

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

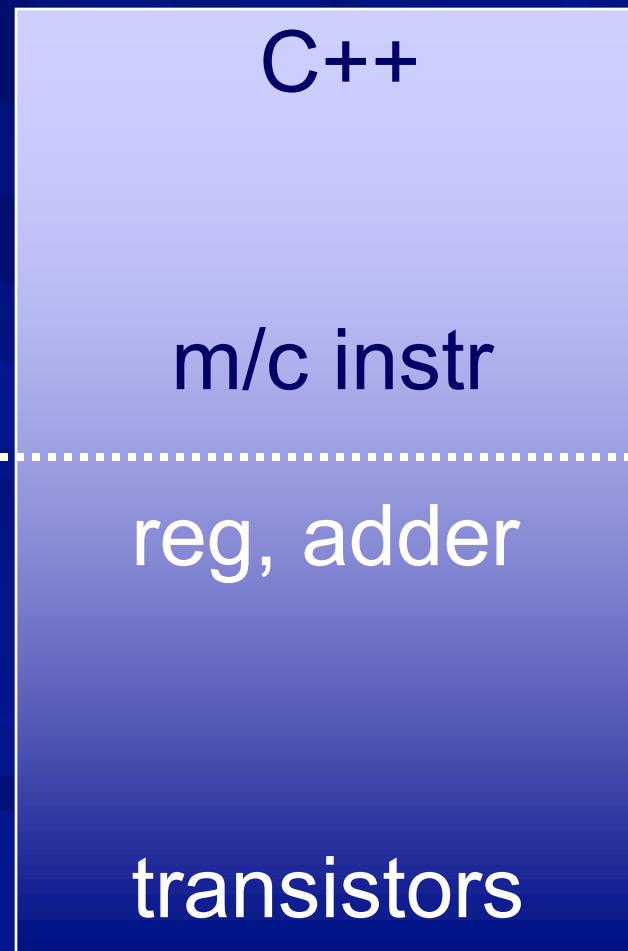
CIRCUITS

- 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:

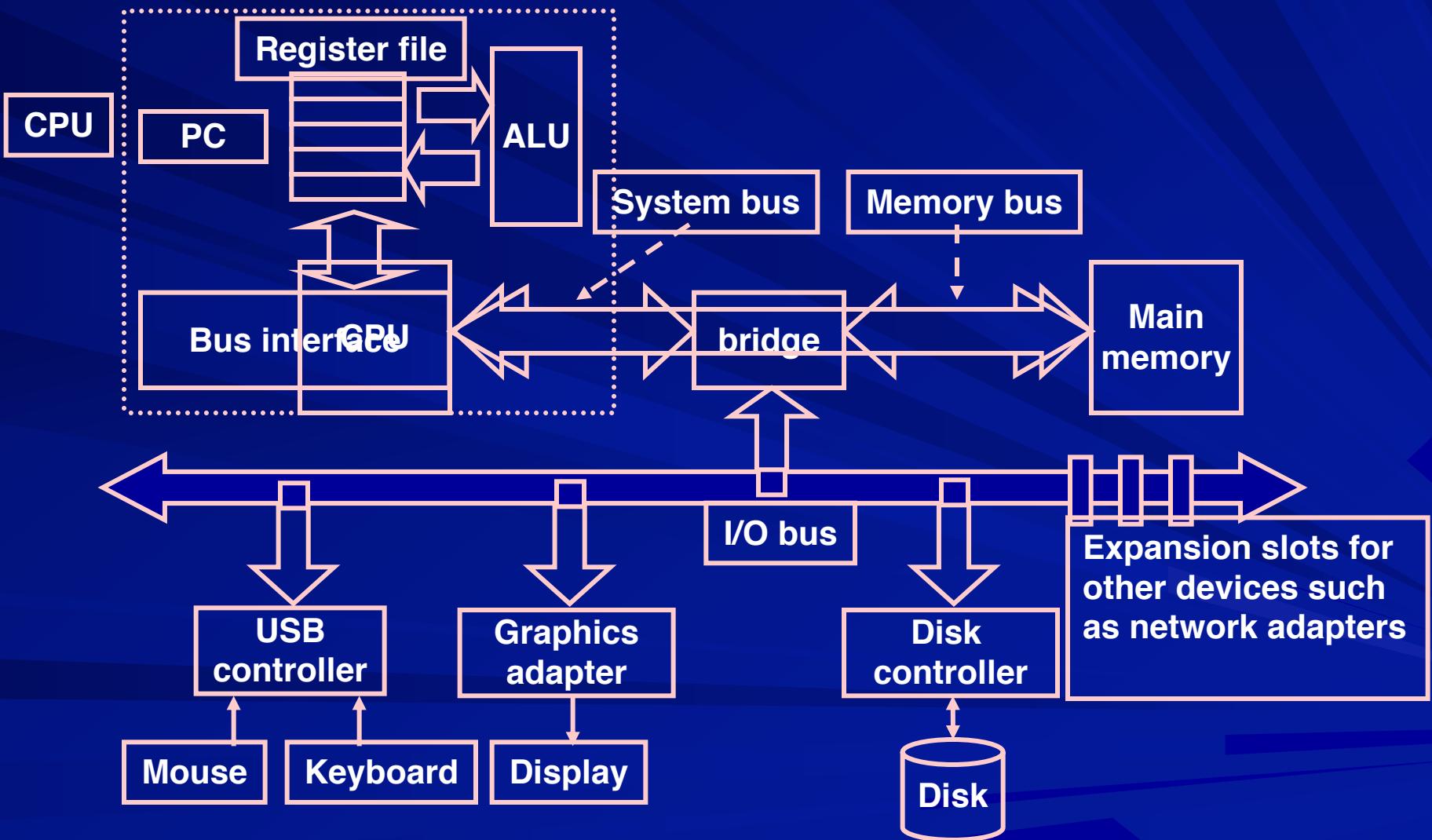
```
mul r3, r5, r4  
add r2, r3, r2  
ldr r6, [r2, #0]  
ldr r7, [r2, #4]  
str r7, [r2, #0]  
str r6, [r2, #4]  
mov pc, lr
```

assembly

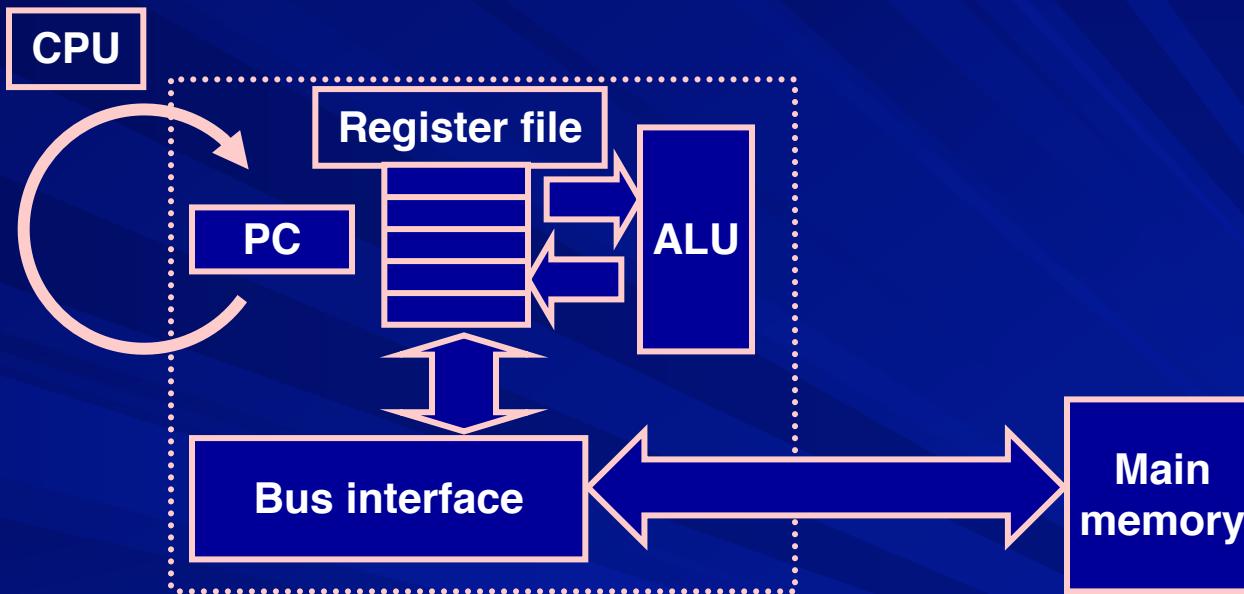
machine
code

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

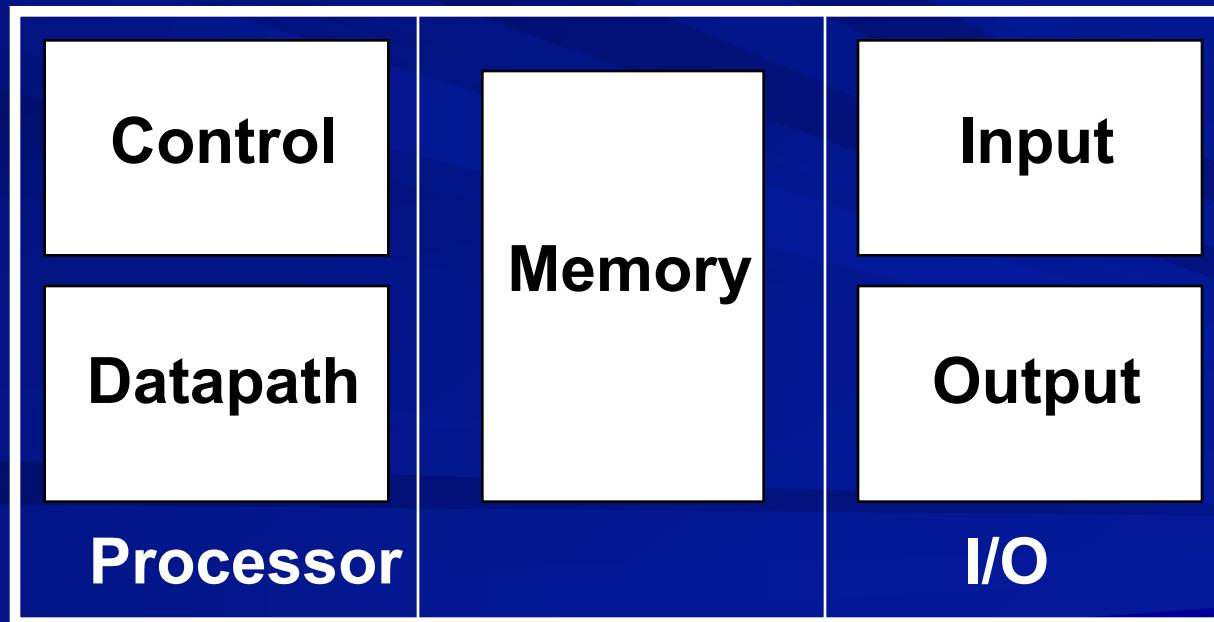
add r1, r7, r4

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

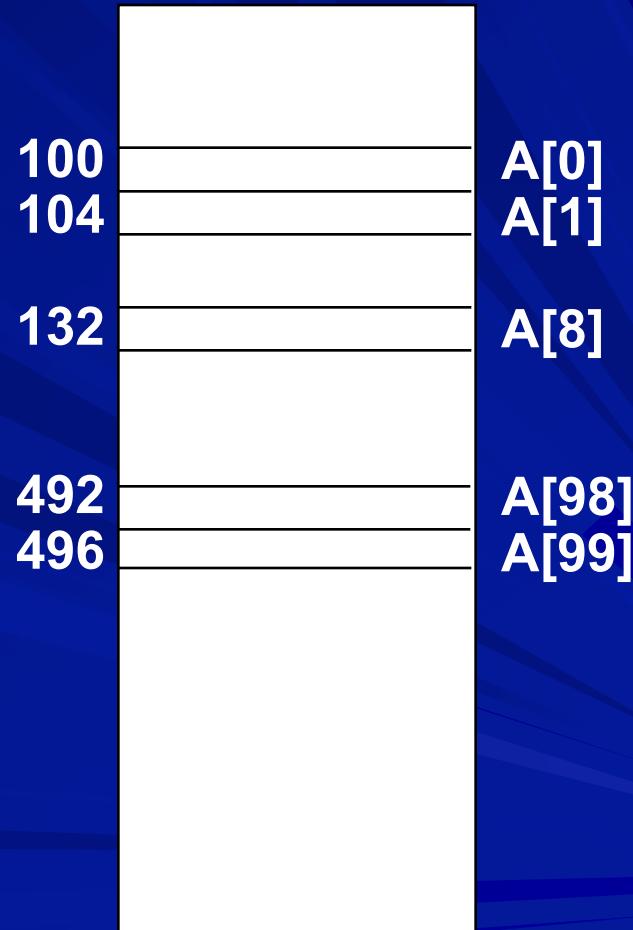
C : $v = A[8];$

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 $\sum_i A[i]$

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

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

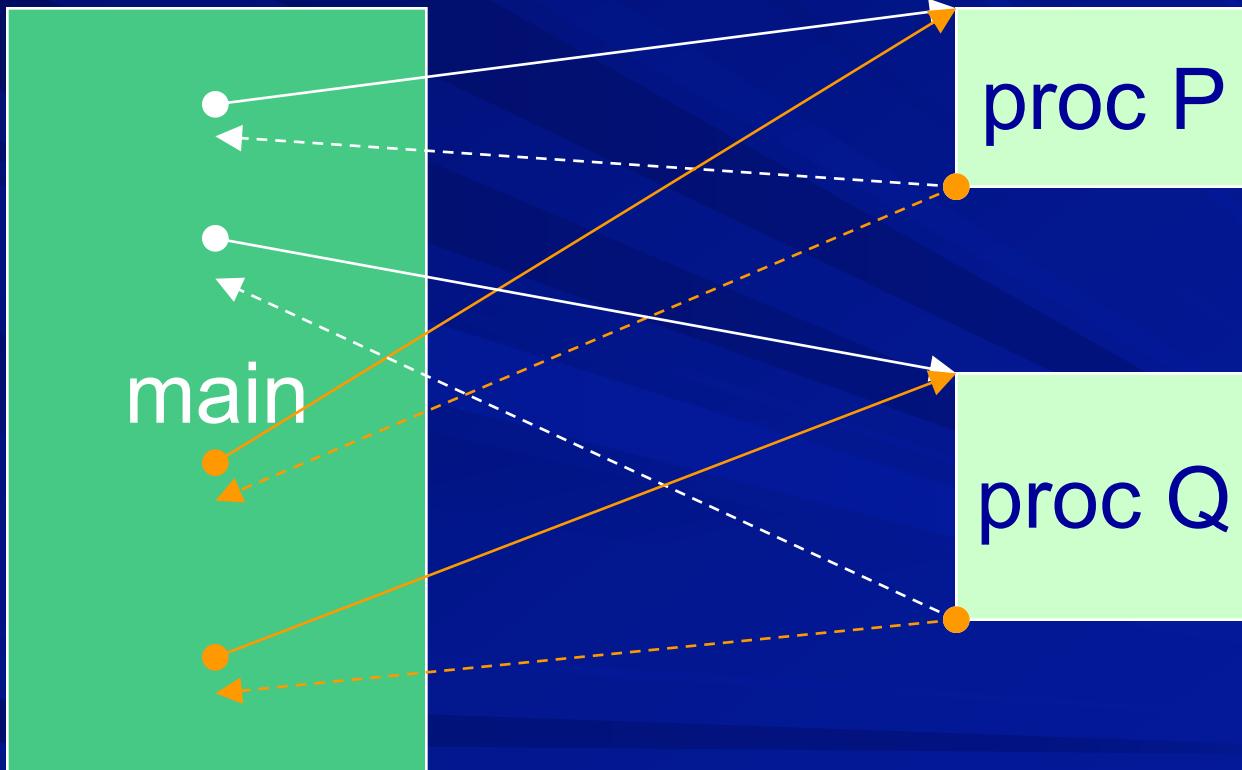
Instruction set design goals

- 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

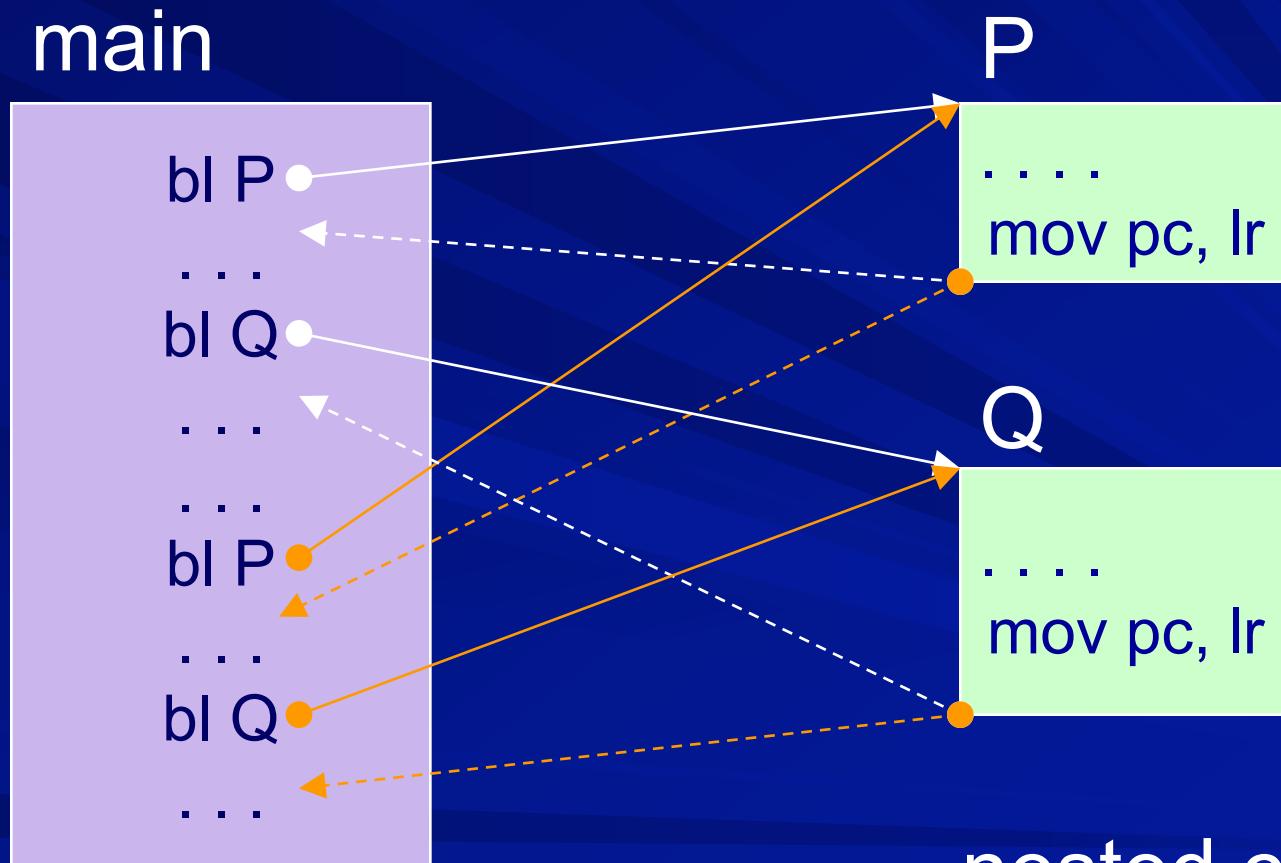
```
void func ( ) {  
    ...  
    ...  
    return;  
}
```

```
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 starts with the first four instructions (mov r1, #0; ldr r3, =AA; add r6, r3, #400) highlighted in a box. An arrow points from this box to another box containing four instructions: ldr r3, =AA; add r6, r3, #400; ldr r1, [r3, #0]; add r3, r3, #4. Another arrow points from the 'add r1, r1, r5' instruction to a box containing the text '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

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

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

add r1, r2, r3

		0	opc	Rn	Rd		Rm
						8	4

add r1, r2, #3

	1	opc	Rn	Rd		Imm
					4	8

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:

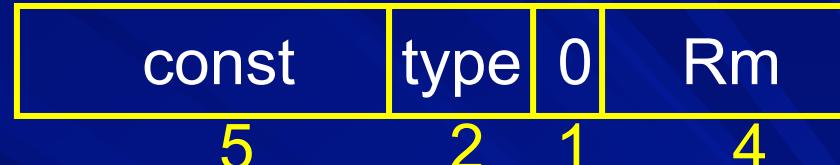
- LSL logical shift left
- LSR logical shift right
- ASR arithmetic shift right
- ROR rotate right

Shift amount

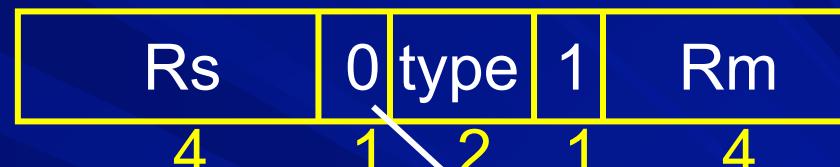
- 5 bit unsigned constant # 0 . . # 31
- 4 bit register number r0 . . r15

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

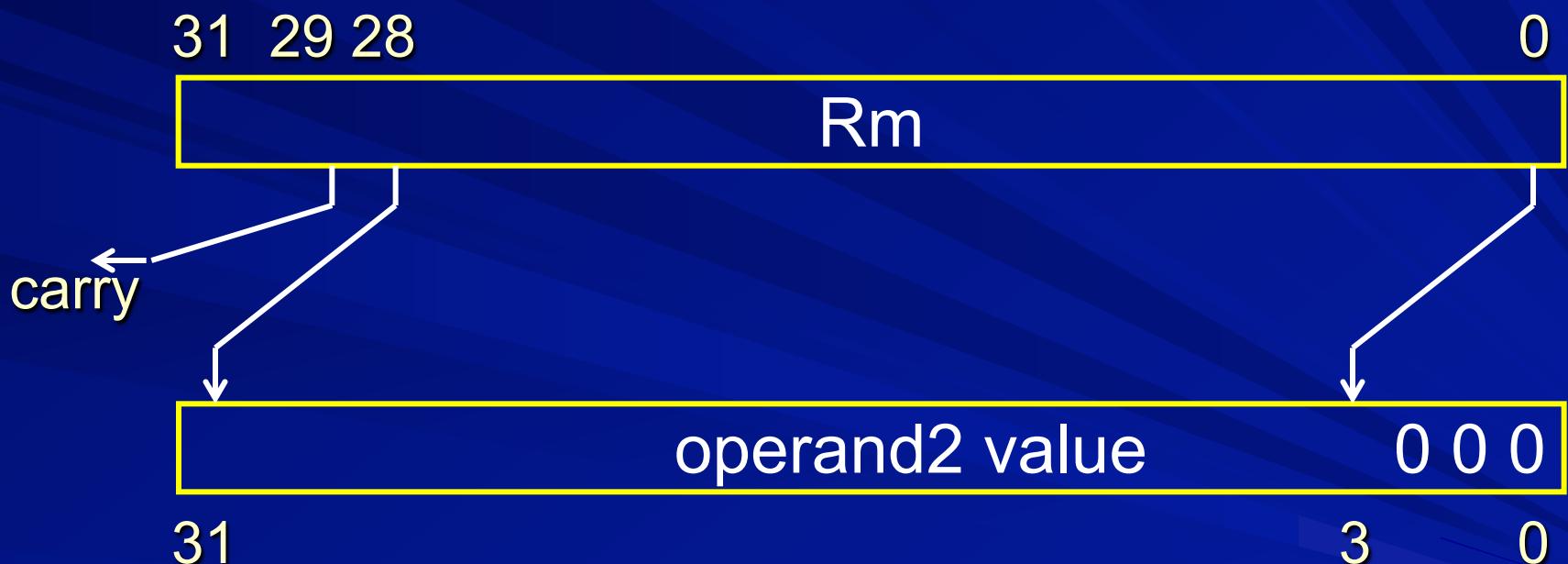
arithmetic shift right

rotate right

1 for multiply and
other instructions

Shift types: Logical Shift Left

■ LSL # 3



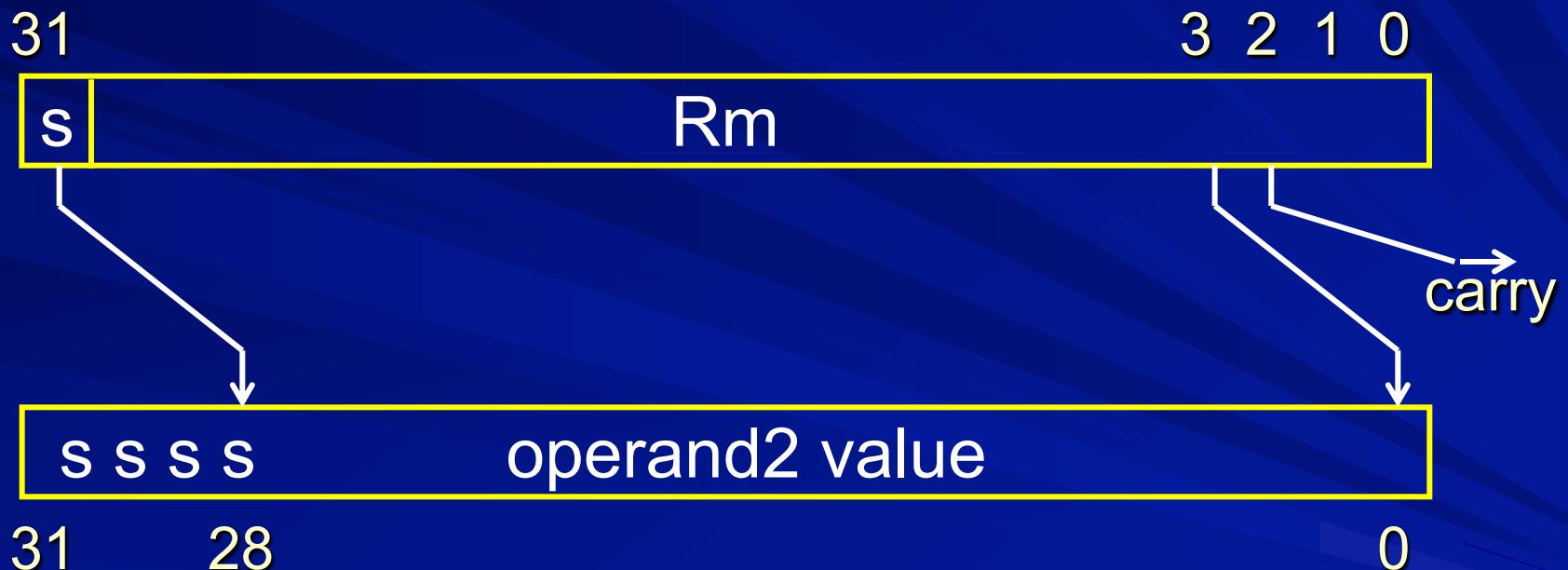
Shift types: Logical Shift Right

■ LSR # 3



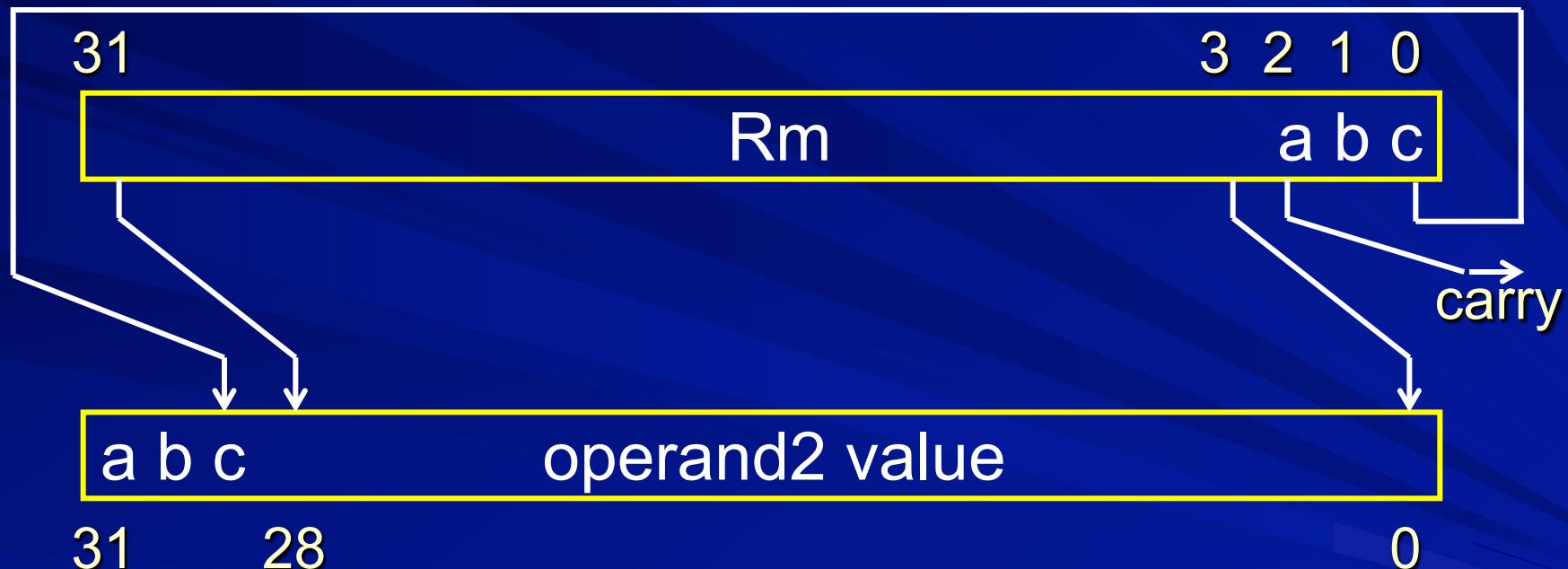
Shift types: Arithmetic Shift Right

■ 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 <= Memory [Rn + 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]

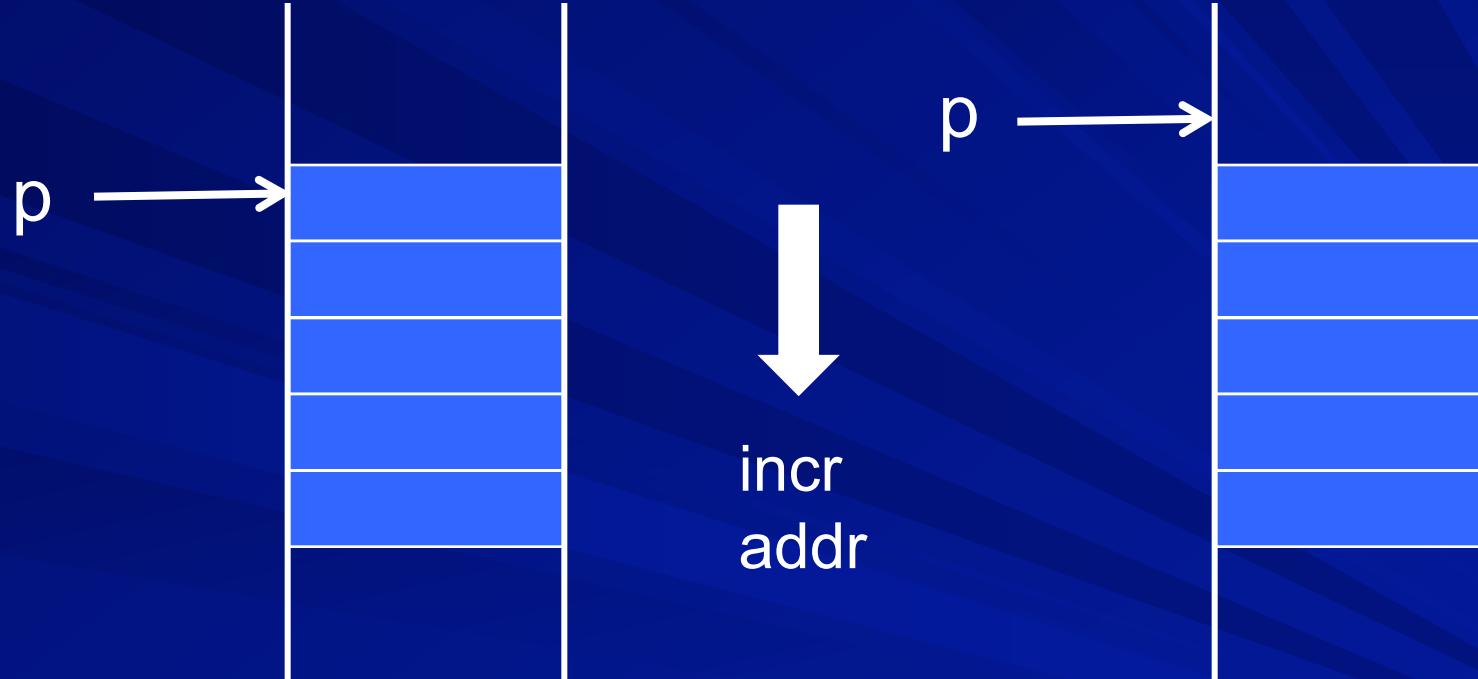
ldrb r4, [r5, # 32] !

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:

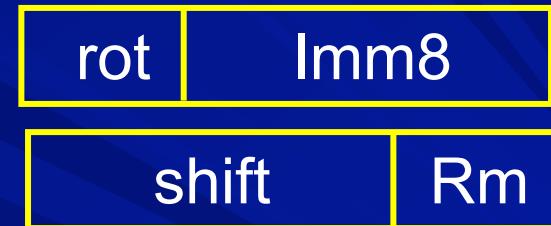
- Absolute or relative?
- Byte or word?

DP instructions



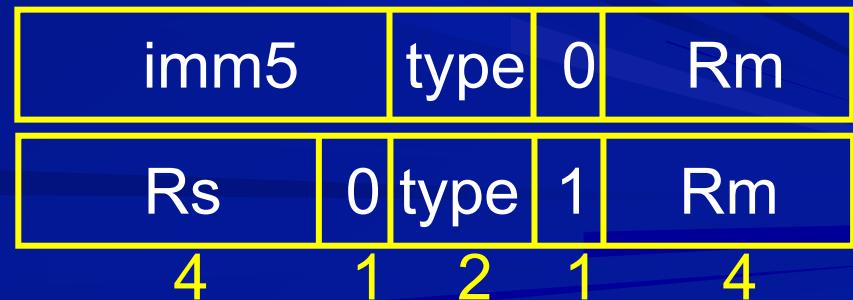
12 bit operand2

- 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

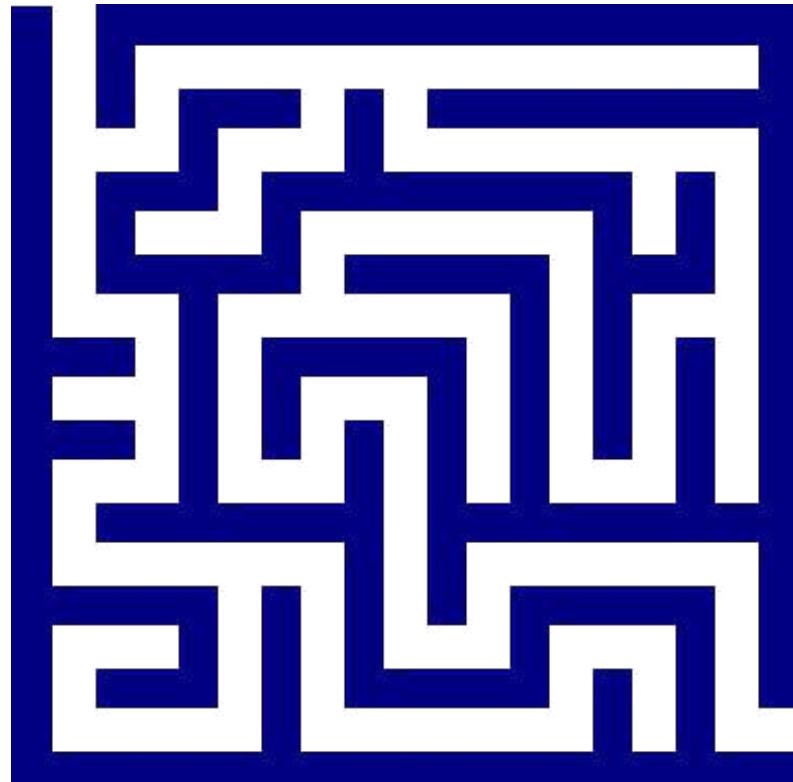
SWI instruction

F=11	cond	11	11	function
4	2	2		24

Software Interrupt
used for invoking OS function

Recursive function example

- Solving a MAZE



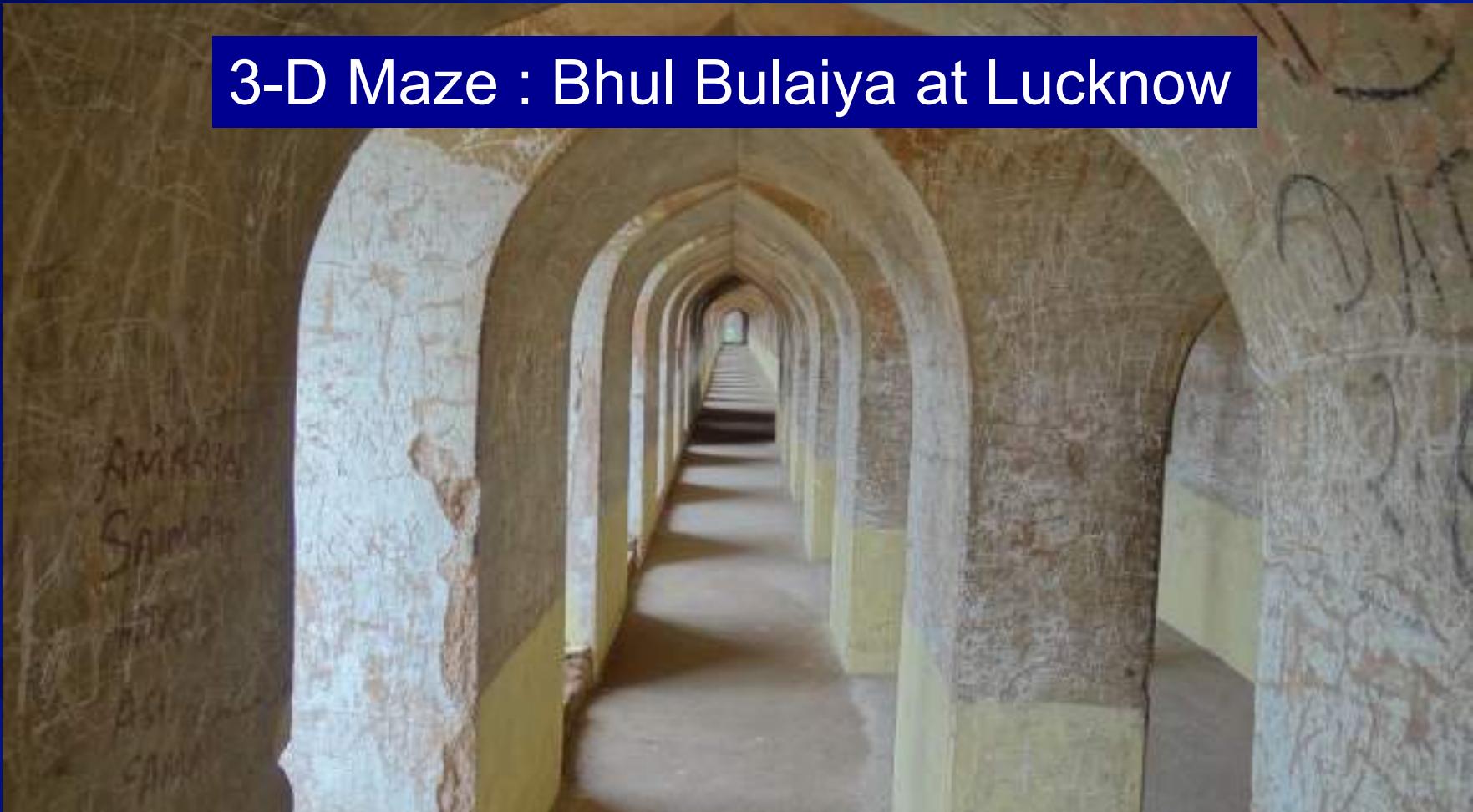
Real Life Maze

Hedge Maze at St Louis, USA



Real Life Maze

3-D Maze : Bhul Bulaiya at Lucknow



Real Life Maze



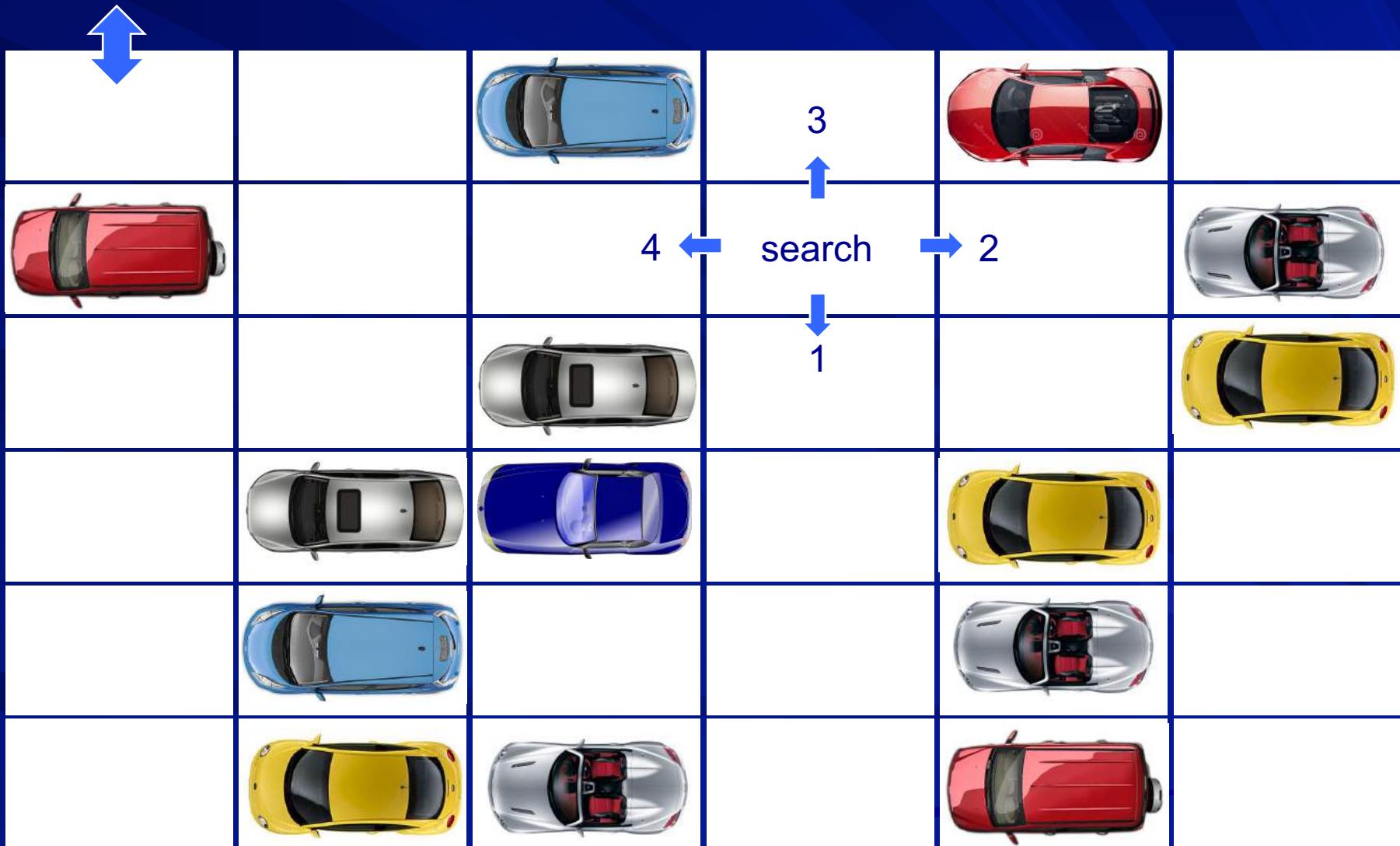
Maze to be actually solved by computer



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]
```

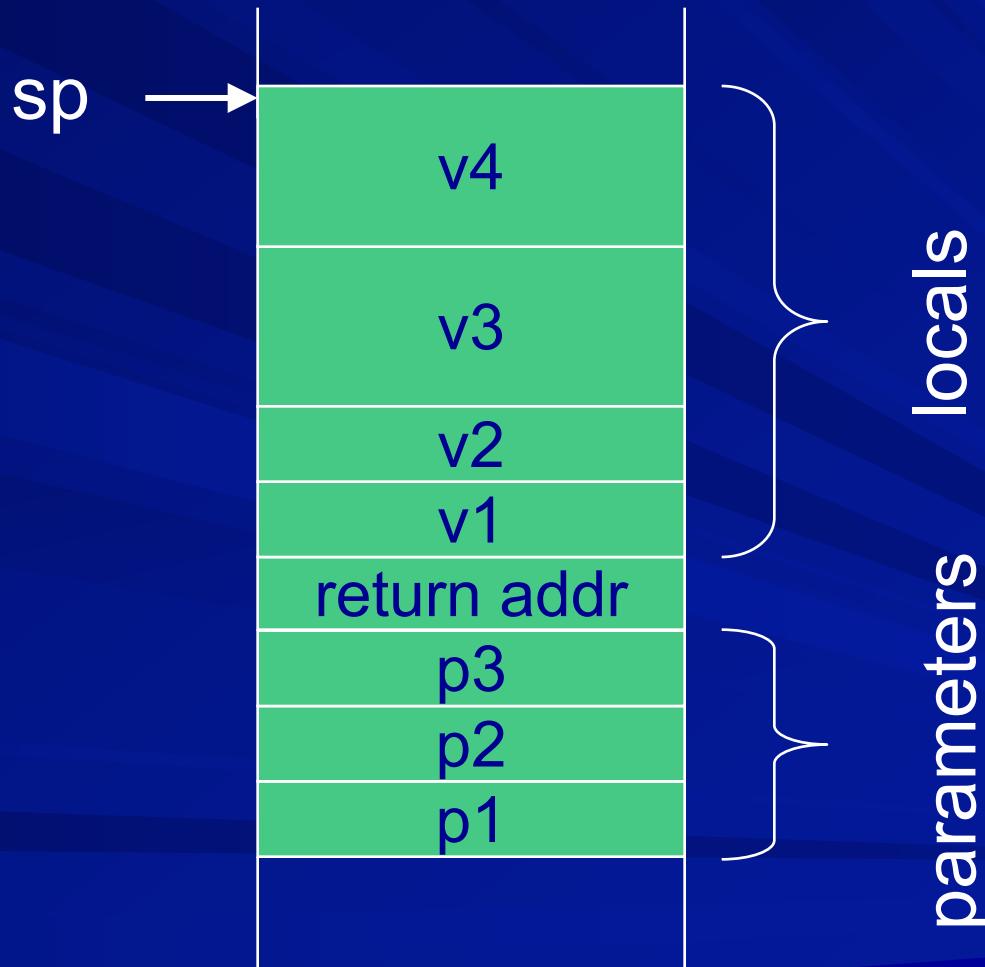
```
    cmp r4, #1
```

```
    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);

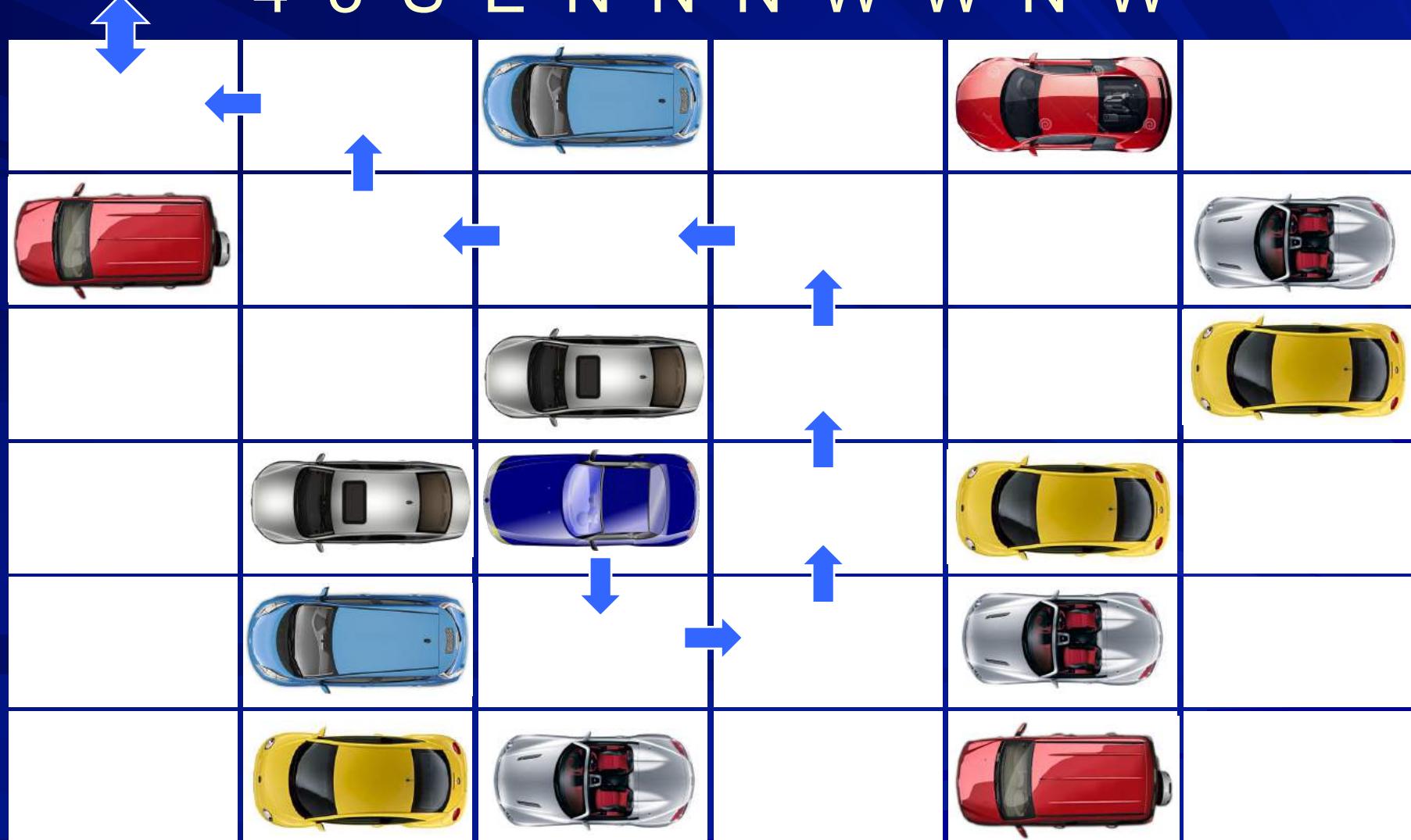
Ret1: mov r0, #1
ldr lr, [sp, #8]
add sp, sp, #12
mov pc, lr

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

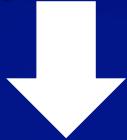
- High level program



- 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

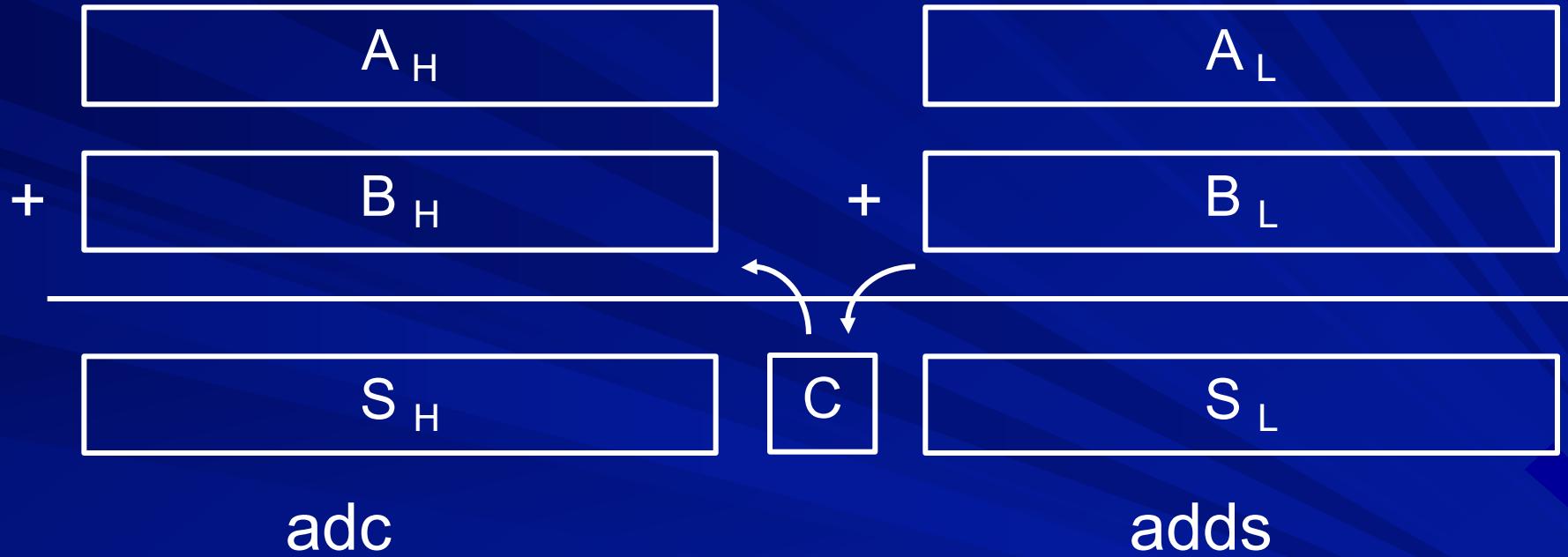
sub r1, r2, r1

add r1, r1, r2

sub r2, r1, r2

sub r1, r1, r2

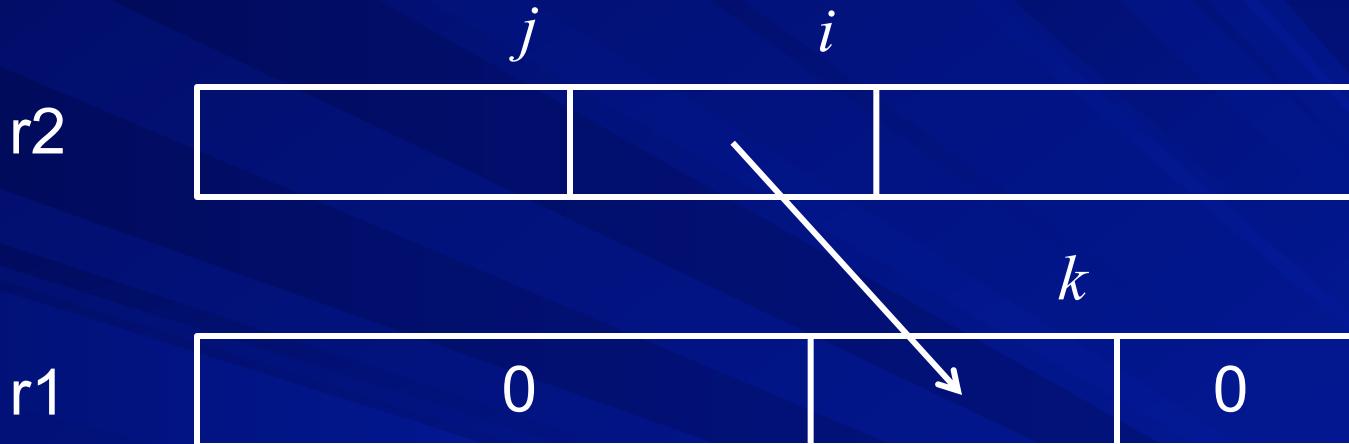
Double word addition



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

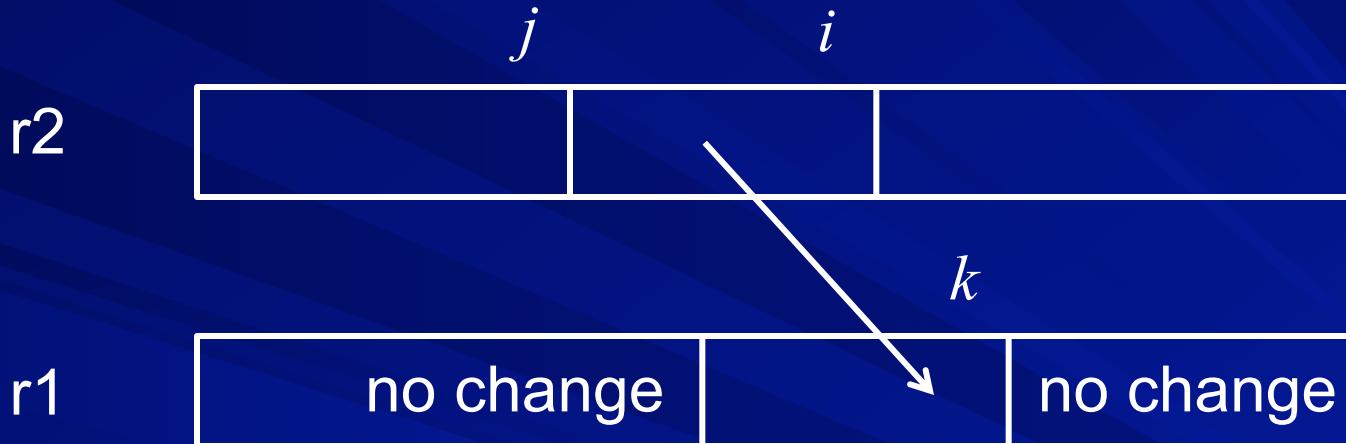


