point

fraction x base power

Numbers with binary point

$$\begin{aligned}101.11 &= 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} \\&= 4 + 1 + 0.5 + 0.25 = 5.75_{10}\end{aligned}$$

$$0.6 = 0.1001\textcolor{red}{1001100110011001\dots}$$

$$.6 \times 2 = 1 + .2$$

$$.2 \times 2 = 0 + .4$$

$$.4 \times 2 = 0 + .8$$

$$.8 \times 2 = 1 + .6$$

FP numbers with base = 2

$$(-1)^S \times F \times 2^E$$

S = sign

F = fraction (fixed point number)

usually called mantissa or significand

E = exponent (positive or negative integer)

- How to divide a word into S, F and E?
- How to represent S, F and E?

IEEE 754 standard

■ Single precision numbers

1	8	23
010110101110101101011000101101101		
S	E	F

■ Double precision numbers

1	11	20+32
010110101110101101011000101101101		
S	E	F
101100010110110010110101110101101		

Representing F in IEEE 754

■ Single precision numbers

23

1. 110101101011000101101101

F

■ Double precision numbers

20+32

1. 101101011000101101101

F

101100010110110010110101110101101

Value range for F

- Single precision numbers

$$1 \leq F \leq 2 - 2^{-23} \quad \text{or} \quad 1 \leq F < 2$$

- Double precision numbers

$$1 \leq F \leq 2 - 2^{-52} \quad \text{or} \quad 1 \leq F < 2$$

These are “normalized”.

Representing E in IEEE 754

■ Single precision numbers

8

10110101

54

E

(+ bias 127)

■ Double precision numbers

11

10110101110

431

E

(+ bias 1023)

Value range for E

- Single precision numbers

$$-126 \leq E \leq 127$$

(all 0's and all 1's have special meanings)

- Double precision numbers

$$-1022 \leq E \leq 1023$$

(all 0's and all 1's have special meanings)

Representing zero

- All bits of F are zero, no explicit “1” to the left
(not a normalized value)
- E can have any value, we choose to have all bits zero
- Sign bit is also zero
- Exactly same as integer zero!

Testing and comparing

- Test for zero, negative and positive for FP numbers is same as that for integers
- Magnitude comparison of FP numbers is same as magnitude comparison for integers

Why?

Because exponent is in biased notation and is located to the left of mantissa

Overflow and underflow

largest positive/negative number (SP) =

$$\pm(2 - 2^{-23}) \times 2^{127} \cong \pm 2 \times 10^{38}$$

smallest positive/negative number (SP) =

$$\pm 1 \times 2^{-126} \cong \pm 2 \times 10^{-38}$$

largest positive/negative number (DP) =

$$\pm(2 - 2^{-52}) \times 2^{1023} \cong \pm 2 \times 10^{308}$$

smallest positive/negative number (DP) =

$$\pm 1 \times 2^{-1022} \cong \pm 2 \times 10^{-308}$$

Floating point operations

■ Add/subtract

$$[(-1)^{S1} \times F1 \times 2^{E1}] \pm [(-1)^{S2} \times F2 \times 2^{E2}]$$

suppose $E1 > E2$, then we can write it as

$$[(-1)^{S1} \times F1 \times 2^{E1}] \pm [(-1)^{S2} \times F2' \times 2^{E1}]$$

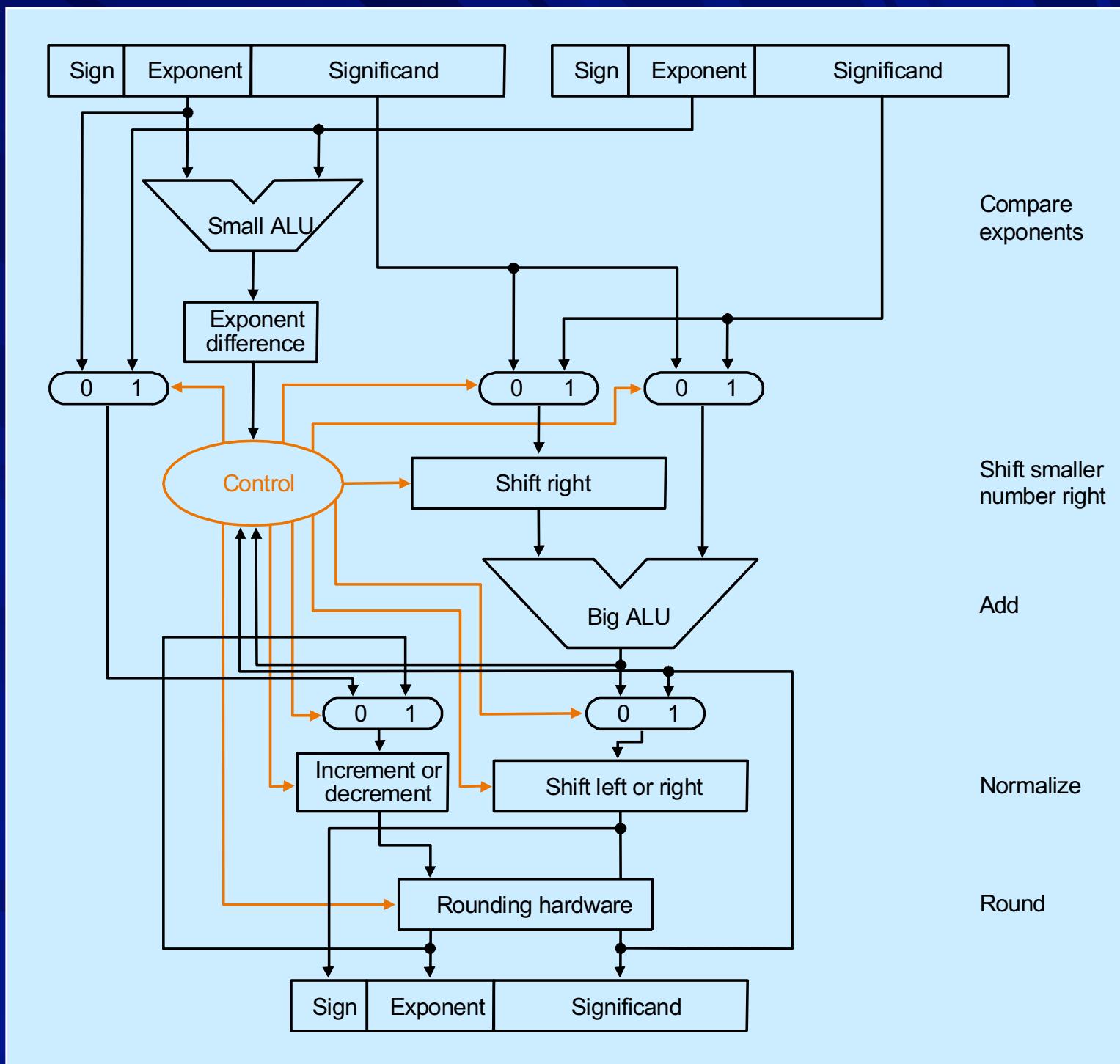
where $F2' = F2 / 2^{E1-E2}$,

The result is

$$(-1)^{S1} \times (F1 \pm F2') \times 2^{E1}$$

It may need to be normalized

FP Add/Sub Unit

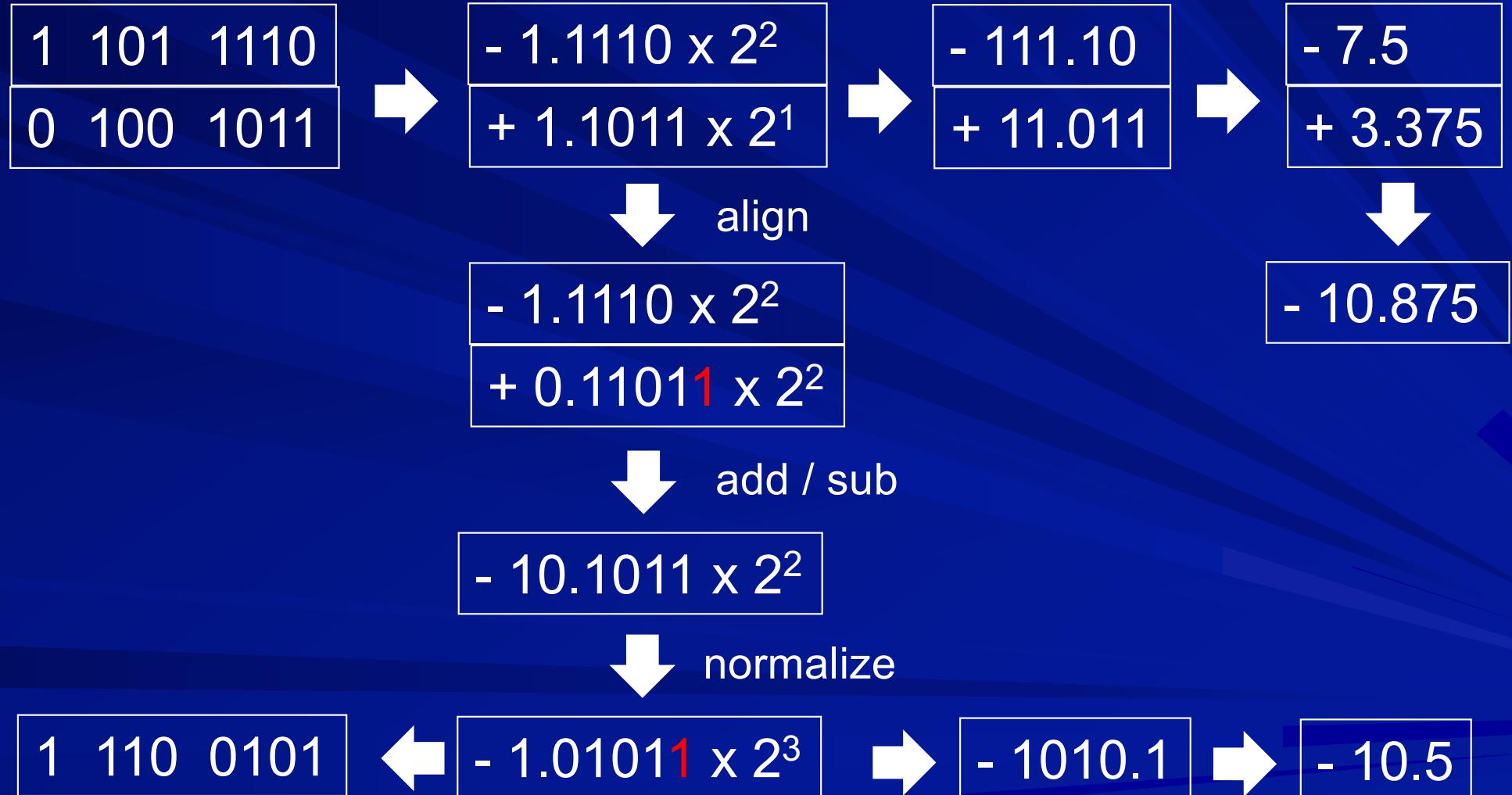


Small word size example

s	e	e	e	f	f	f	f
---	---	---	---	---	---	---	---

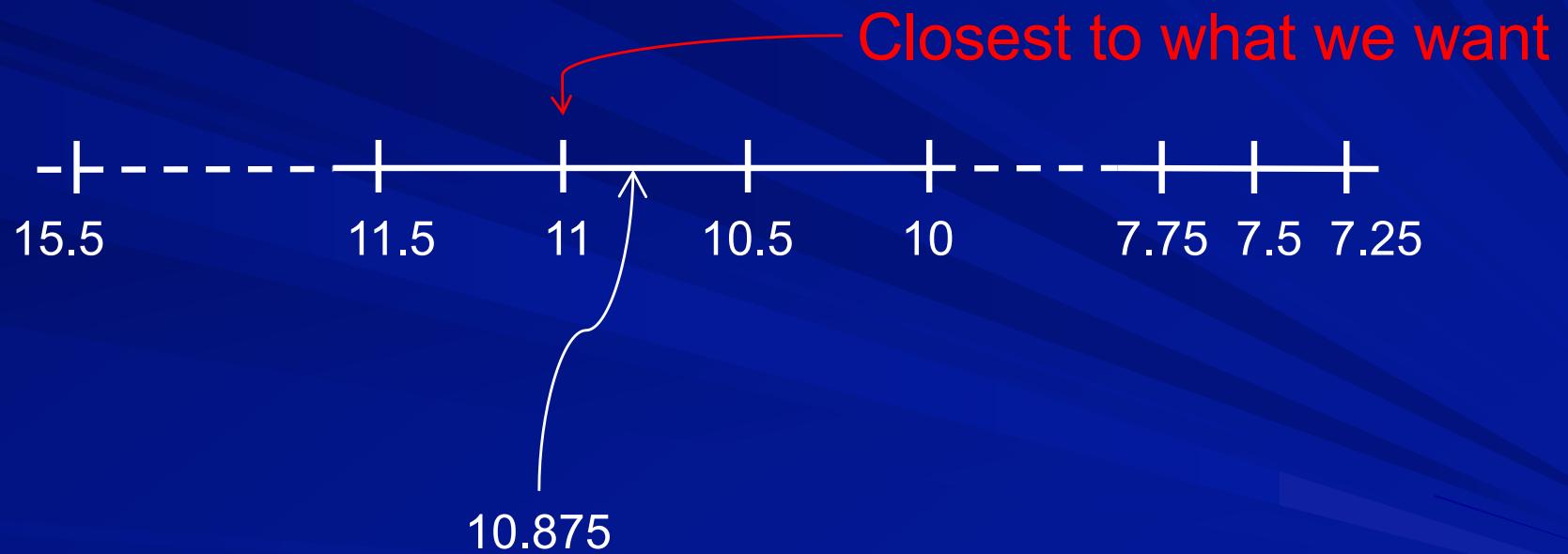
- Range of E = -2 to 3
 - represented by 001 to 110 (bias = 3)
- Range of F = 1.0 to 1.9375
 - represented by 1.0000 to 1.1111
- resolution = $.0625 \times 2^E$