```
mov r1, r2, LSL # 32 - j
```

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

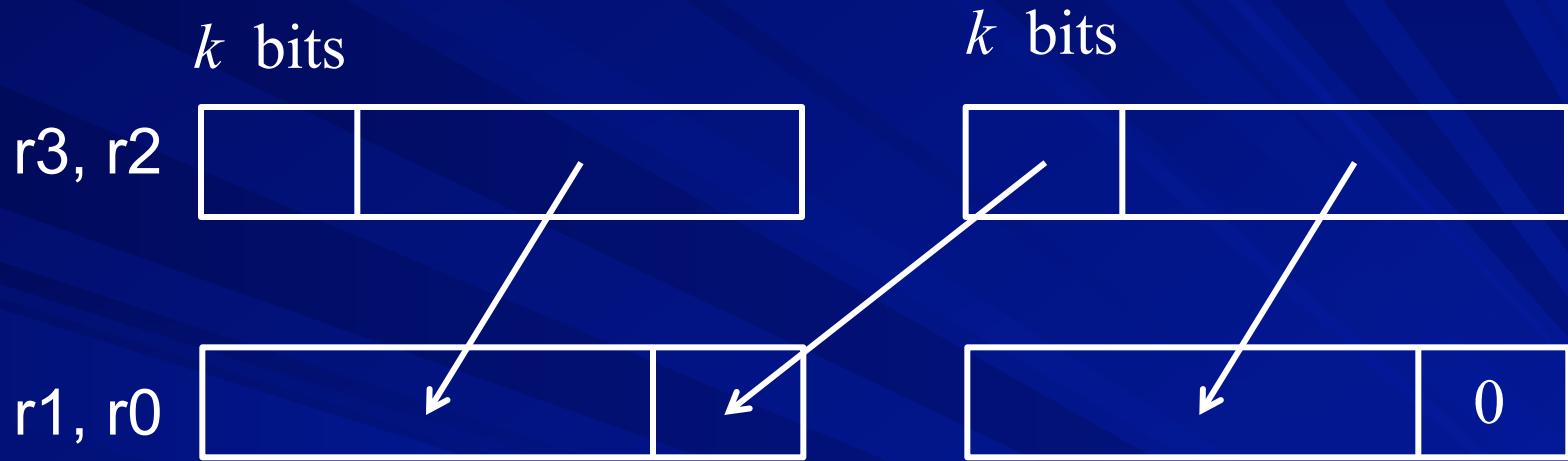
```
mov r1, r1, LSL # k
```

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

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

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;};
```

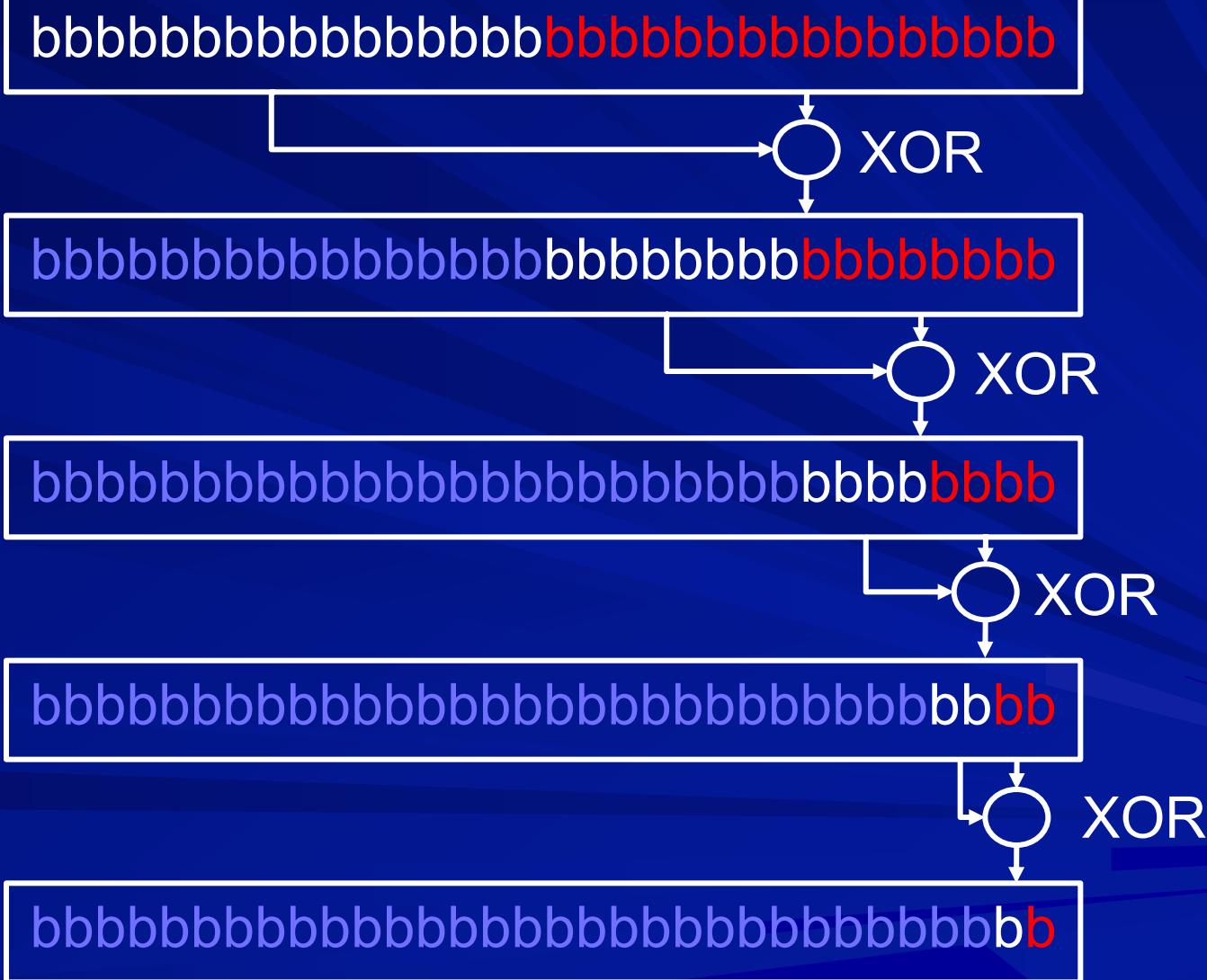
Find k where $v = 2^k$

```
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

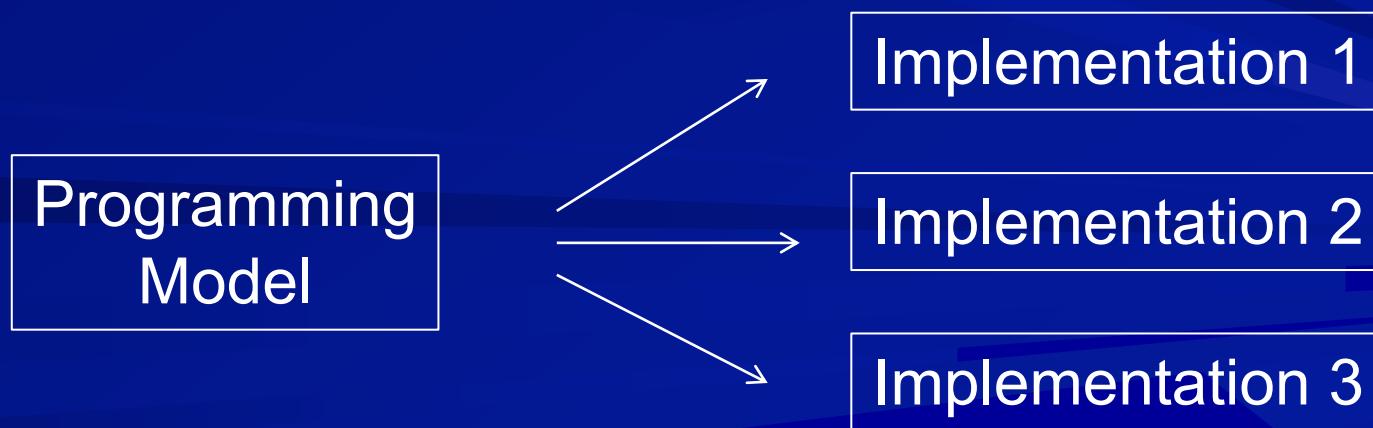
What is “Computer Architecture”?

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



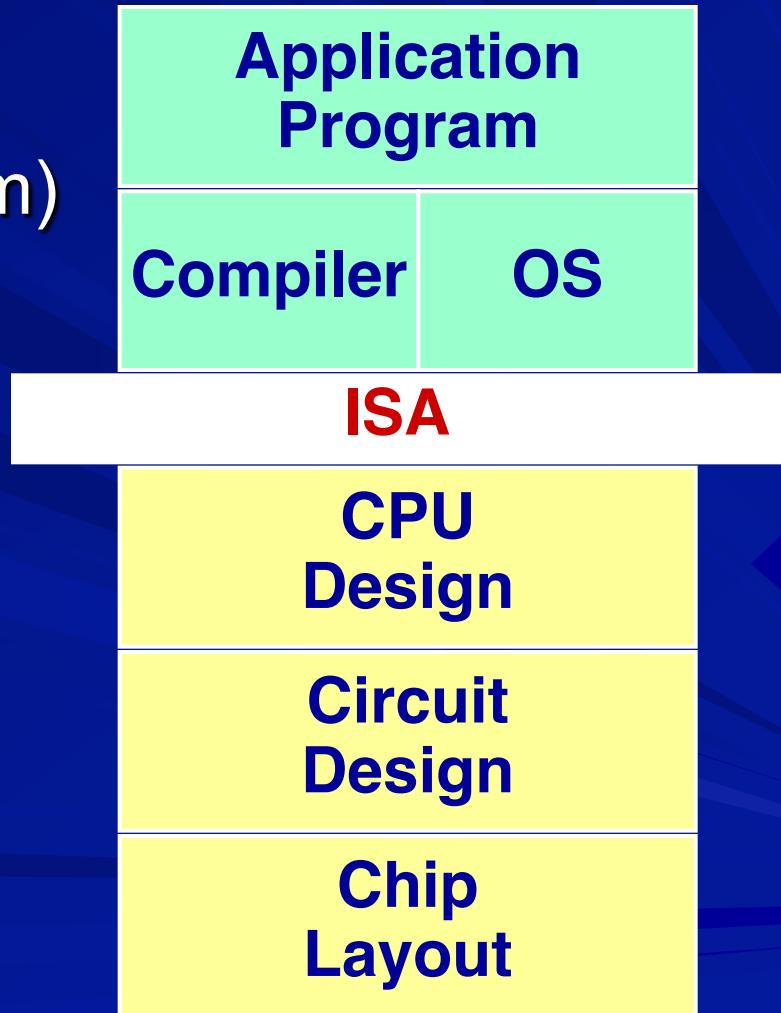
Architecture levels

- 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

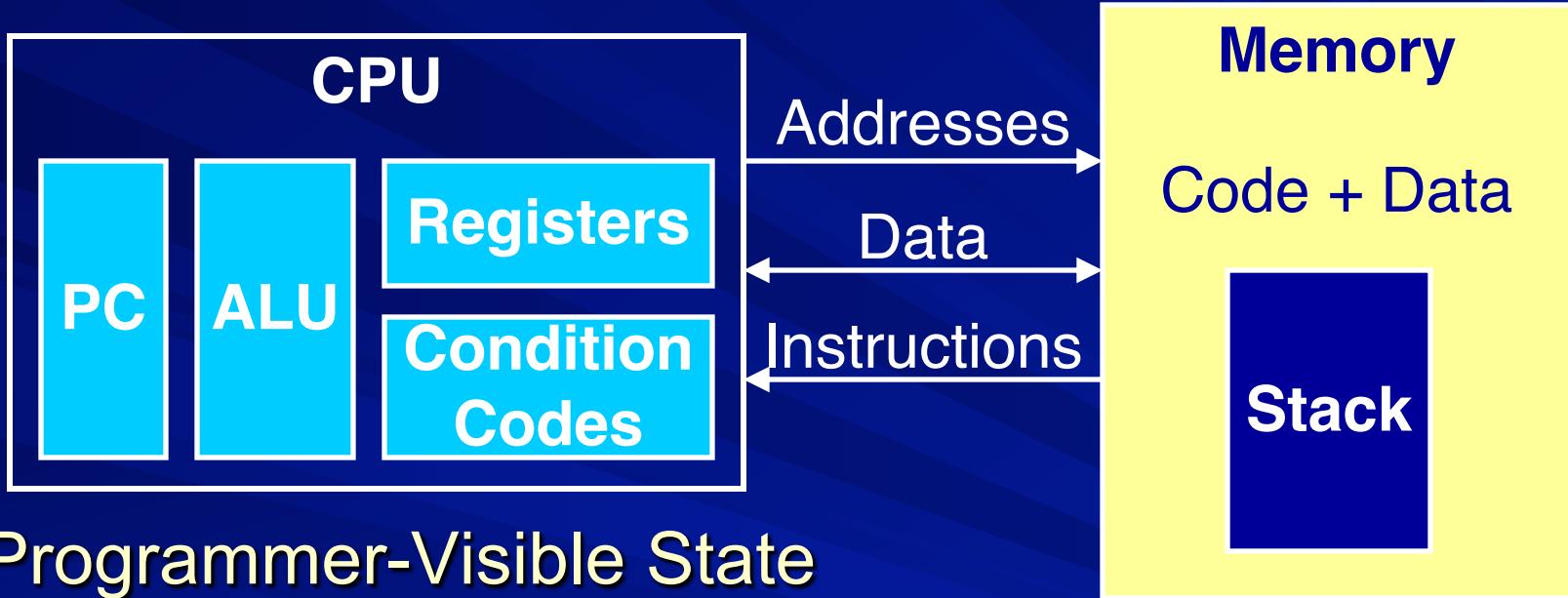
Instruction Set Architecture

- 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

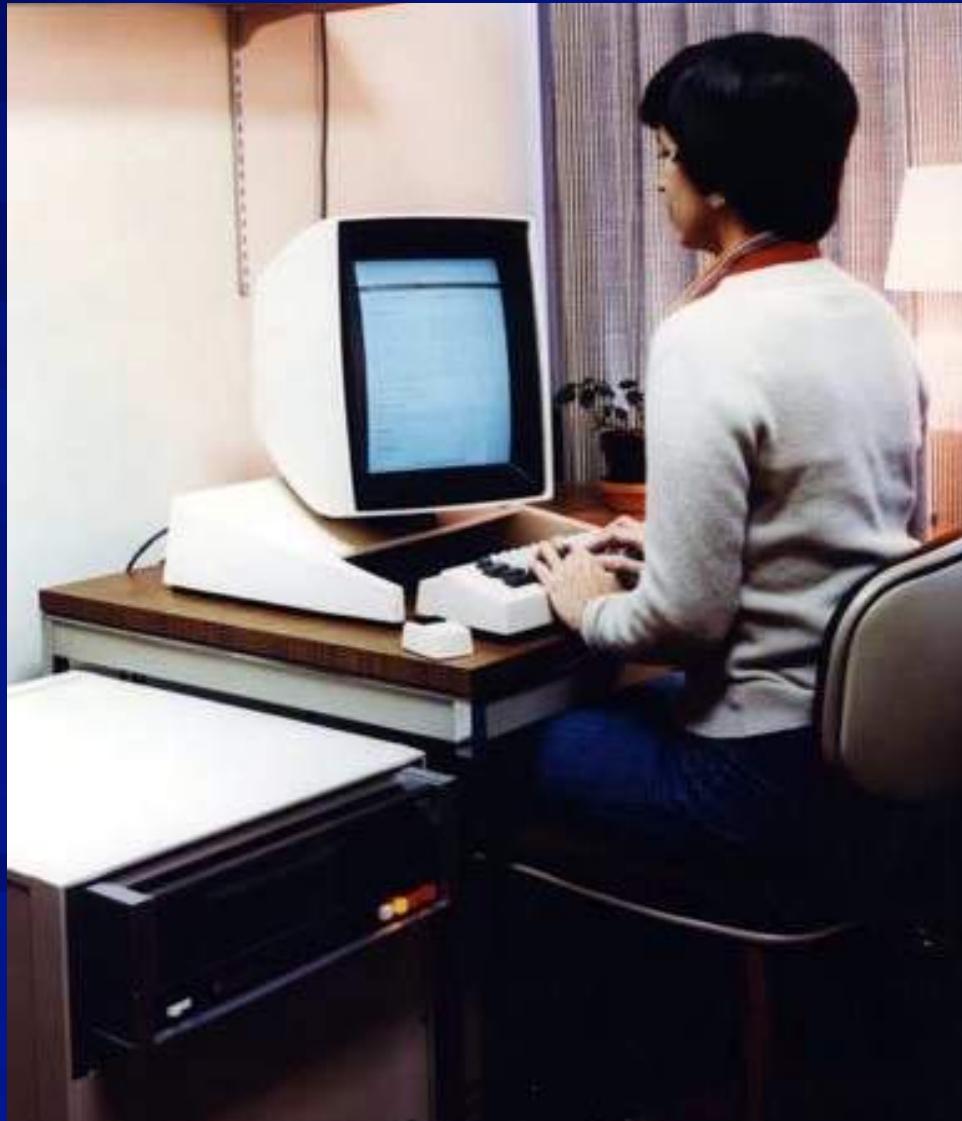


Mini
computer

IBM 360, mid 1960s



Xerox PARC Alto, mid 1970s



Personal
computer

IBM PC, early 1980s



Apple Macintosh, mid 1980s



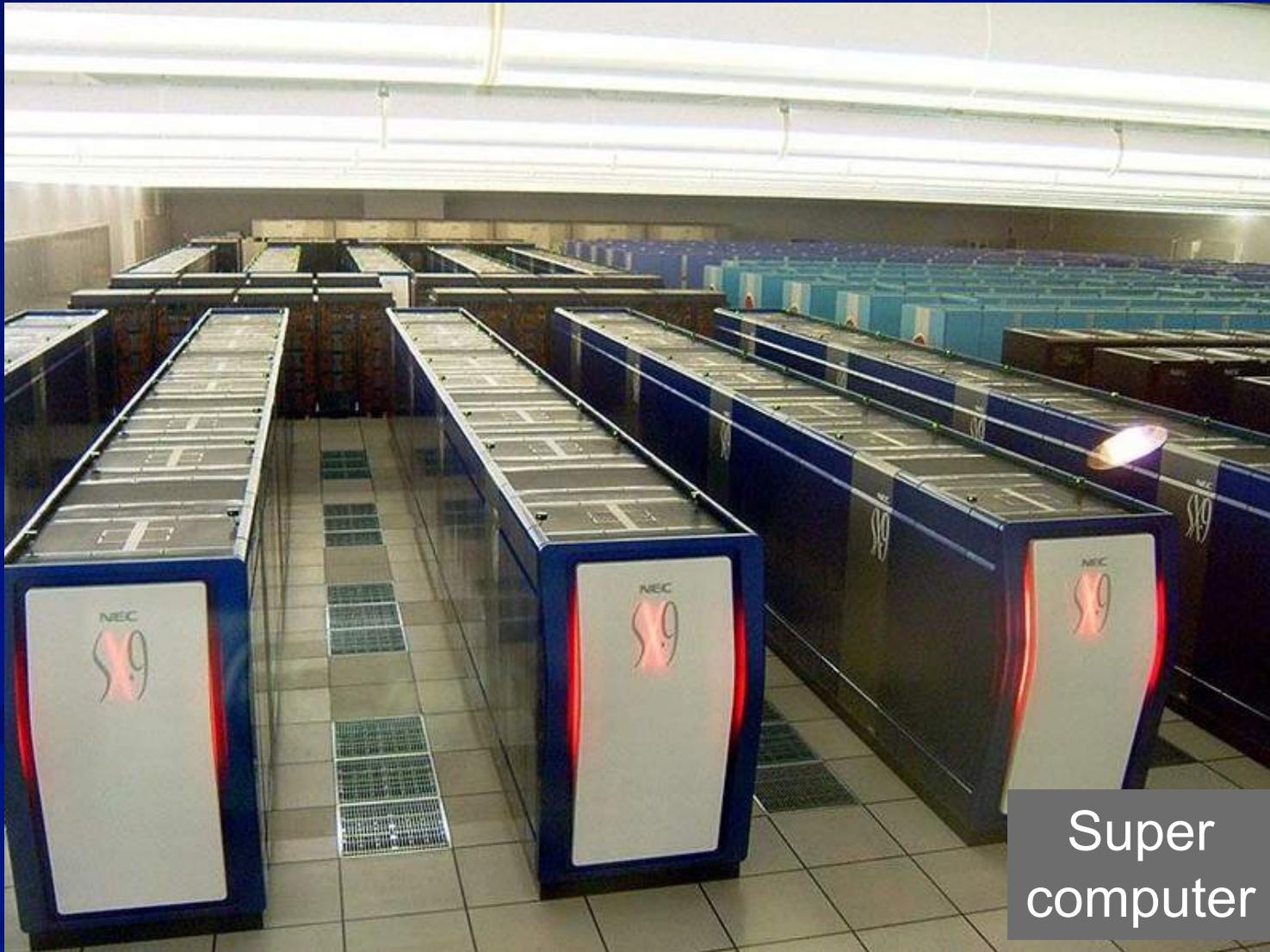
Apple PowerBook, early 1990s



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

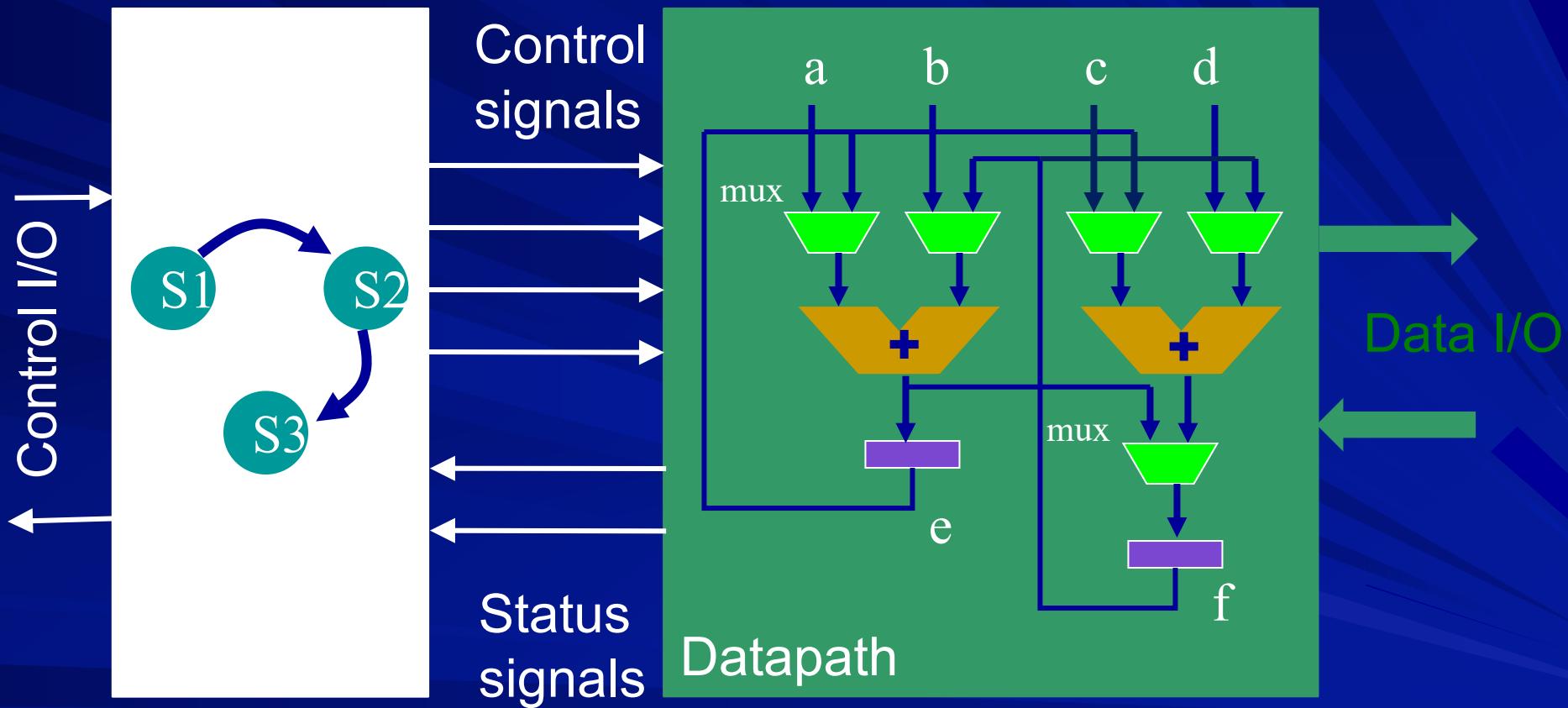
Motivations for complex processors

- 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

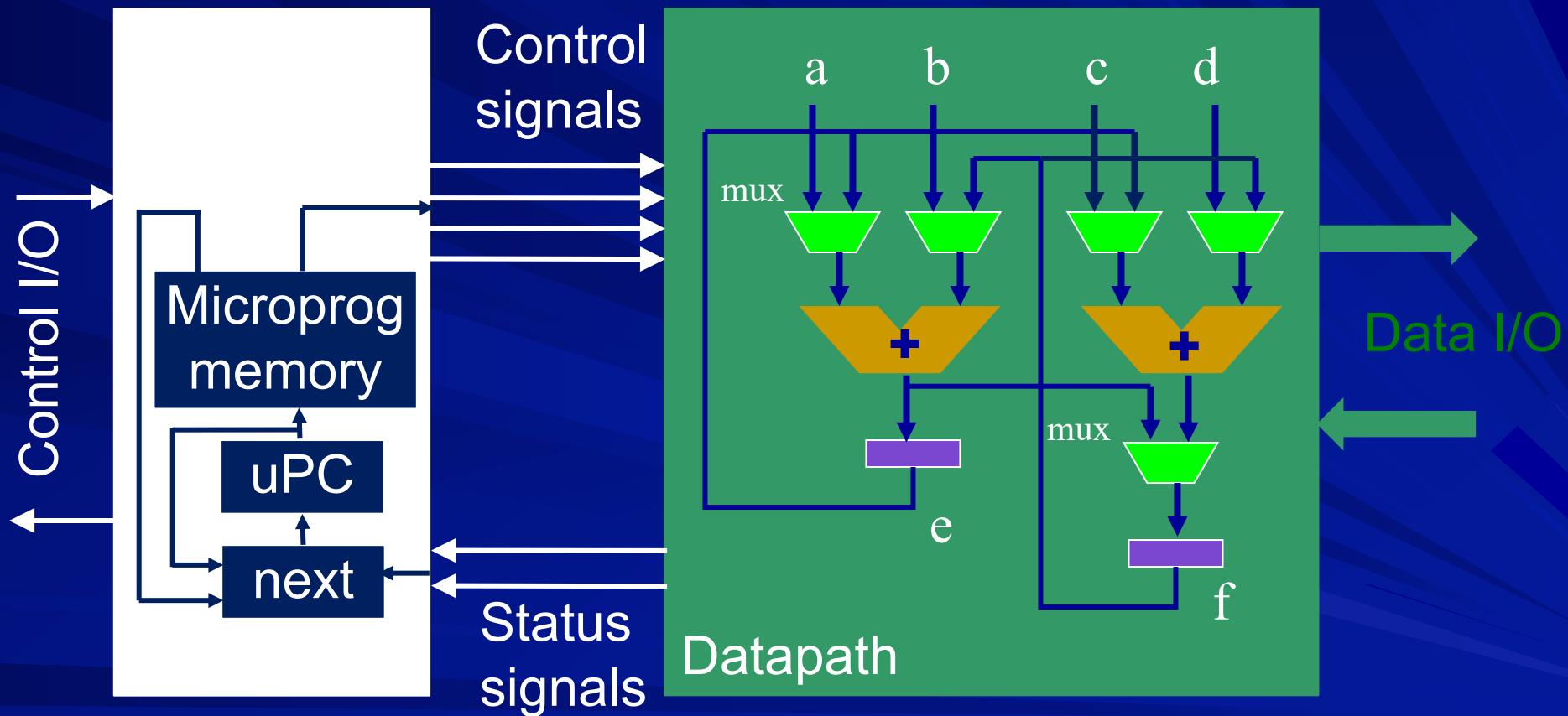
What facilitated complex designs?

- 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

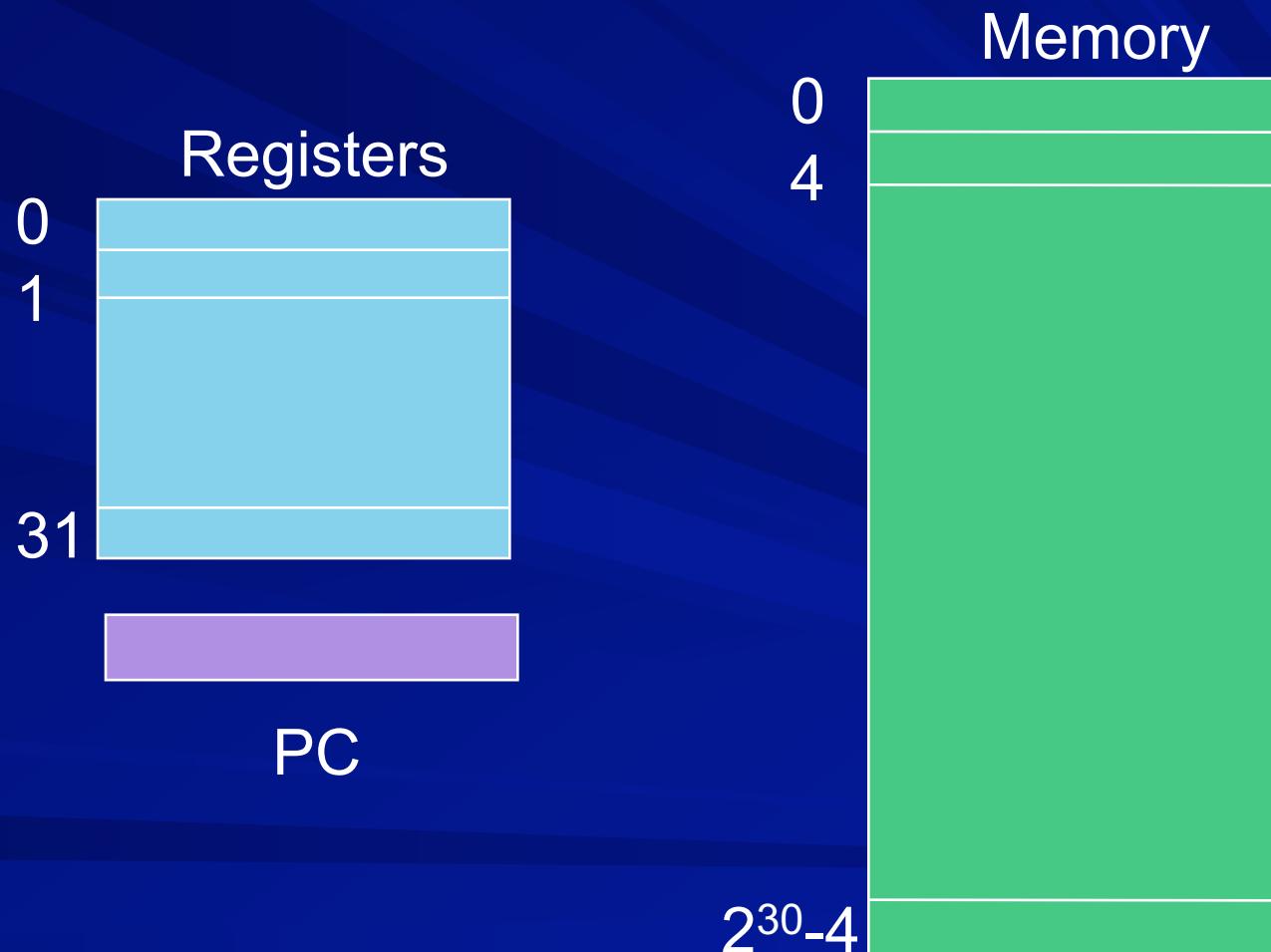
Advanced RISC Machine

- 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

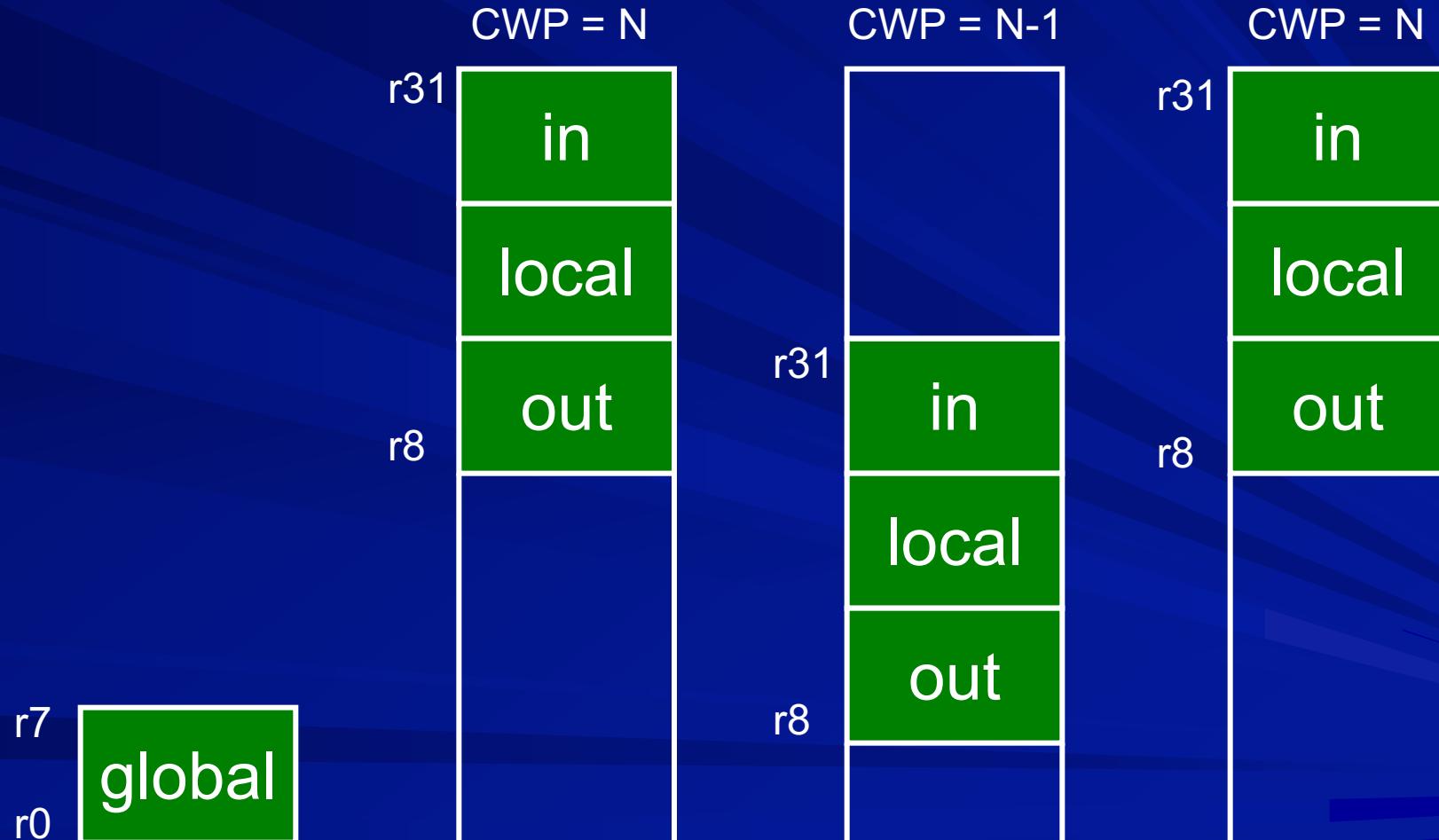
PowerPC Registers

- 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	

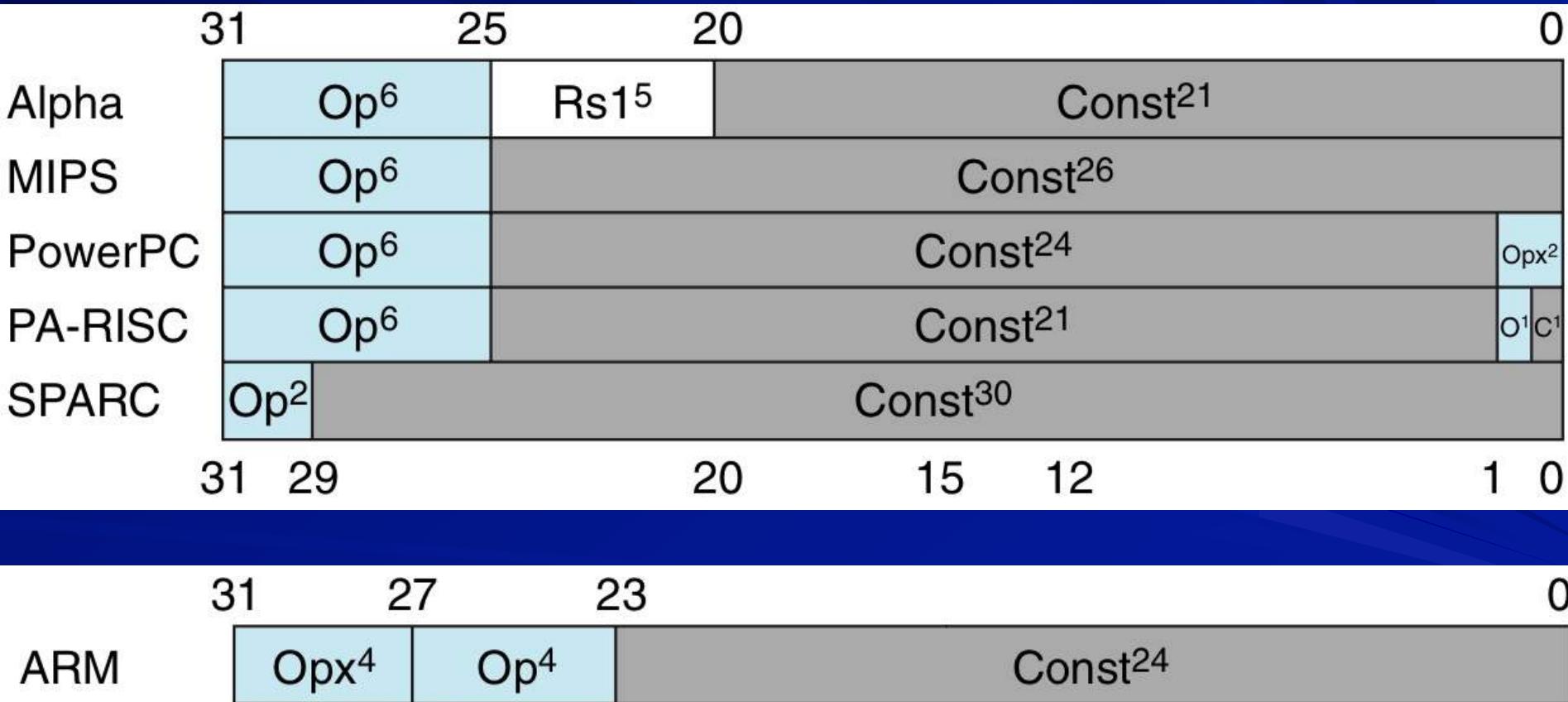
Conditional Branch

	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
PA-RISC		Op ⁶	Rs2 ⁵	Rs1 ⁵	Opx ³ Const ¹¹ OC
SPARC	Op ²	Opx ¹¹		Const ¹⁹	

31 29 18 12 1 0

	31	27	23	0
ARM	Opx ⁴	Op ⁴	Const ²⁴	

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

Intel x86 registers

Size	Accumulator		Counter		Data		Base		Stack pointer		Base pointer		Source index		Dest index	
64	RAX		RCX		RDX		RBX		RSP		RBP		RSI		RDI	
32	EAX		ECX		EDX		EBX		ESP		EBP		ESI		EDI	
16	AX		CX		DX		BX		SP		BP		SI		DI	
8	AH AL		CH CL		DH DL		BH BL									

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

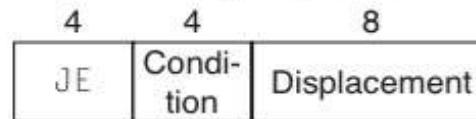
- override default data size
- override default segment register
- lock bus for a semaphore
- repeat the following instruction
- override default address size

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

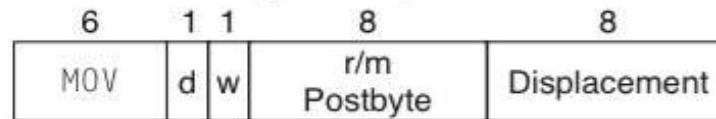
a. JE EIP + displacement



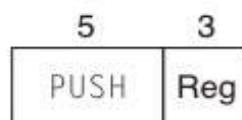