FP subtraction example



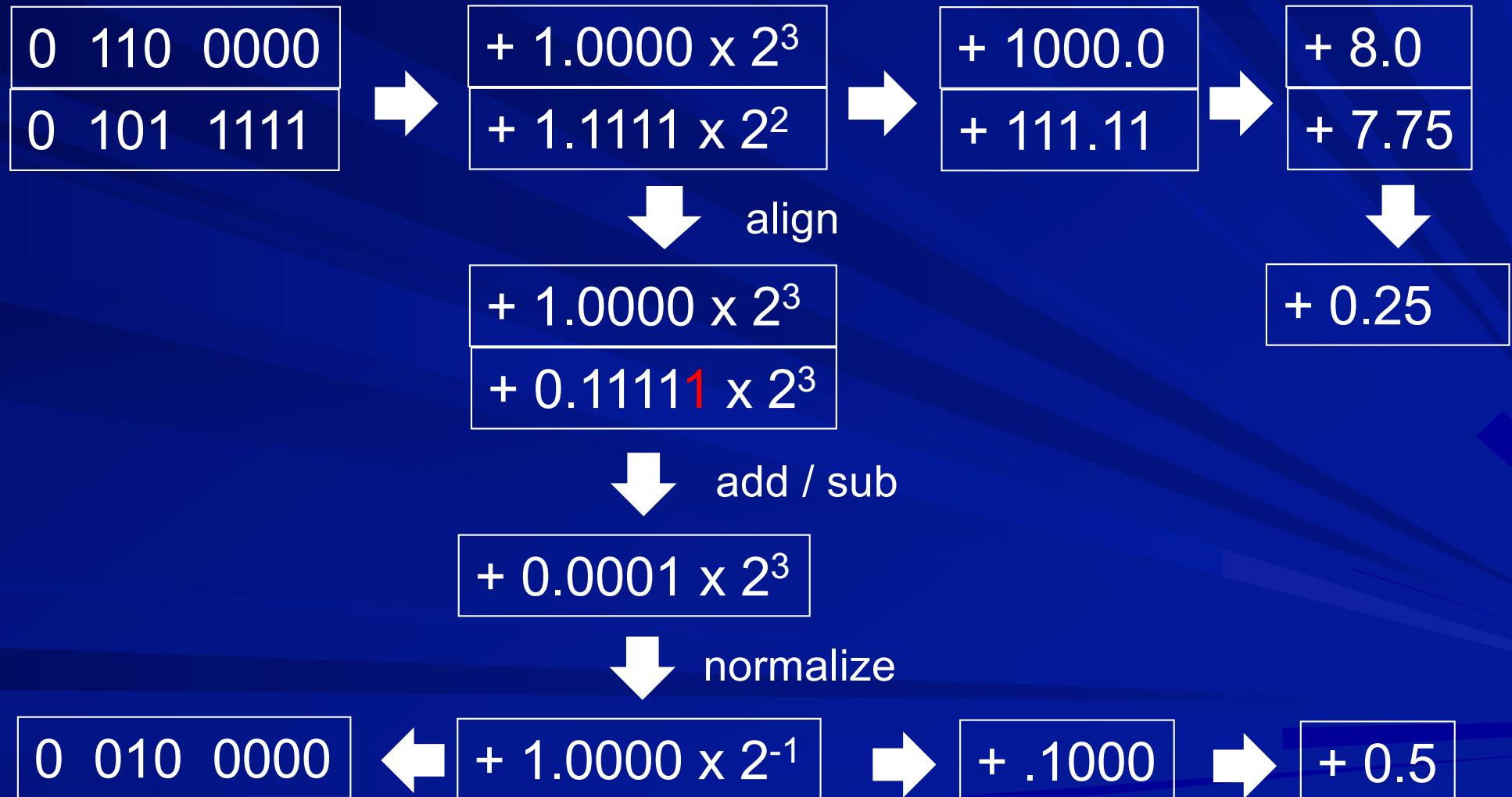
Can we do better?

The values that can be represented



The value we want
to represent

Another FP subtraction example



Can we represent 0.25 ?