b. CALL



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

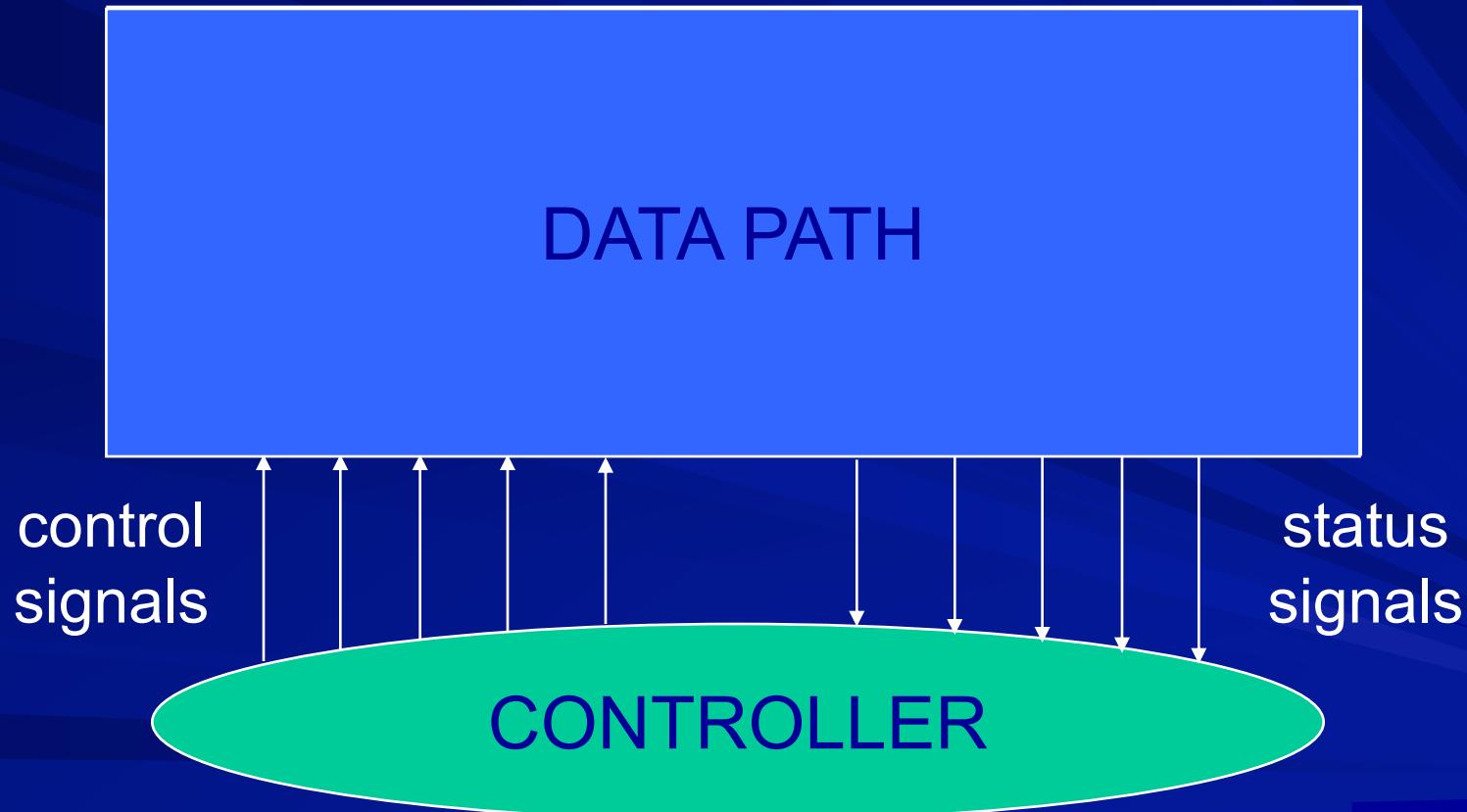
31st January, 2022

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

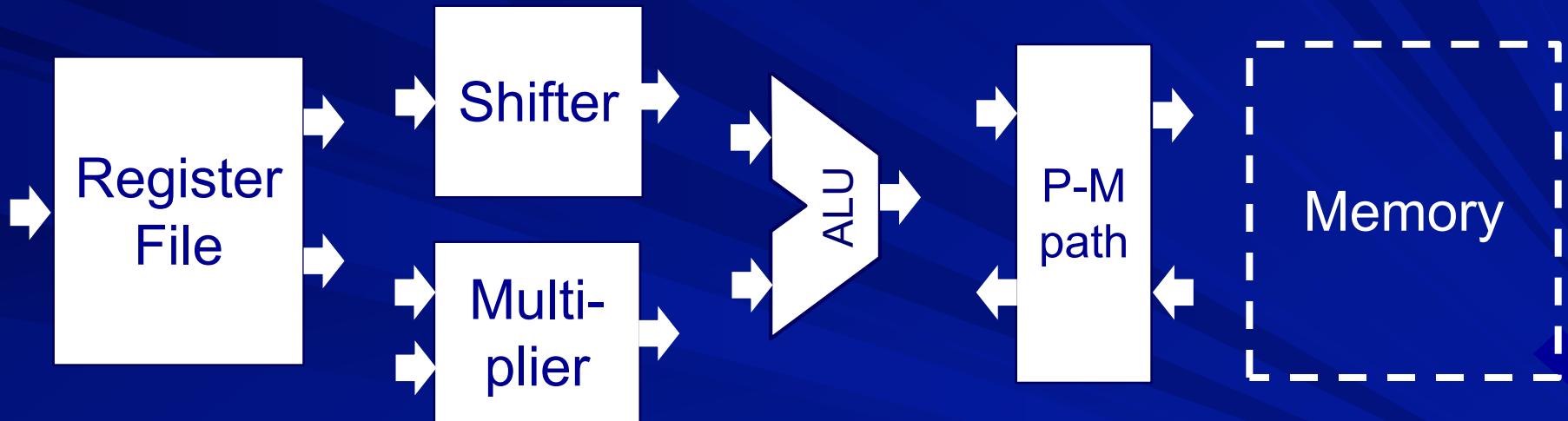
CPU datapath + controller

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

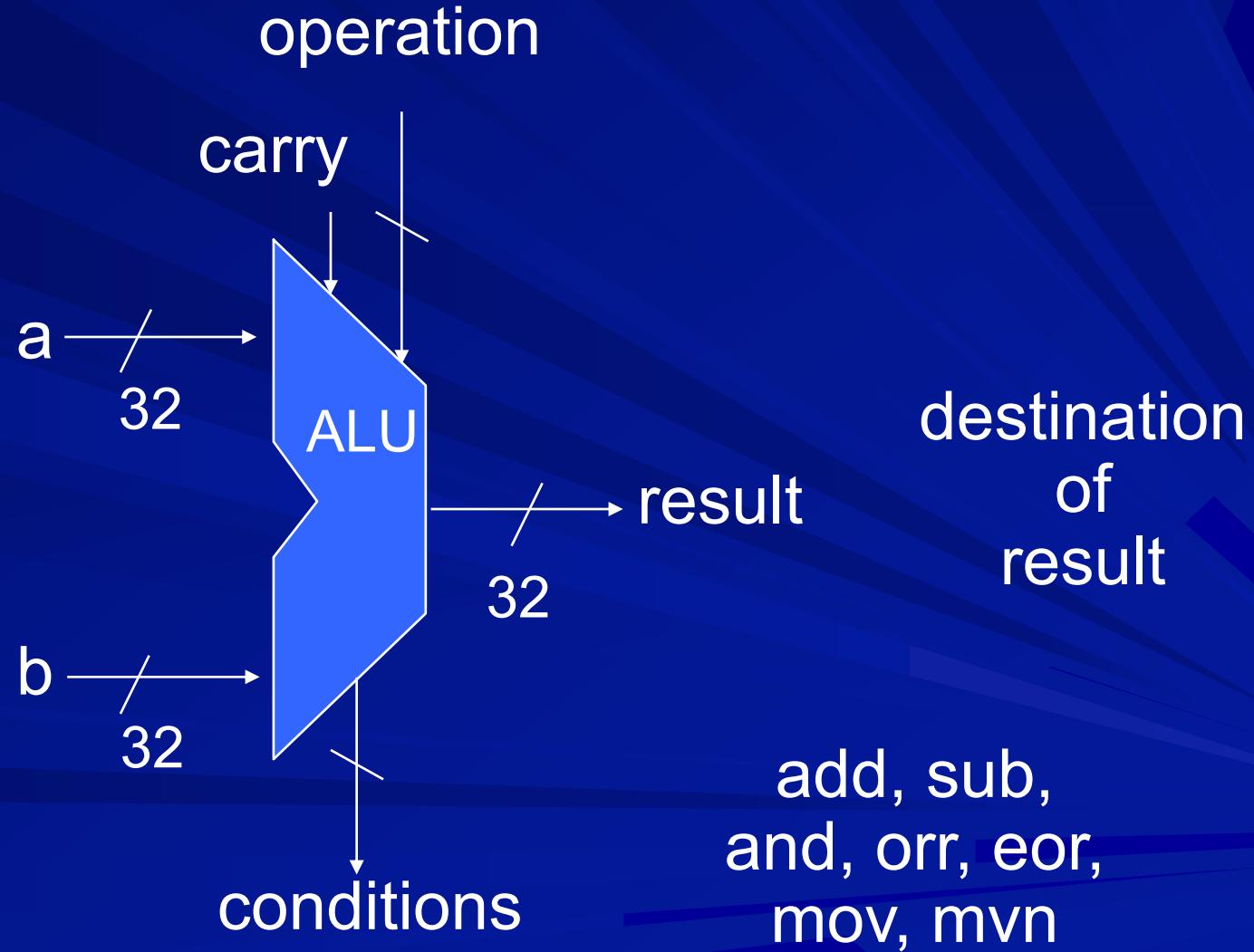
- 12 bit unsigned constant

- or

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

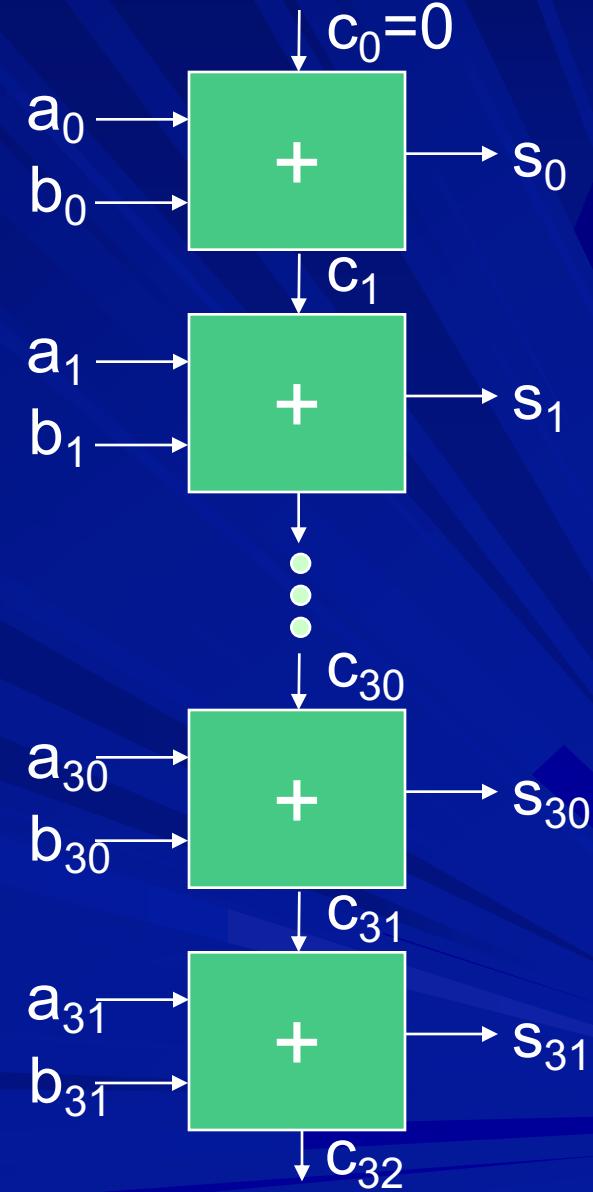
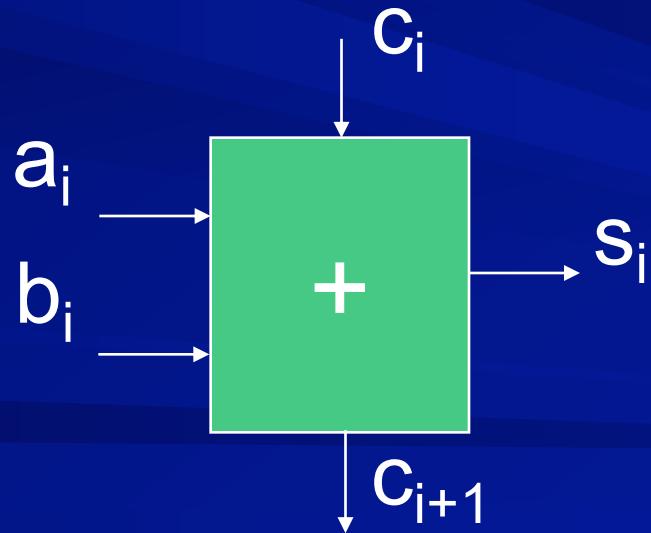
Arithmetic and Logical operations

sources
of
operands



Adder circuit

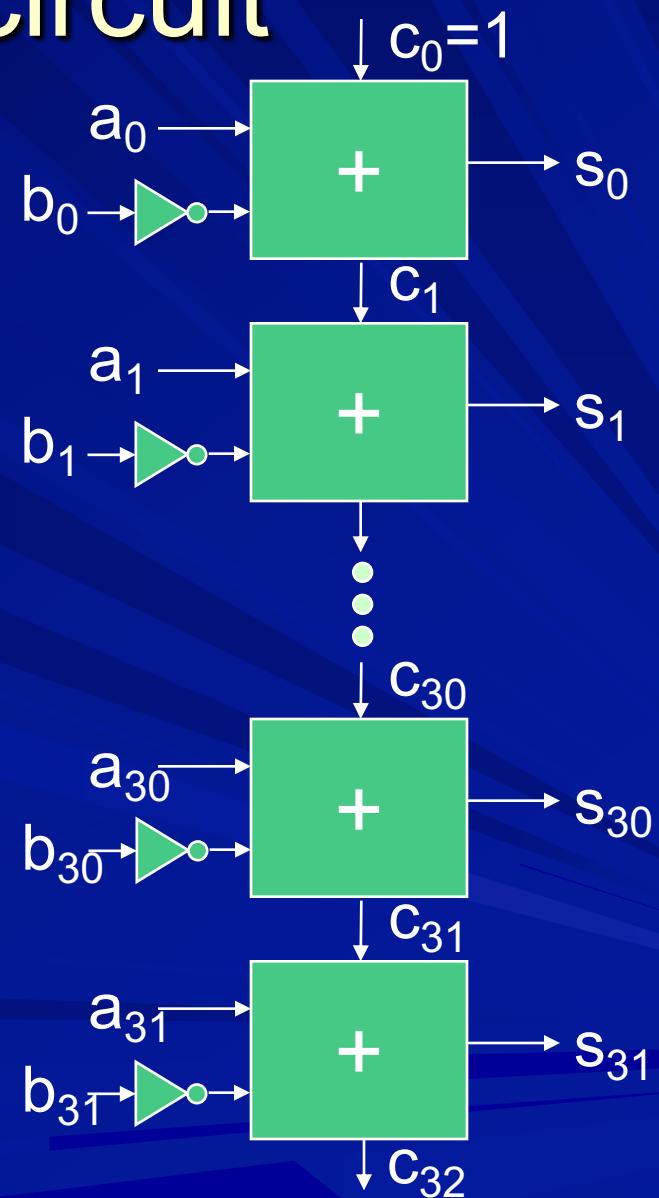
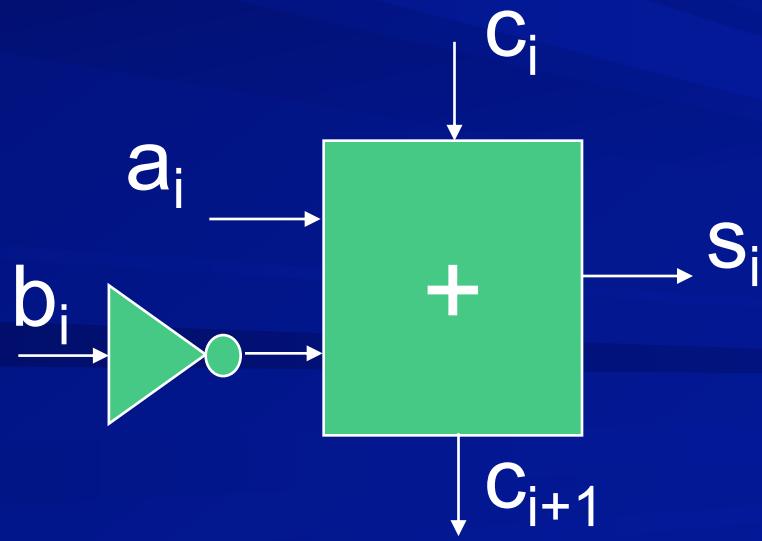
- Perform addition with carry propagation or carry look ahead



Subtraction circuit

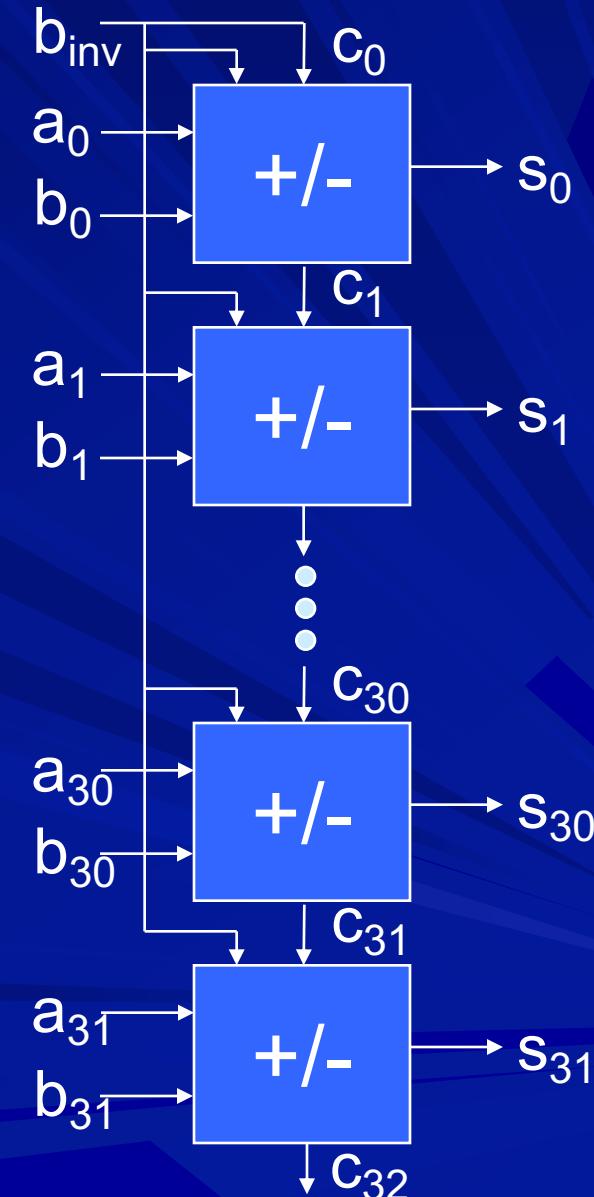
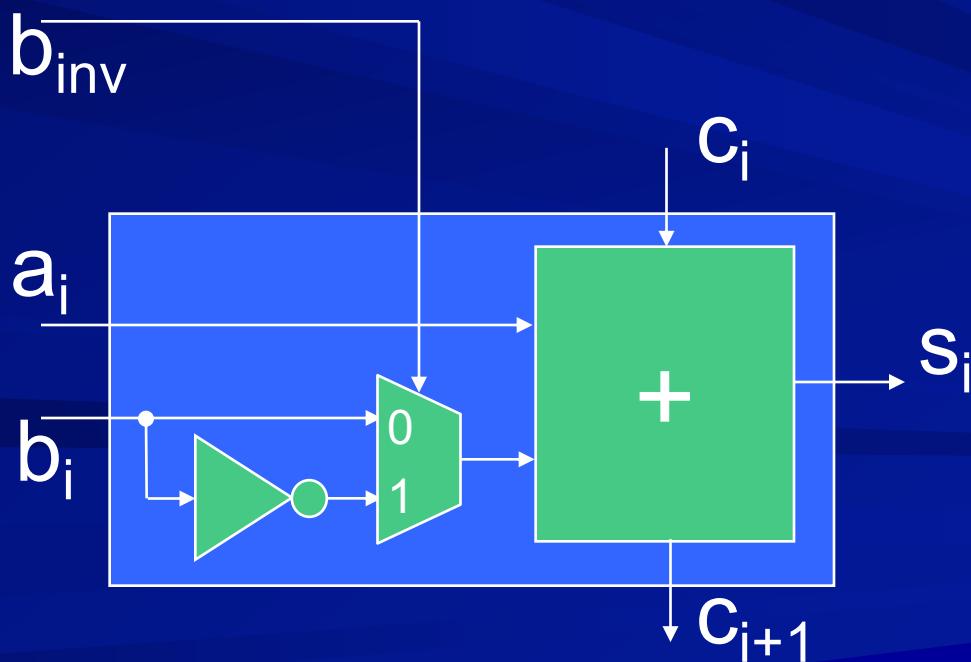
- Use the formula

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

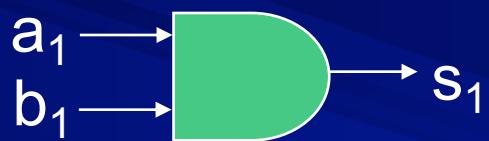
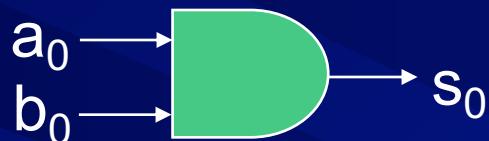


Combining addition and subtraction

- Use a multiplexer circuit



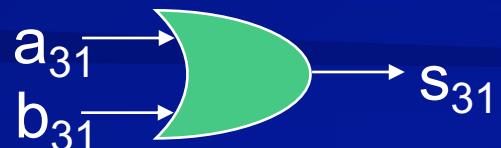
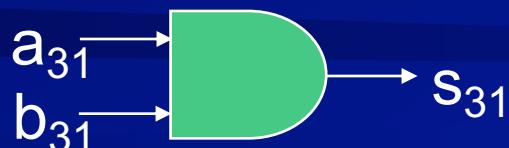
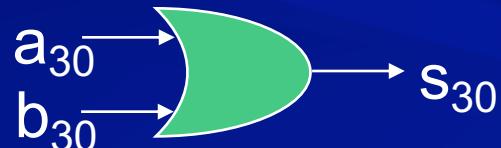
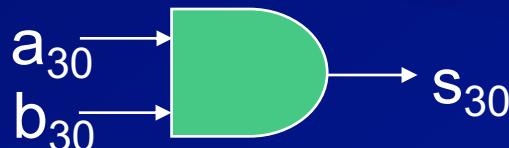
Circuit for and, or, xor



⋮

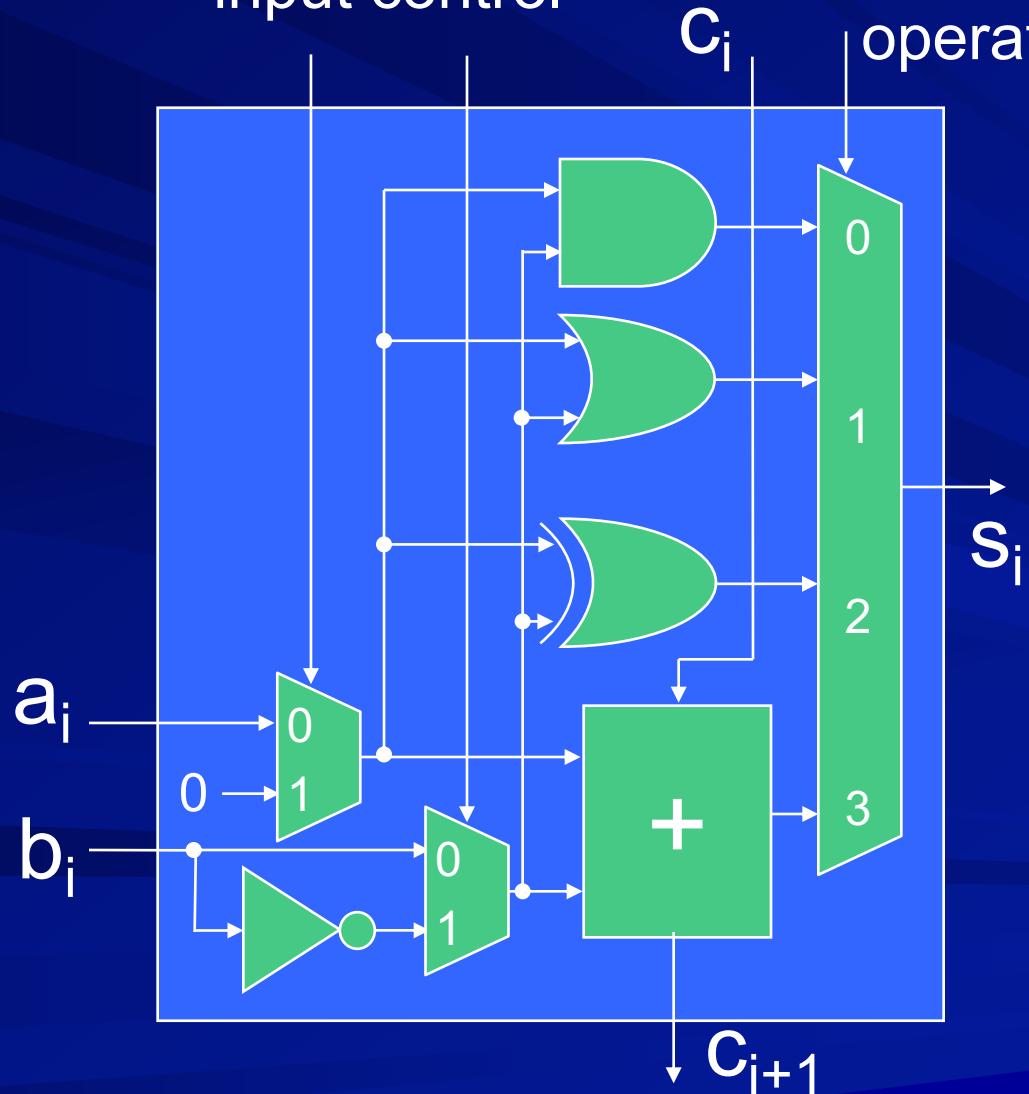
⋮

⋮



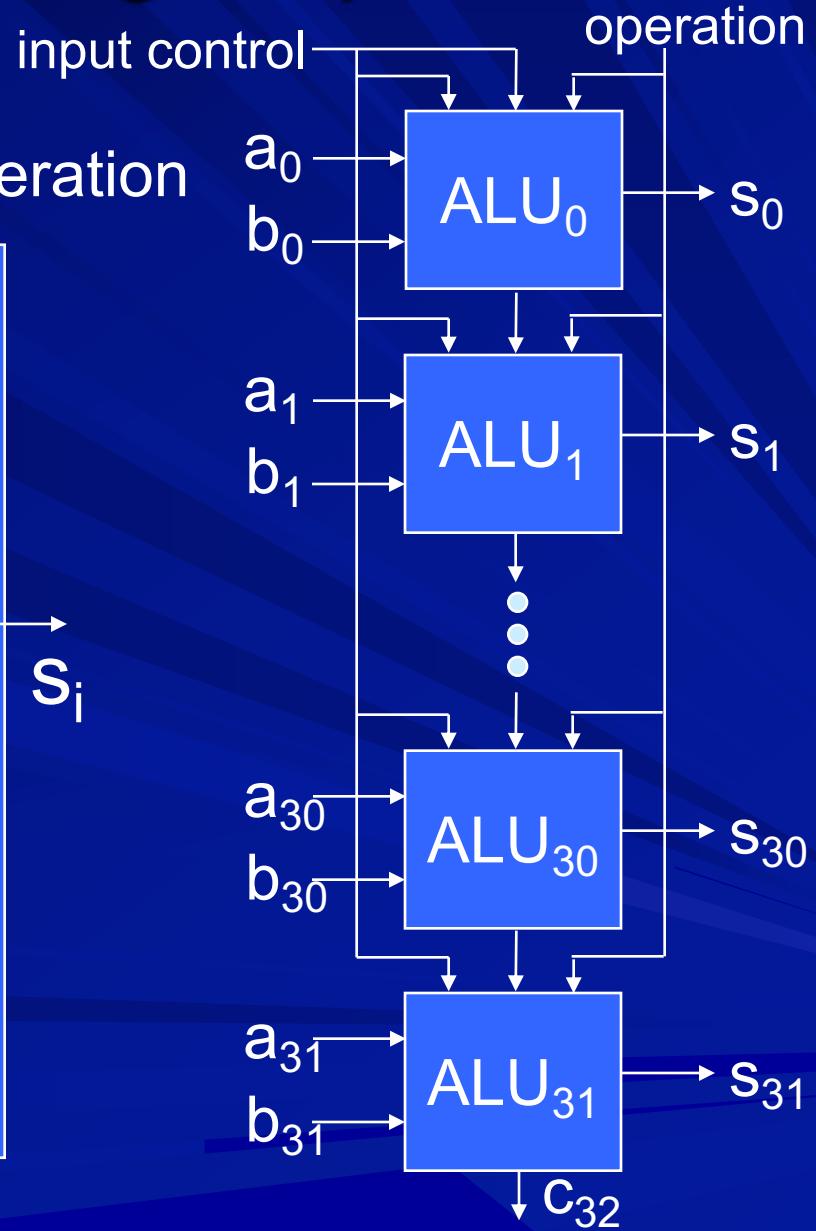
Combining arith & logic operations

input control

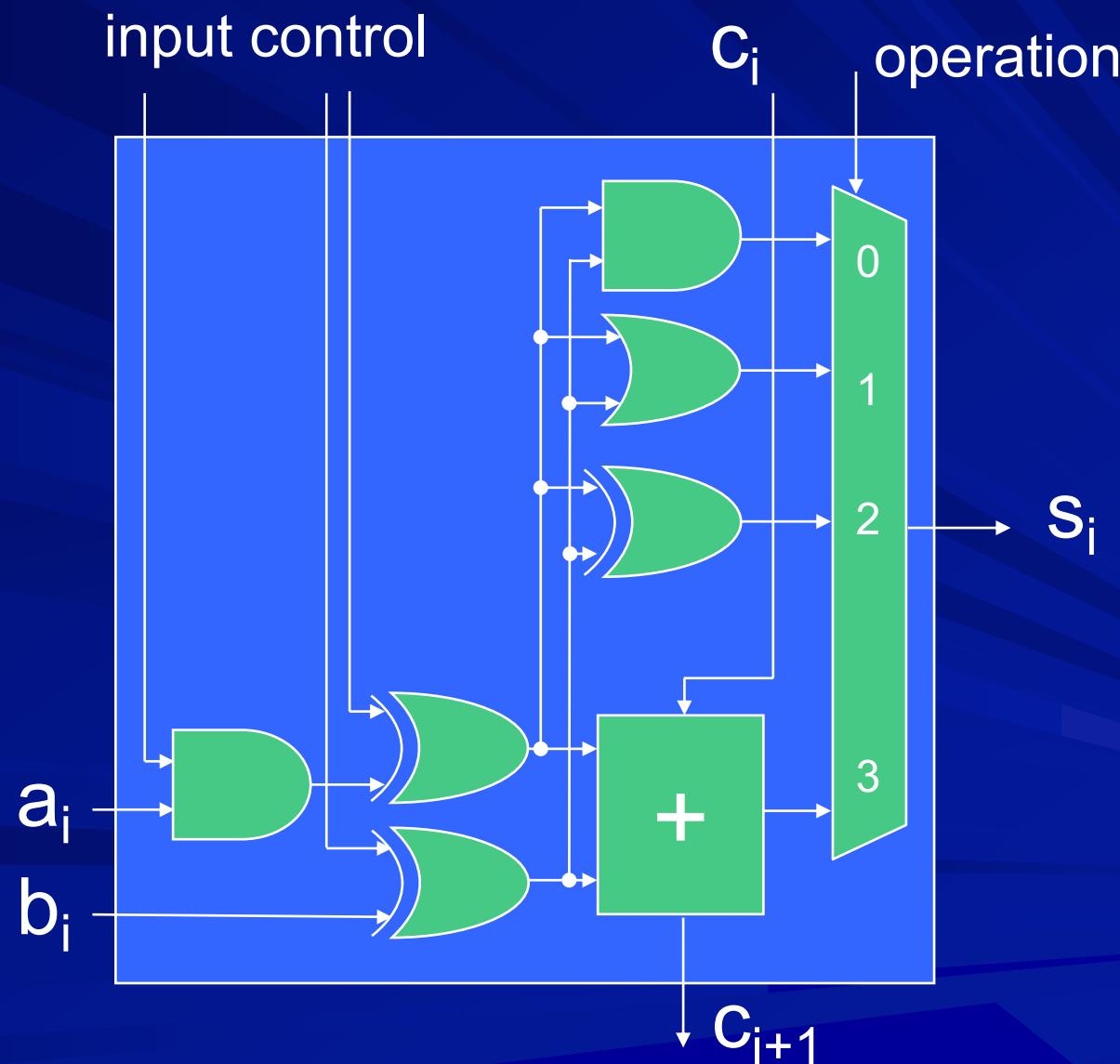


input control

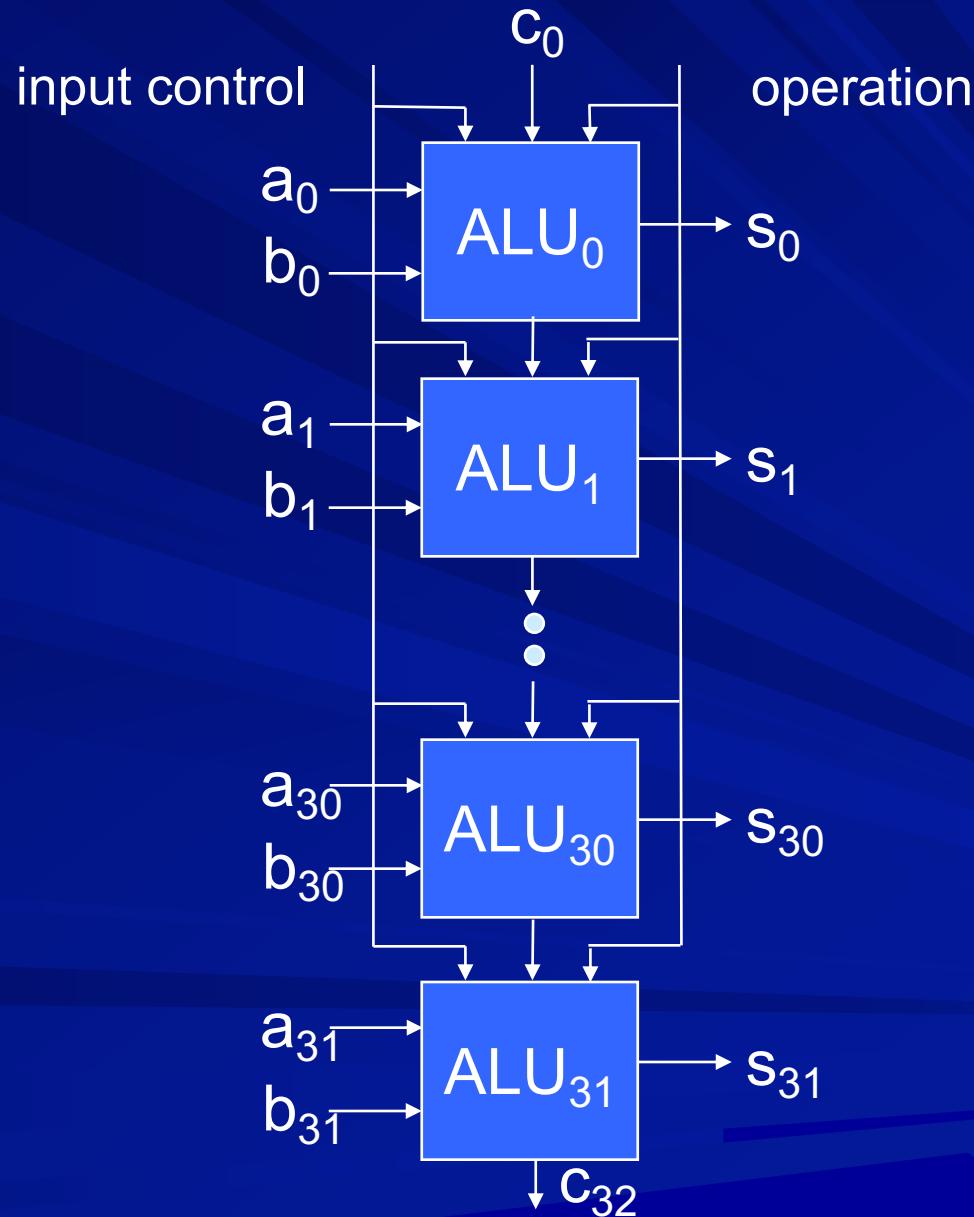
operation



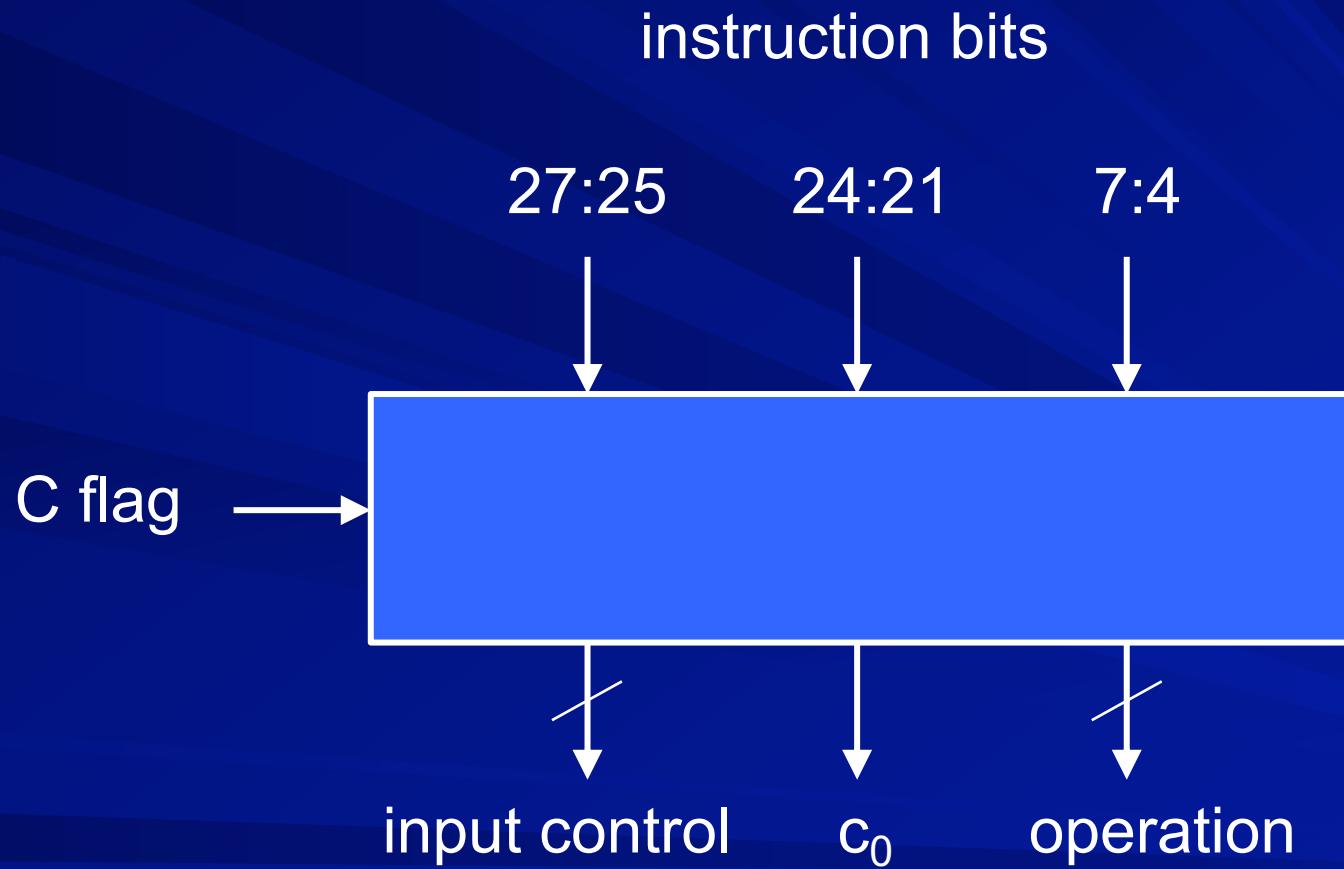
ALU Cell with reverse subtract



ALU – 32 bits



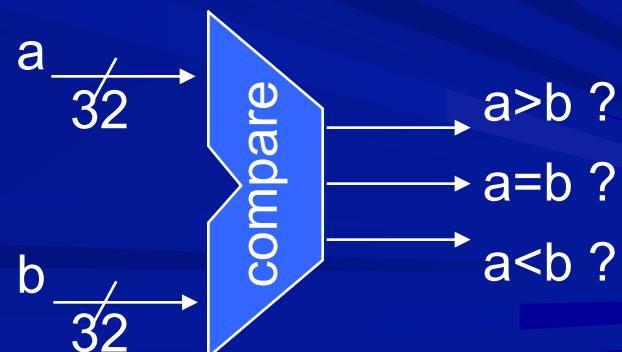
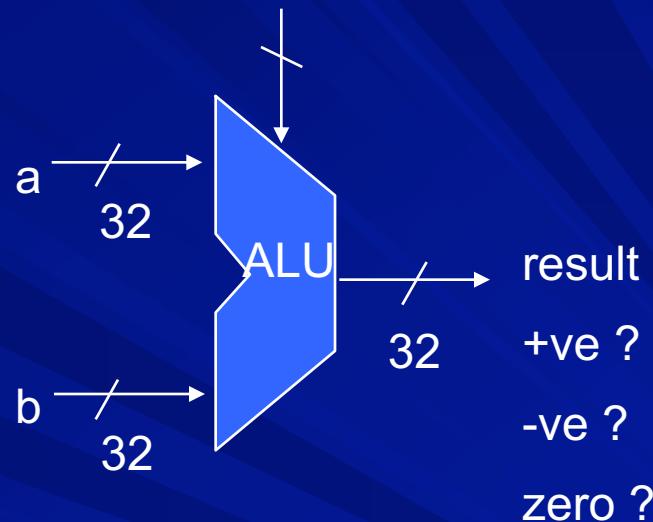
ALU control inputs



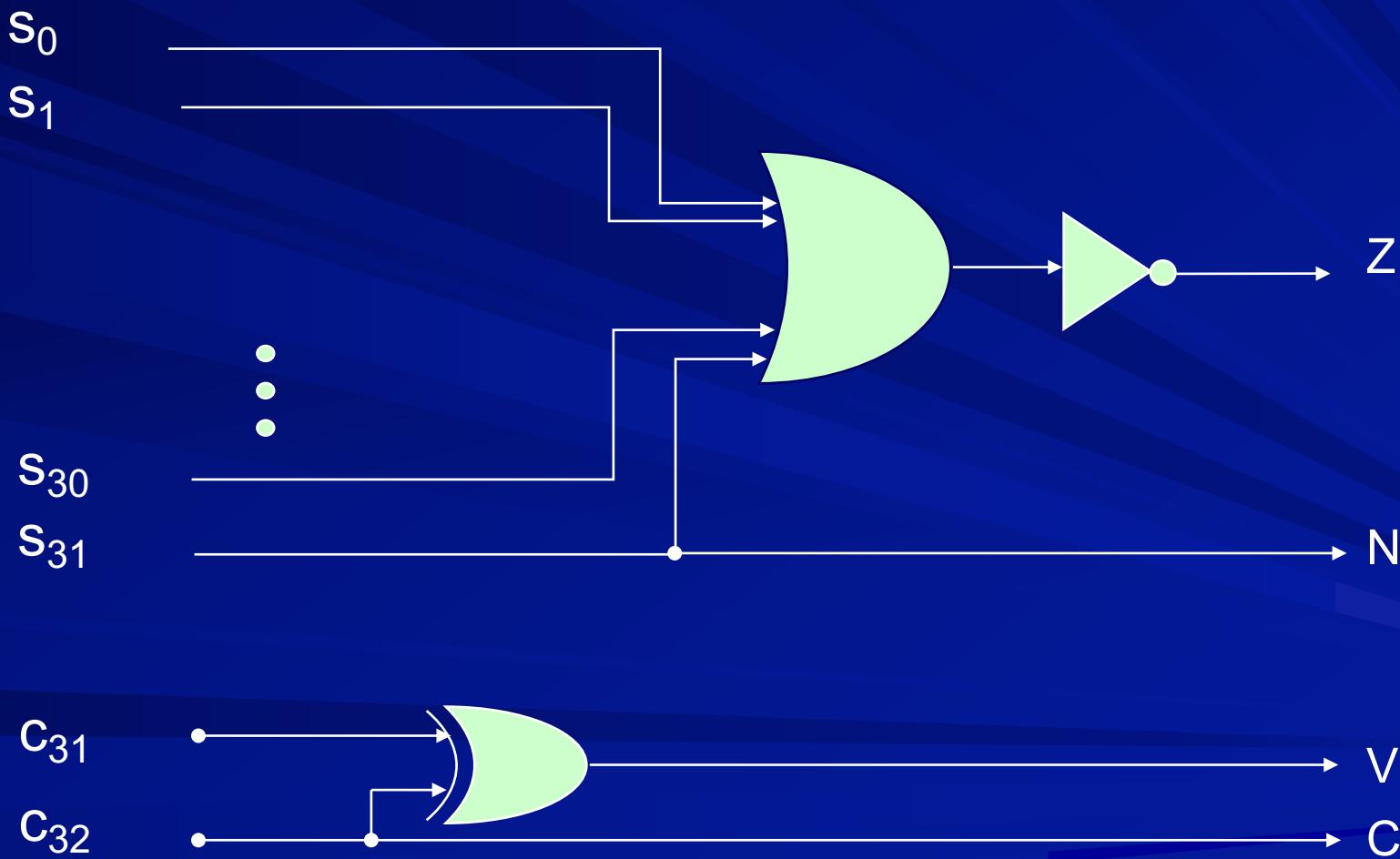
Comparing two integers

- Subtract and check the result
- Compare directly

operation = subtract



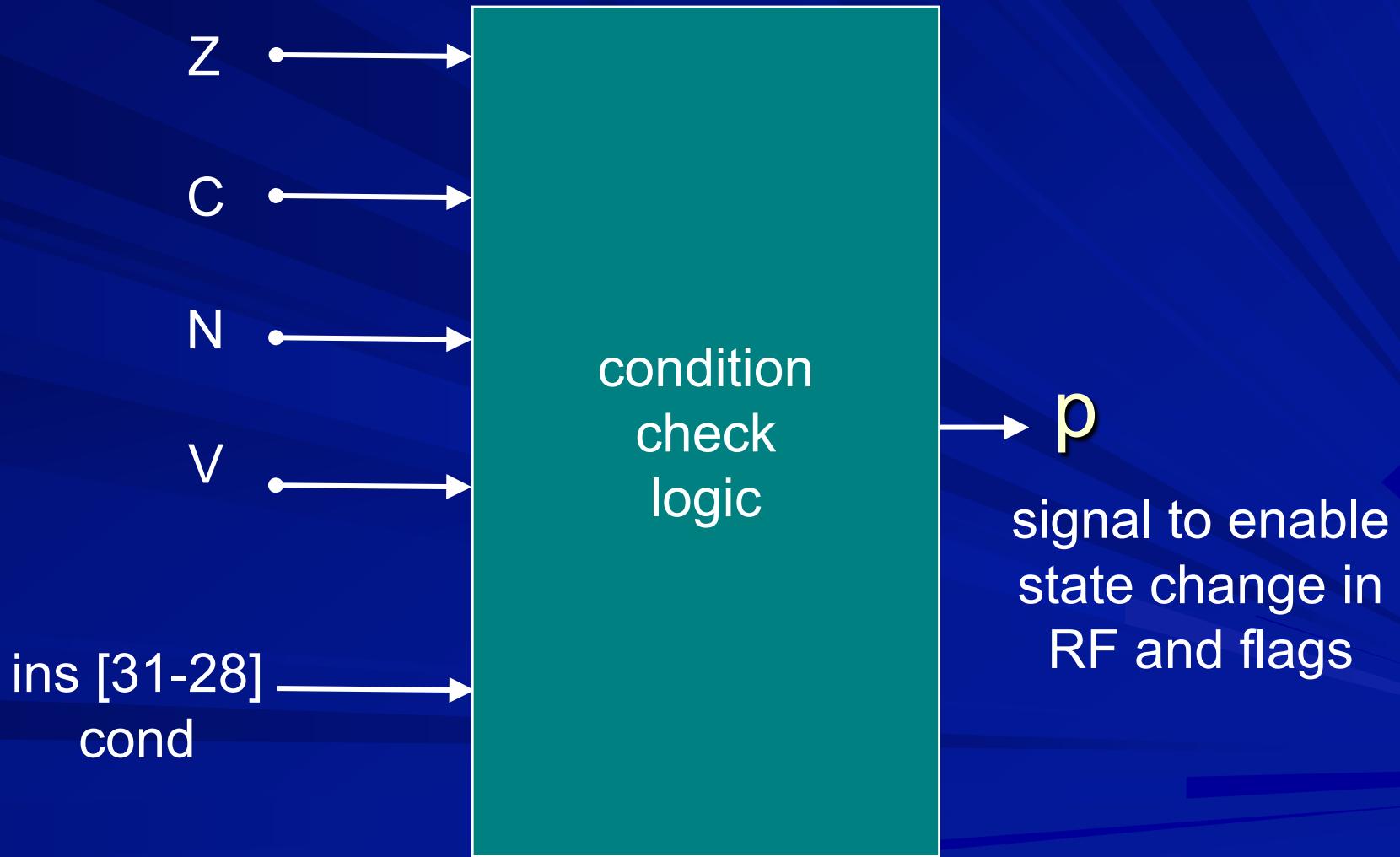
Setting flags



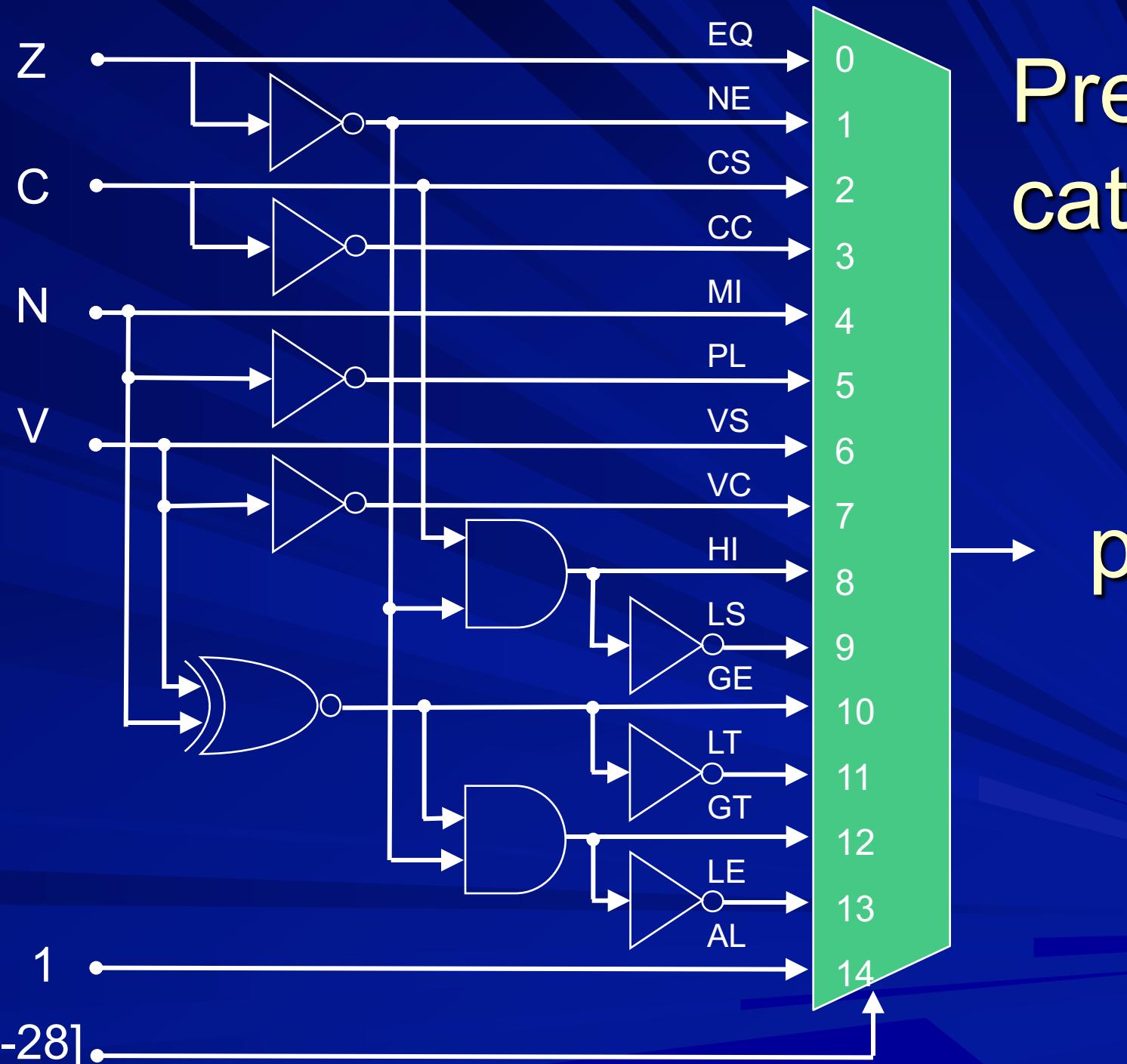
Predication

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



ins [31-28]

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

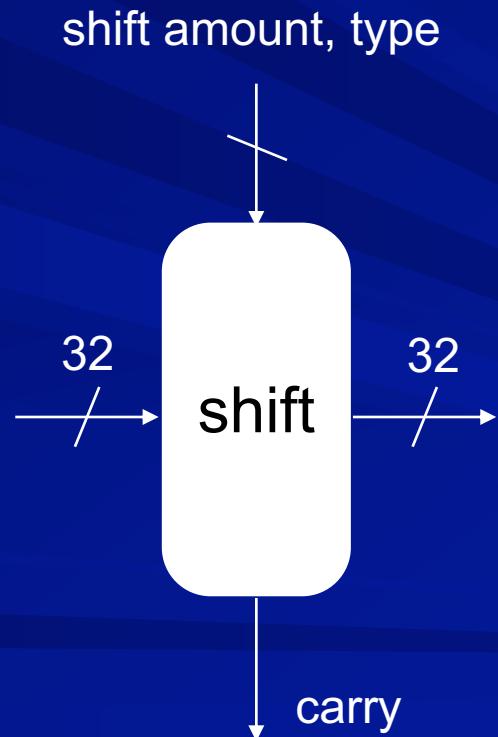
⇒ Rm, <Shift type>, Rs

⇒ 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
---	---	---	----------	----------	-----	-------	-------

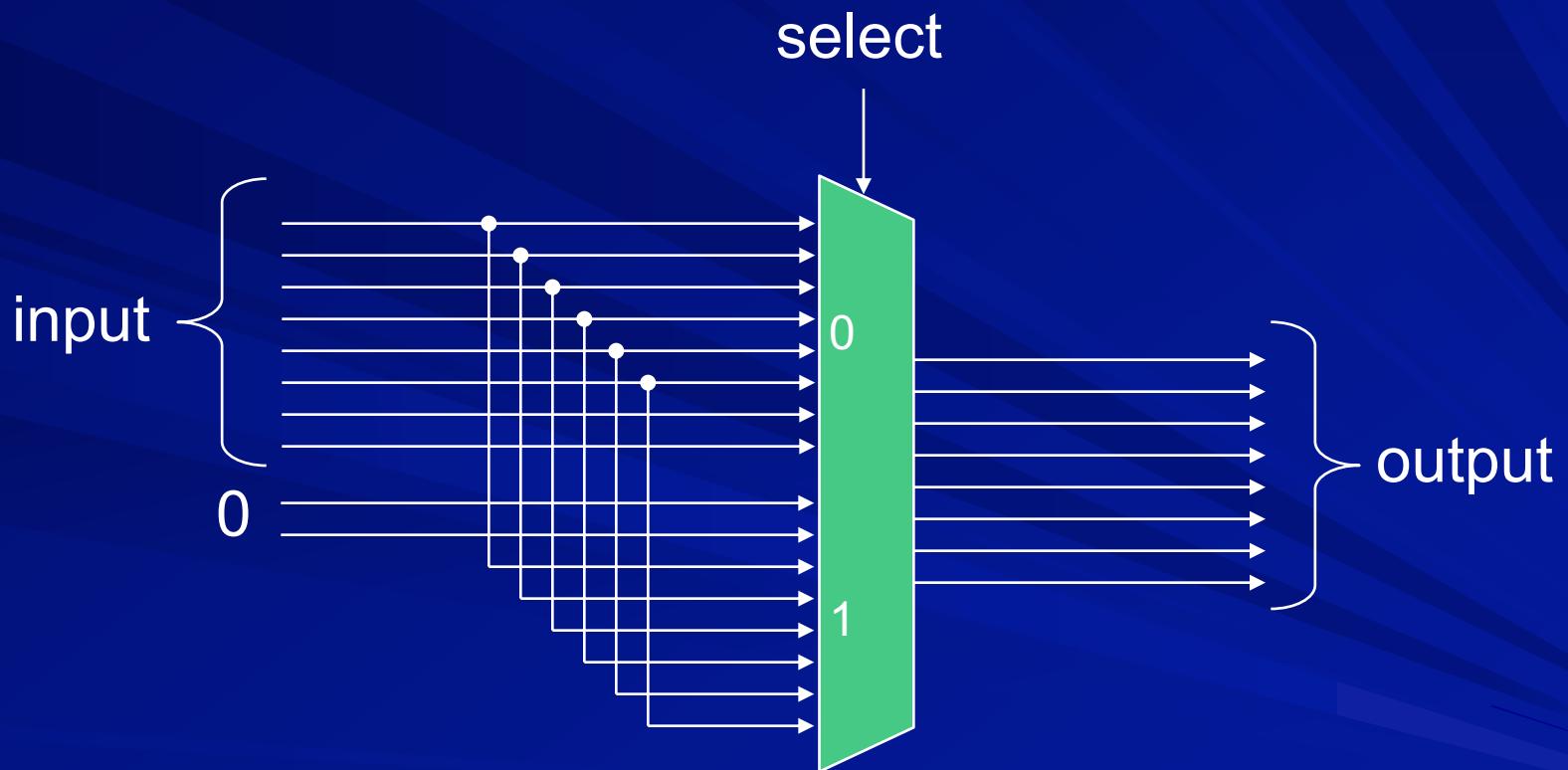