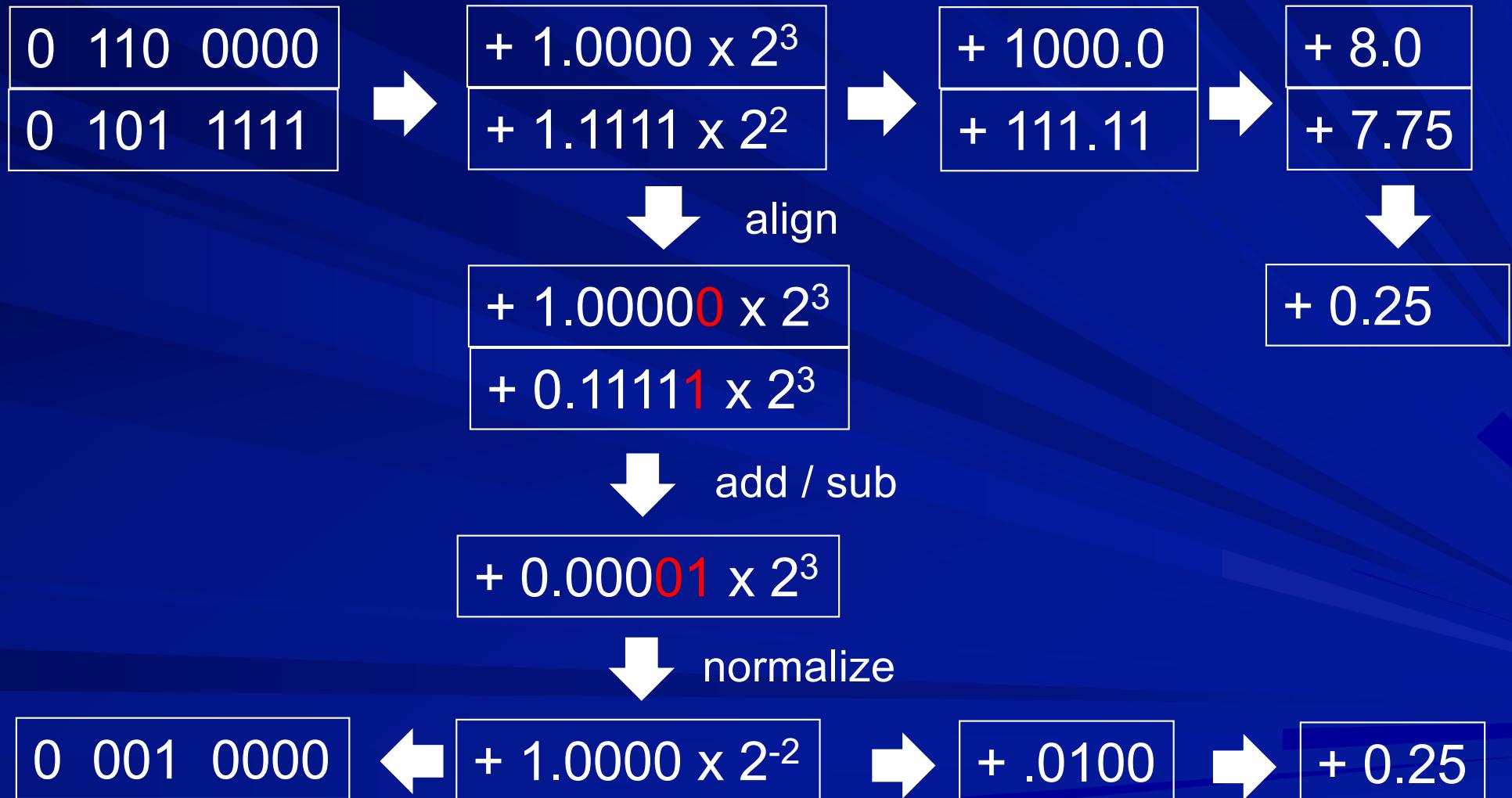
Yes.

$$0.25 = 1.0 \times 2^{-2}$$

0 001 0000

So where is the problem?

If we had more bits to work with...



Improving precision of computation

- Precision is lost when some bits are shifted to right of the rightmost bit or are thrown

How many bits more?

- How many bits we lose in alignment that can potentially be recovered during normalization?
- many bits lost during alignment when difference in operands is large
- many bits brought in during normalization when difference is small

Improving precision of computation

- Three extra bits are used internally -
G (guard), R (round) and S (sticky)
 - G and R are simply the next two bits after LSB
 - S = 1 iff any bit to right of R is non-zero

1. 110101101011000101101101 GRS

Rounding using G, R and S

- if $G=1 \ \& \ R=1$, add 1 to LSB
- if $G=1 \ \& \ R=0$, look at S
 - if $S=1$, add 1 to LSB
 - if $S=0$, round to the nearest “even”
i.e., add 1 to LSB if $LSB = 1$
- if $G=0 \ \& \ R=0$ or 1, no change

Floating point operations

■ Multiply

$$\begin{aligned} & [(-1)^{S1} \times F1 \times 2^{E1}] \times [(-1)^{S2} \times F2 \times 2^{E2}] \\ &= (-1)^{S1 \oplus S2} \times (F1 \times F2) \times 2^{E1+E2} \end{aligned}$$

Since $1 \leq (F1 \times F2) < 4$,
the result may need to be normalized

Floating point operations

■ Divide

$$[(-1)^{S1} \times F1 \times 2^{E1}] \div [(-1)^{S2} \times F2 \times 2^{E2}]$$

$$= (-1)^{S1 \oplus S2} \times (F1 \div F2) \times 2^{E1-E2}$$

Since $.5 < (F1 \div F2) < 2$,

the result may need to be normalized

(assume $F2 \neq 0$)

Important points

- Floating point representation is an approximation of the real values
- Floating point operations are not exact real operations
- Associativity of operations need not hold
 - $(A + B) + C$ may differ from $A + (B + C)$

Special numbers

Single precision	Double precision	object		
exponent	mantissa	exponent	mantissa	
0	0	0	0	zero
0	nonzero	0	nonzero	denorm
1-254	any	1-2046	any	norm
255	0	2047	0	infinity
255	nonzero	2047	nonzero	NaN

Co-processors / processor extensions

- Extend the capability of main processors
- Modularize the design – available as options
- Co-processor registers
- Load/store instructions
- Instructions for movement of data between co-processor registers and main registers
- Co-processor operations

Examples

■ ARM

- Floating point co-processor
- VFP
- Neon

■ Intel

- Floating point co-processor
- MMX (multimedia extension)
- SSE (streaming SIMD extension)

Floating Point processors

- Floating point registers
- Load/store instructions
- Instructions for movement of data between FP and integer registers
- Arithmetic instructions (SP, DP, ...)
- Comparison instructions
- Instructions for format conversion

SIMD extensions

- SIMD : Single Instruction Multiple Data
 - same operation repeated over multiple units of data
 - example – addition of two vectors
- A large register may be interpreted as an array of smaller registers
- Hardware may perform operations on small vectors of larger registers or large vectors of smaller registers

SIMD extensions

1 x 128-bit



2 x 64-bit



4 x 32-bit



8 x 16-bit



16 x 8-bit



COL216

Computer Architecture

Concluding Remarks:
What lies beyond the present course
31st March 2022

What have we studied

- Instruction set architecture and assembly language programming
- Arithmetic operations, how to build an ALU
- Constructing a processor to execute instructions – Micro architecture
- Performance issues
- Pipelining to improve performance
- Memory: caches and virtual memory
- Input / output

What we have not studied

- Advanced pipelining techniques
- Advanced concepts in memory hierarchy design
- Superscalars
- VLIW architectures
- Vector processors
- Multiprocessors
- GPUs

How to achieve high performance?