shift right arithmetic 2 bits

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

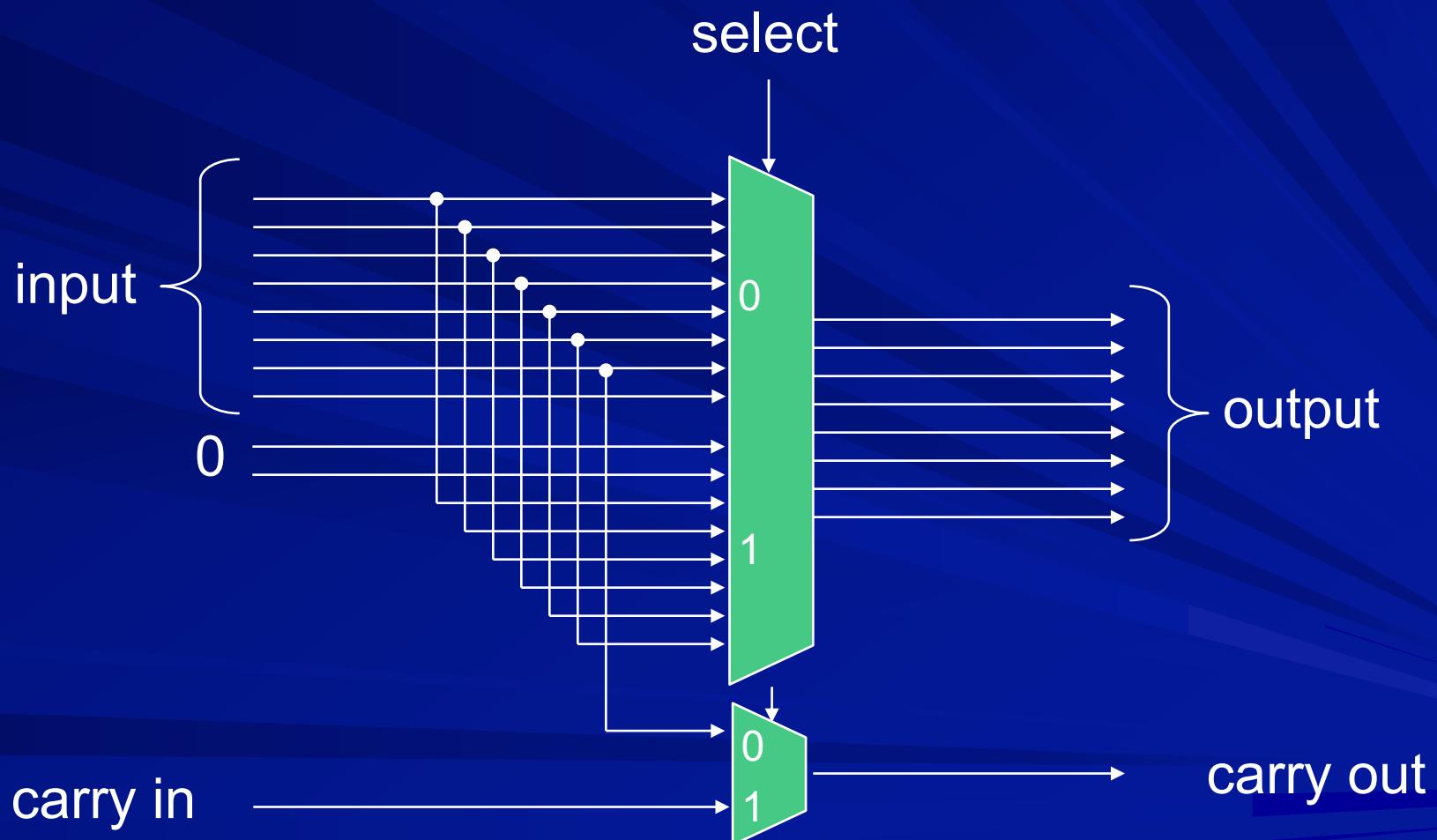


a_{31}	a_{31}	a_{31}	a_{30}	...	a_3	a_2
----------	----------	----------	----------	-----	-------	-------

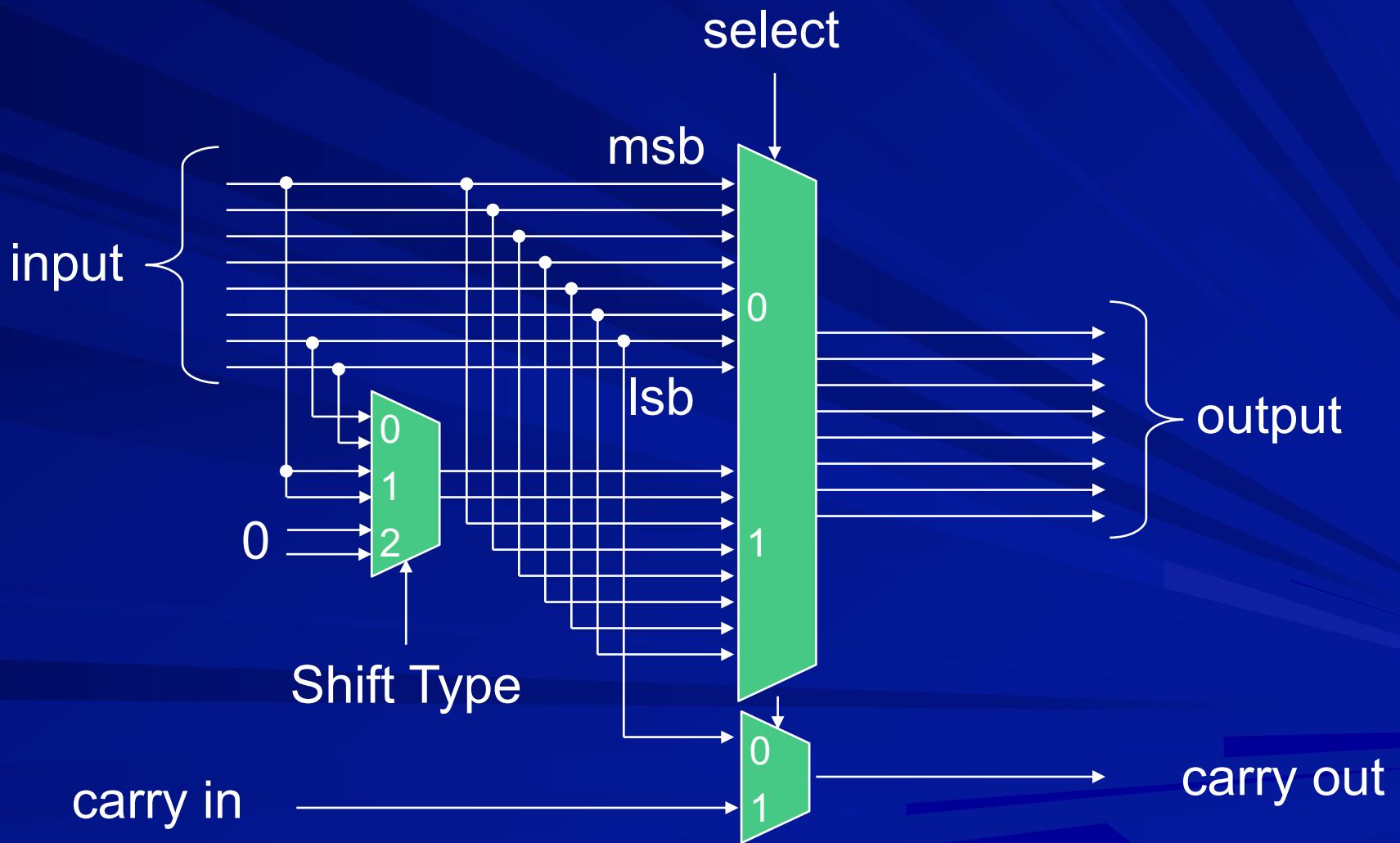
Circuit for shifting by 2 bits



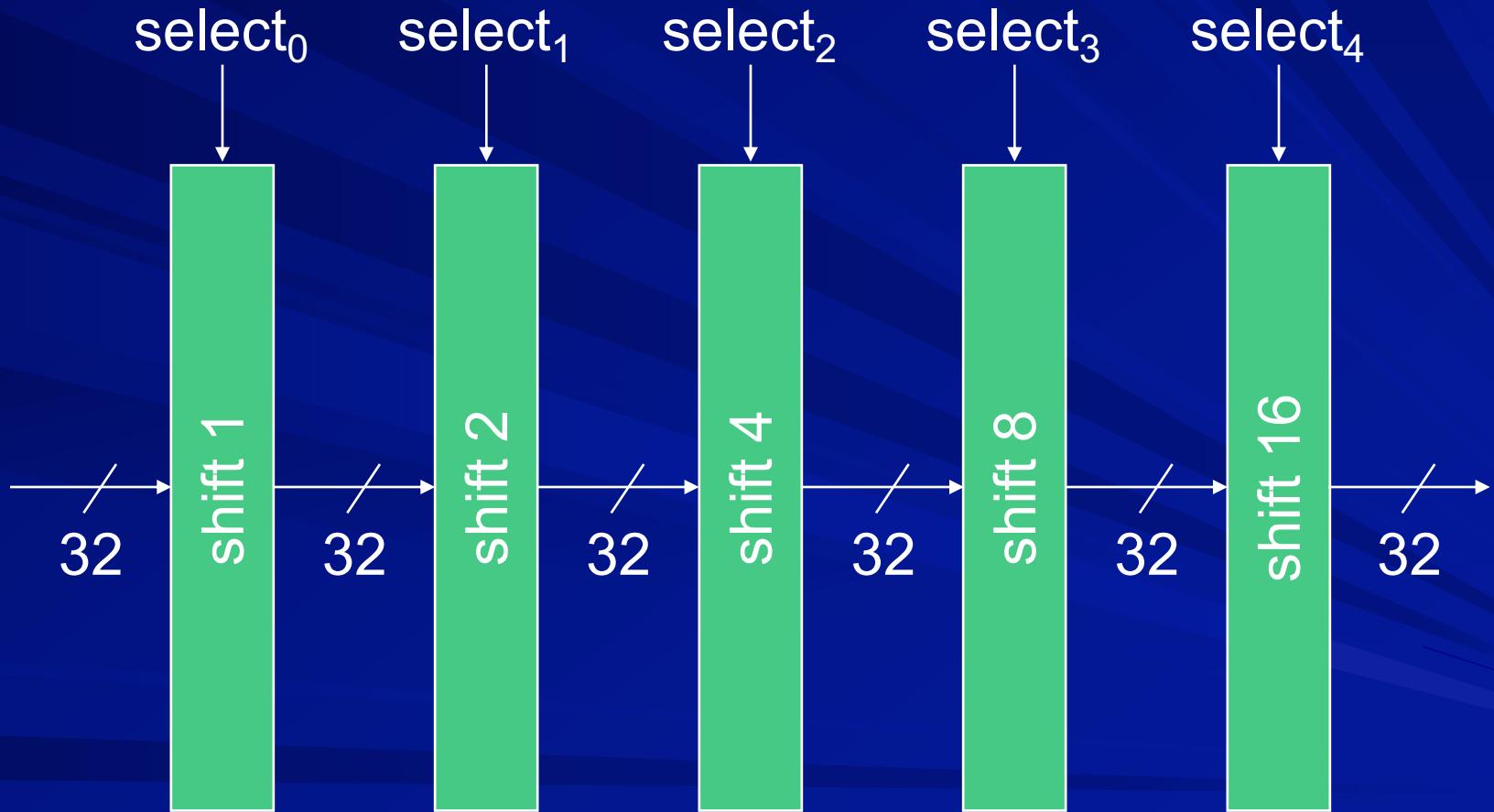
Circuit for shifting by 2 bits



Combining ROR, ASR, LSR

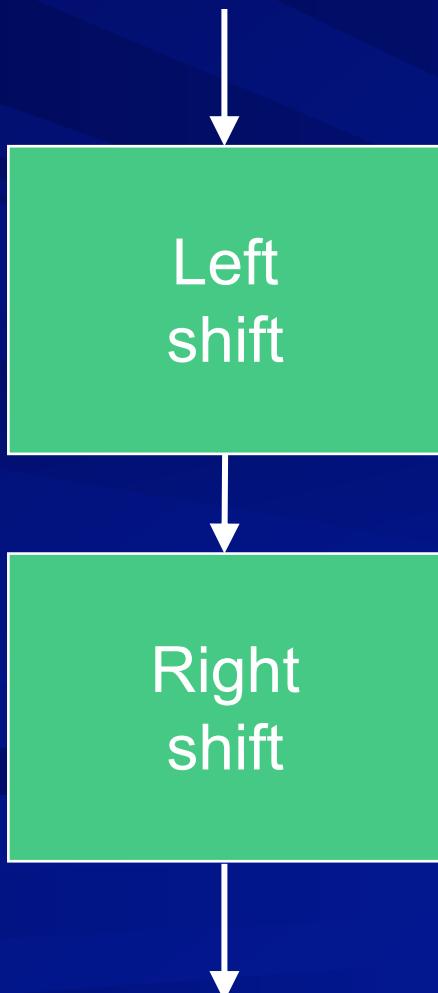


Circuit for shifting by 0 to 31 bits

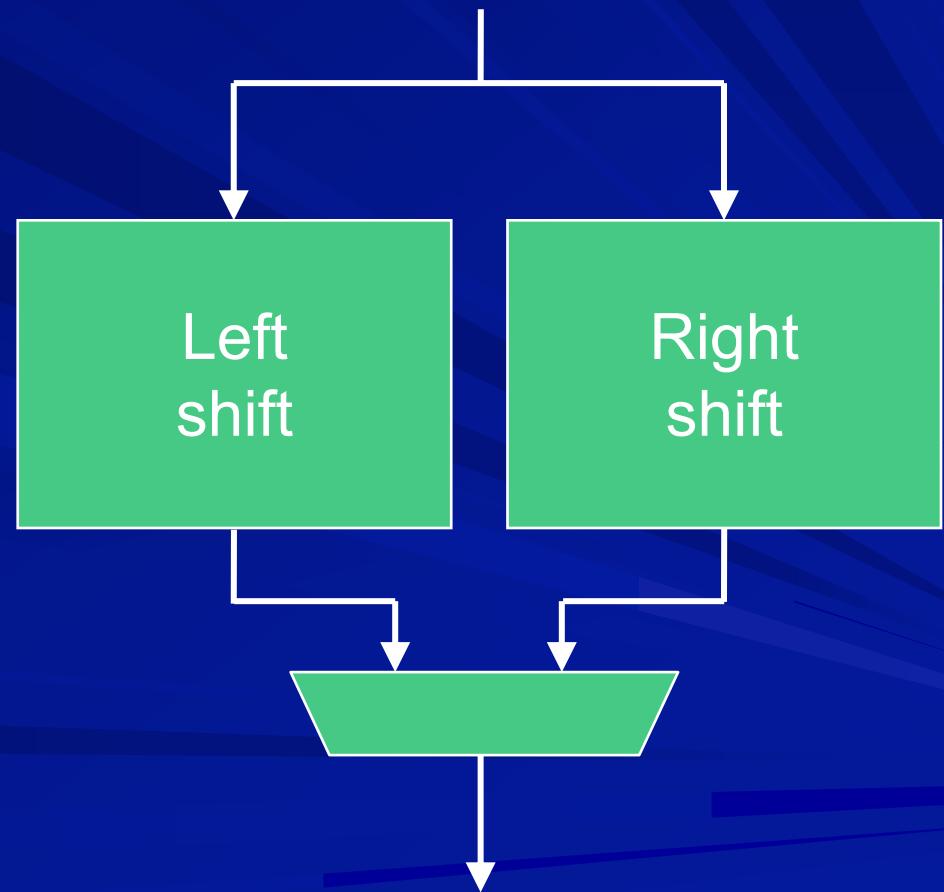


Combining Left and Right Shift

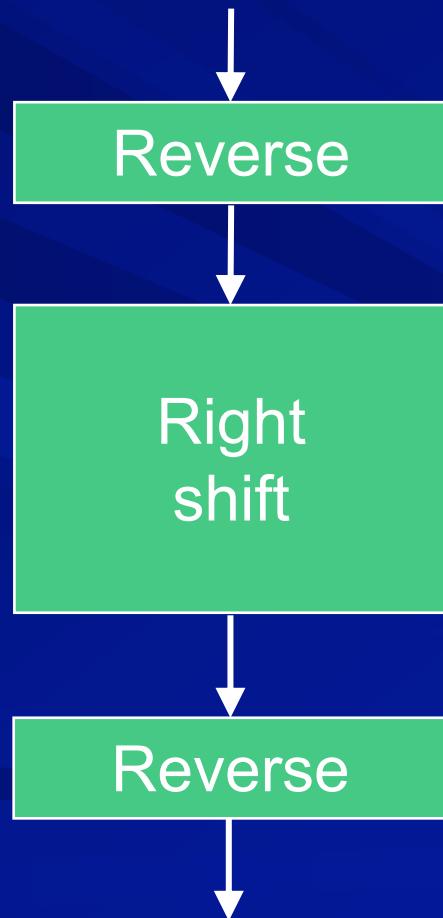
cascade



parallel



Combining Left and Right Shift



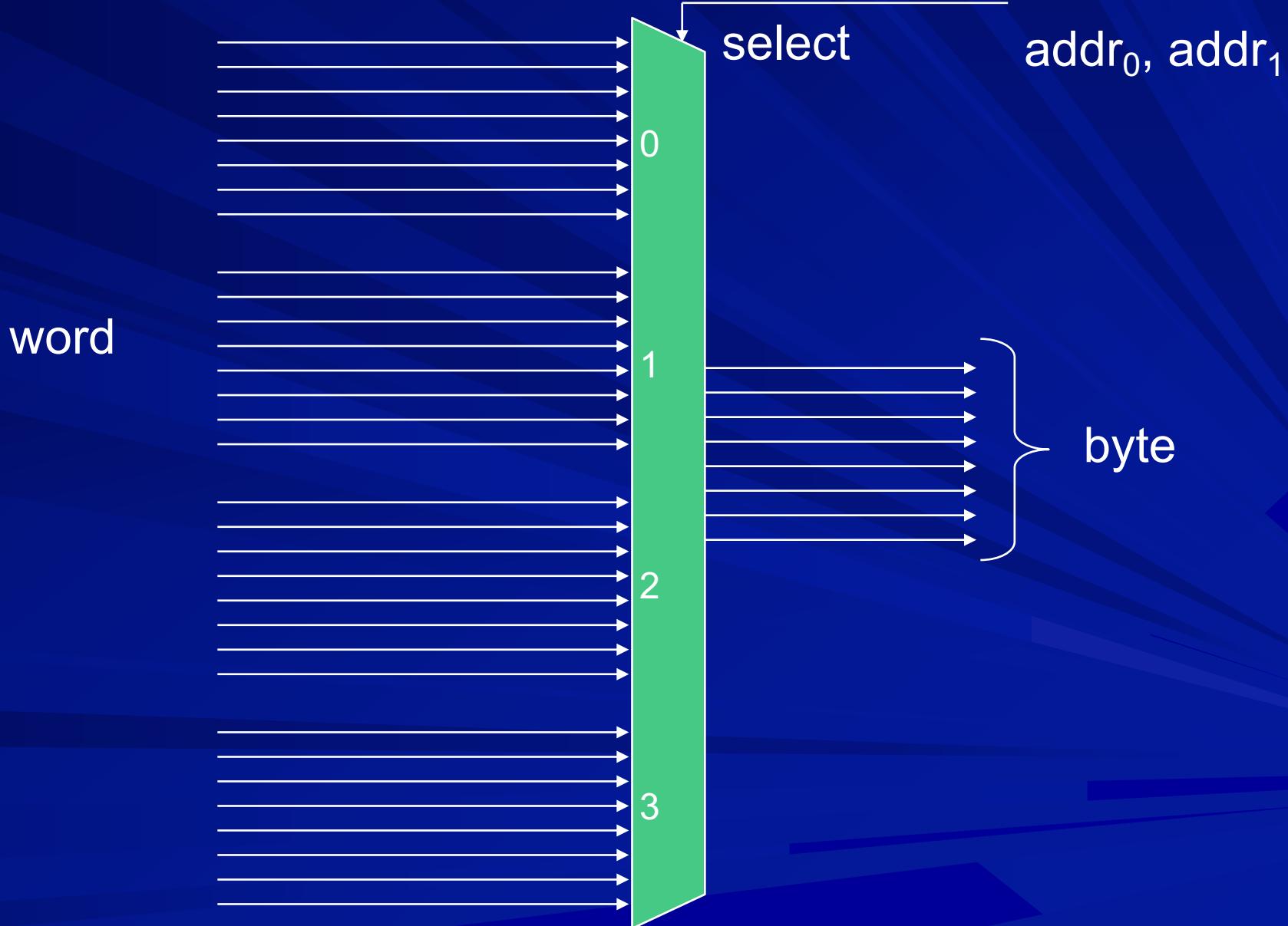
Pre/post increment/decrement

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

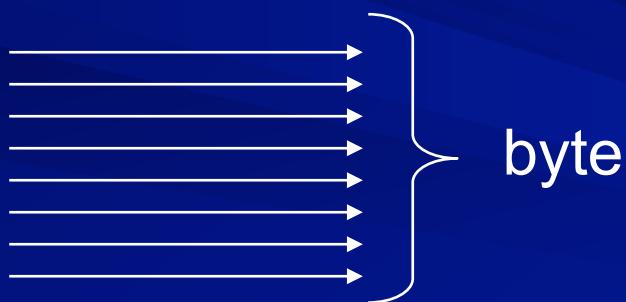
Auto increment/decrement

- 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

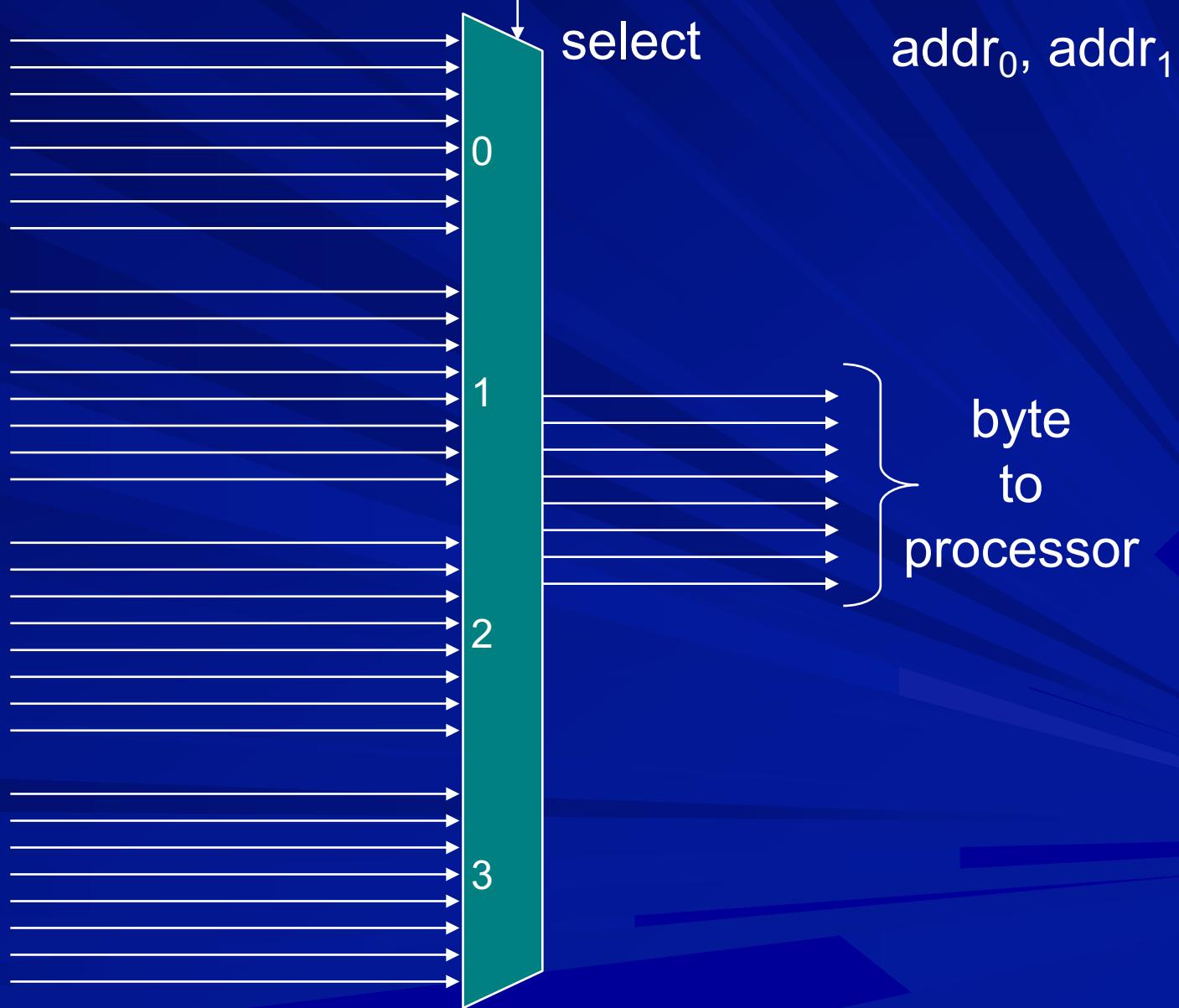


Word / half-word / byte transfer



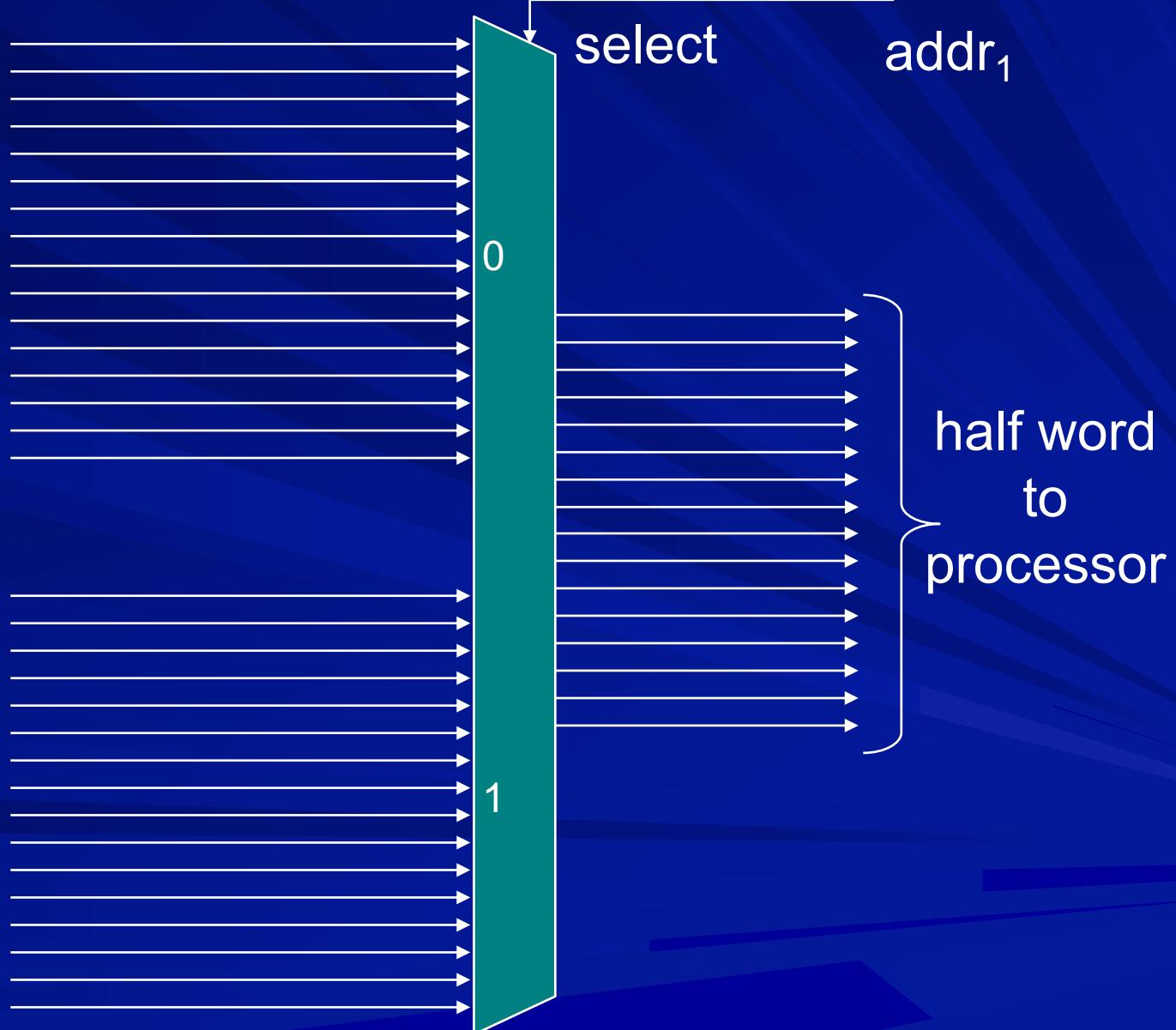
Selecting byte for ldrb / ldrsb

word
from
memory

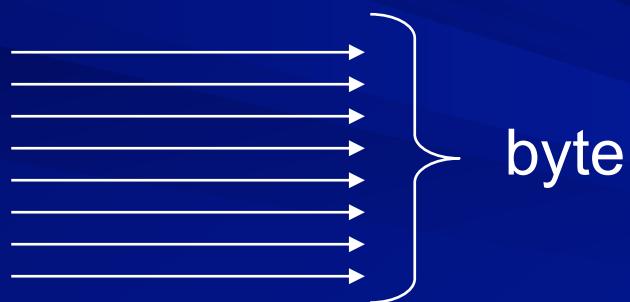


Selecting half word for ldrh / ldrsh

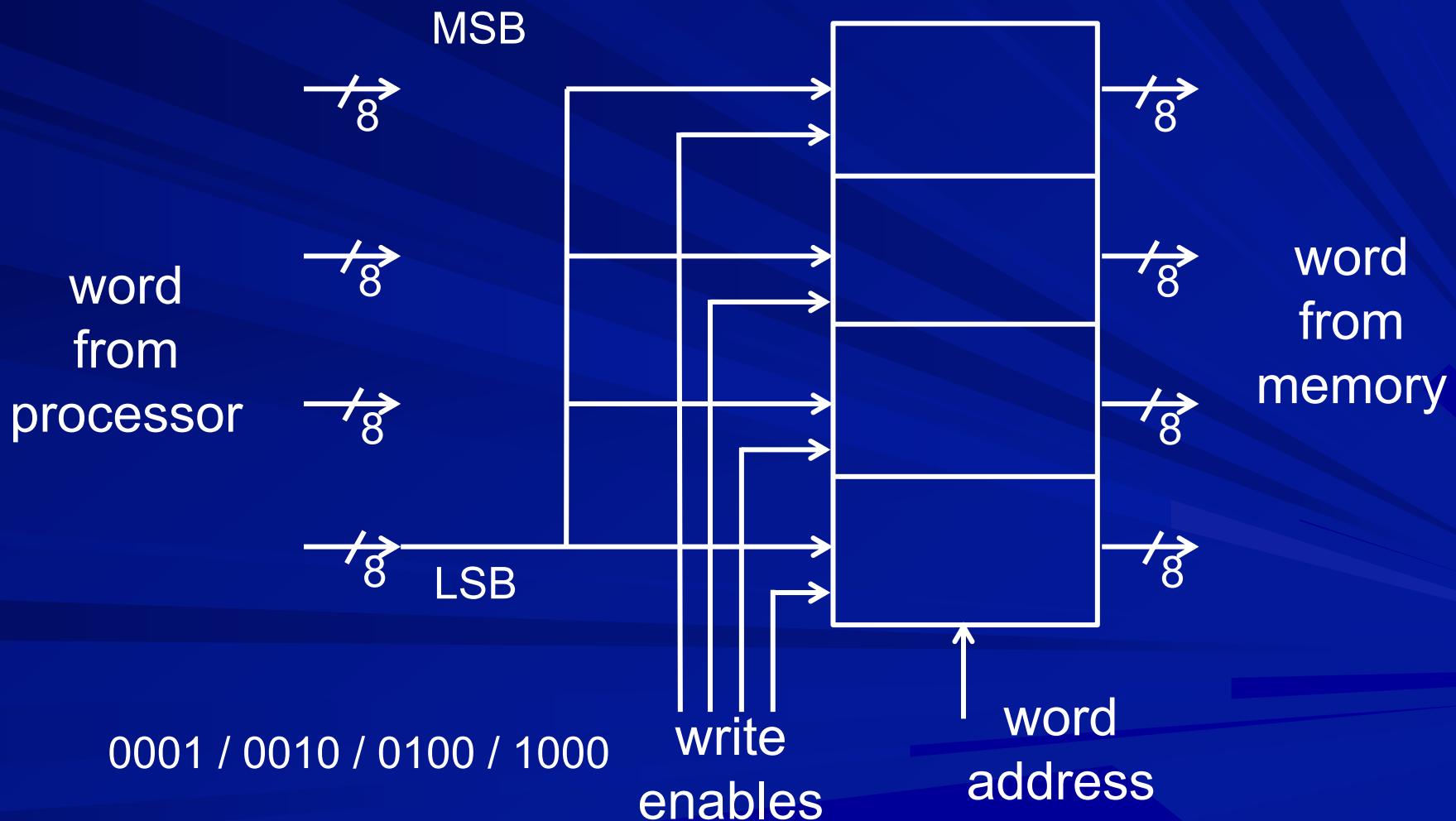
word
from
memory



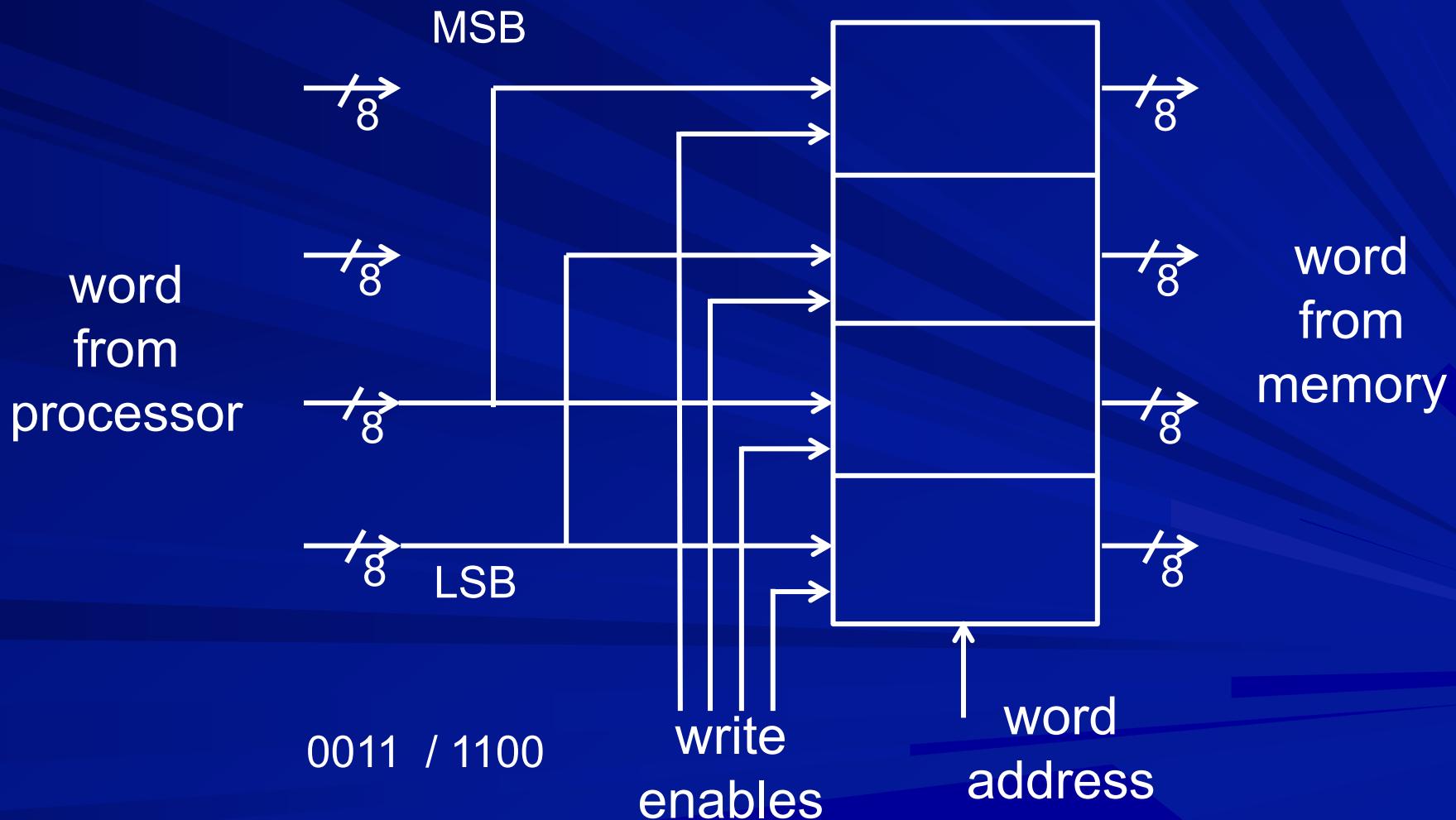
Extending from byte to word



Writing a byte in memory



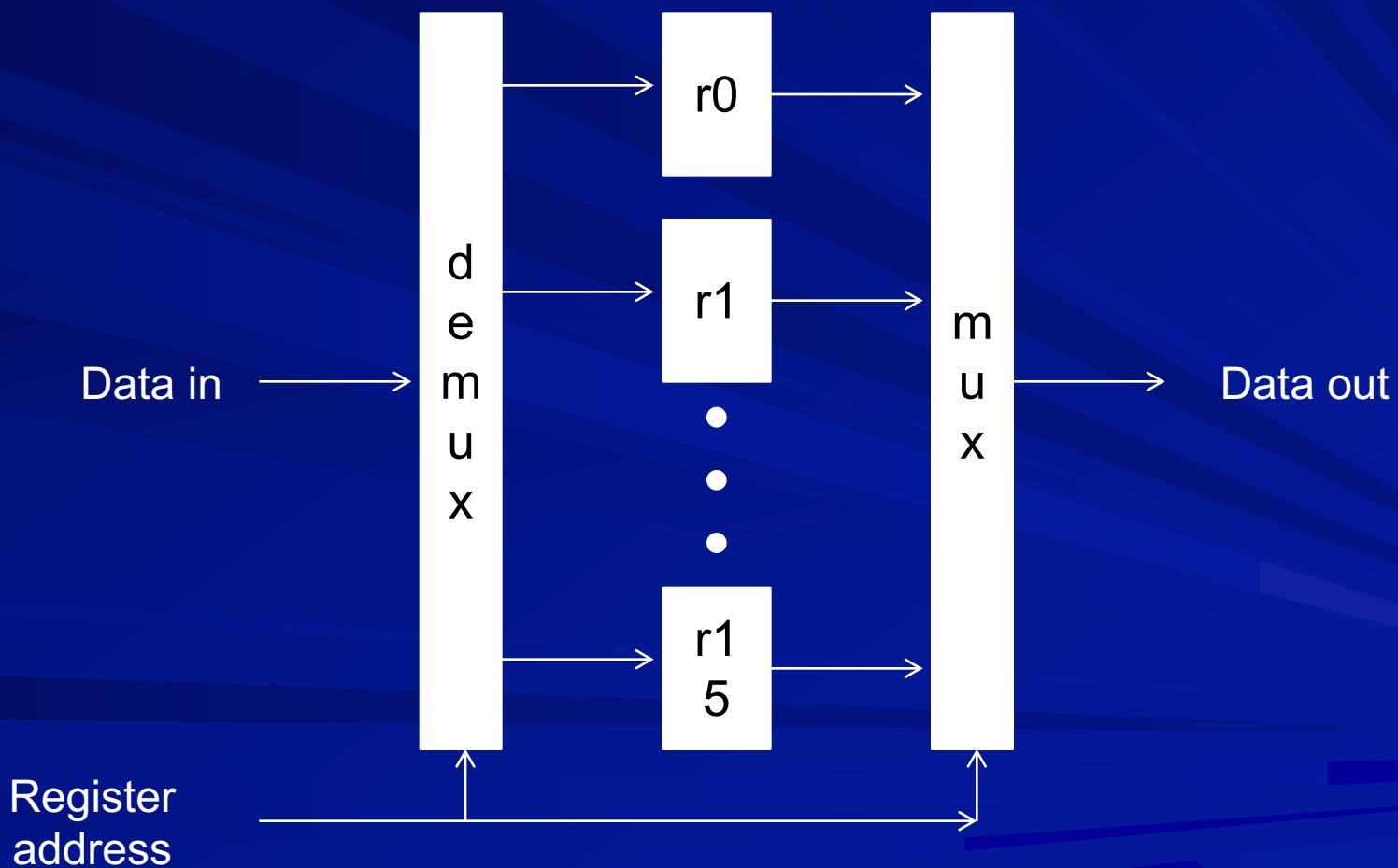
Writing a half word in memory



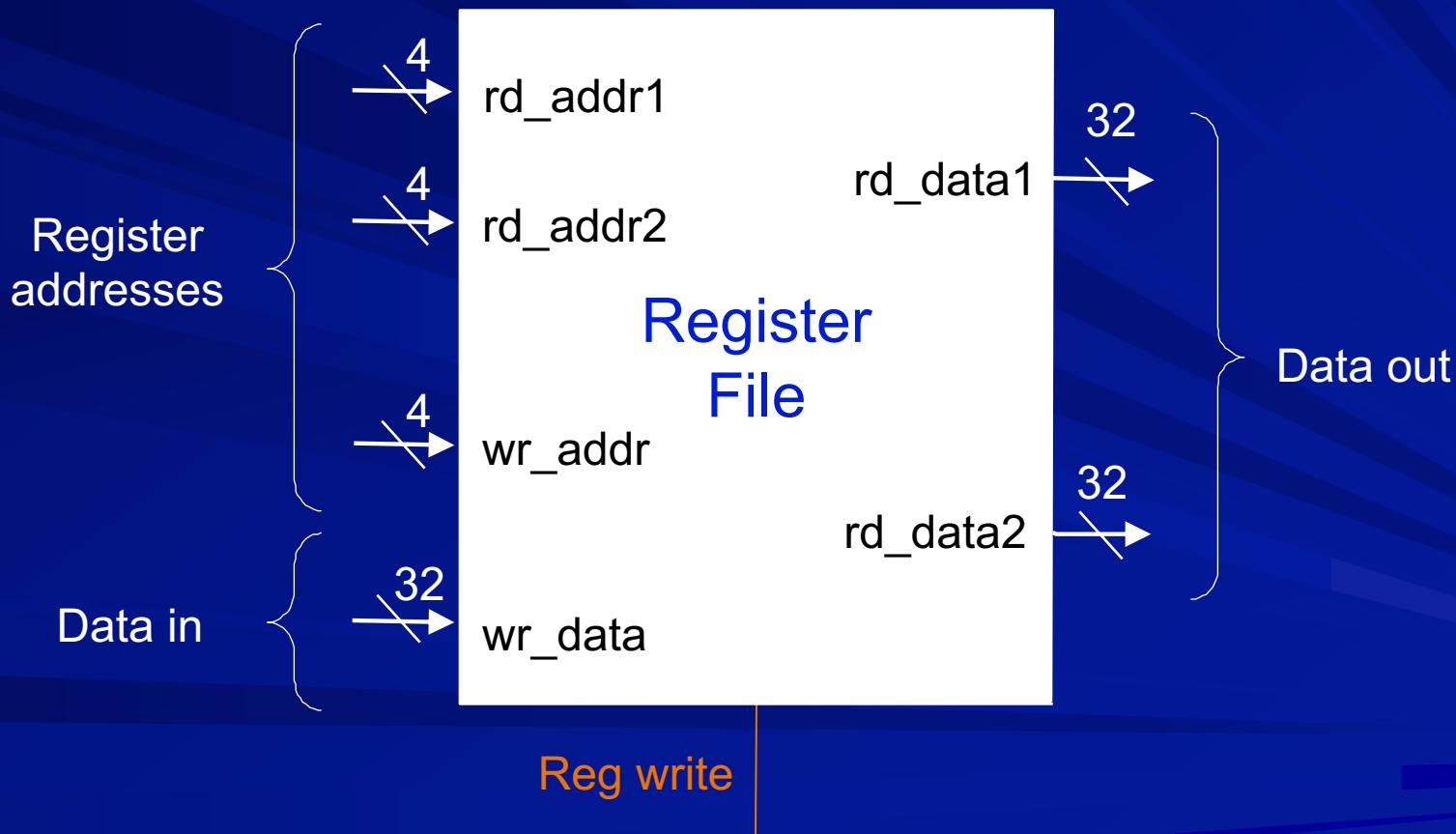
Register file



Register file



Register file (2R + 1W ports)

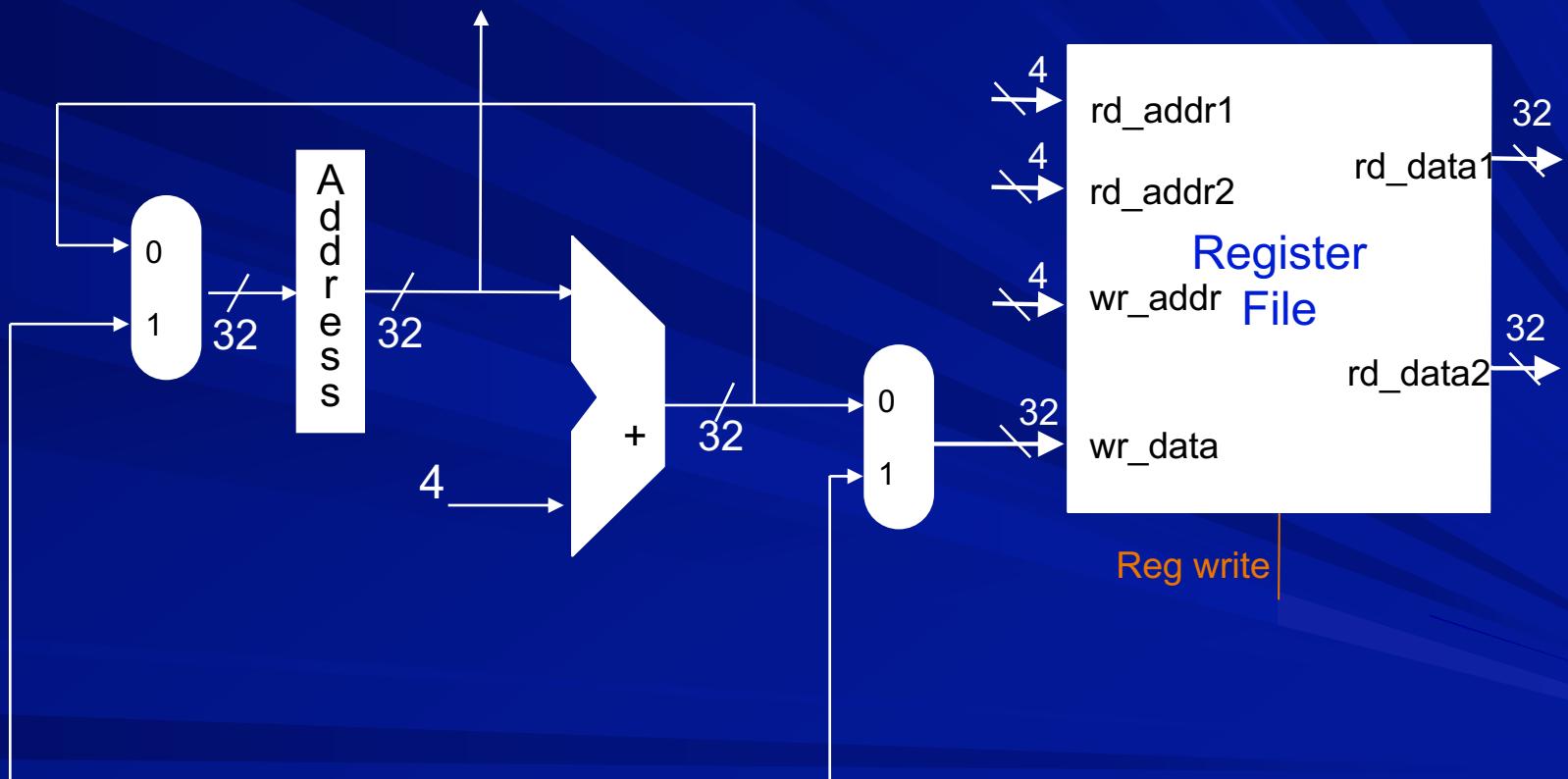


RF ports for ARM

- 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

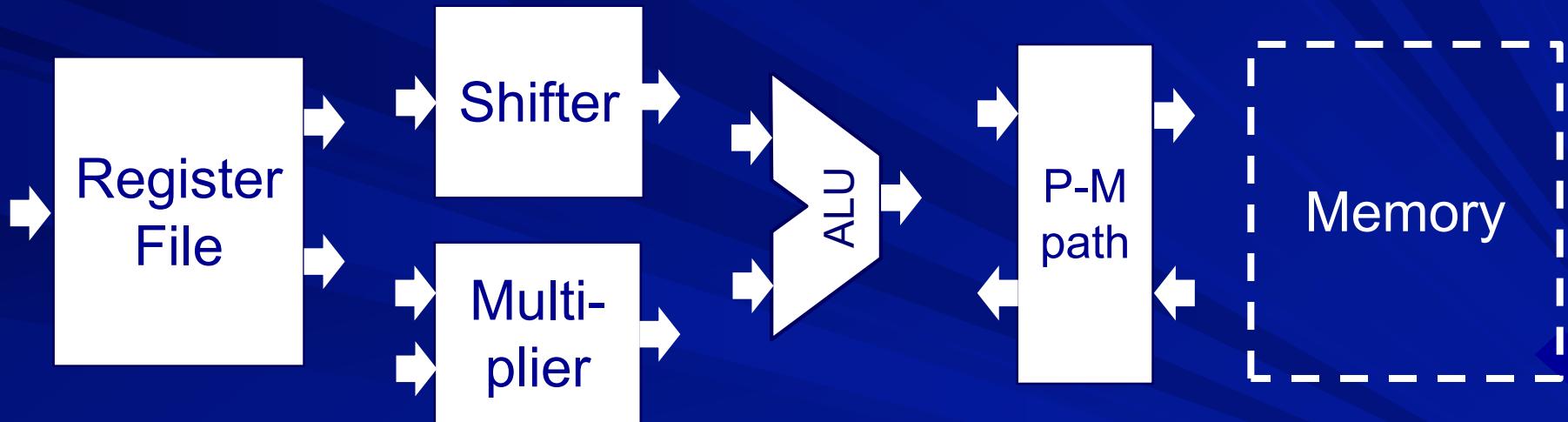
COL216

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 x 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)

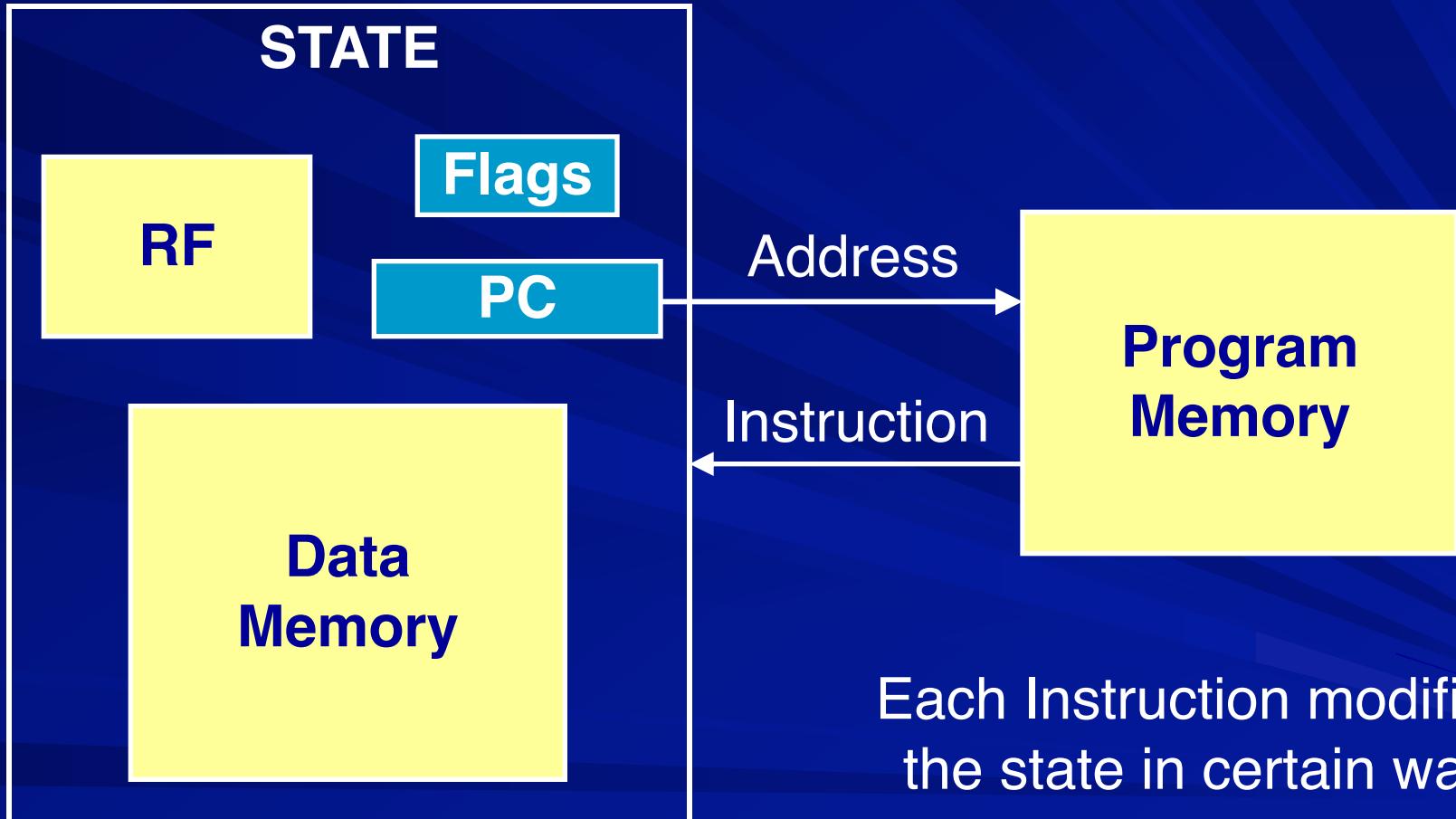
Lowest level
visible to a
programmer

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

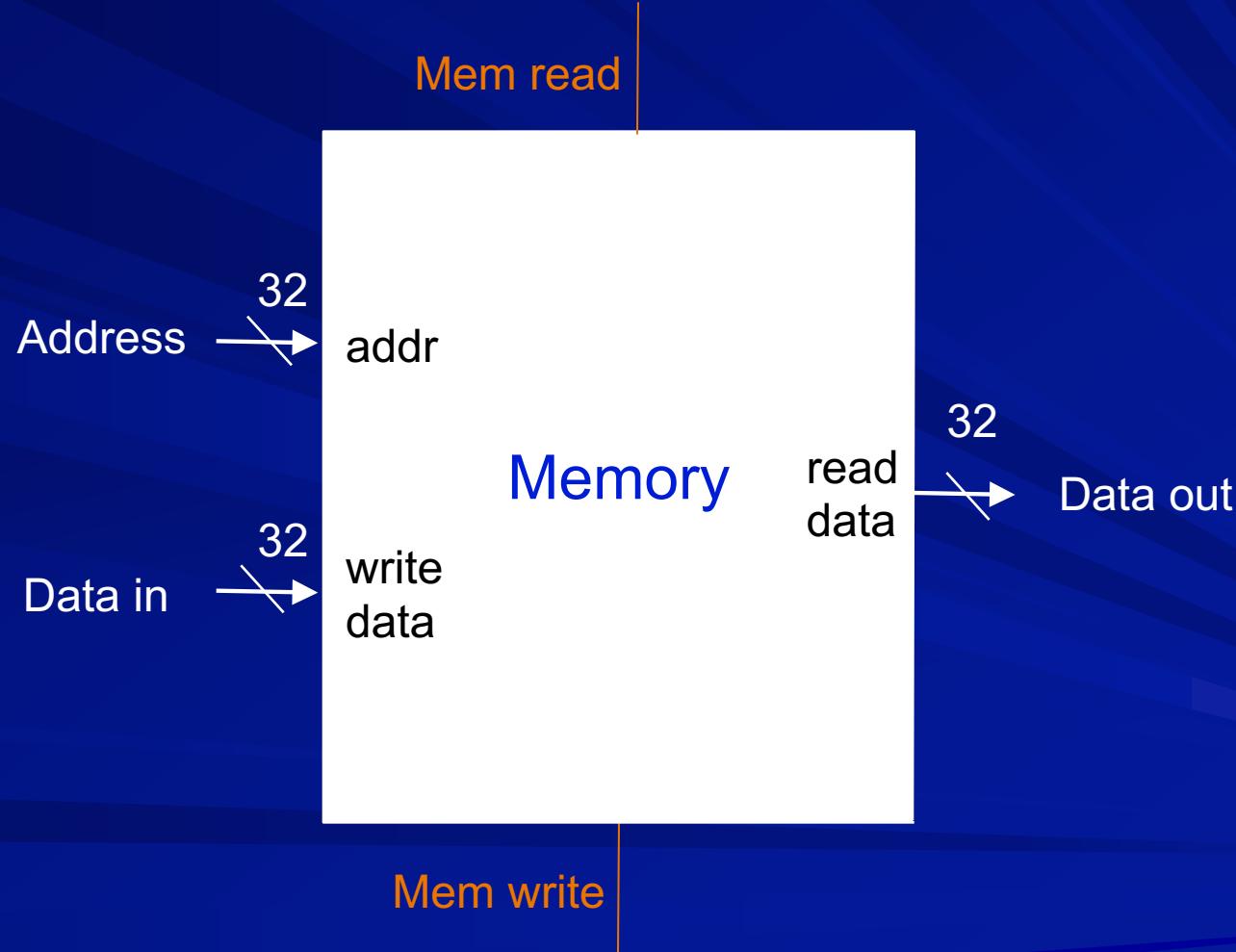
Timings across

- 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

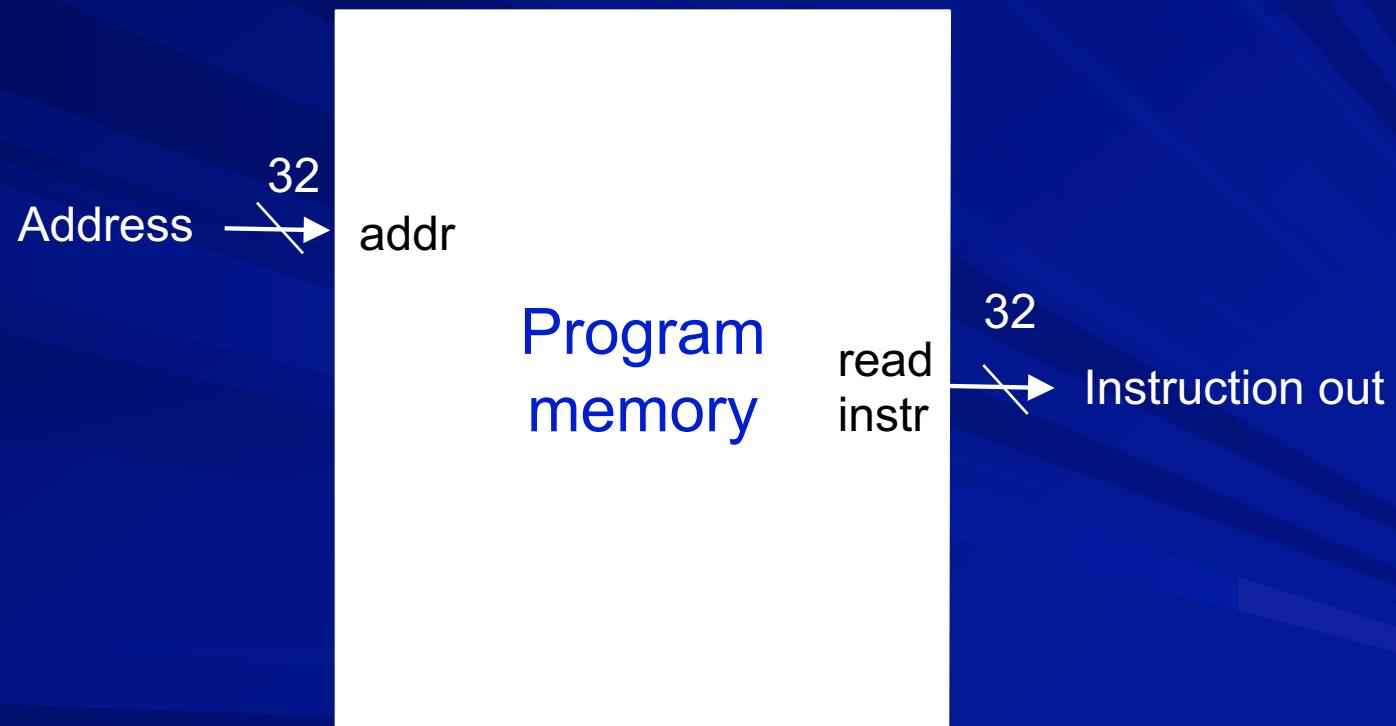
Memory



Program and data memory

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

Program memory

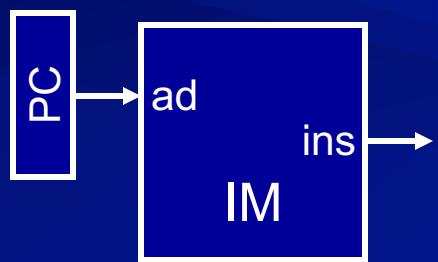


Building datapath

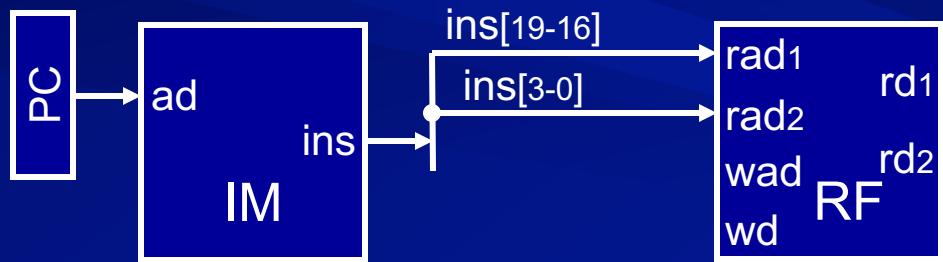
Actions for DP instructions

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

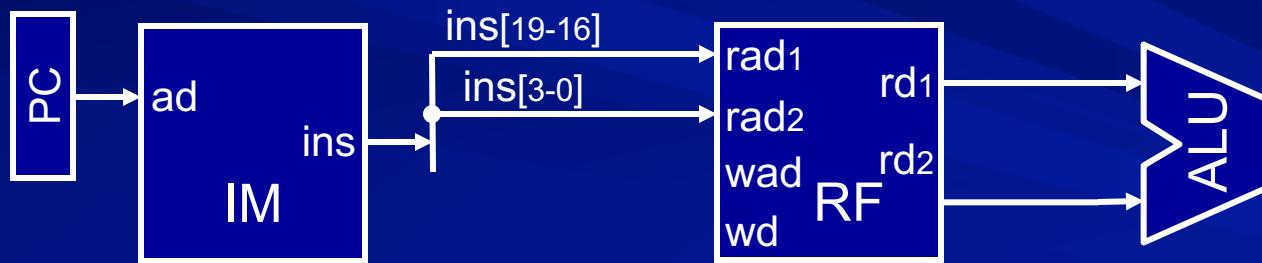
Fetching instruction



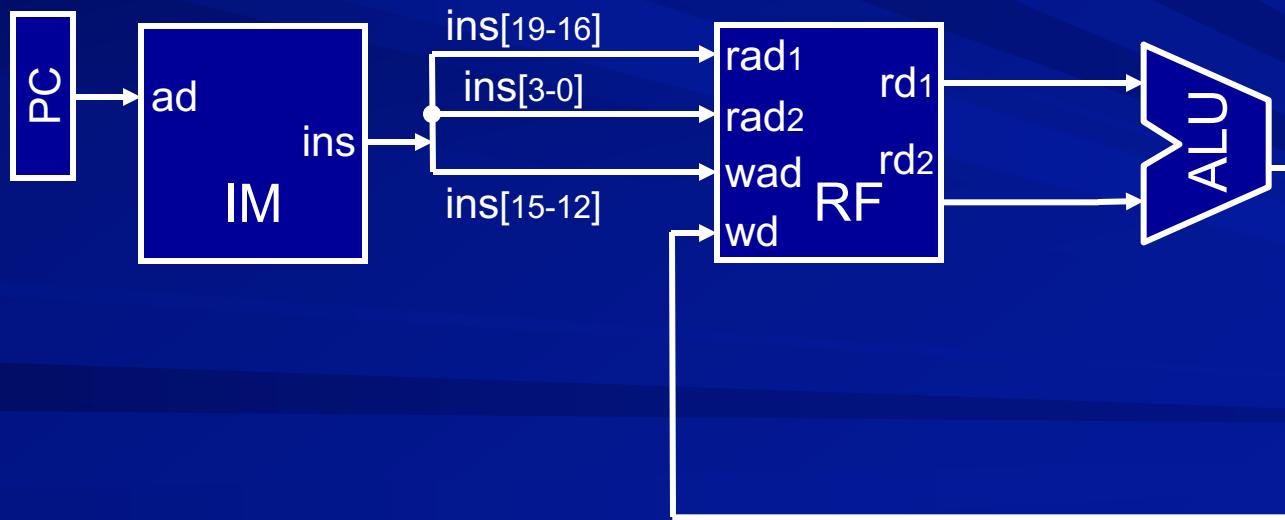
Accessing RF



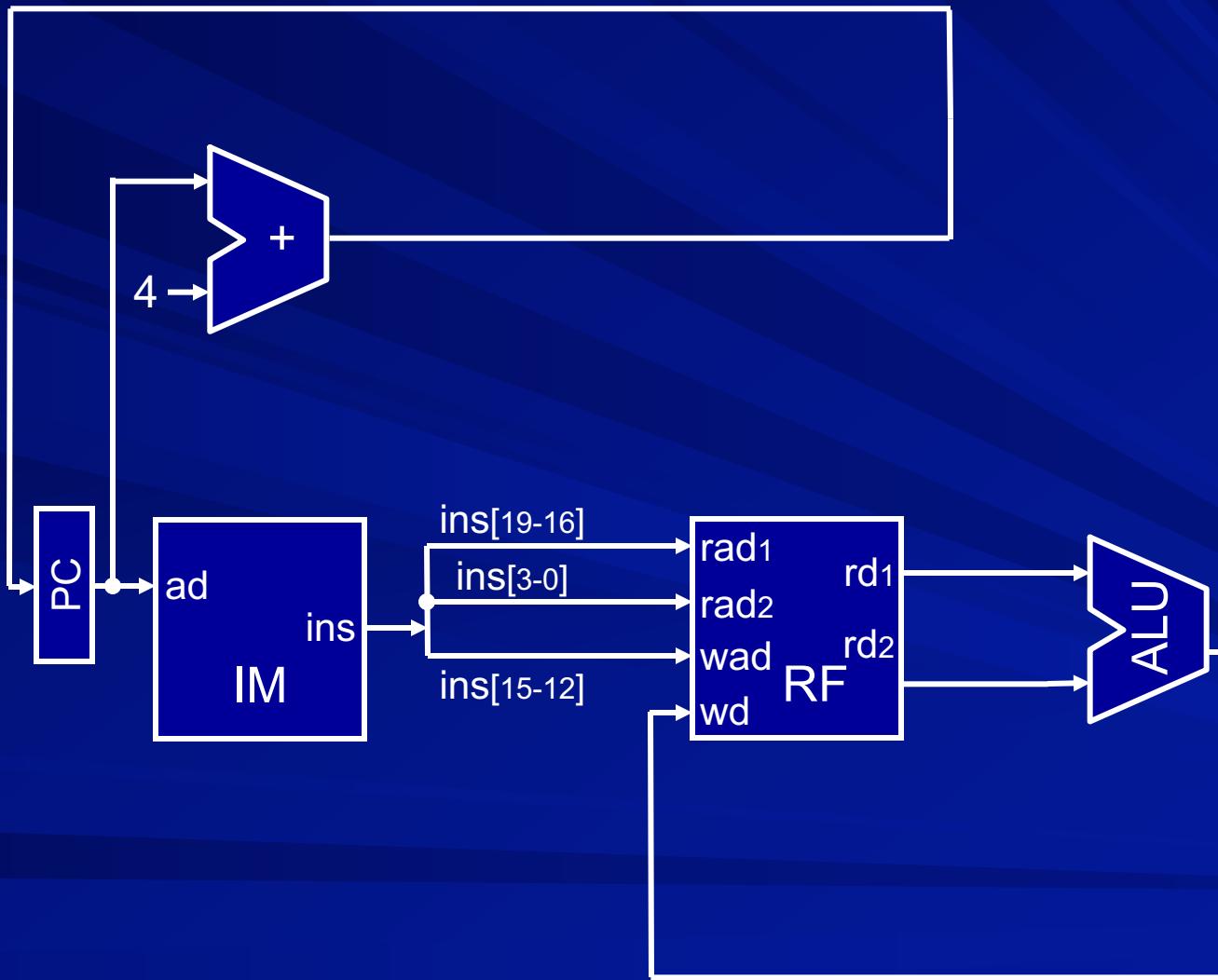
Passing operands to ALU



Passing the result to RF



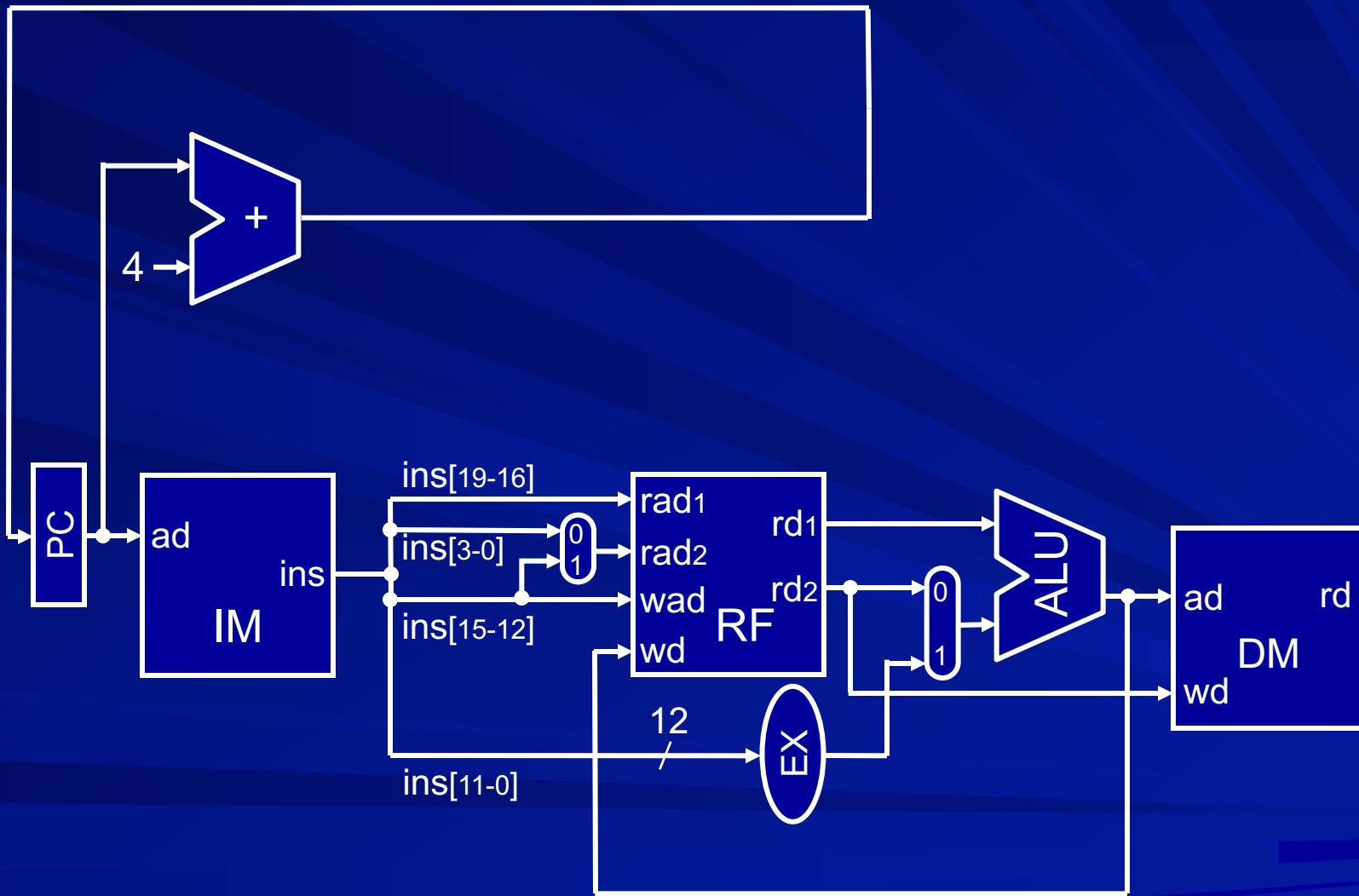
Incrementing PC



Actions for str instructions

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

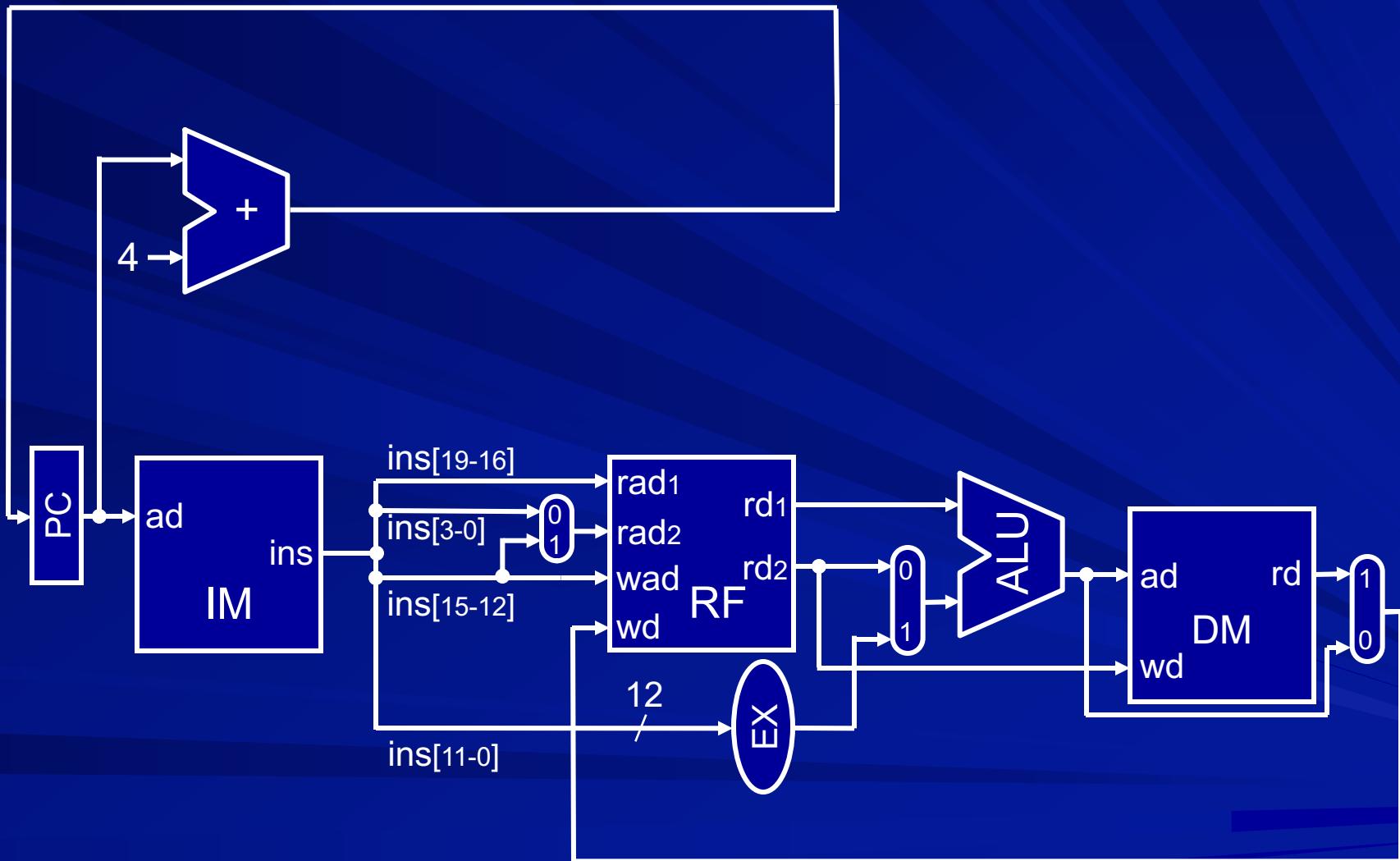
Adding “str” instruction



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

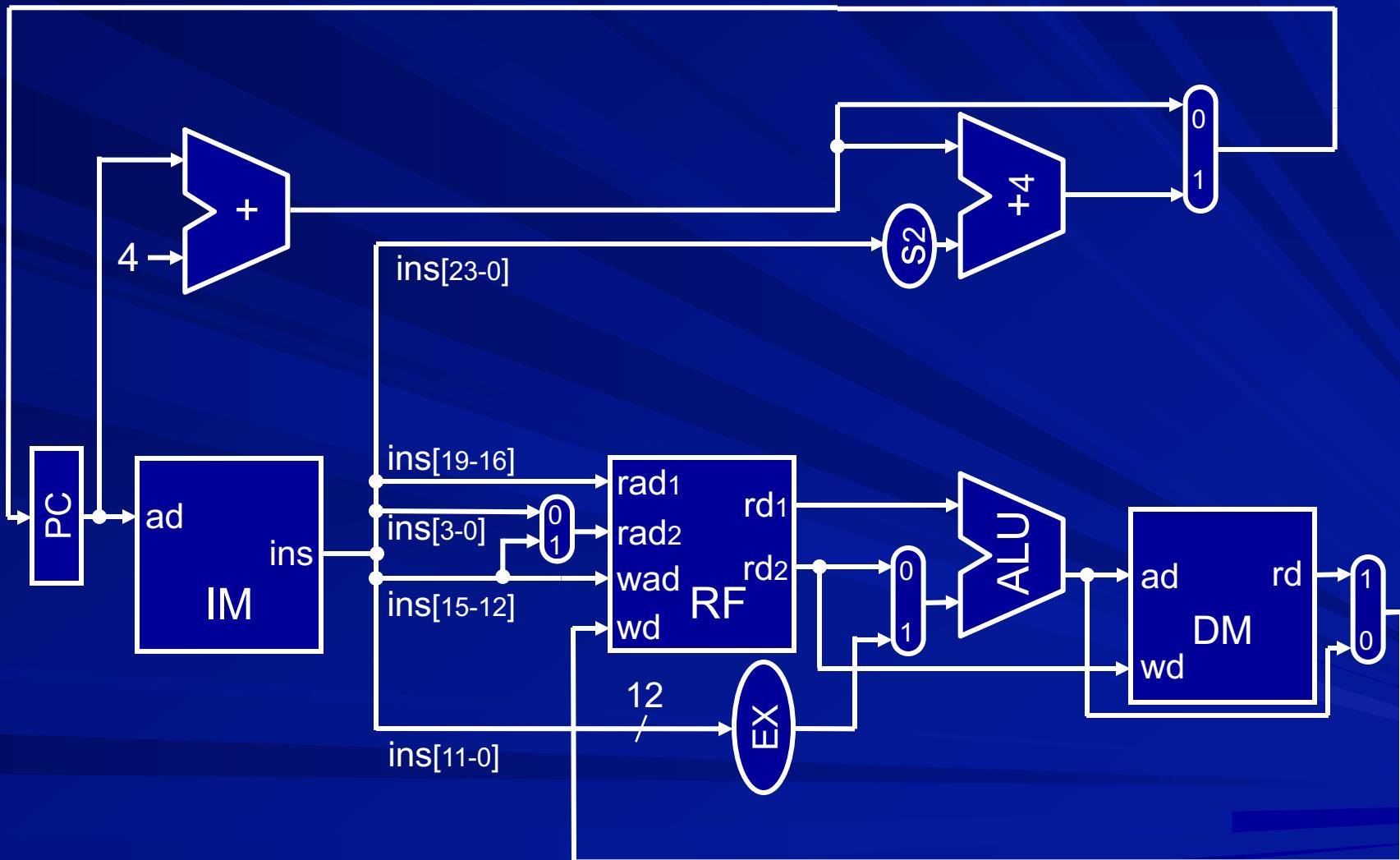
Adding “ldr” instruction



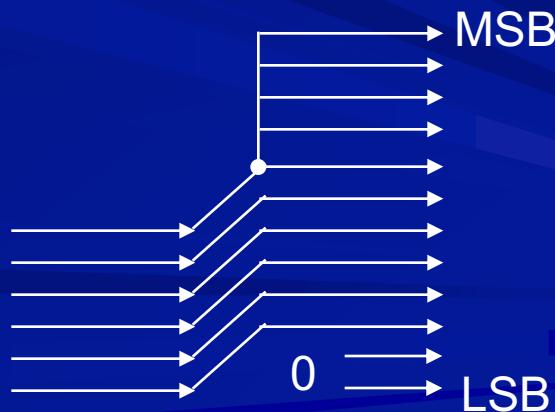
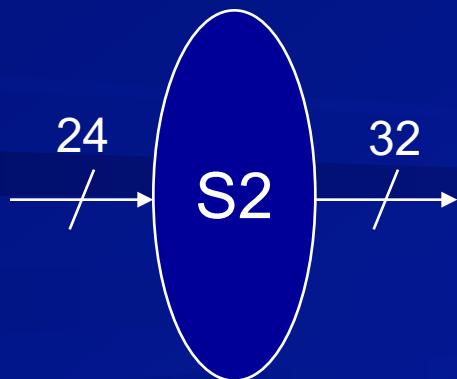
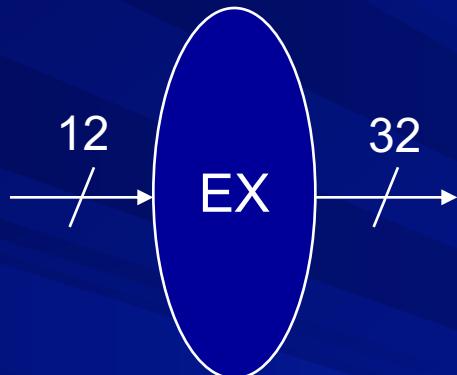
Actions for branch instruction

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

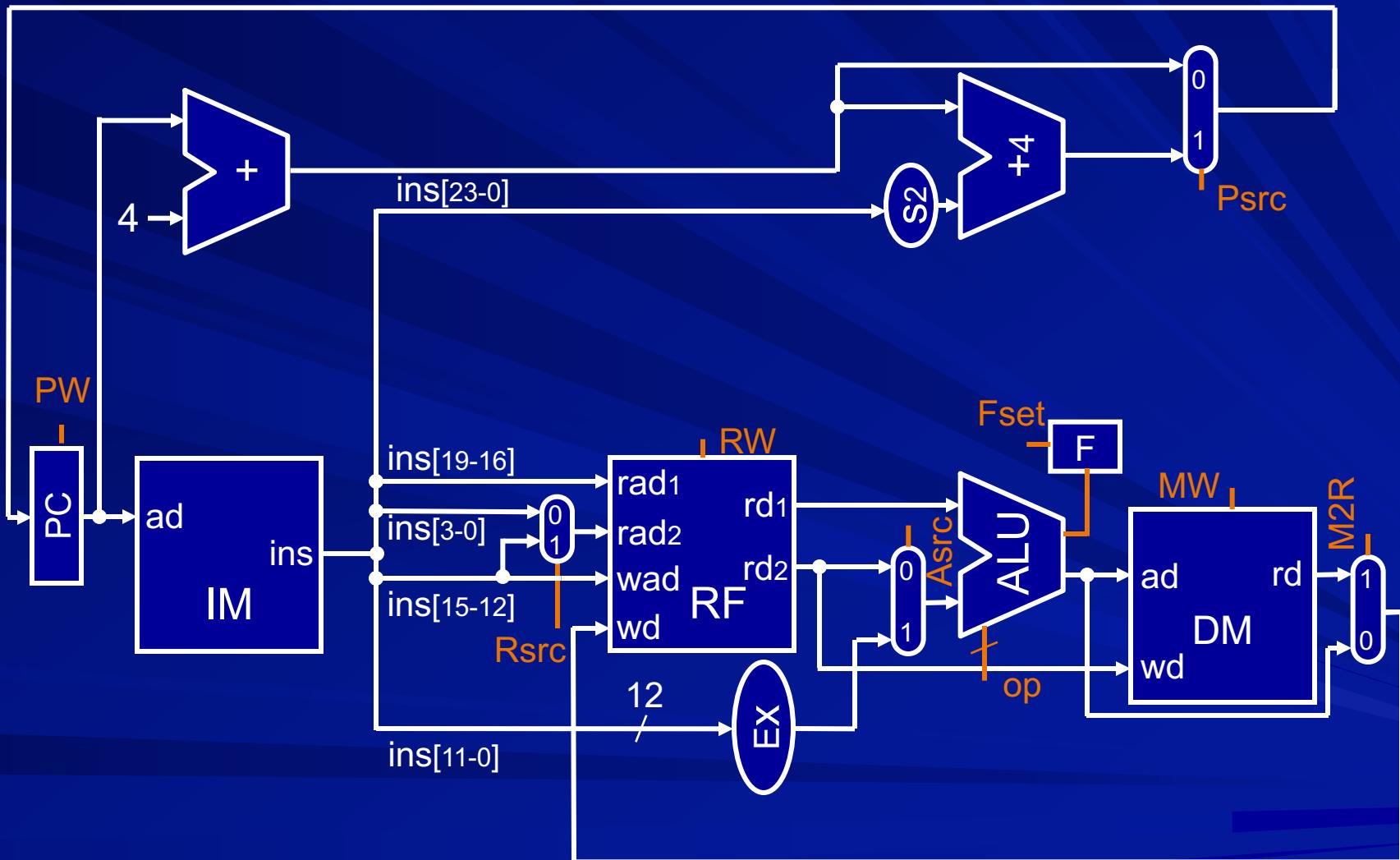
Adding “b” instruction



Extending offsets



Single cycle Datapath



Problems with single cycle design

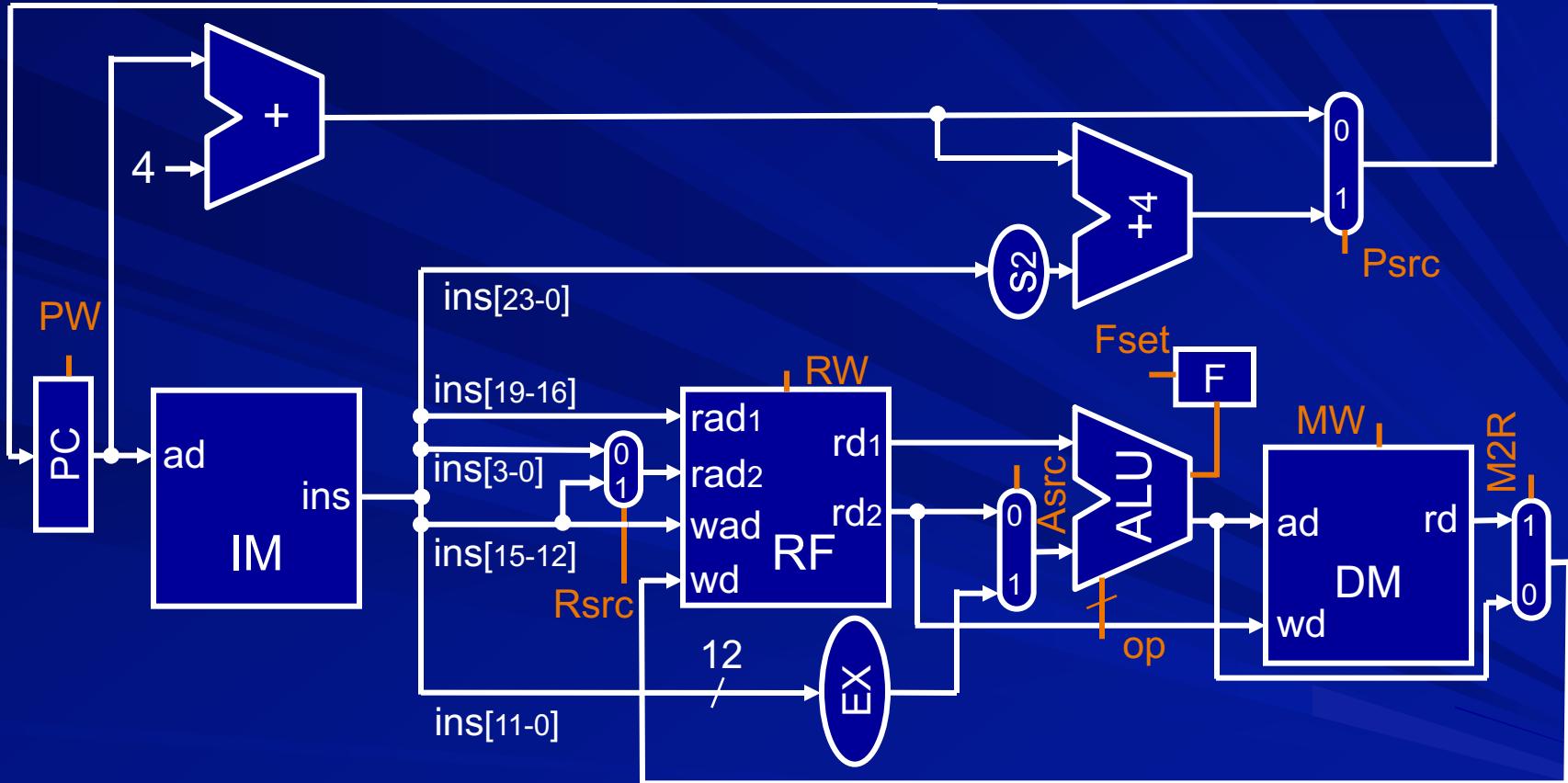
- Slowest instruction pulls down the clock frequency
- Resource utilization is poor
- There are some instructions which are impossible to be implemented in this manner

Analyzing performance

Component delays

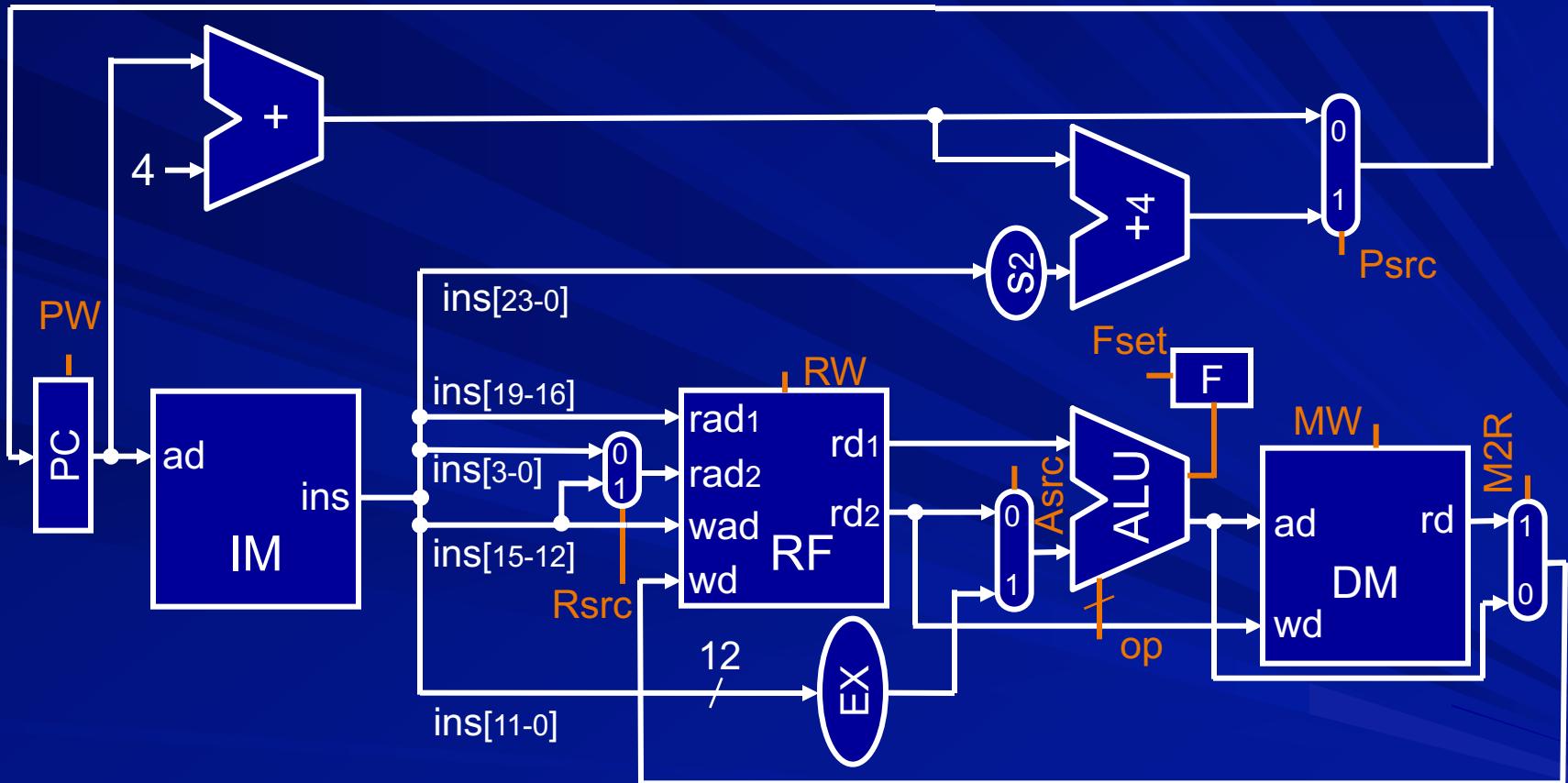
■ Register	0
■ Adder	t_+
■ ALU	t_A
■ Multiplexer	0
■ Register file	t_R
■ Program memory	t_I
■ Data memory	t_M
■ Bit manipulation components	0

Delay for {add, sub, ...}



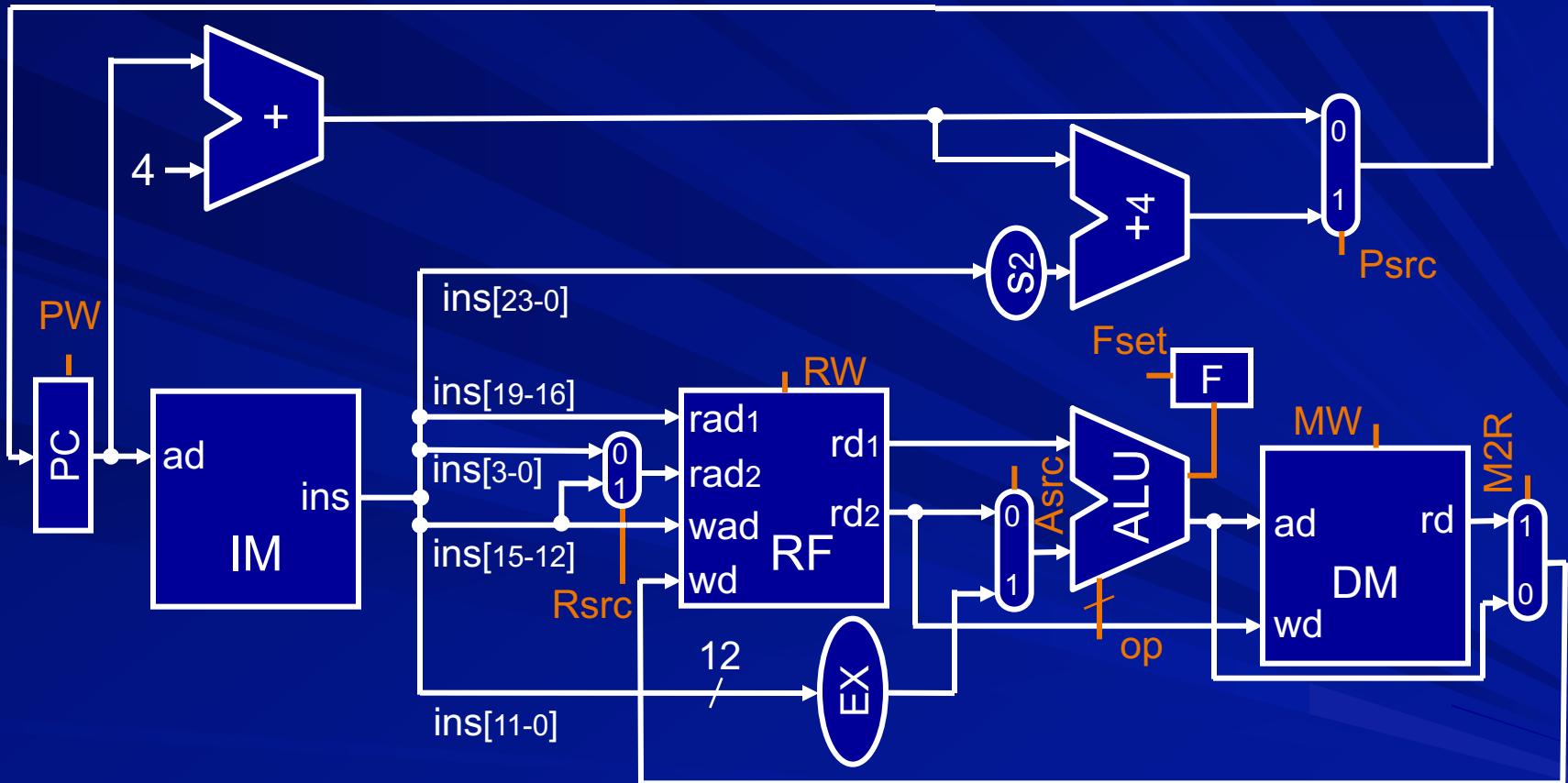
$$\max \left\{ \begin{array}{c} t_+ \\ t_I + t_R + t_A + t_R \end{array} \right\}$$

Delay for {cmp, tst, ...}



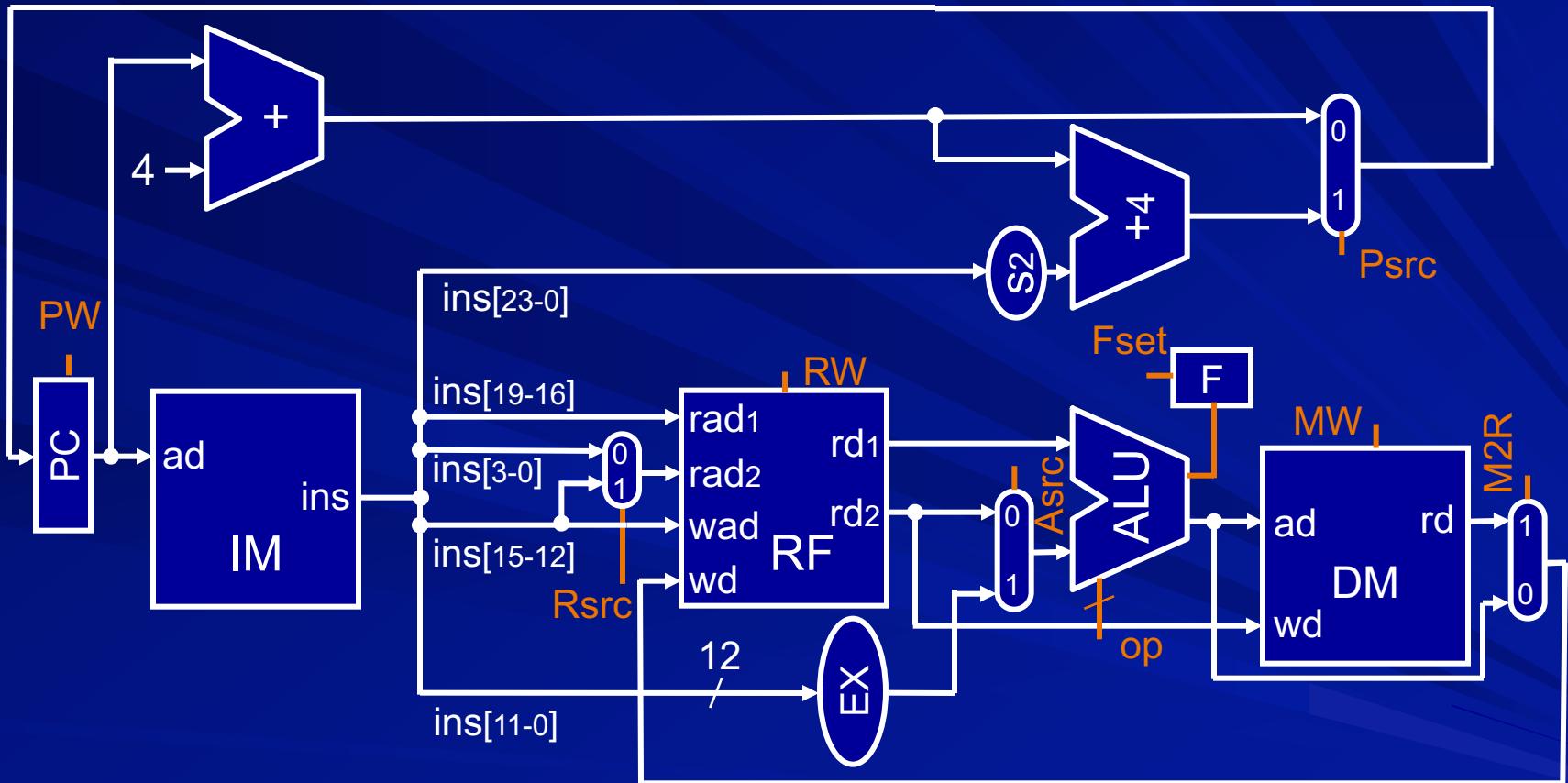
$$\max \left\{ \begin{array}{l} t_+ \\ t_I + t_R + t_A \end{array} \right\}$$

Delay for {str}



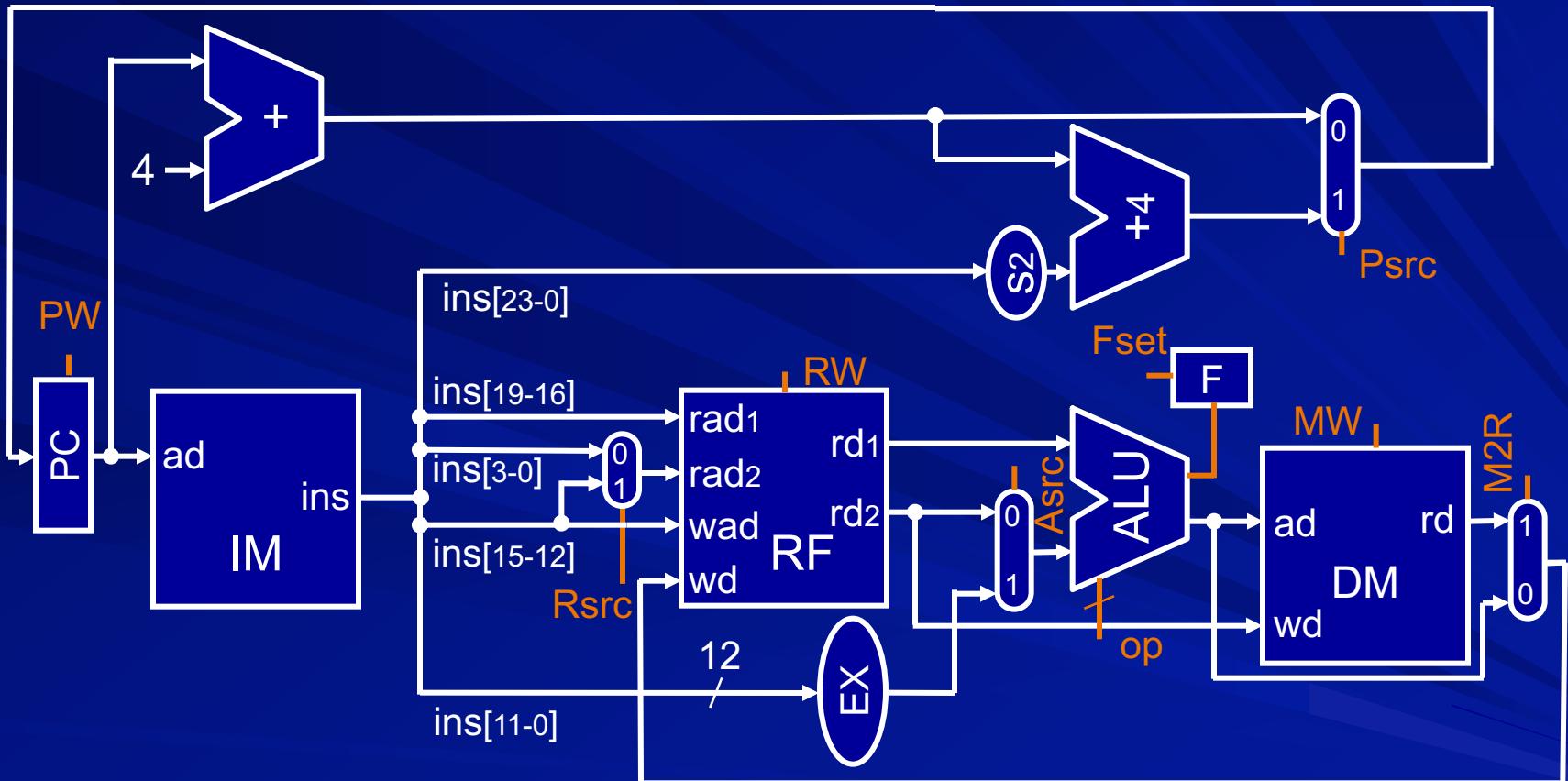
$$\max \left\{ \begin{array}{l} t_+ \\ t_I + t_R + t_A + t_M \end{array} \right\}$$

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

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

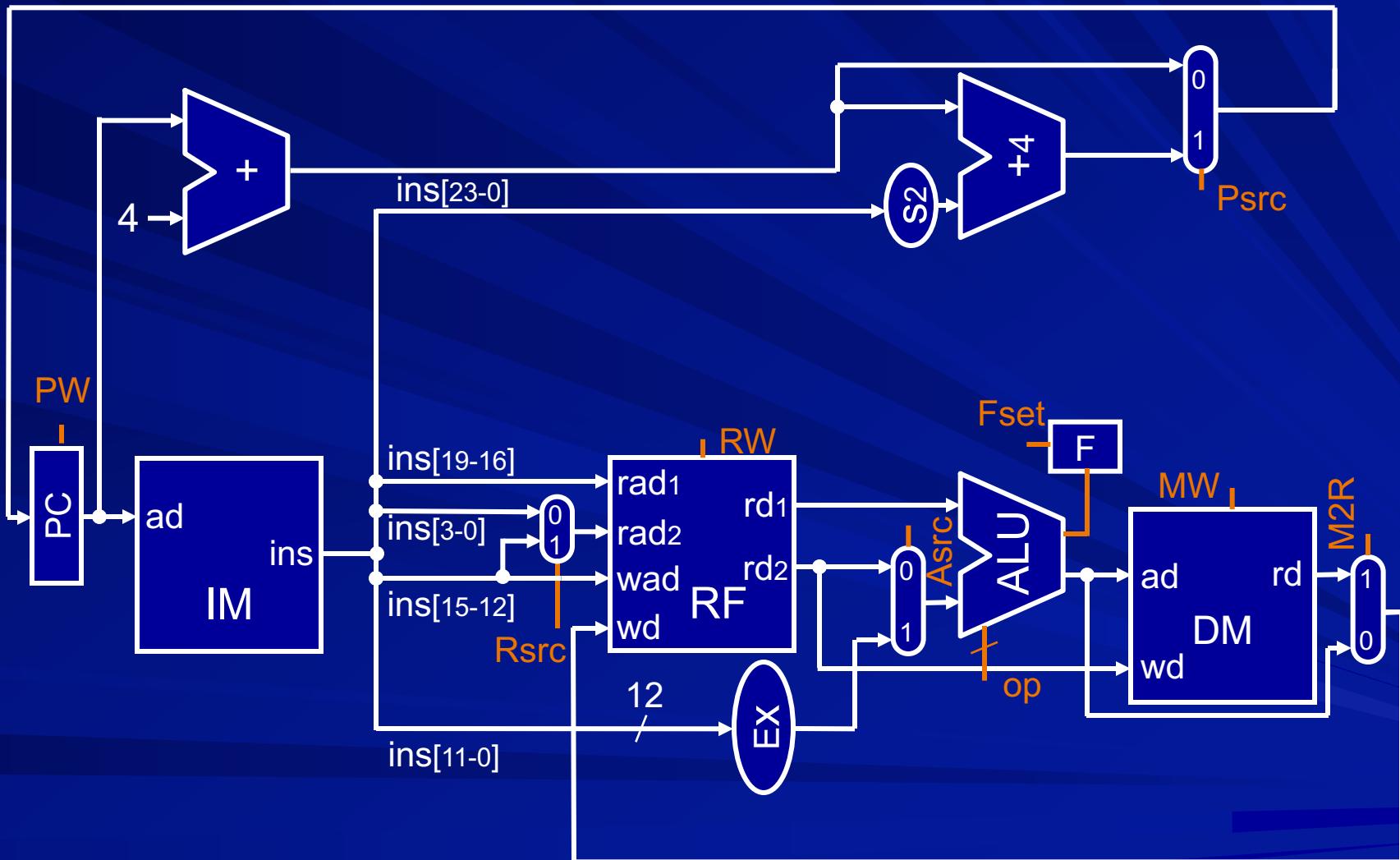
Thanks

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

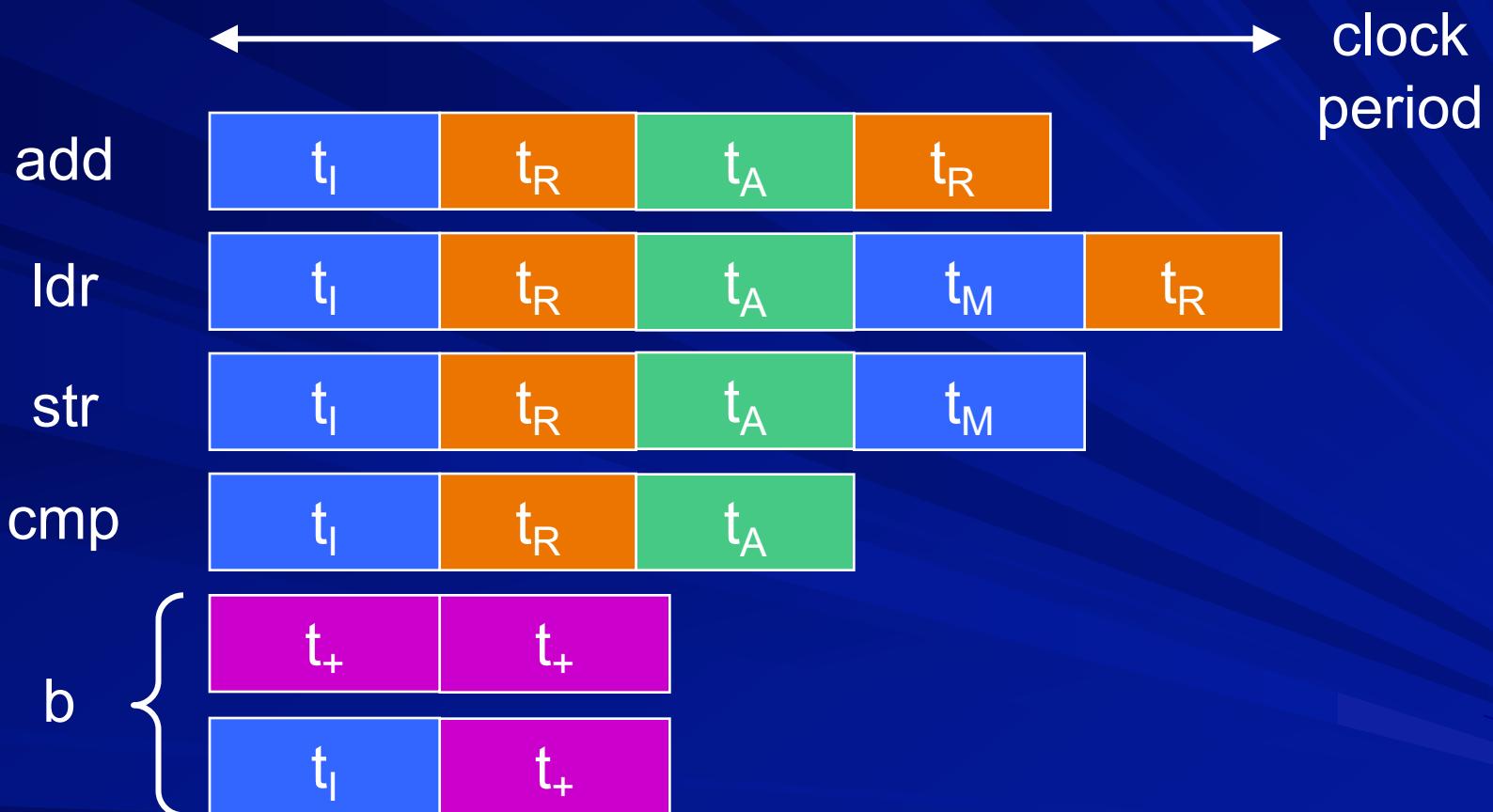
Component delays

■ 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

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

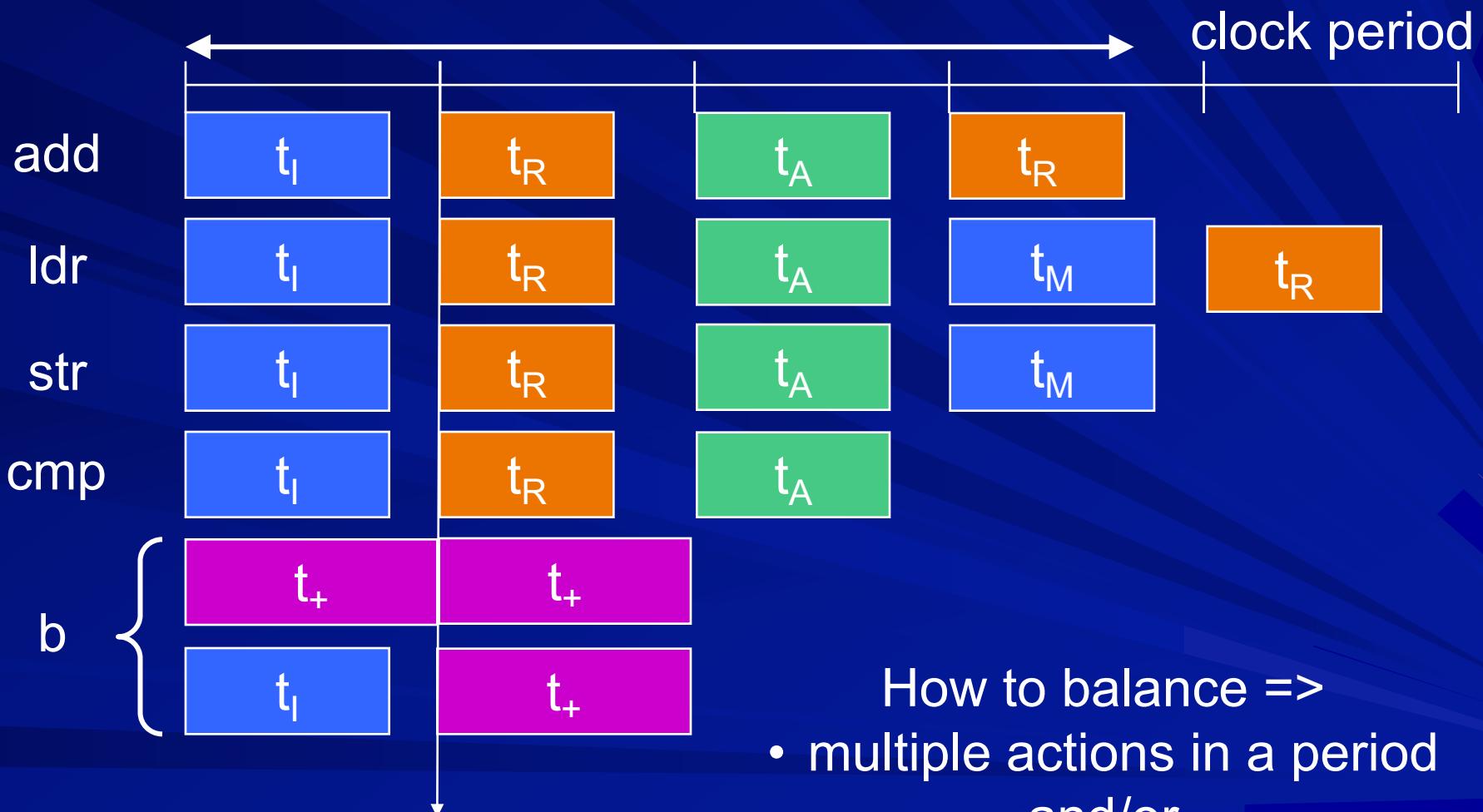
Clock period in single cycle design