- Do things faster
- Do more things at the same time
- Achieve same thing by doing less
- Remove bottle necks – achieve balance
- What do N_{instr} , CPI_{avg} and T_{clock} depend on?
 - Technology
 - Circuit / logic design
 - Architecture
 - Compiler, OS
 - Algorithm

Technology trends (increase/year)

- Transistor density : 35%
- Die size : 10-20%
- Transistor count : 40-55% (double in 18-24 mo)
 - Moore's Law
- DRAM capacity : 25-40% (double in 2-3 yr)
- Flash chip capacity : 50-60%
- Disk capacity : 40% ($30 \Rightarrow 60 \Rightarrow 100 \Rightarrow 40$)

Forms of Parallelism

- Data Parallelism
- Functional Parallelism
- Fine Grain Parallelism
- Coarse Grain Parallelism

Data Parallel Architectures

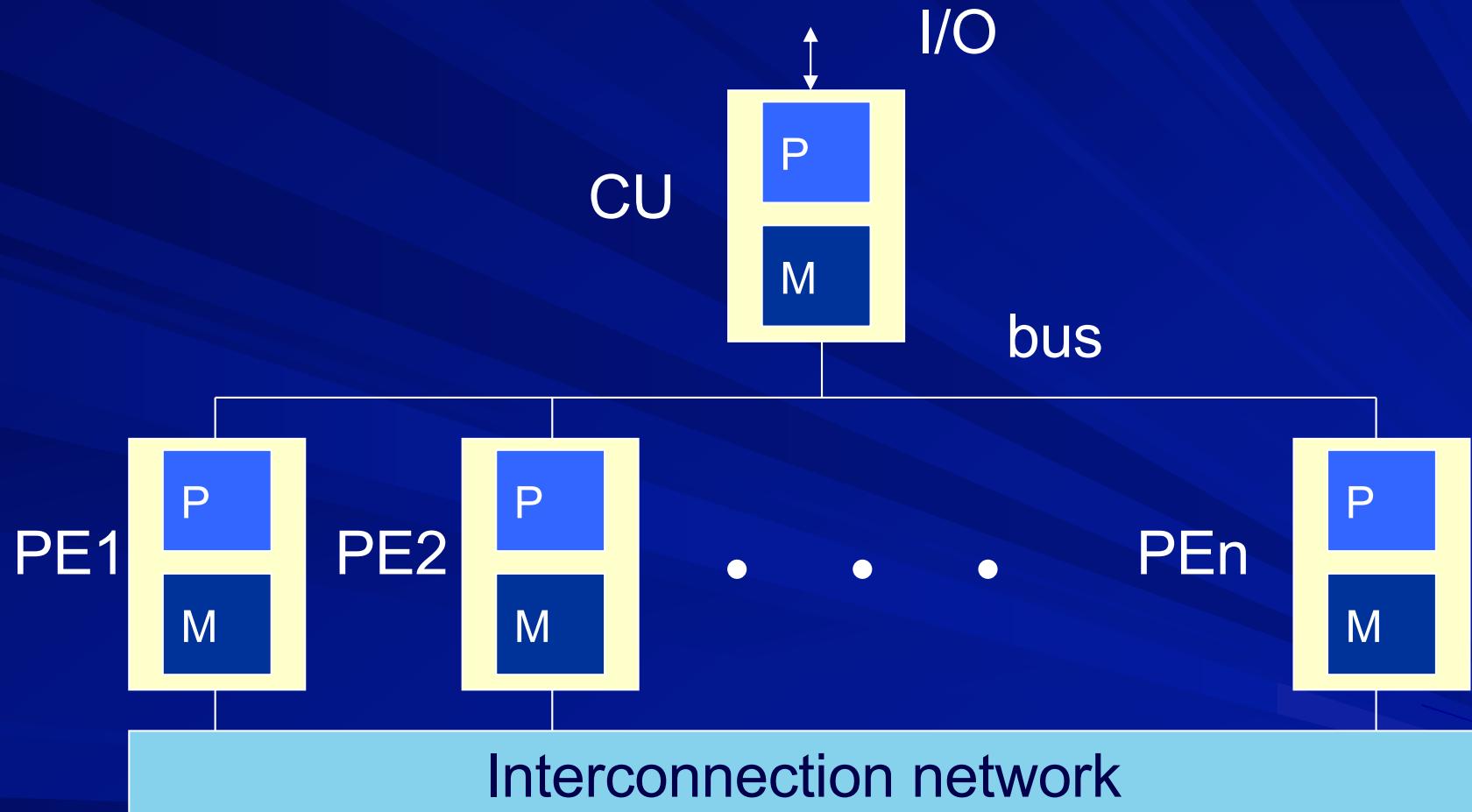
- SIMD Processors

- Multiple processing elements driven by a single instruction stream

- Vector Processors

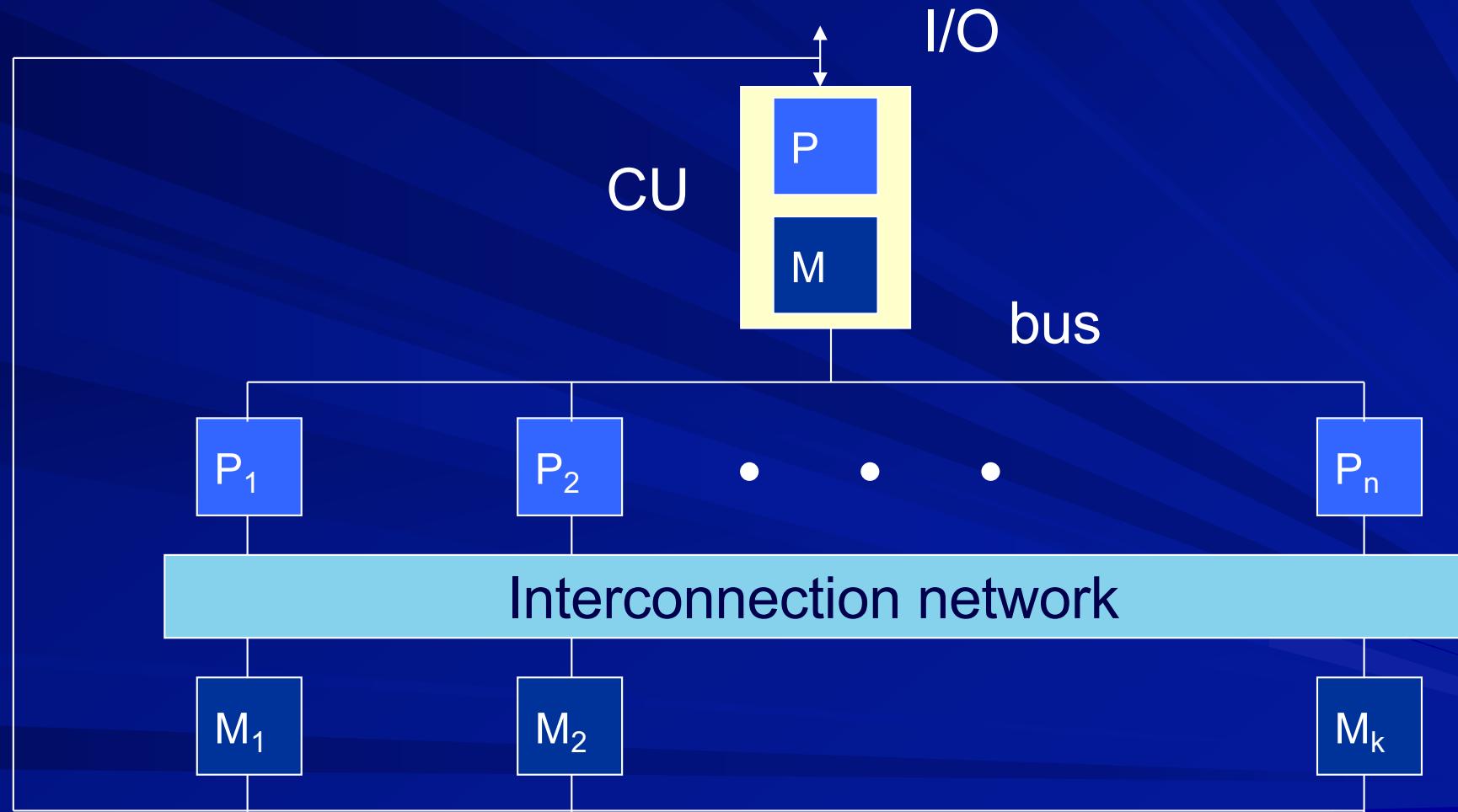
- Uni-processors with vector instructions