Split the cycle into multiple cycles



Unbalanced delays



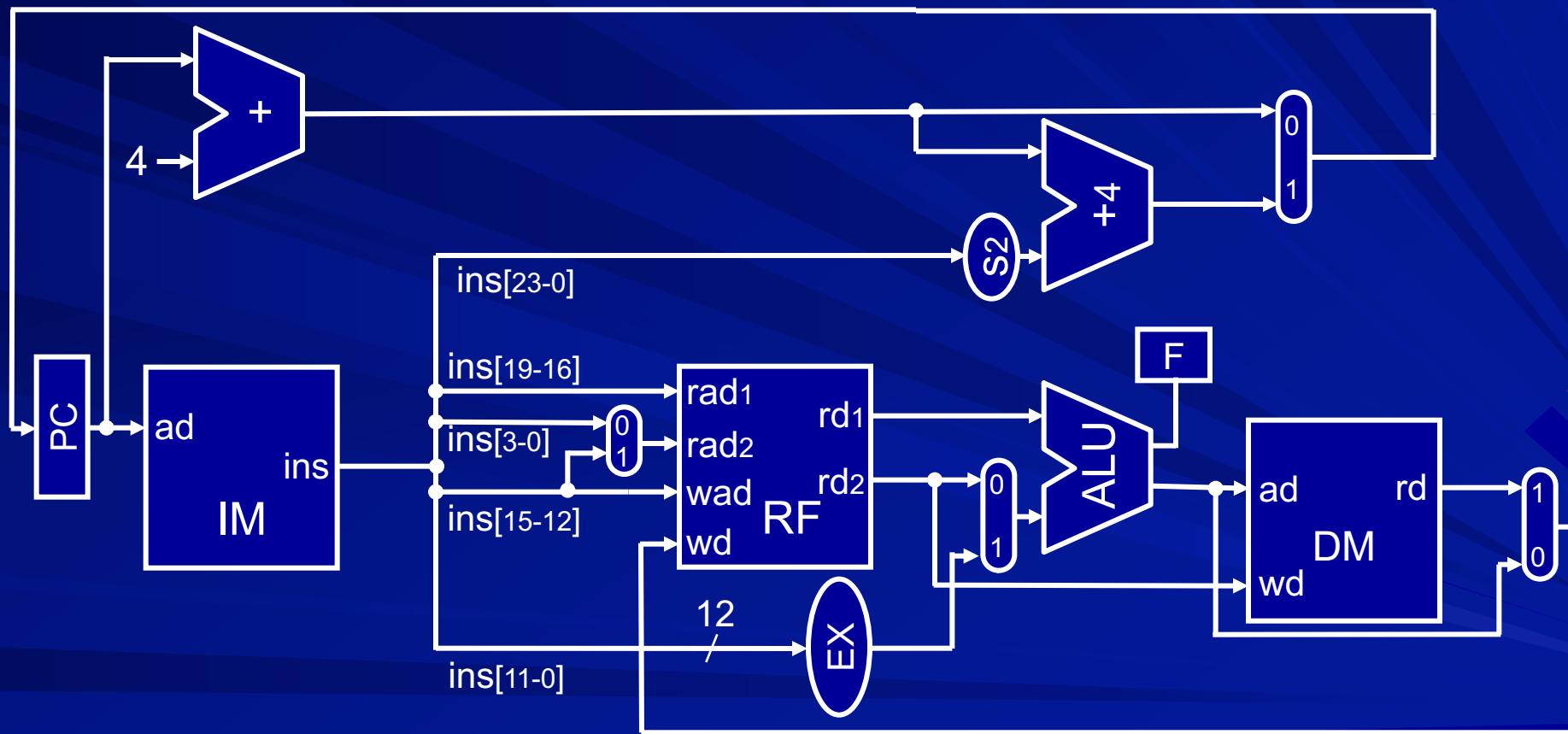
How to balance =>

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

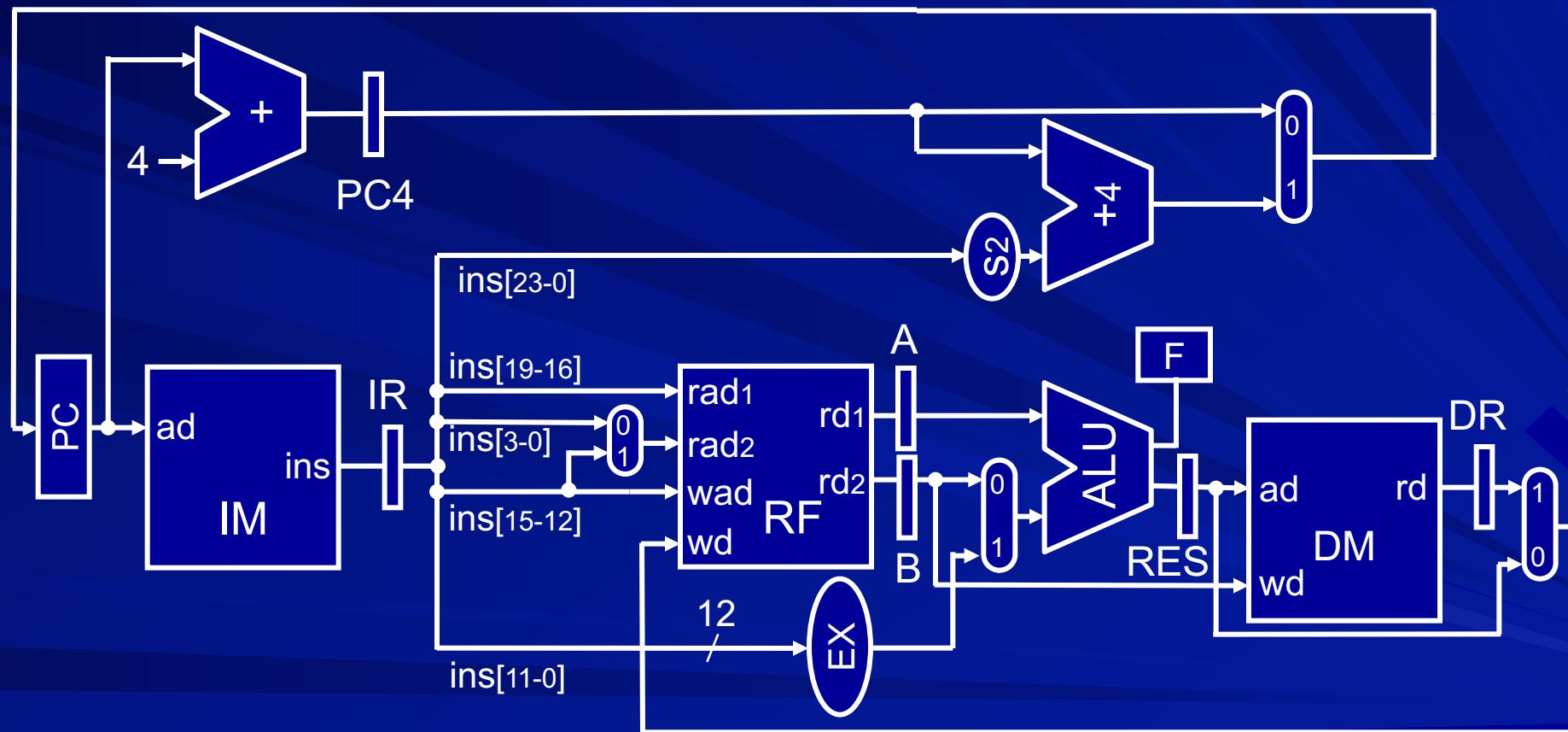
Improving resource utilization

- 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

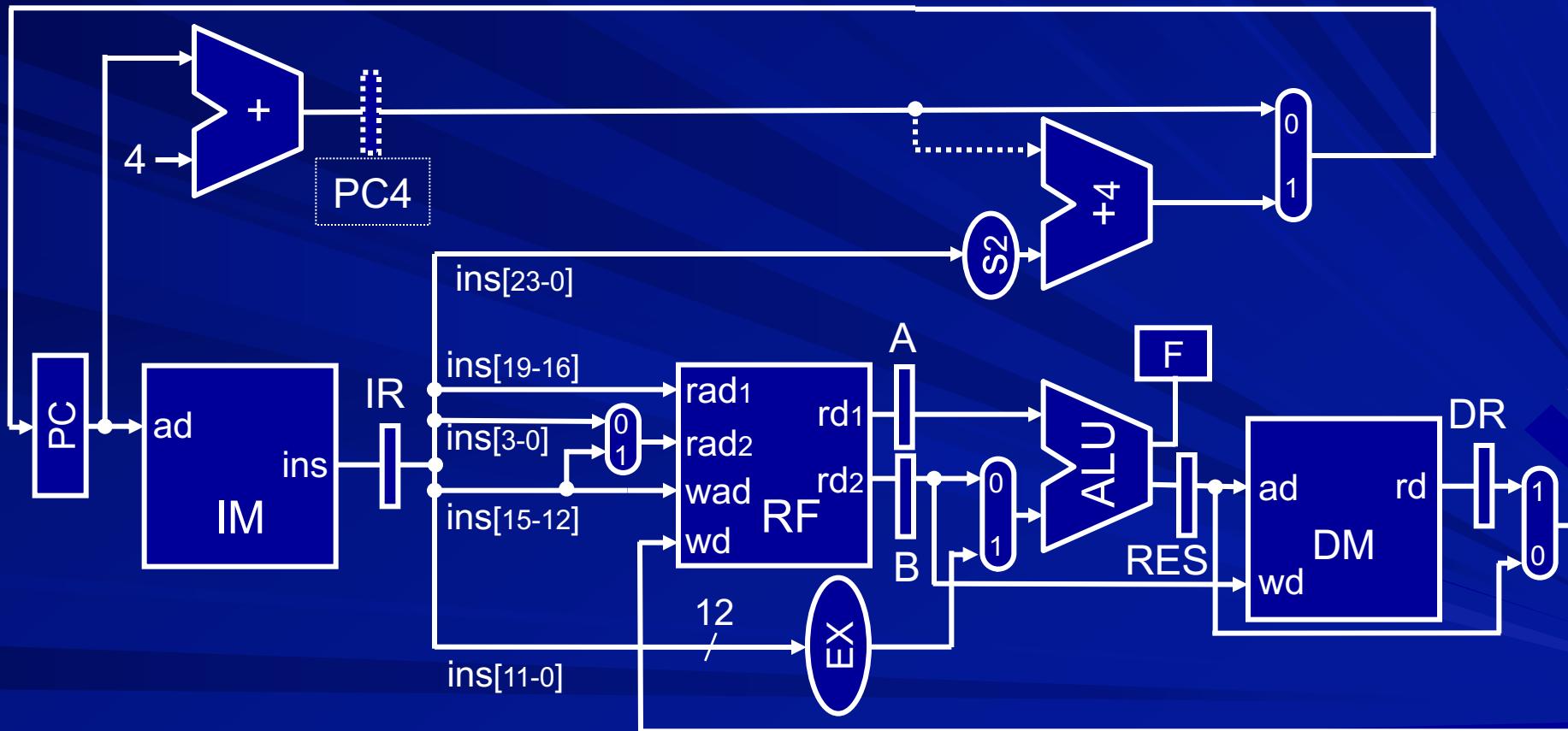


Introducing registers

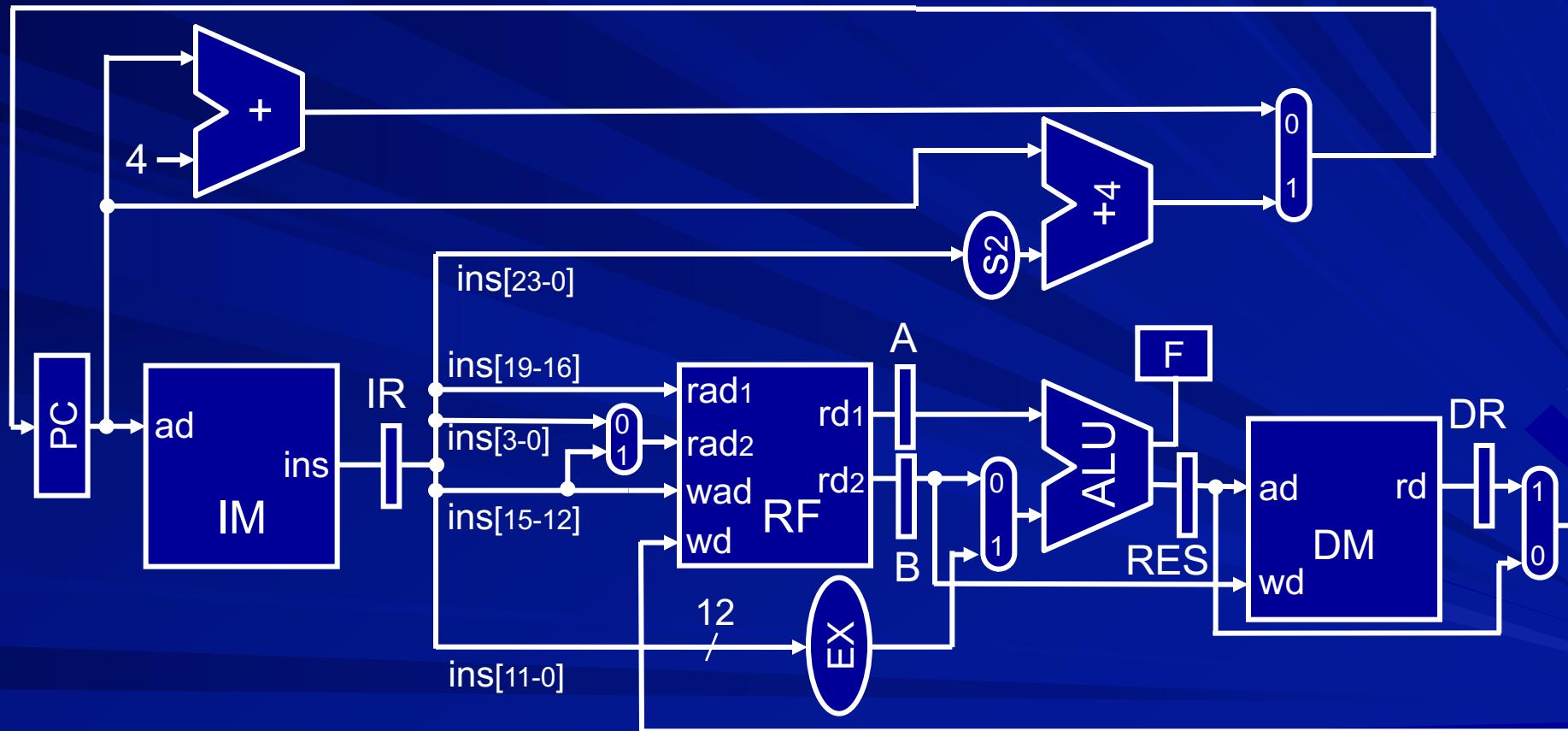


PC4 can be eliminated

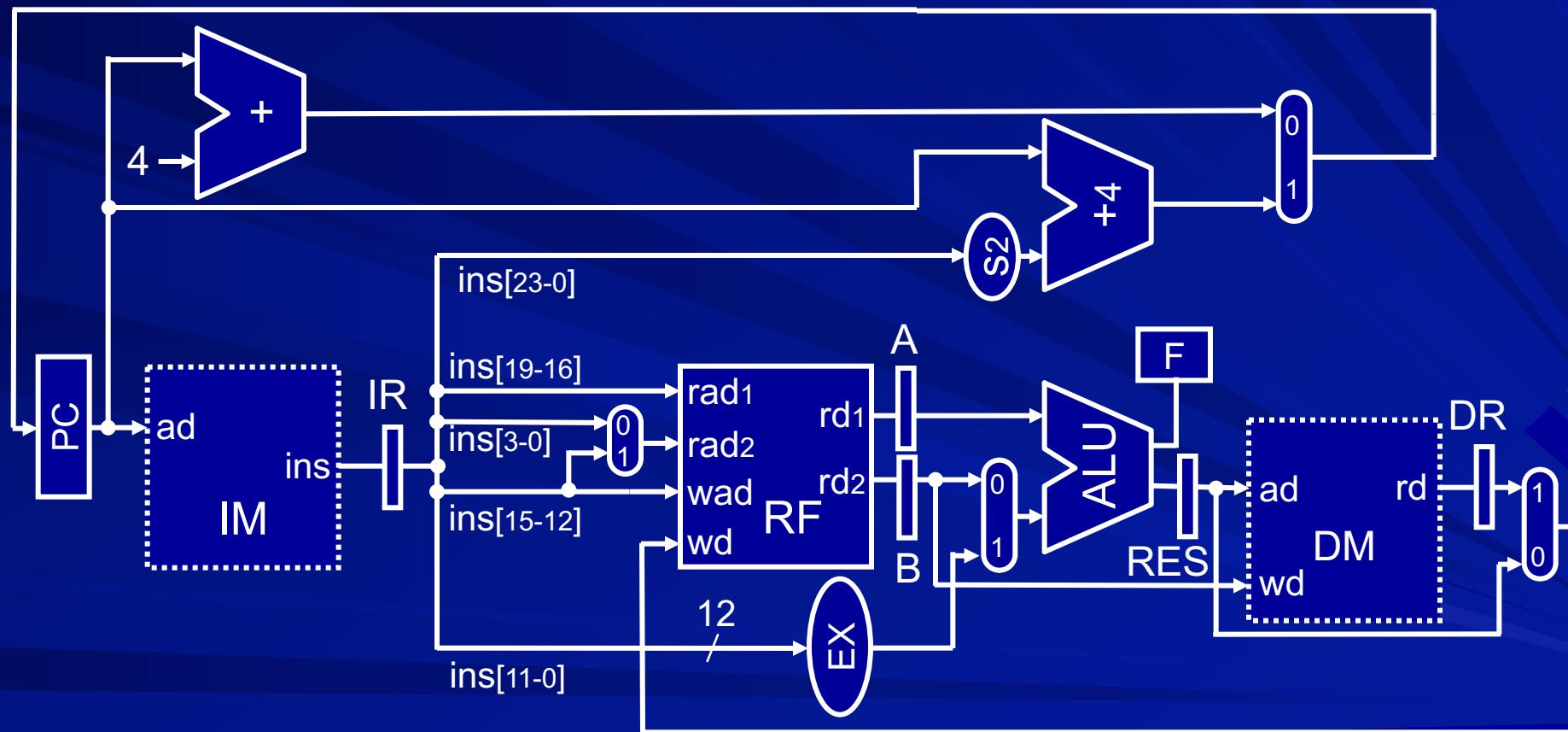
Store PC + 4 in PC itself



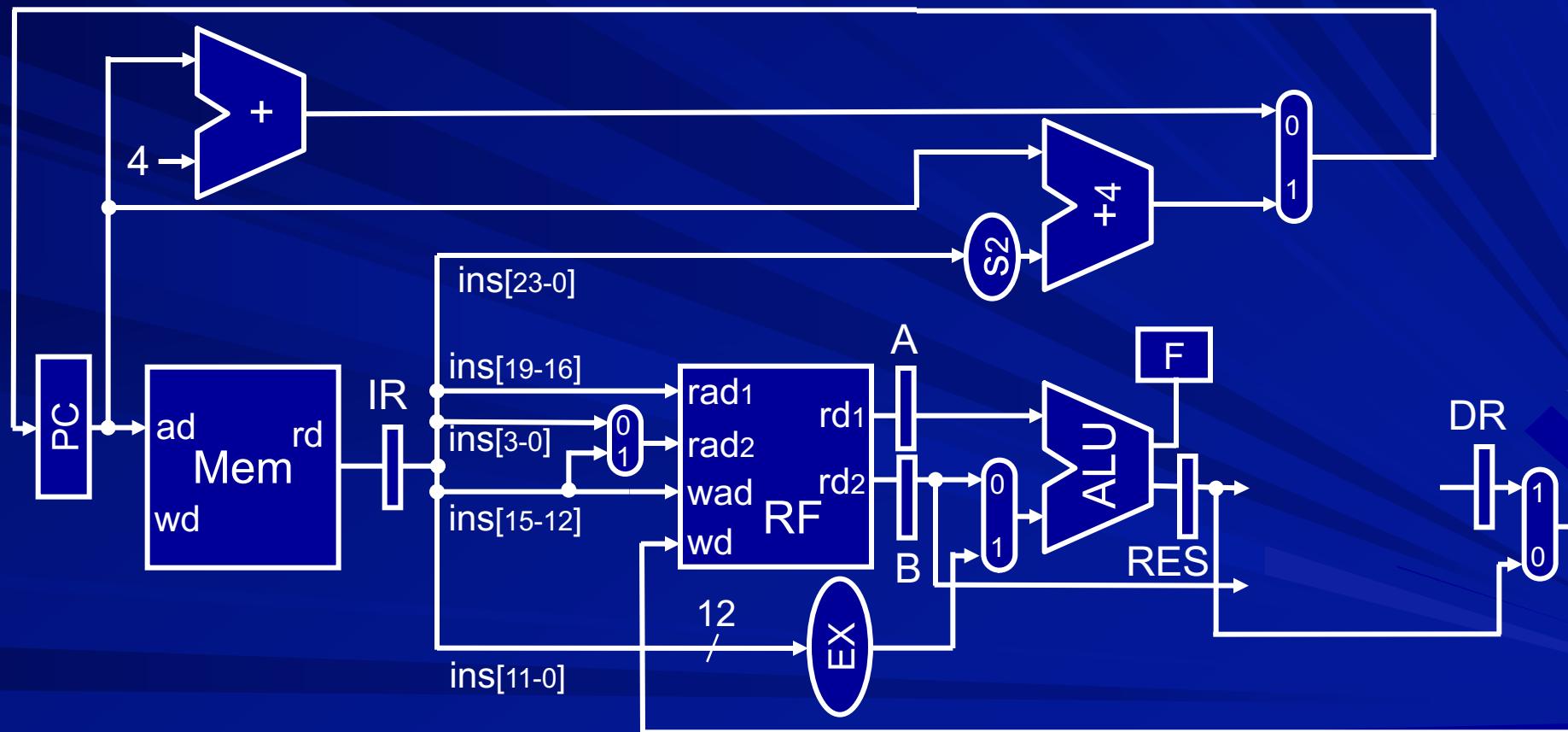
PC4 can be eliminated



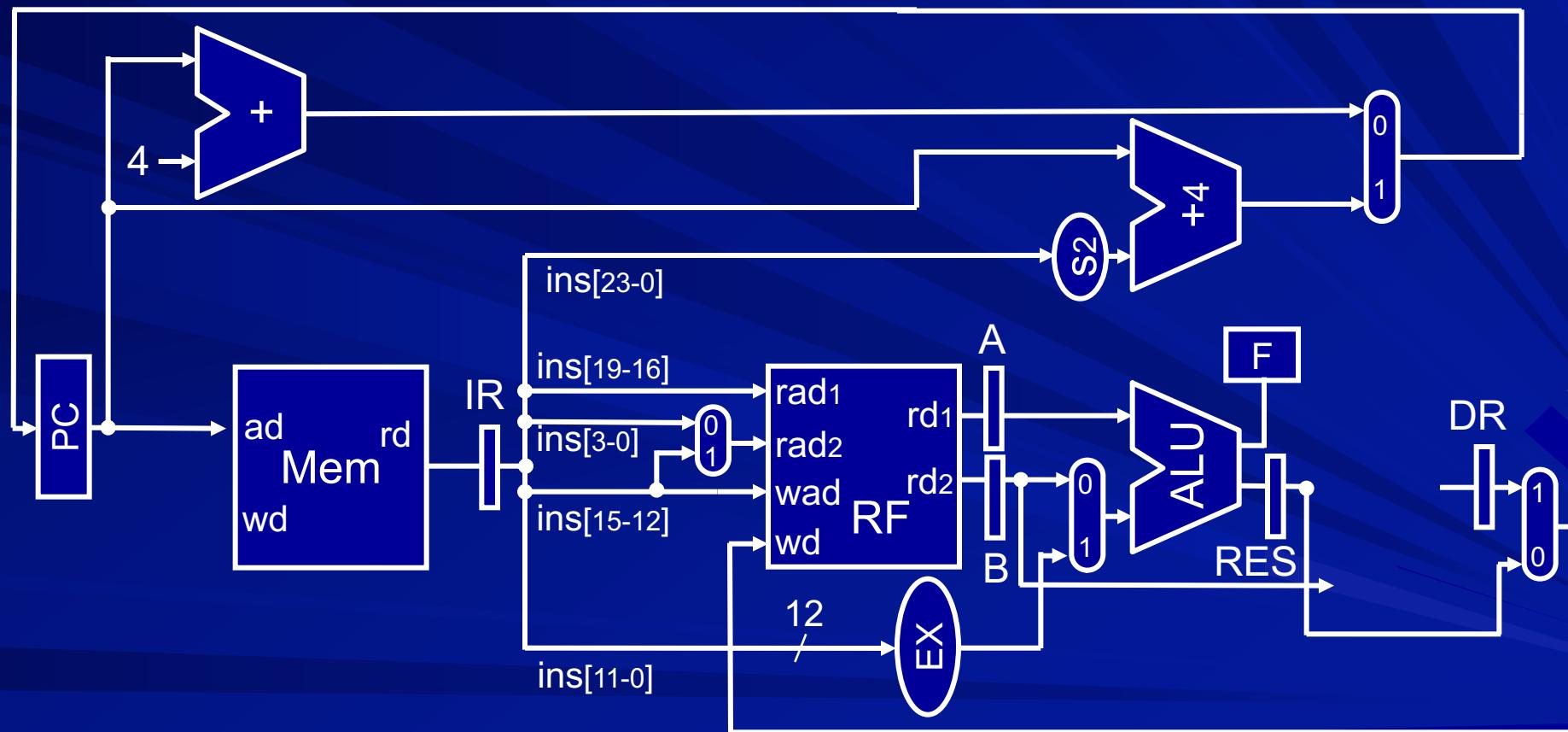
Merge IM and DM



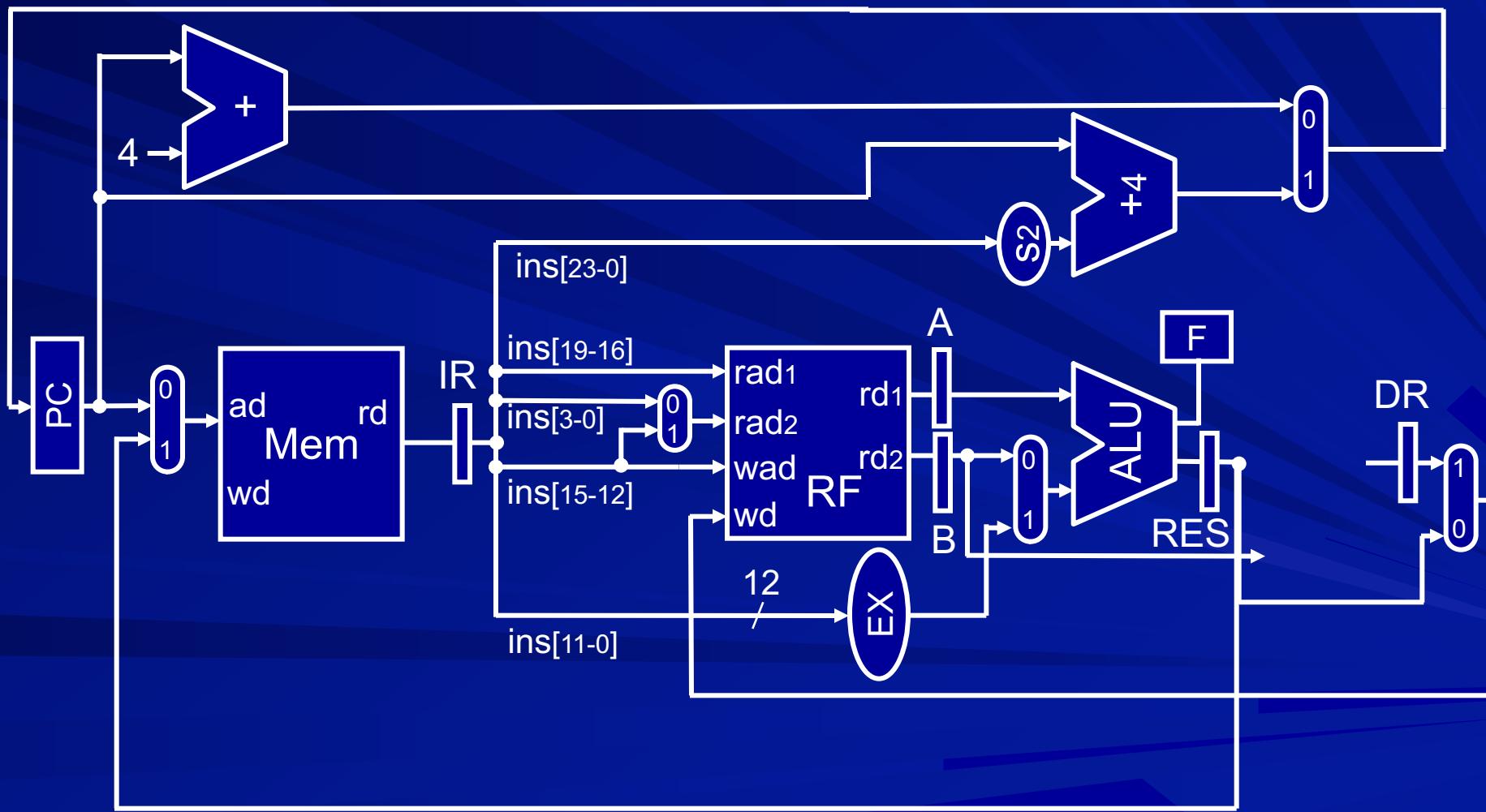
Merge IM and DM



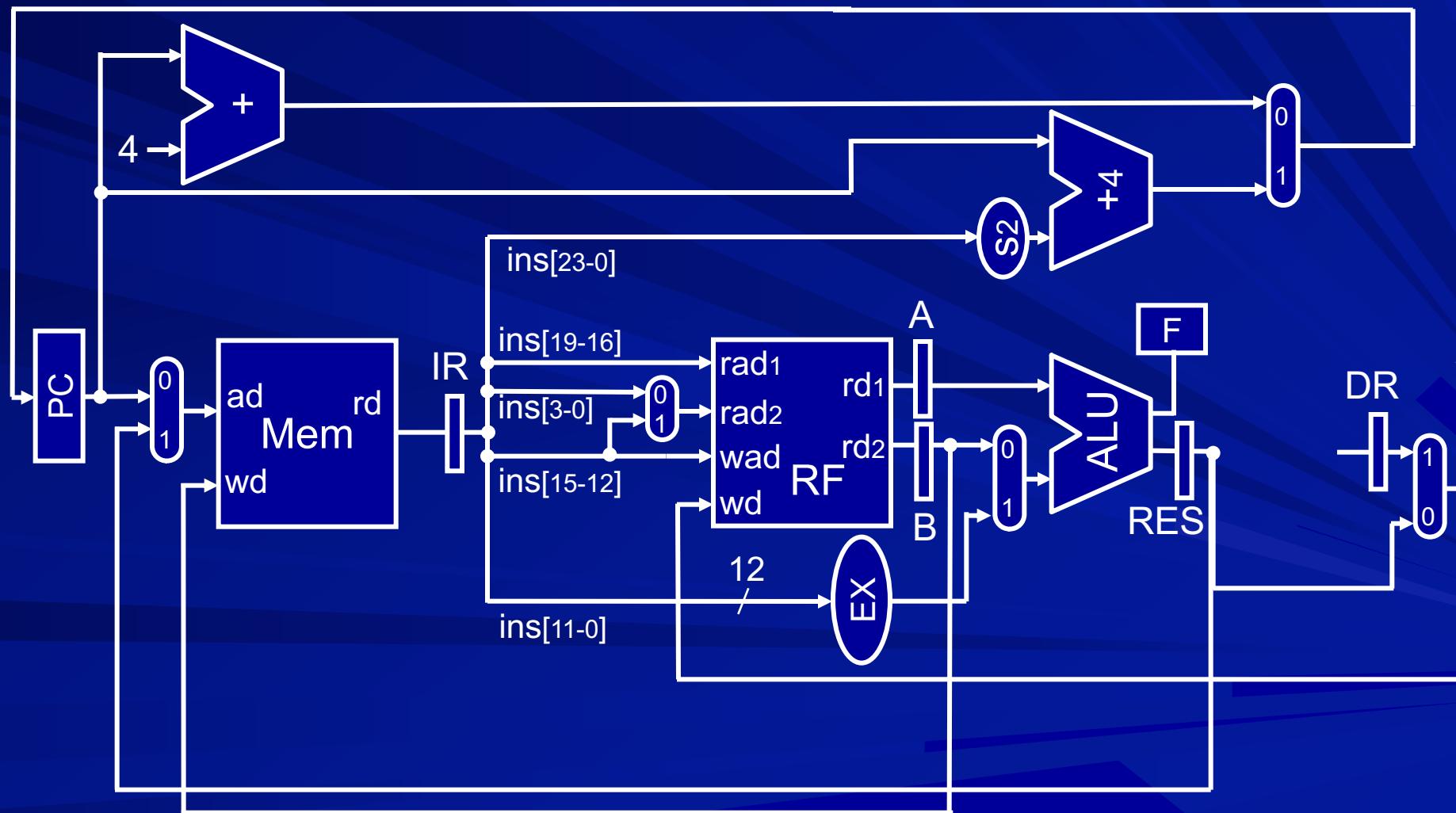
Merge IM and DM



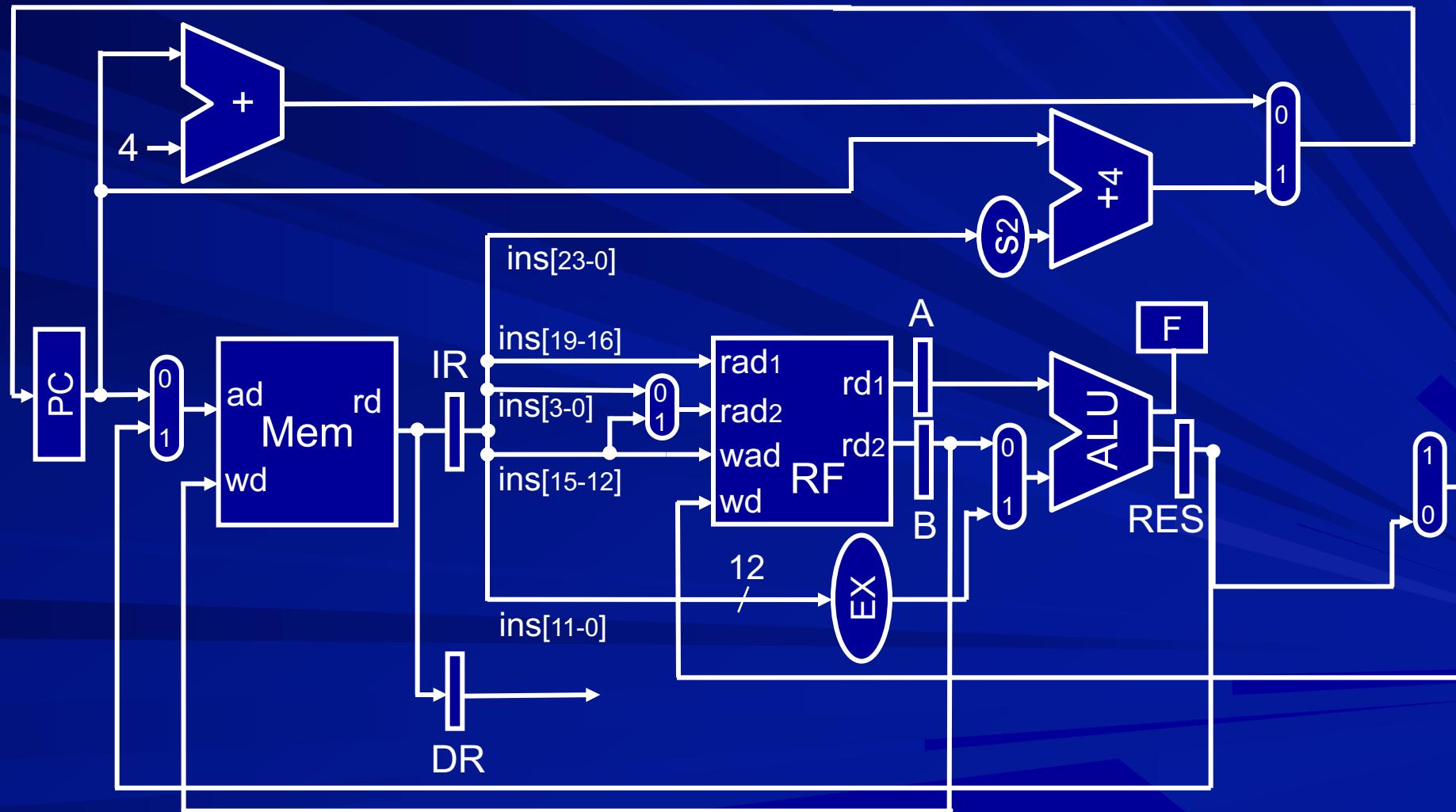
Merge IM and DM



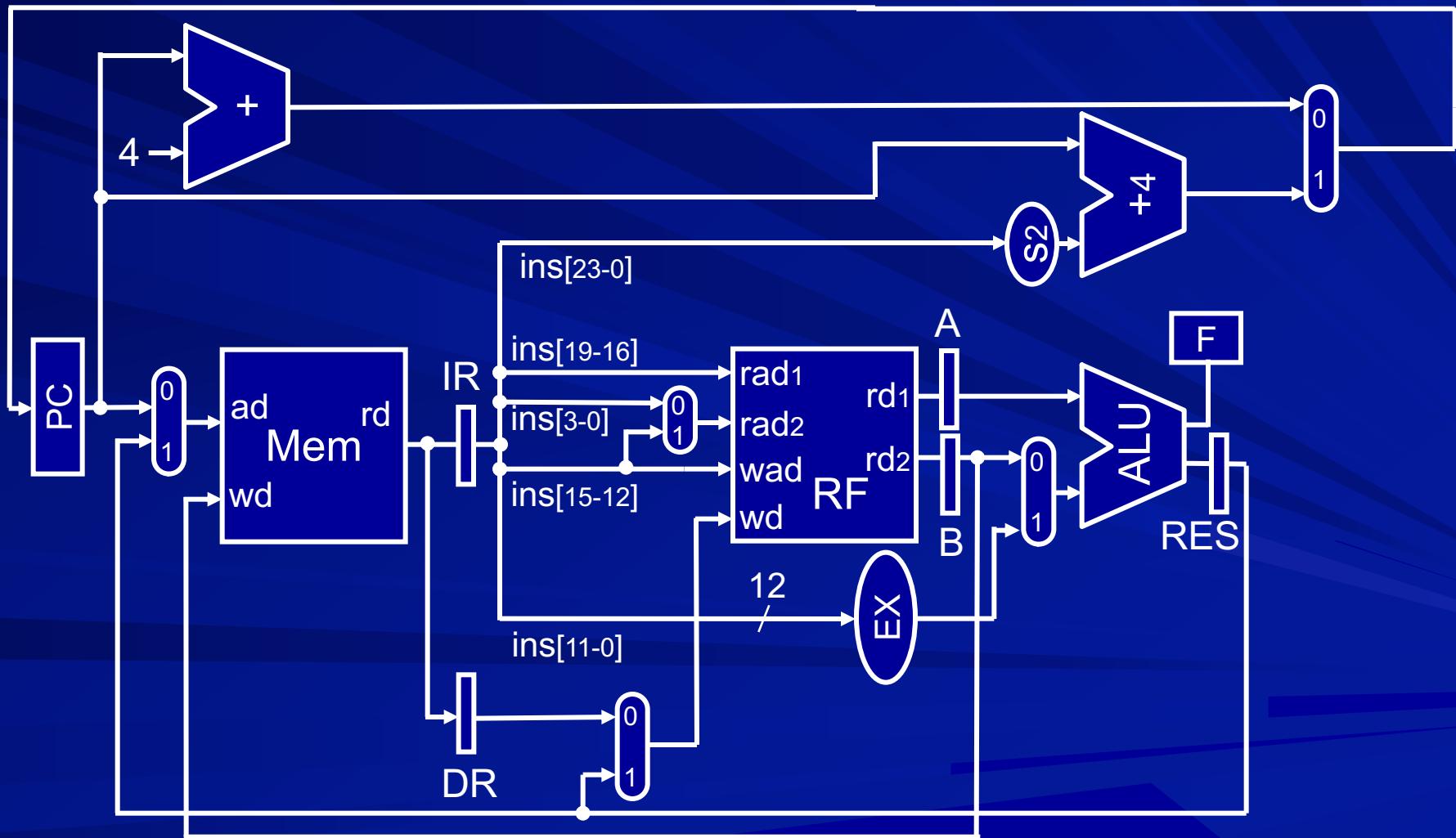
Merge IM and DM



Merge IM and DM

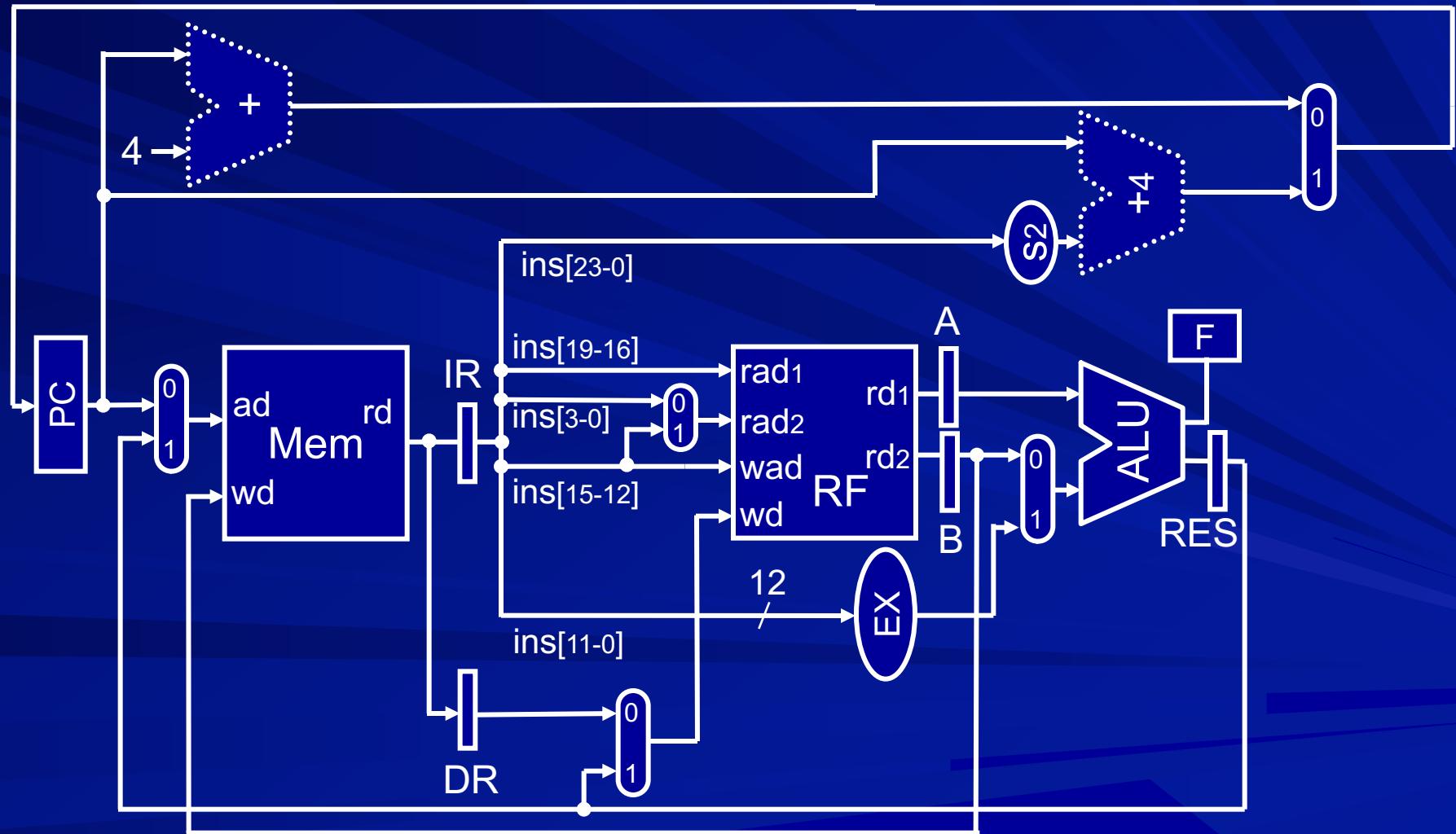


Merge IM and DM

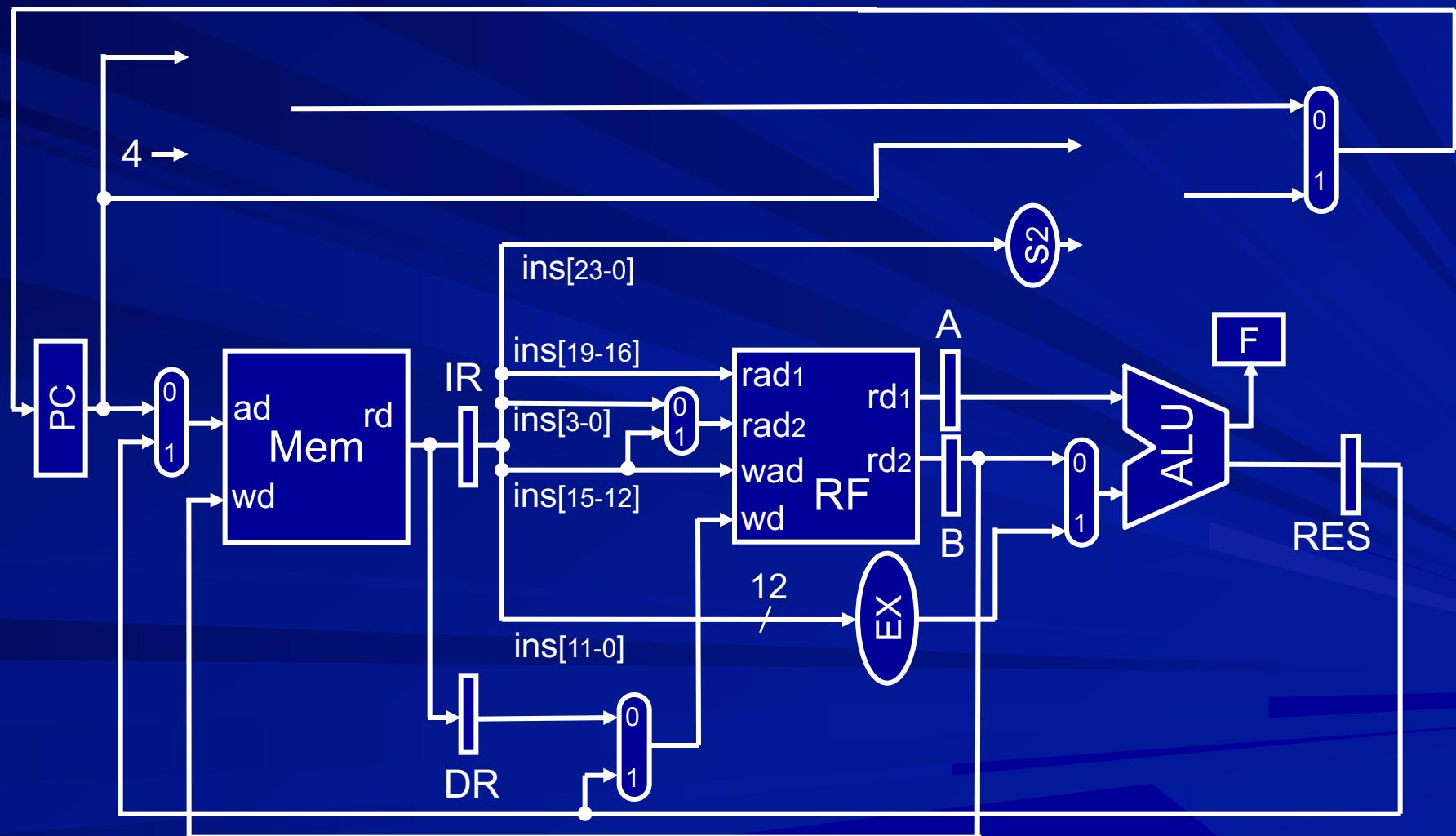


Eliminate adders

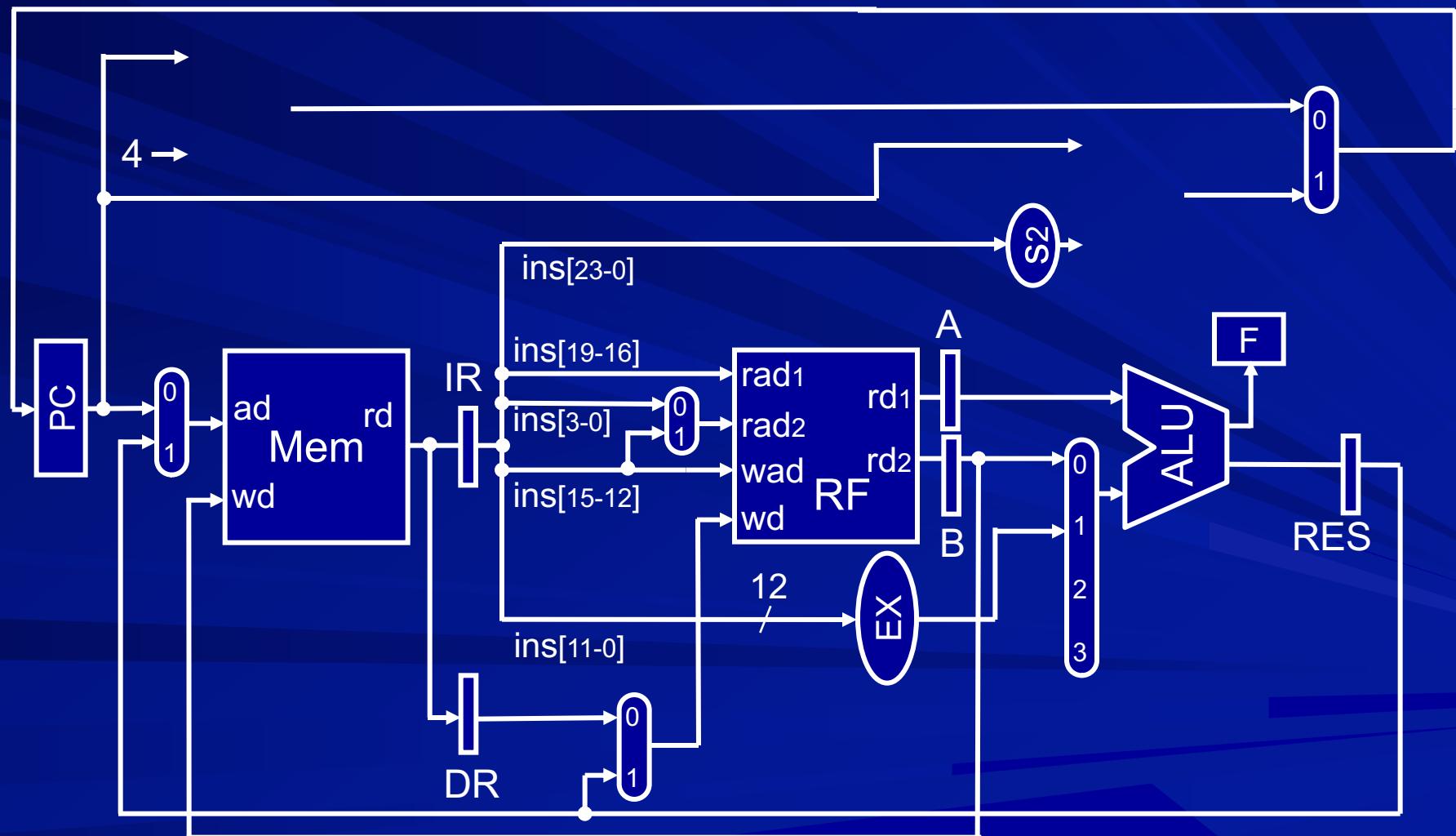
Use ALU



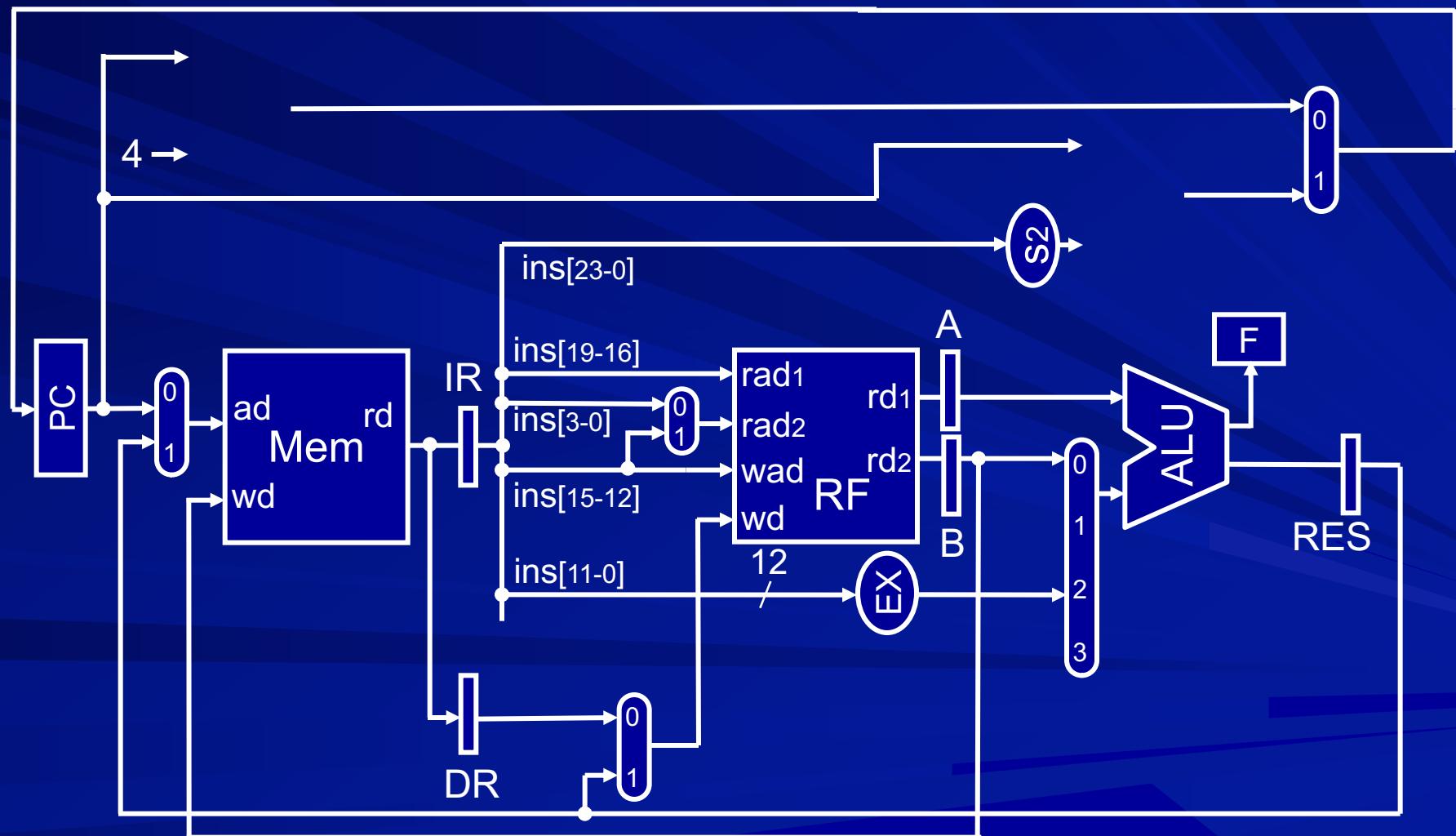
Eliminate adders



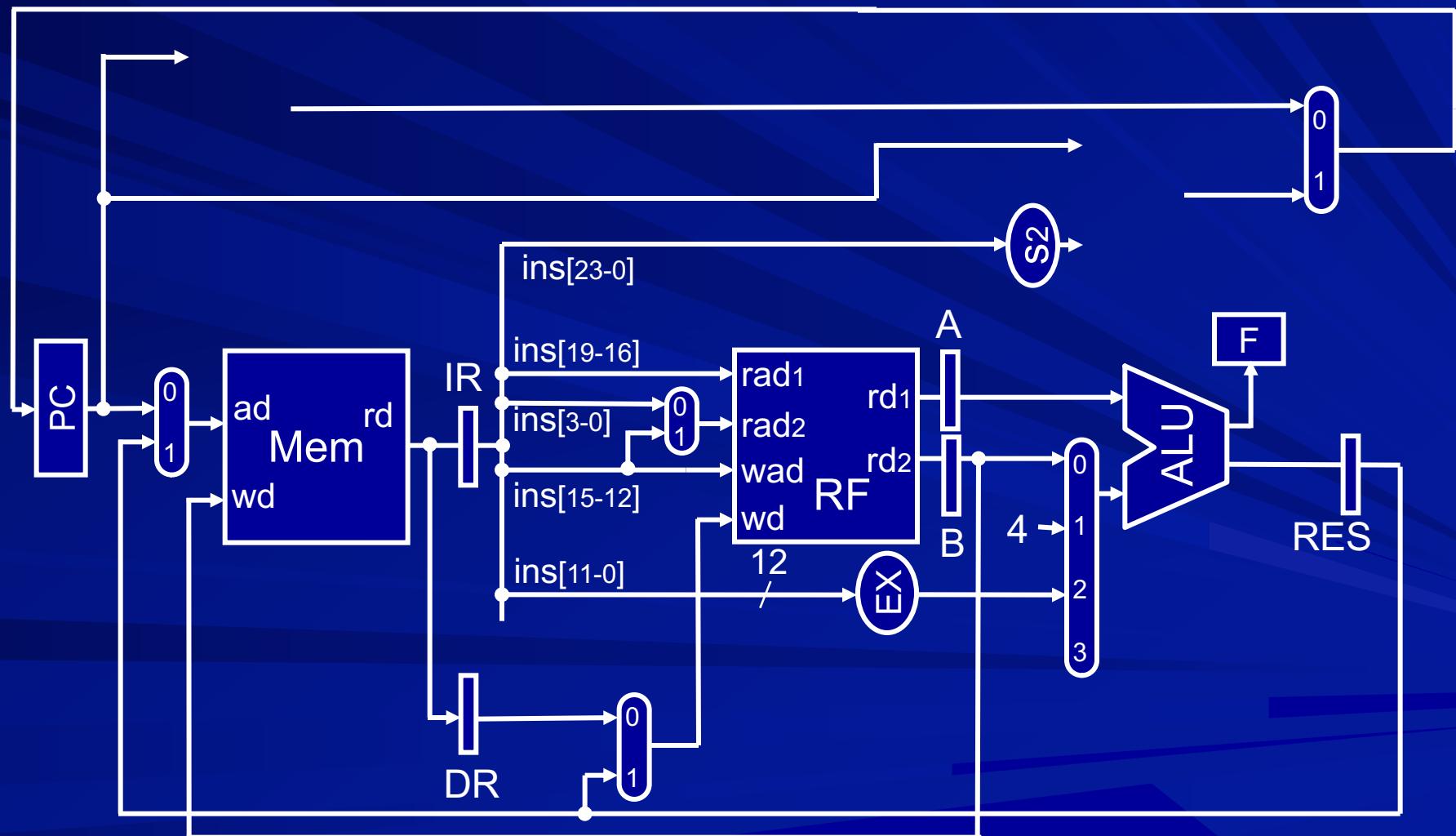
Eliminate adders



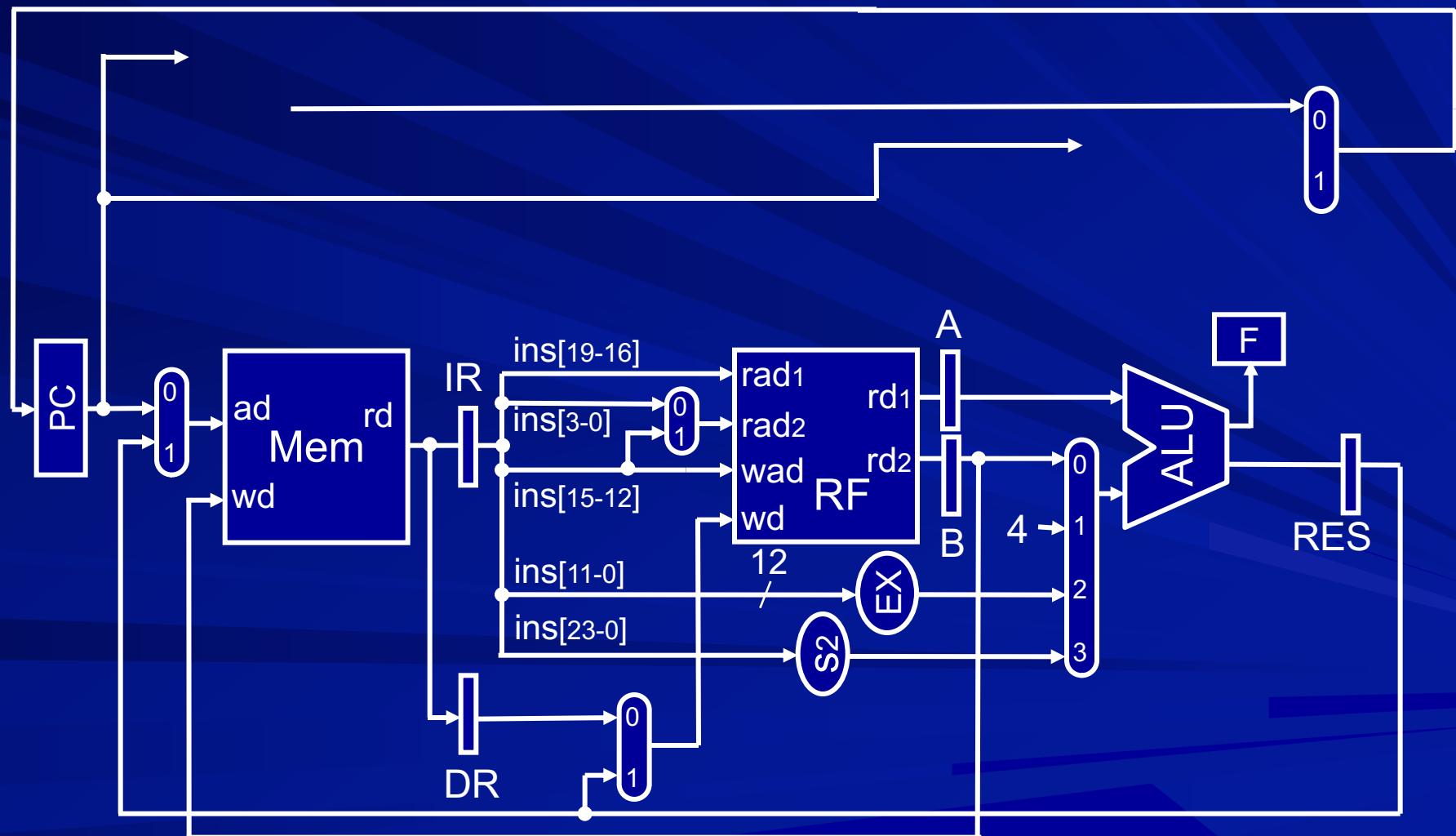
Eliminate adders



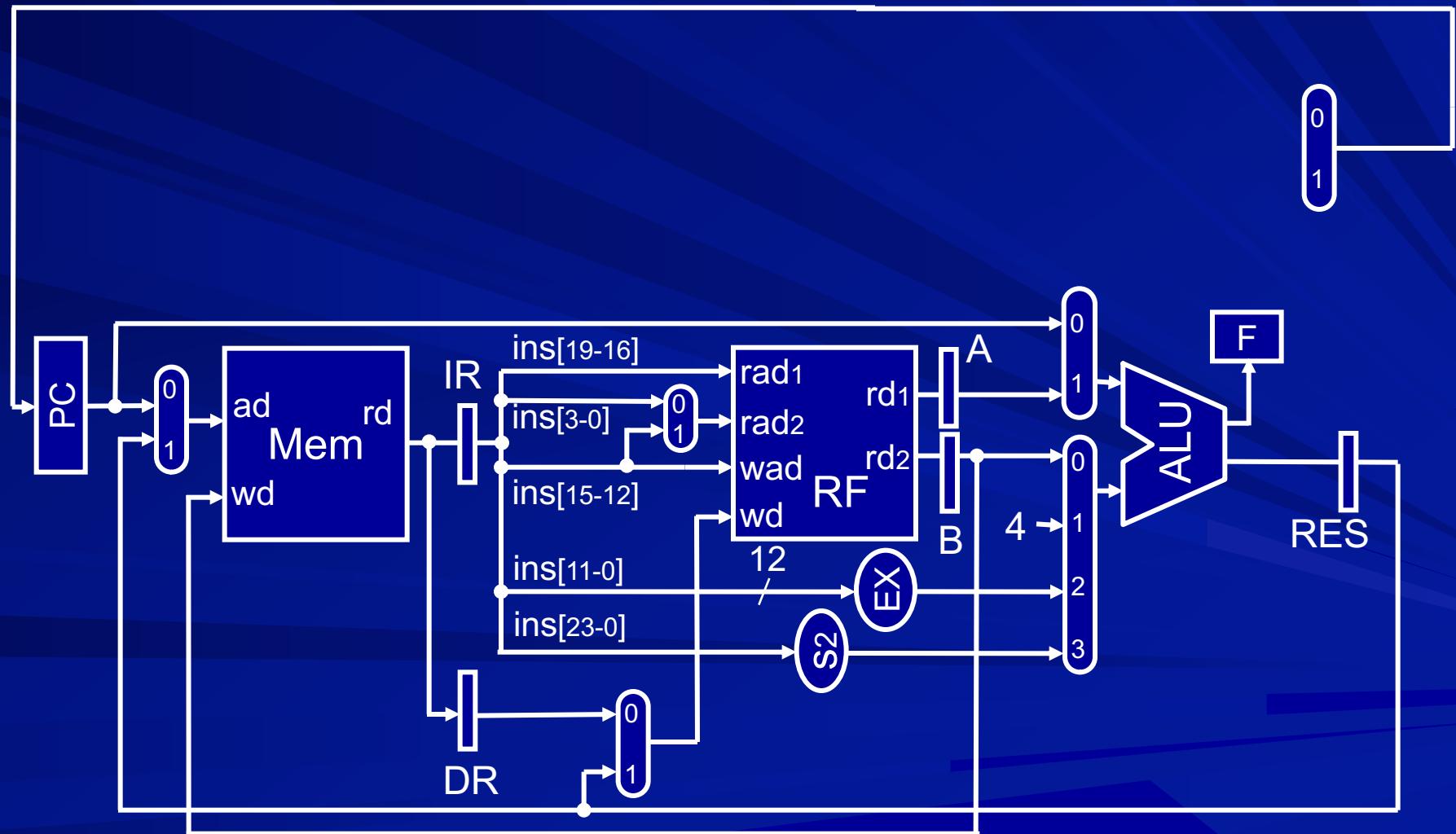
Eliminate adders



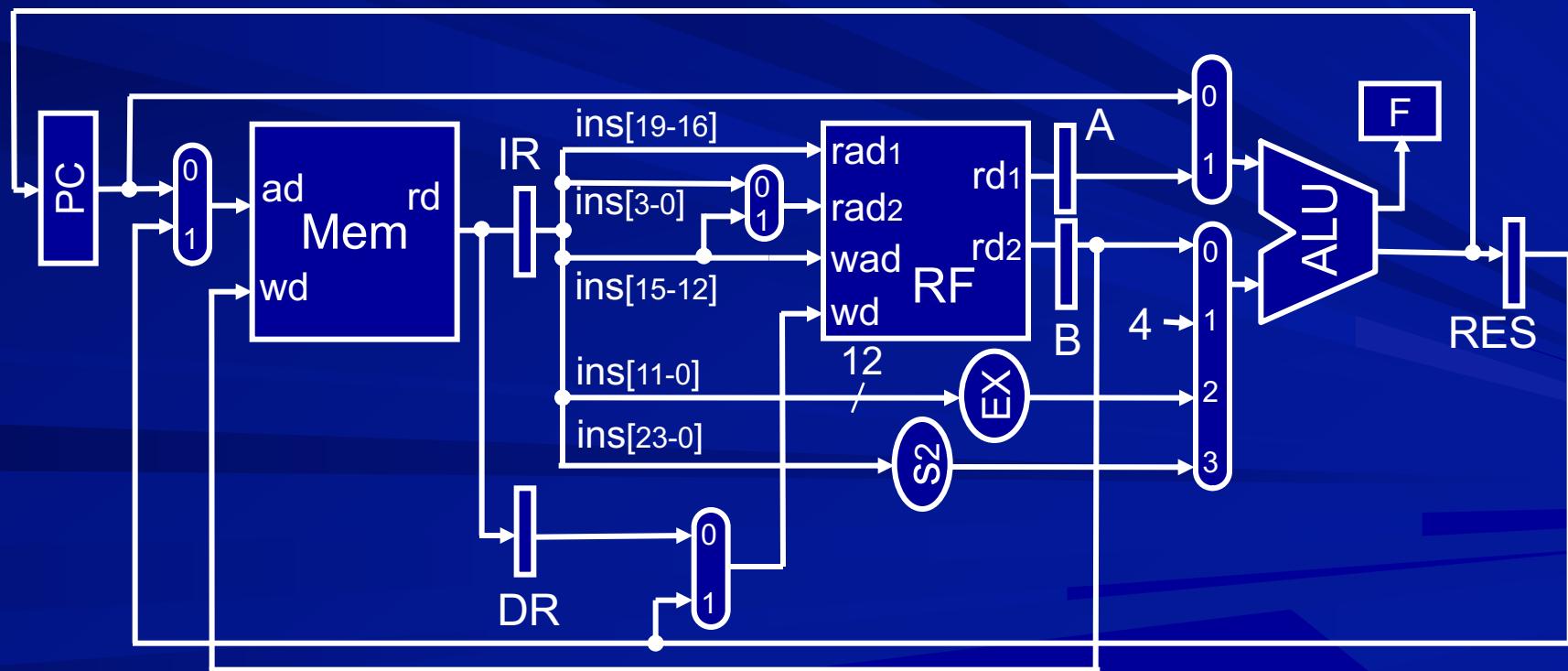
Eliminate adders



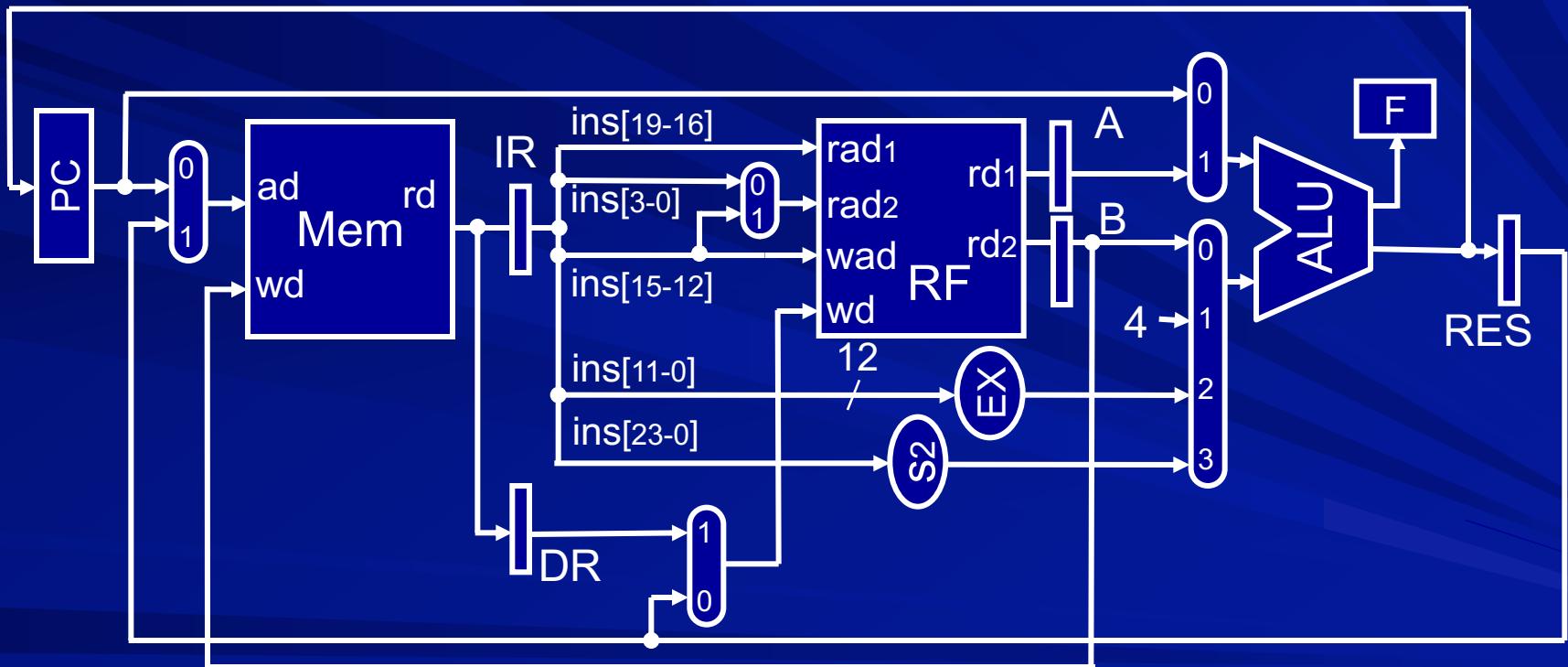
Eliminate adders



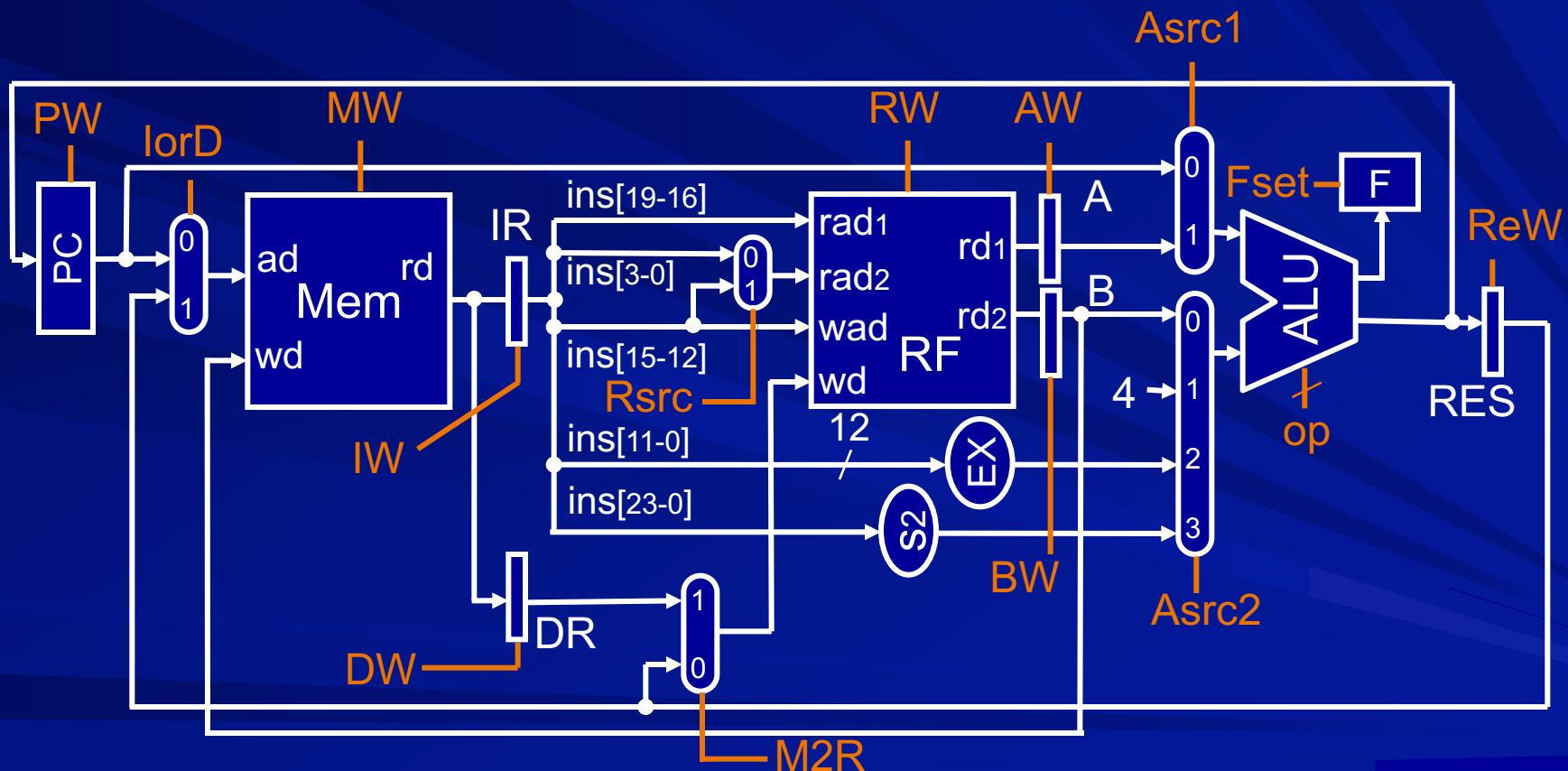
Eliminate adders



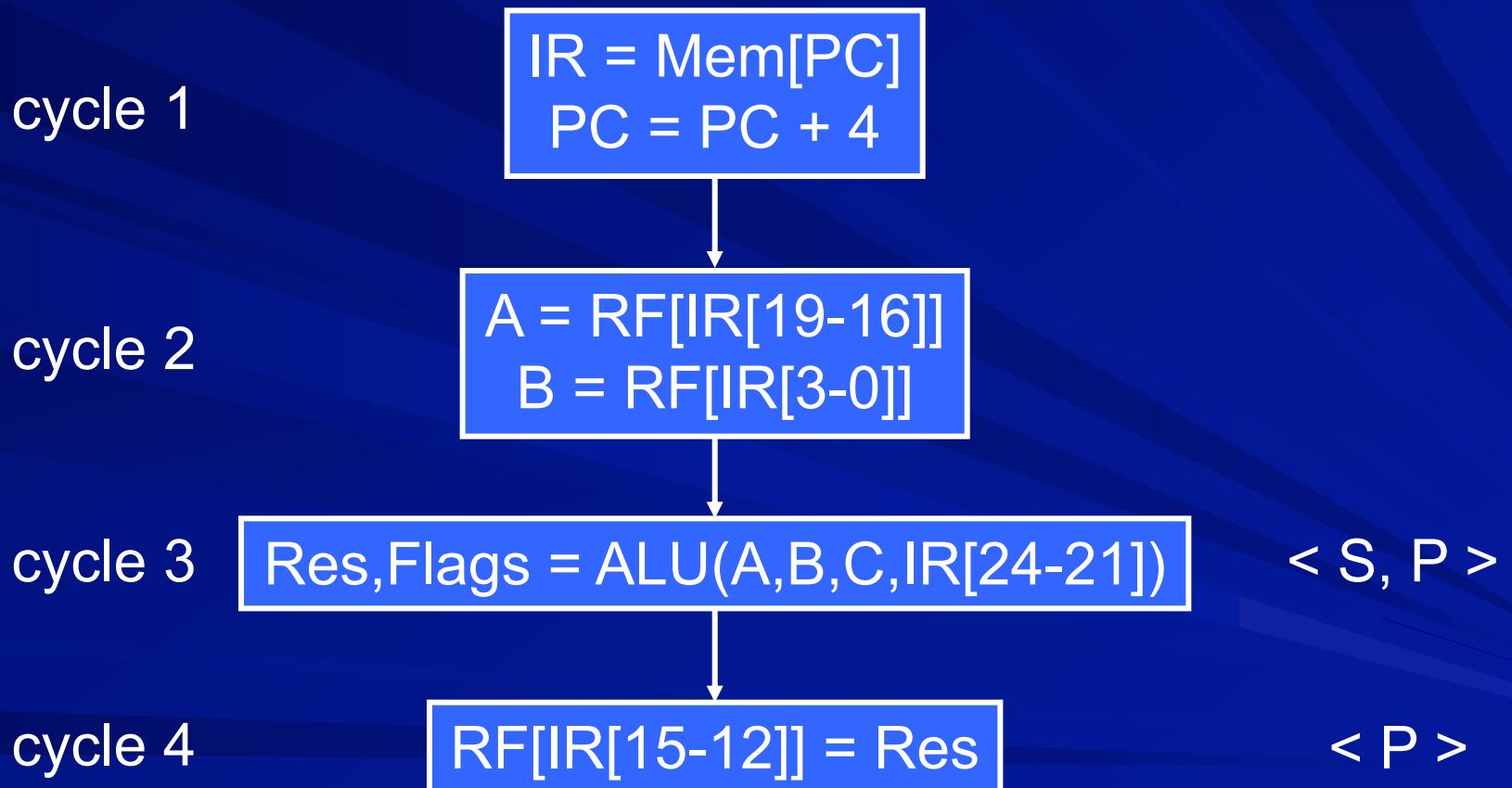
Multi-cycle Datapath



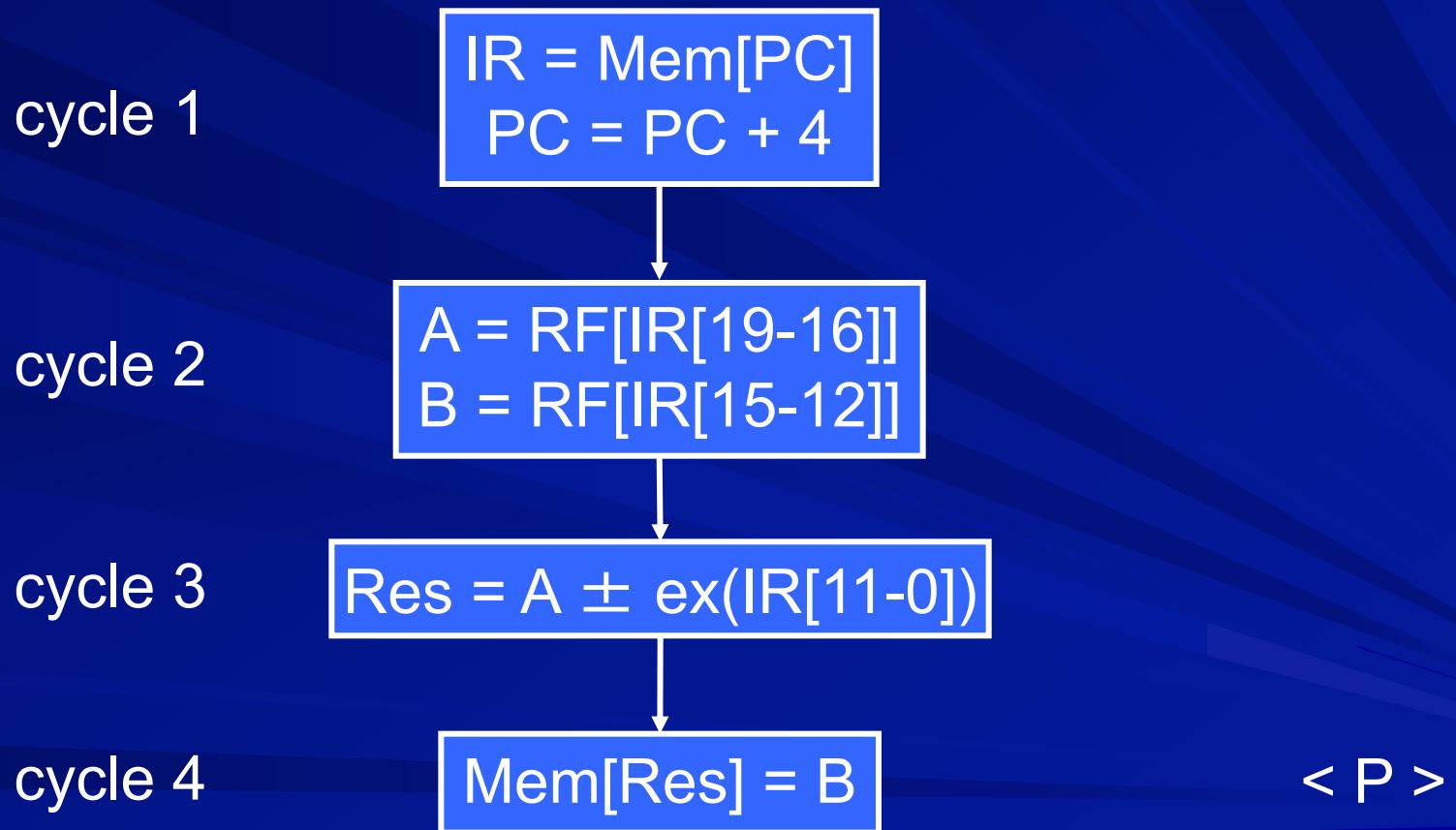
Control signals in multi-cycle datapath



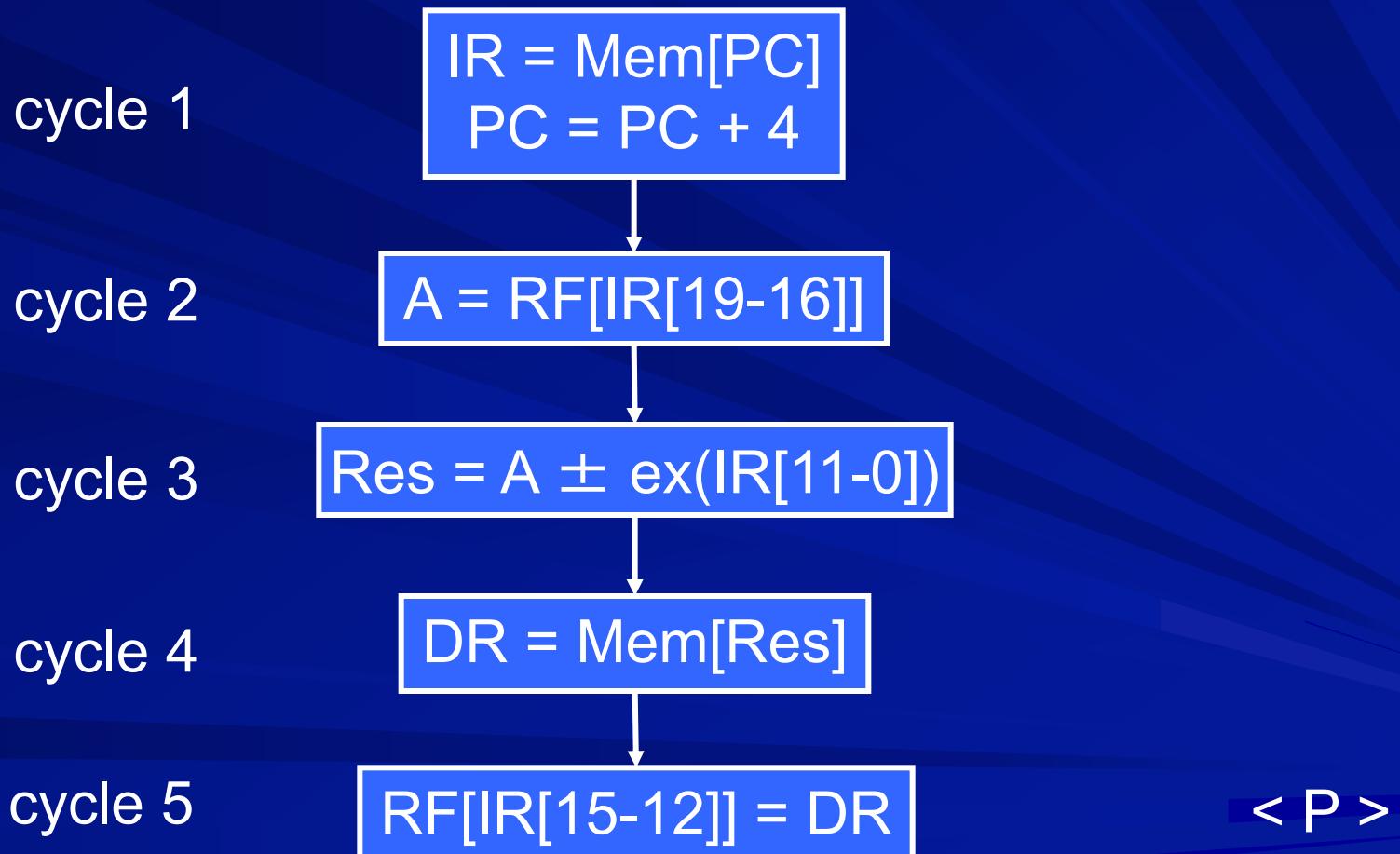
Break Instruction Execution into Cycles: DP instructions



Break Instruction Execution into Cycles: str instruction



Break Instruction Execution into Cycles: ldr instruction



Break Instruction Execution into Cycles: b instruction

cycle 1

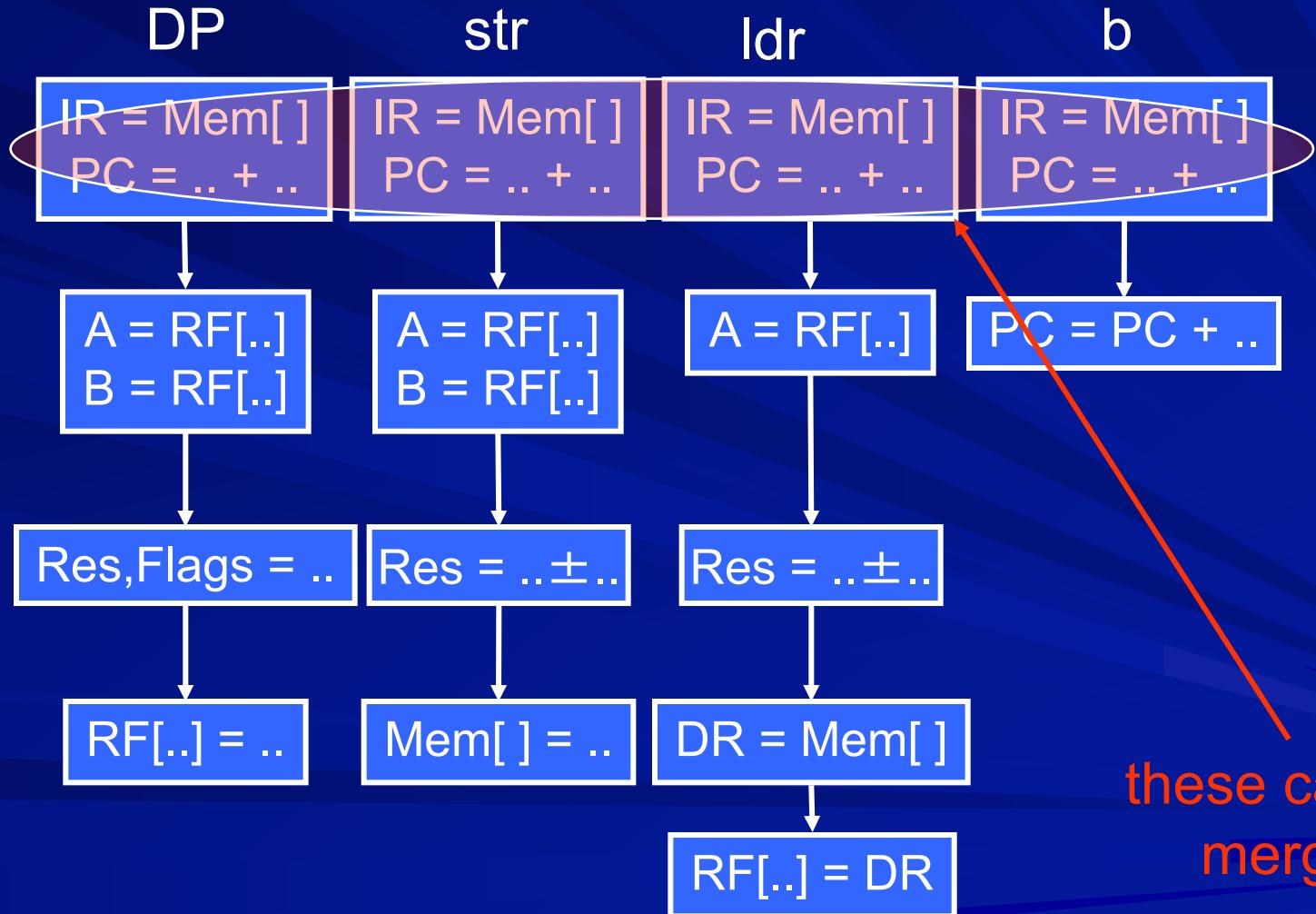
```
IR = Mem[PC]  
PC = PC + 4
```

cycle 2

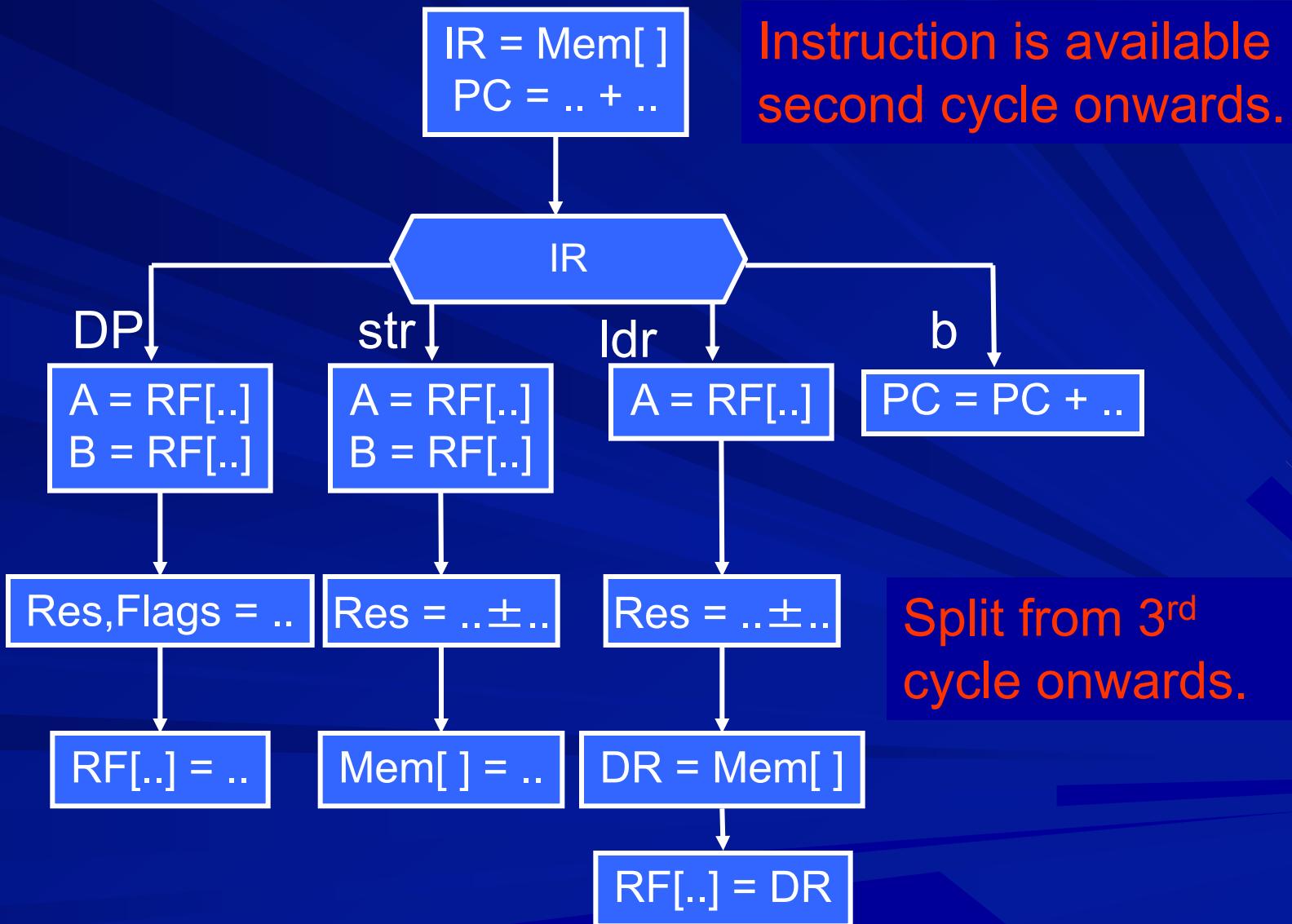
```
PC = PC + S2(IR[23-0]) + 4
```

< P >

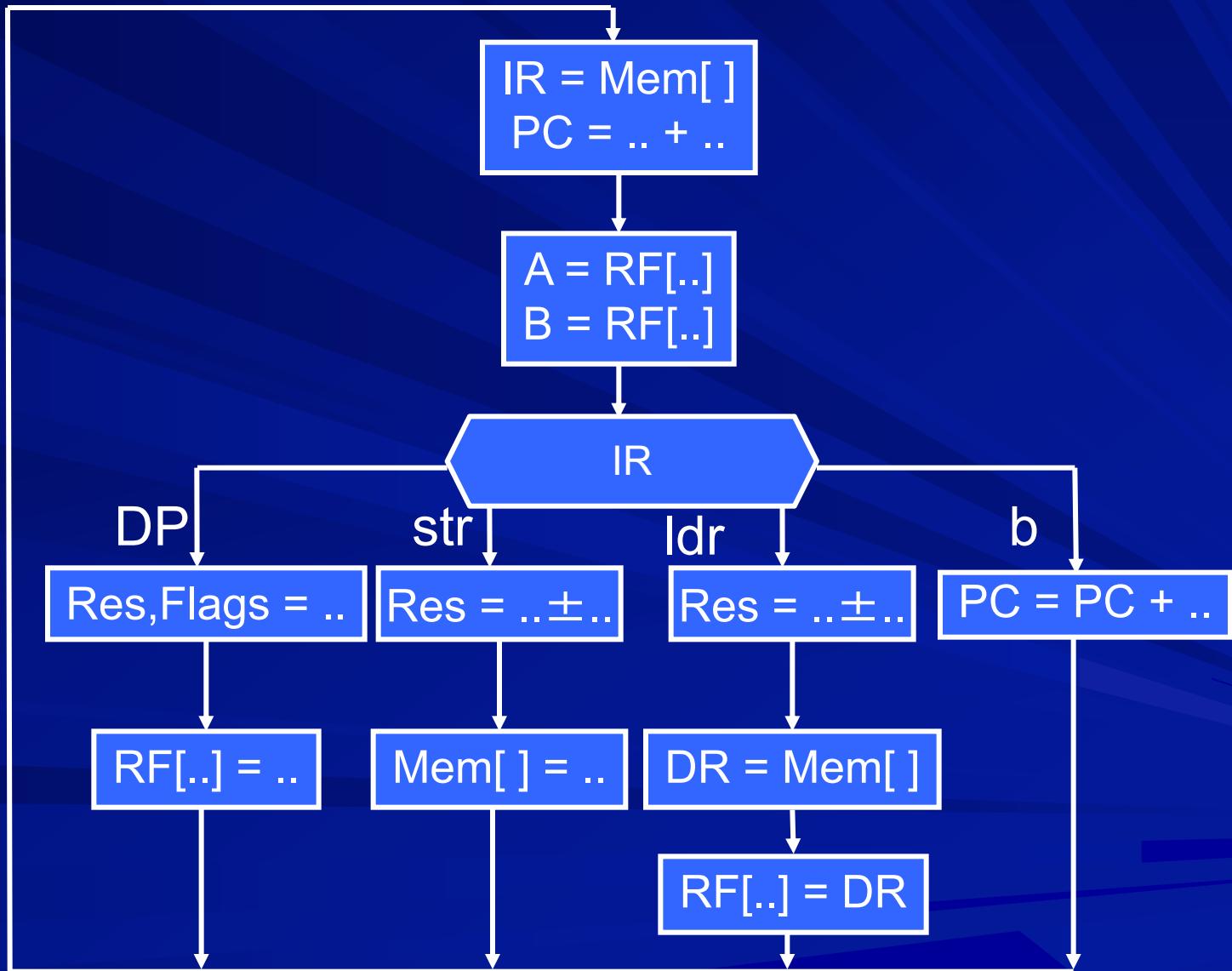
Put cycle sequences together



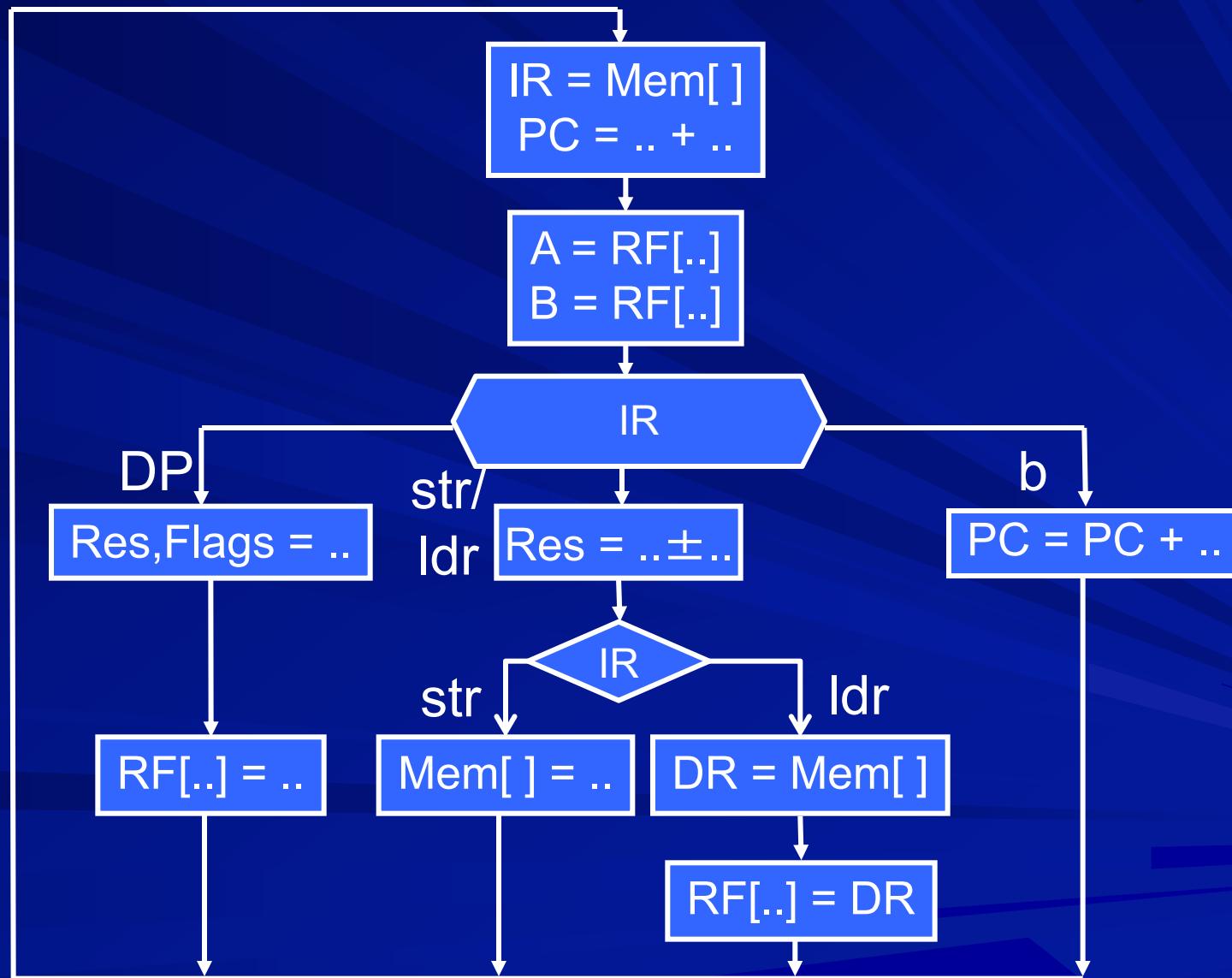
After merging fetch cycle



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
```