ILLIAC IV SIMD



Planned for 64 x 4 PEs, built only 64

Burroughs Scientific Processor



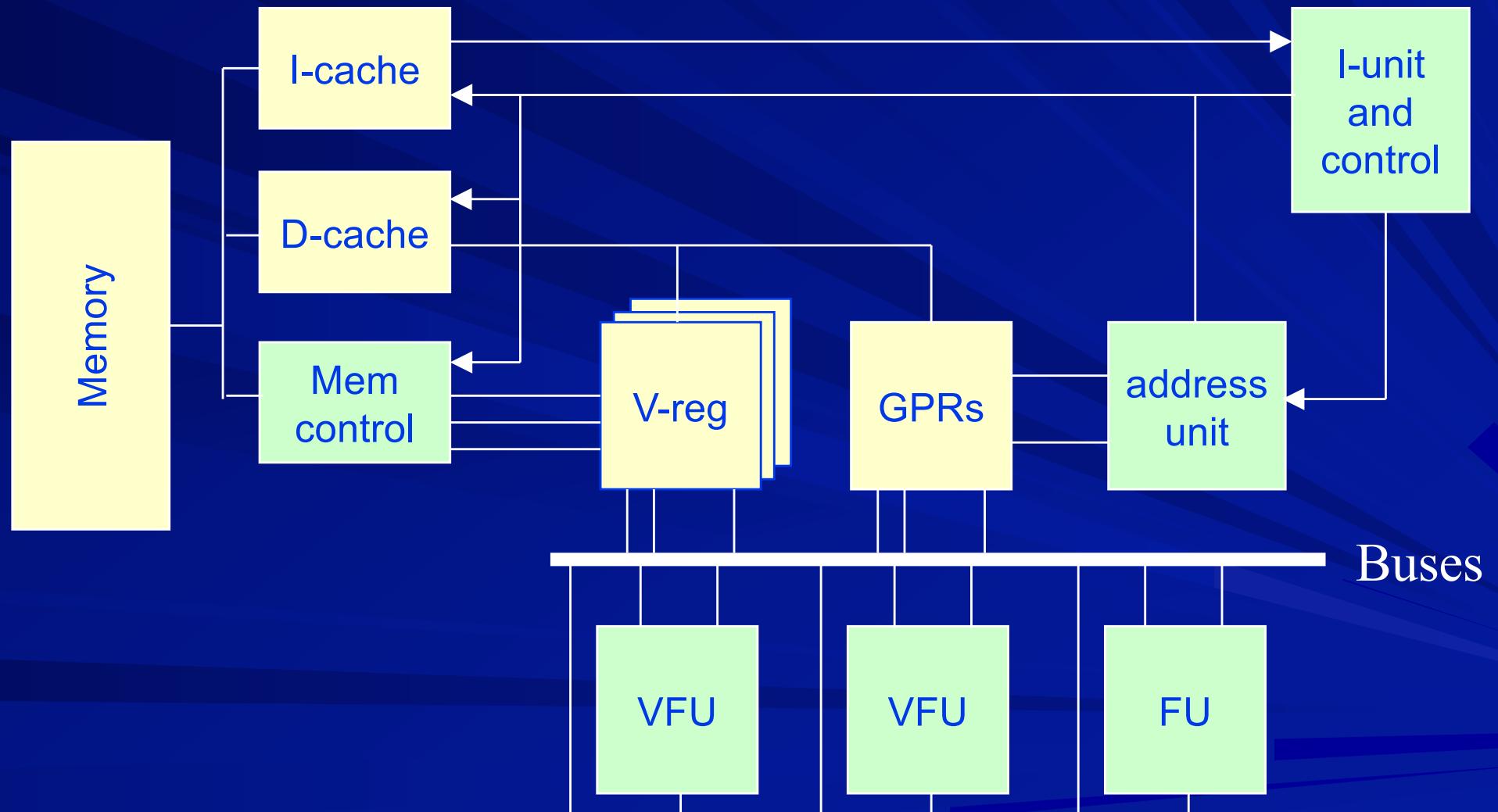
Rise and fall of SIMDs

- Introduced in 60' s (e.g. Illiac, BSP)
- Problems:
 - not cost effective
 - serial fraction and Amdahl's law
 - I/O bottle neck
- Overshadowed by Vector Processors
- Resurrected in 80' s
- Did not survive because of high cost
- Basic ideas used in some modern systems

Vector Processors

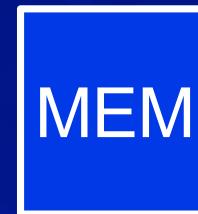
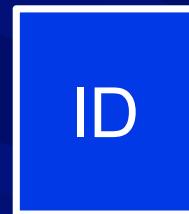
- Instructions to operate on vectors
- Vectors are streamed into and out of highly pipelined functional units
- Vector registers hold vector data
- Vectors are streamed from memory into vector registers and from vector registers into memory, no cache
- Sequences of vector operations are chained

Vector Processor Architecture

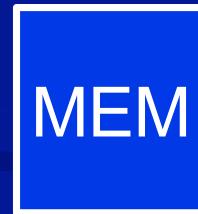
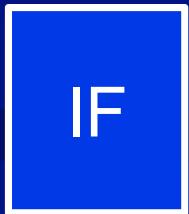


Fine Grain Functional Parallelism

- Scalar pipeline : IPC ≤ 1



- Instruction Level Parallelism



Finding Instruction Level Parallelism

- Dynamic (Superscalar)

- Using hardware
- Expensive (hardware cost, power consumption)
- Analyze instruction window
- All delays and dependencies known

- Static (VLIW)

- Using software (compiler)
- Inexpensive
- Analyze whole program
- All delays and dependencies not known

How superscalars work

- Multiple pipelined EUs for parallel execution
- Fetch multiple instructions in a buffer, parallel decode – instruction window
- Dynamic scheduling - out of order
- Dependency check and resource check before execution
- Speculation
- Preserving the sequential consistency and exception processing - in order retirement

Limits on ILP

- Limited ILP in the application (available ILP)
- Limitation of HW and SW in exploiting given ILP (achieved ILP)
 - limited EUs
 - limited instruction window
 - limited issue capability
 - renaming limitations
 - imperfect branch/jump prediction
 - imperfect load/store speculation
 - imperfect cache

Beyond ILP

- Multithreading over ILP Datapath
- Multithreading over multiple cores

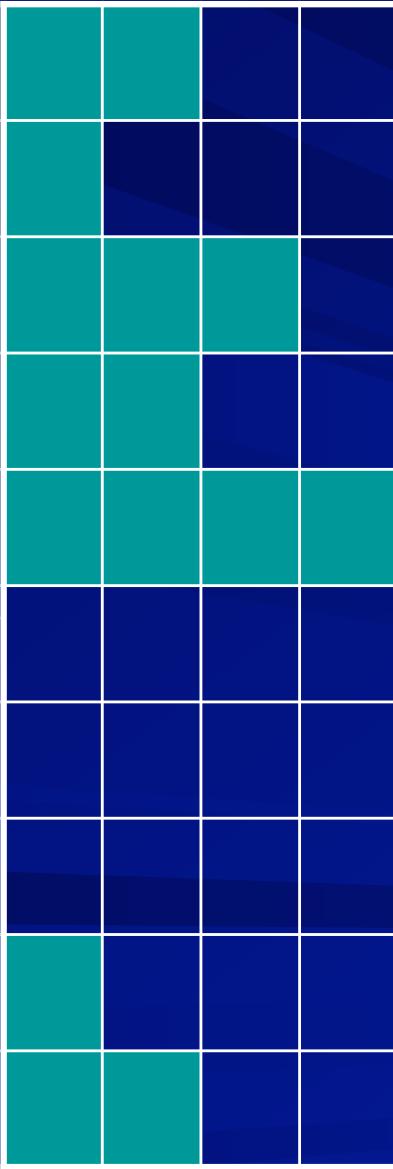
Multithreading over ILP datapath

- The state needs to be replicated for each thread
 - program counter
 - registers
- Memory is shared
- Require ability to switch from one thread to other quickly

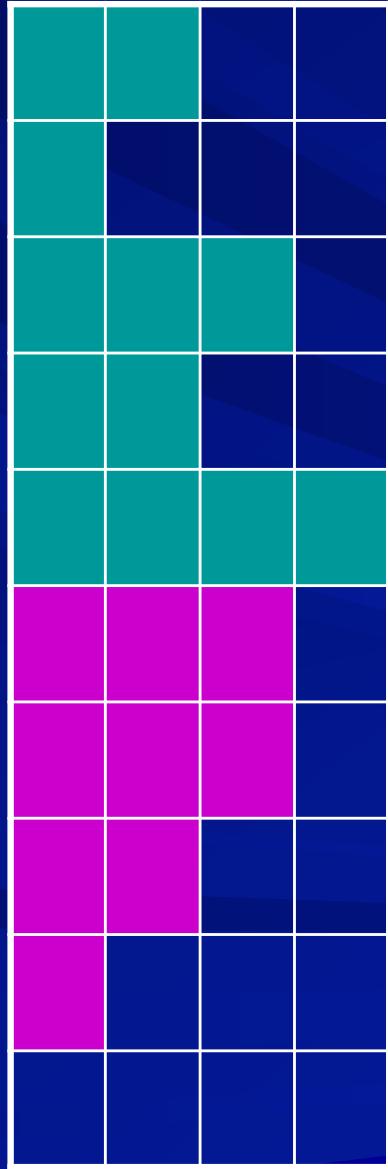
ILP and Multithreading

ILP

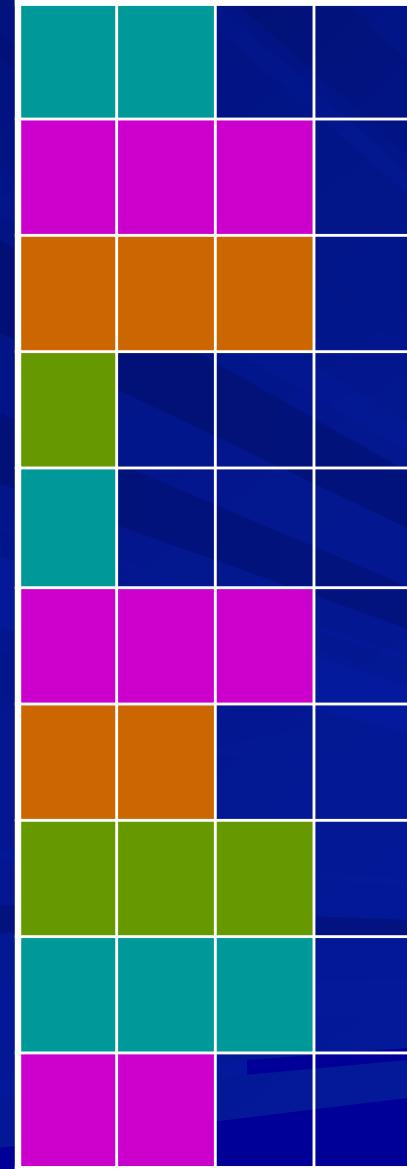
Hennessy and Patterson



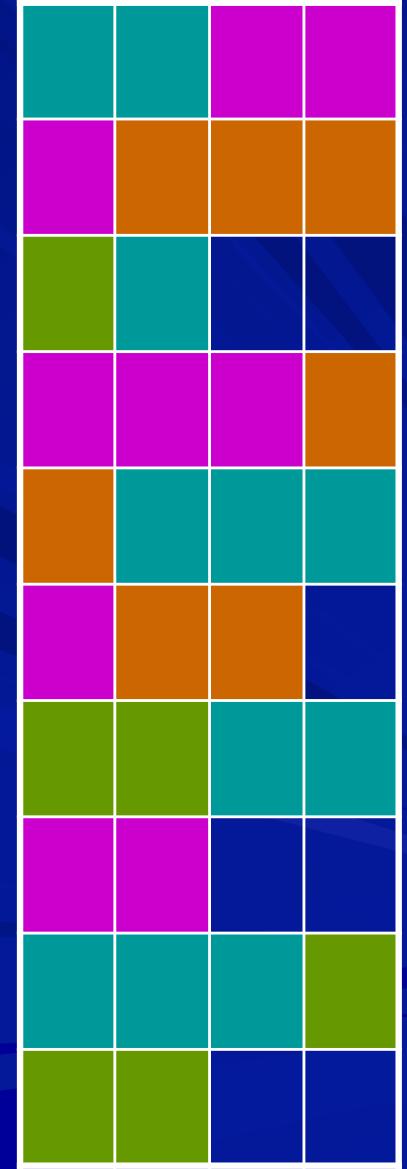
Coarse MT



Fine MT



SMT



ILP, SMT and Multicores

■ ILP

- Multiple execution units
- RF, caches and memory shared

■ SMT

- Multiple execution units and RFs
- Caches and memory shared

■ Multicore

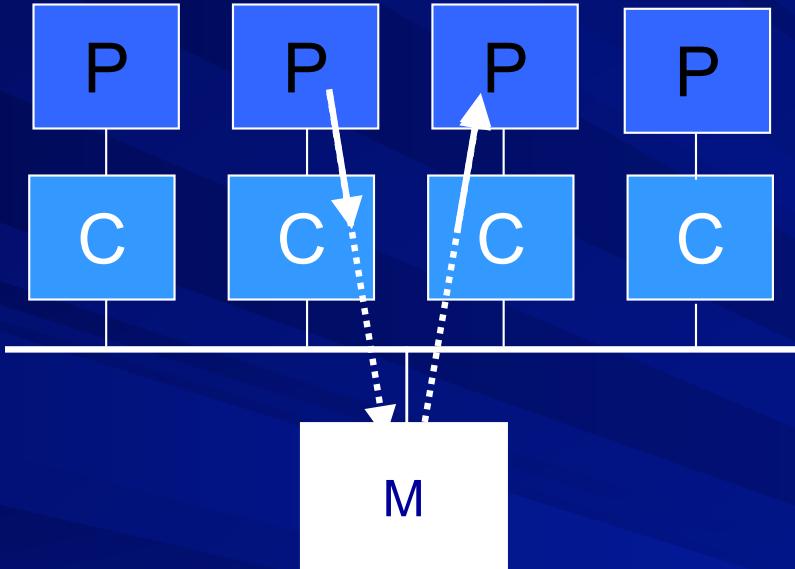
- Multiple EUs, RFs and caches (upper levels)
- Caches (lower levels) and memory shared

more sharing



ease of design

Memory hierarchy in Multicores



- Multiple copies of data in caches may exist
- ⇒ Problem of cache coherence

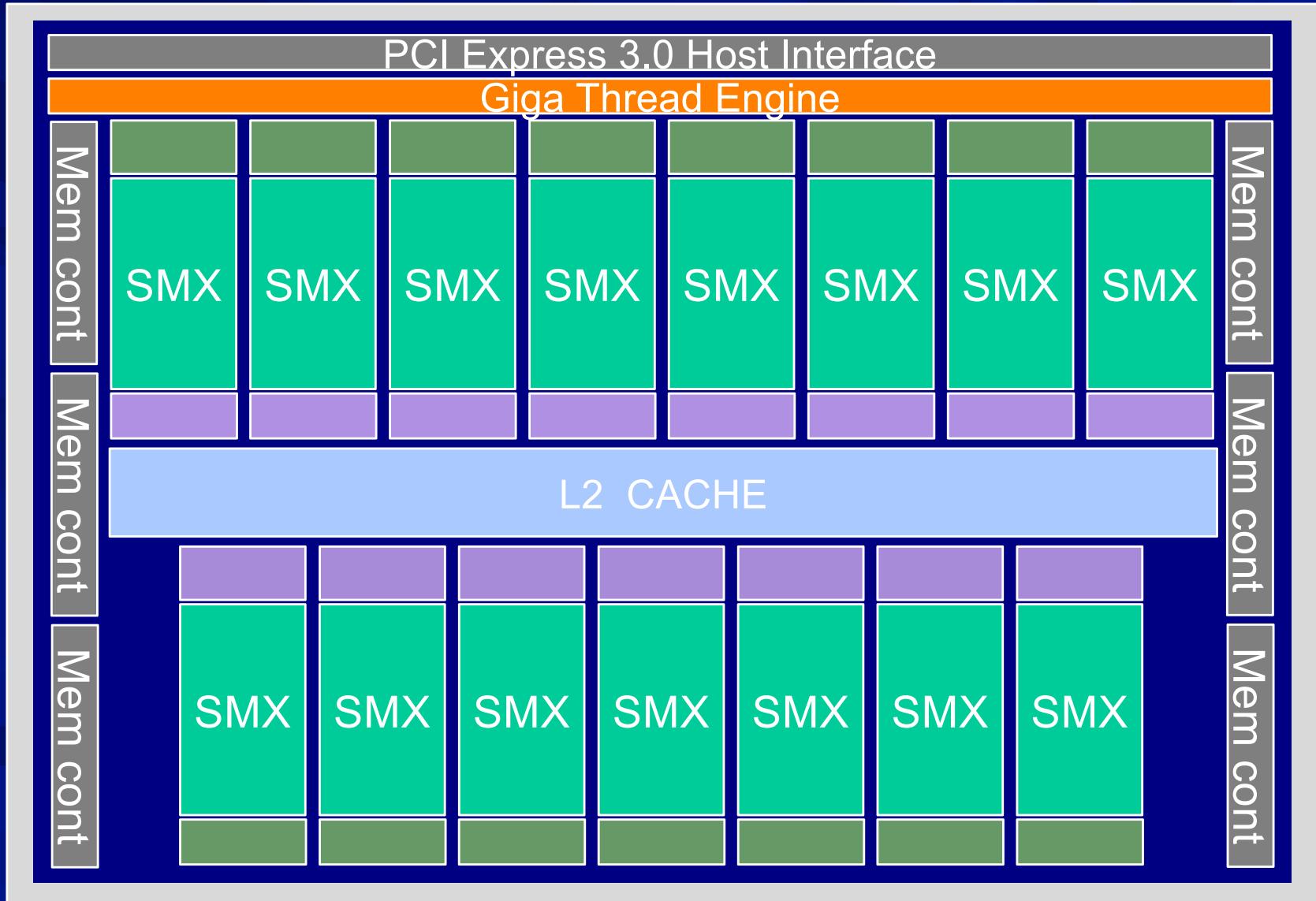
Solution:

- Cache controllers follow “coherence protocol”
- Interaction and communication in a mutually agreed manner

Graphics processors

- Developed originally for accelerating graphics
- Now used for General Purpose computing as well
- Major manufacturers : Nvidia, AMD, Intel
- Support for programming : CUDA (for Nvidia devices), OpenCL (general)

Nvidia's Kepler GK110



SMX: Streaming Multiprocessor



16 Rows

192 CUDA cores, 64 DPUs

32 LSUs, 32 SFUs



Thanks