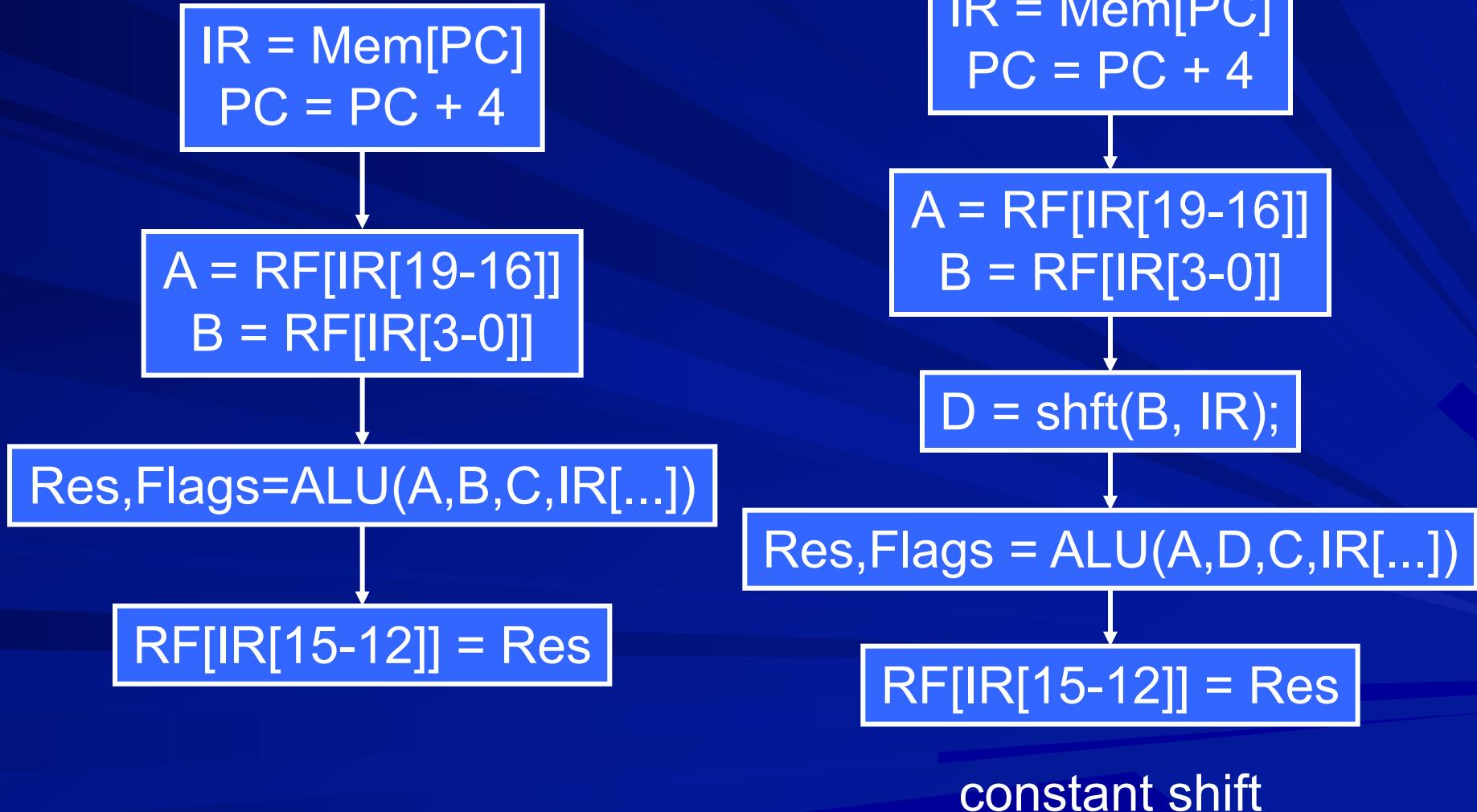
cycle 2

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

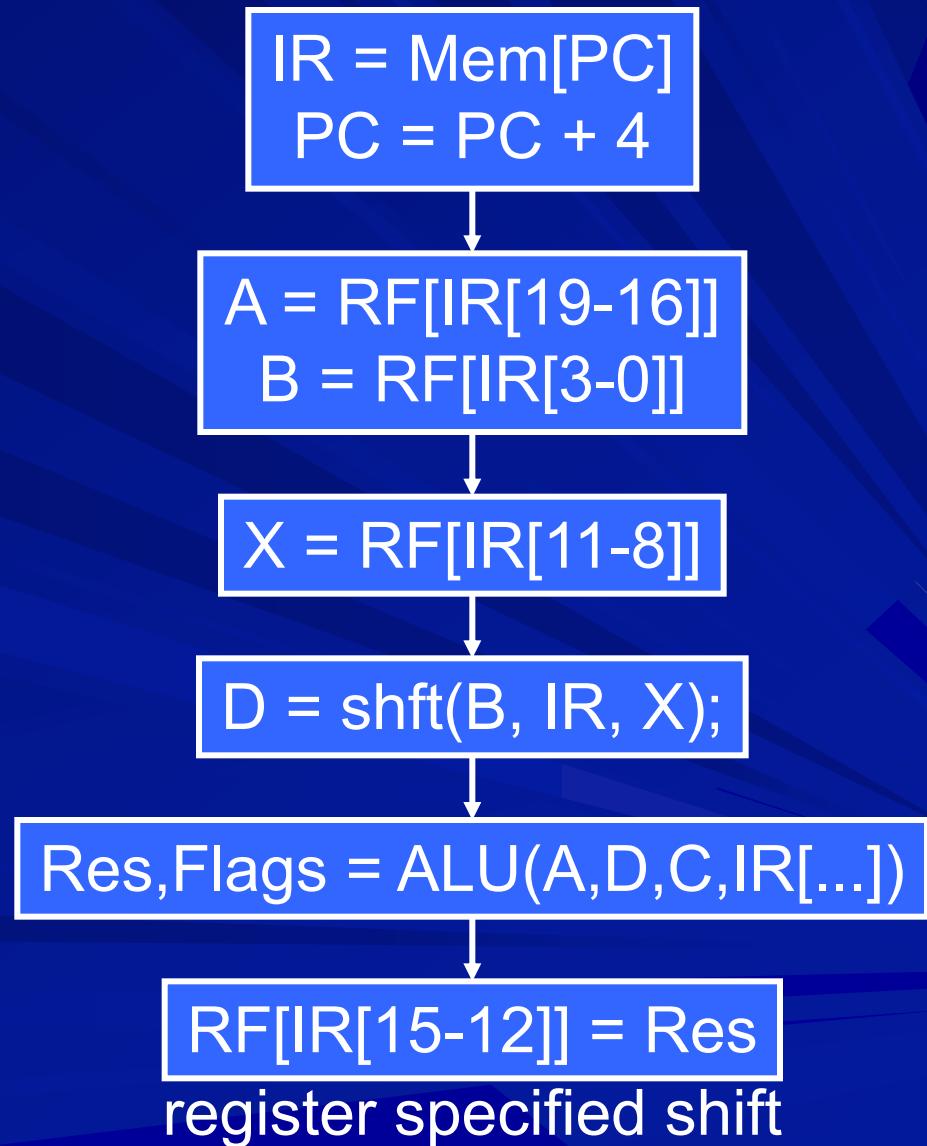
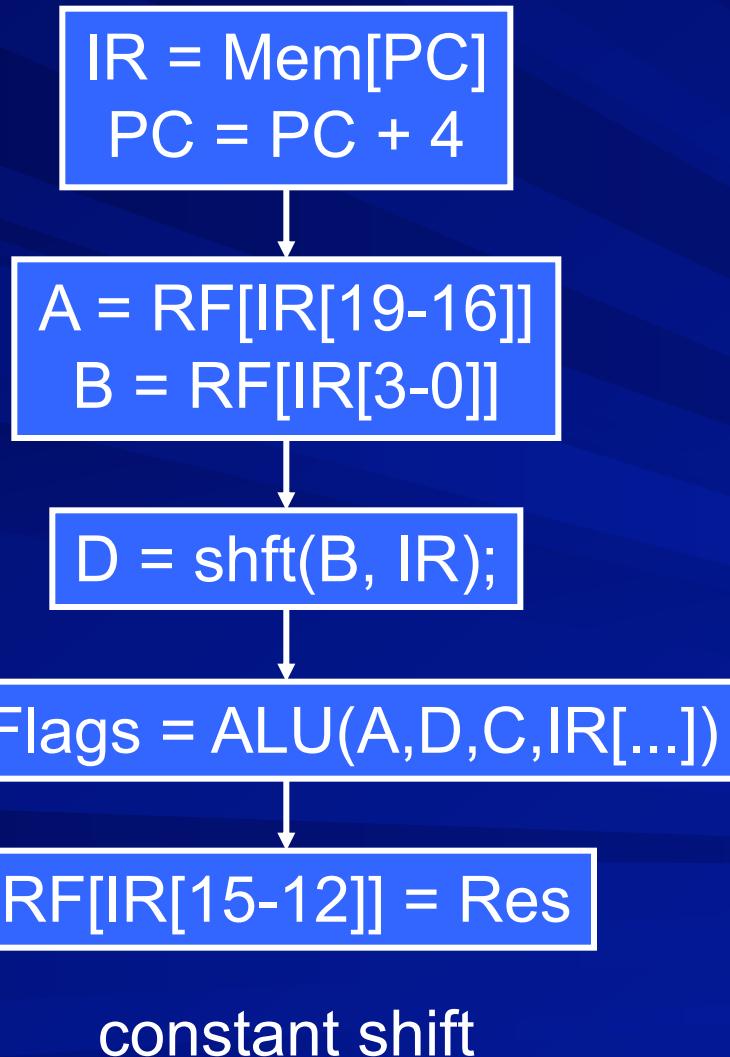
cycle 3

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

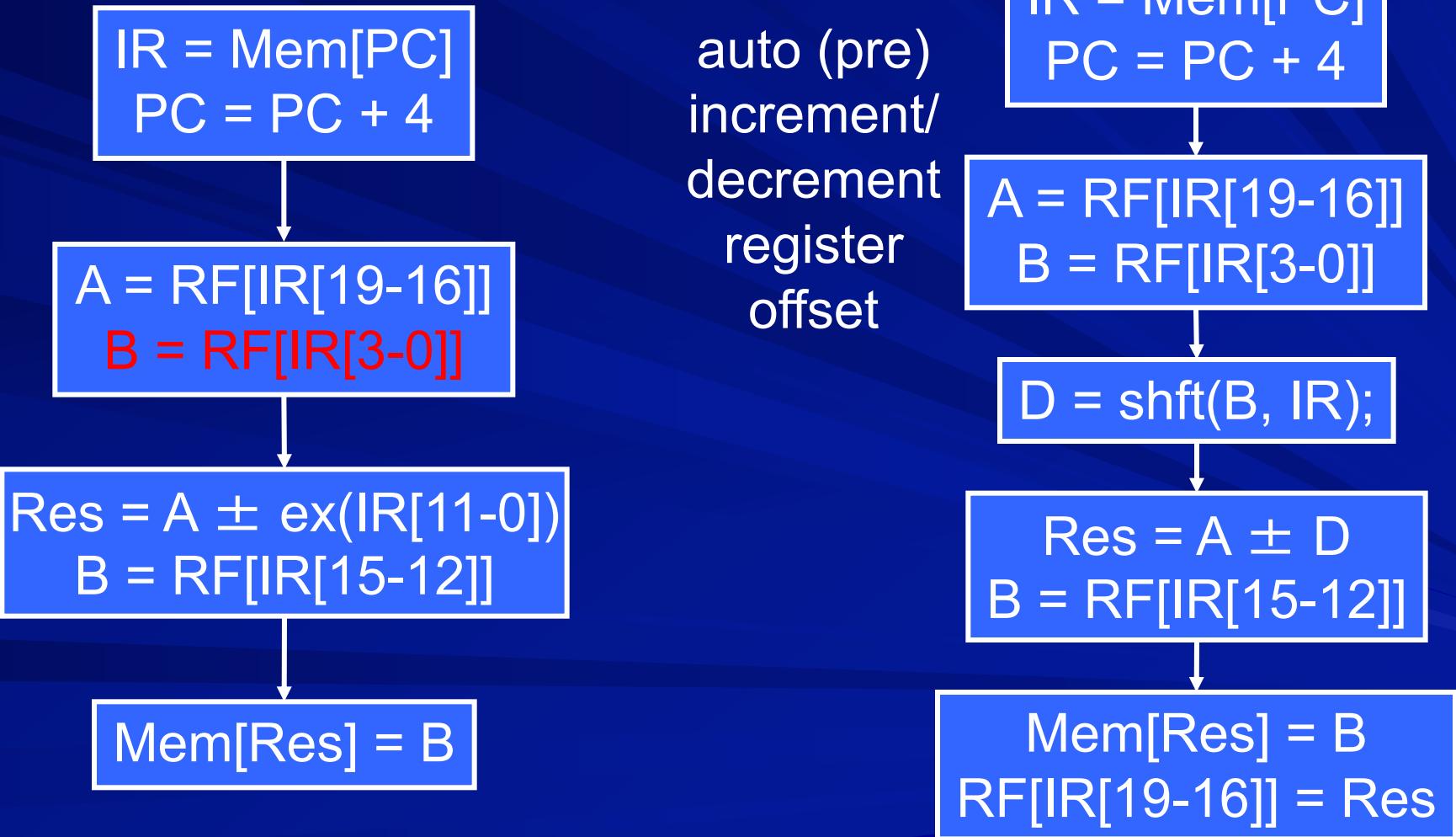
Other forms of DP instructions



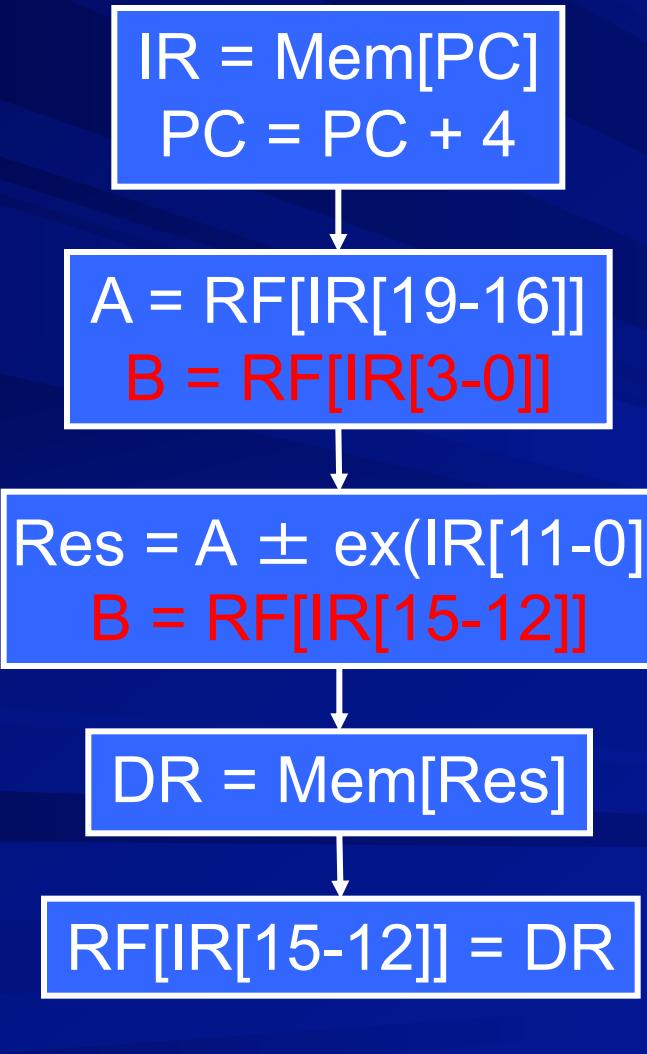
Other forms of DP instructions



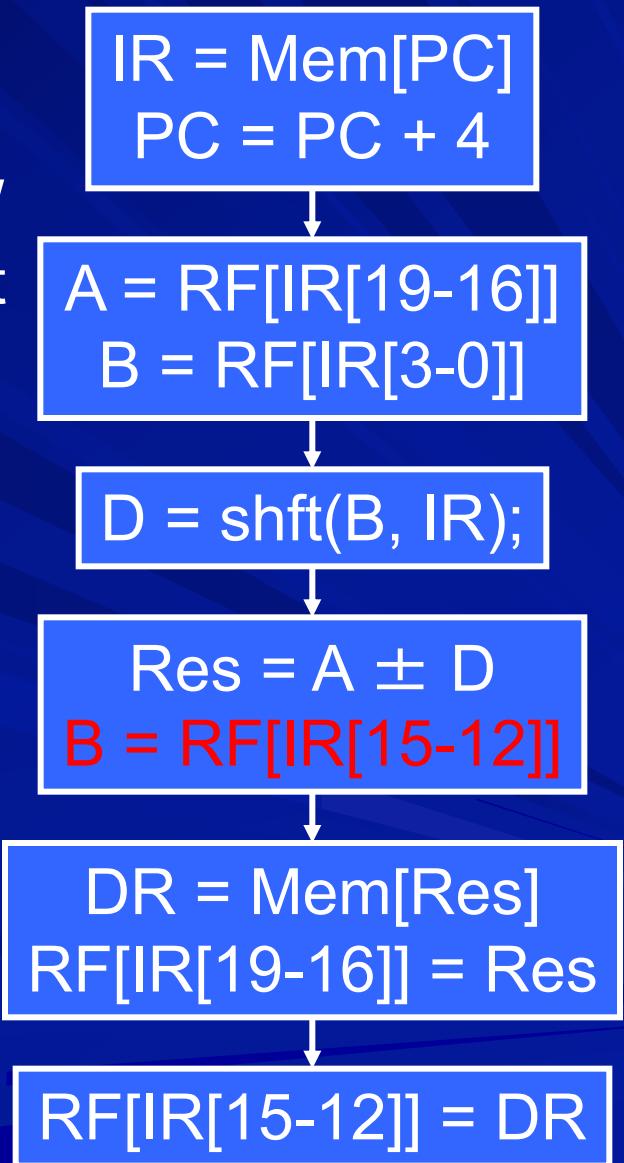
Other forms of str instruction



Other forms of ldr instruction



auto (pre)
increment/
decrement
register
offset



Thanks

COL216

Computer Architecture

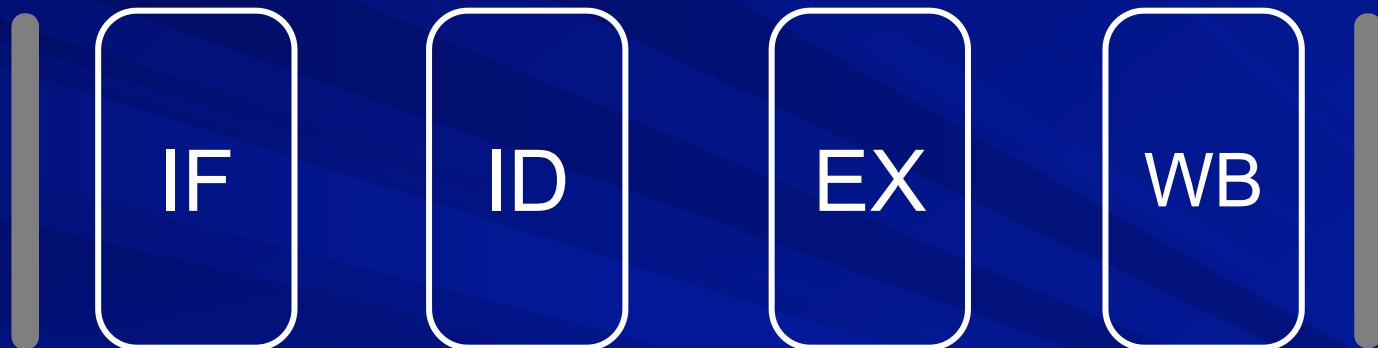
Processor design -
Pipelining

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

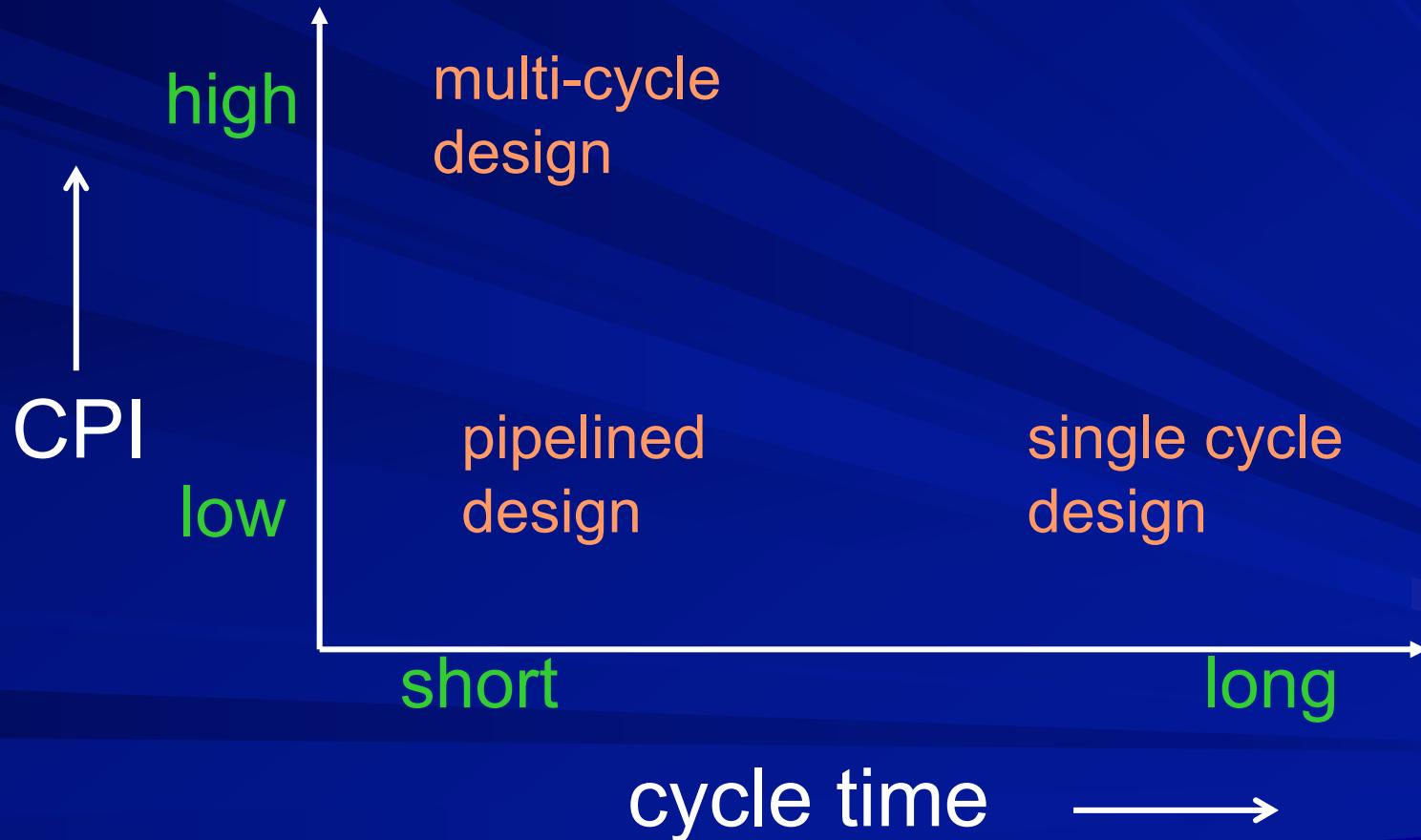
■ Multi-cycle design

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

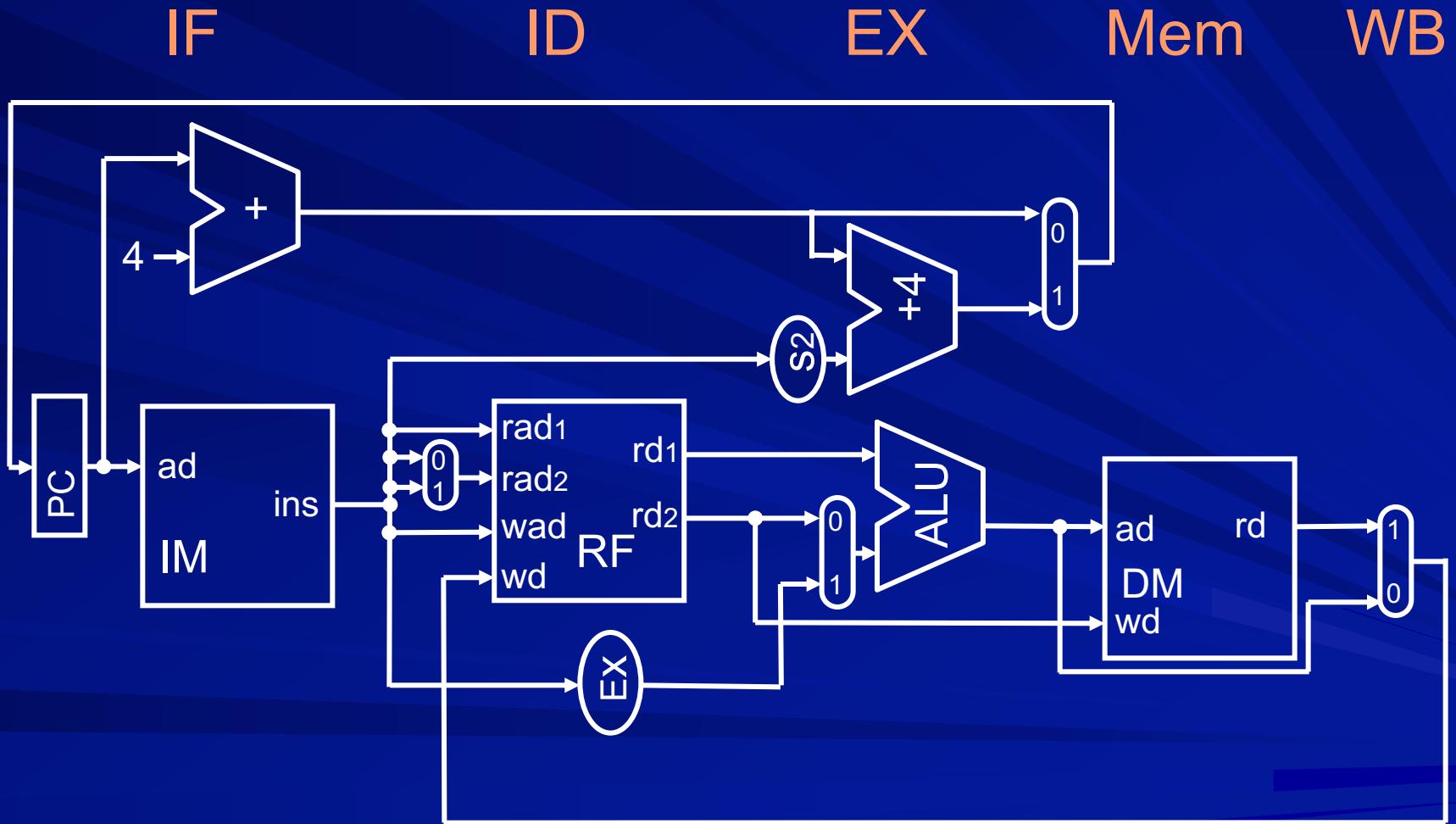
■ Pipelined design

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

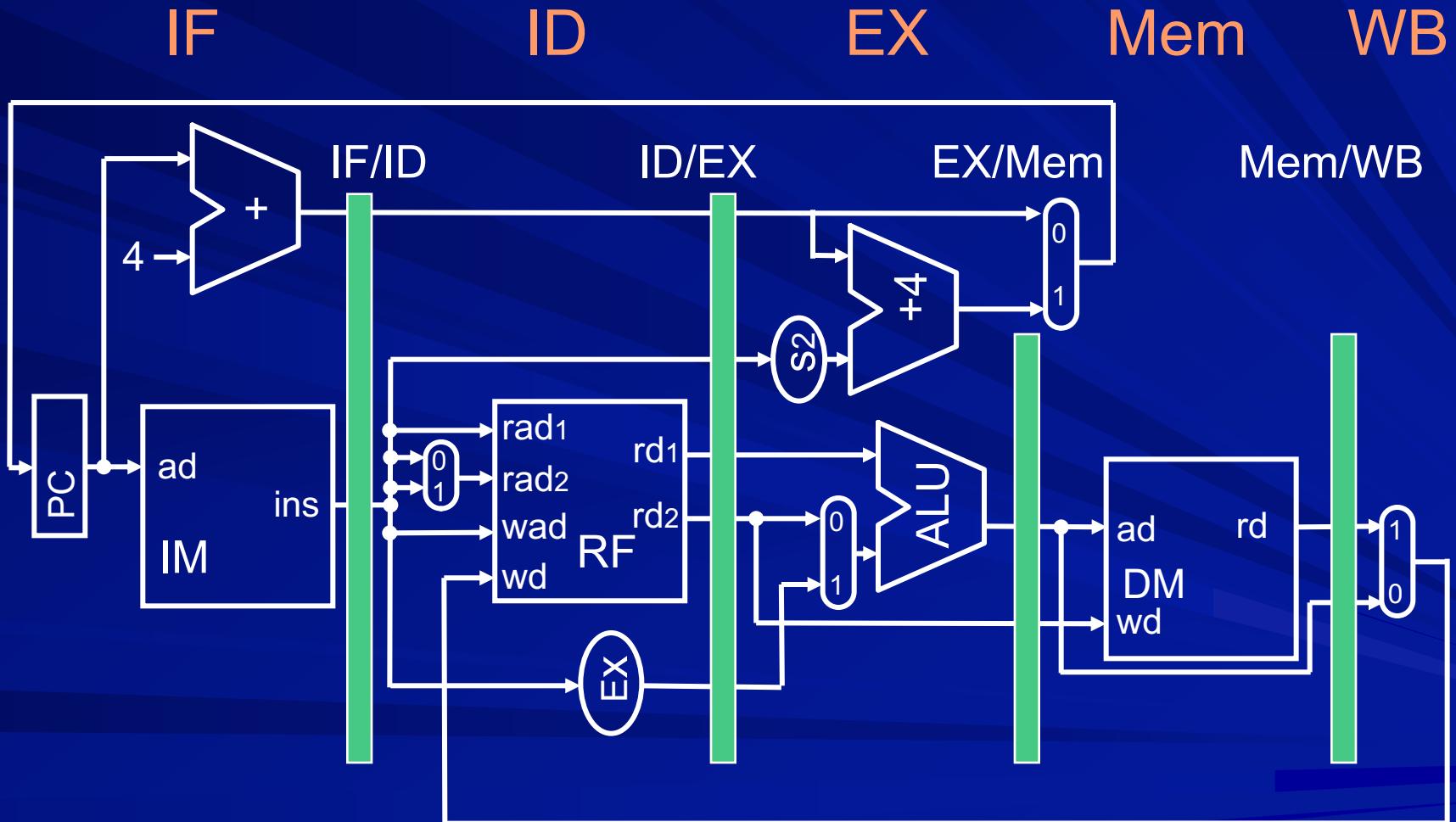
Performance of different designs



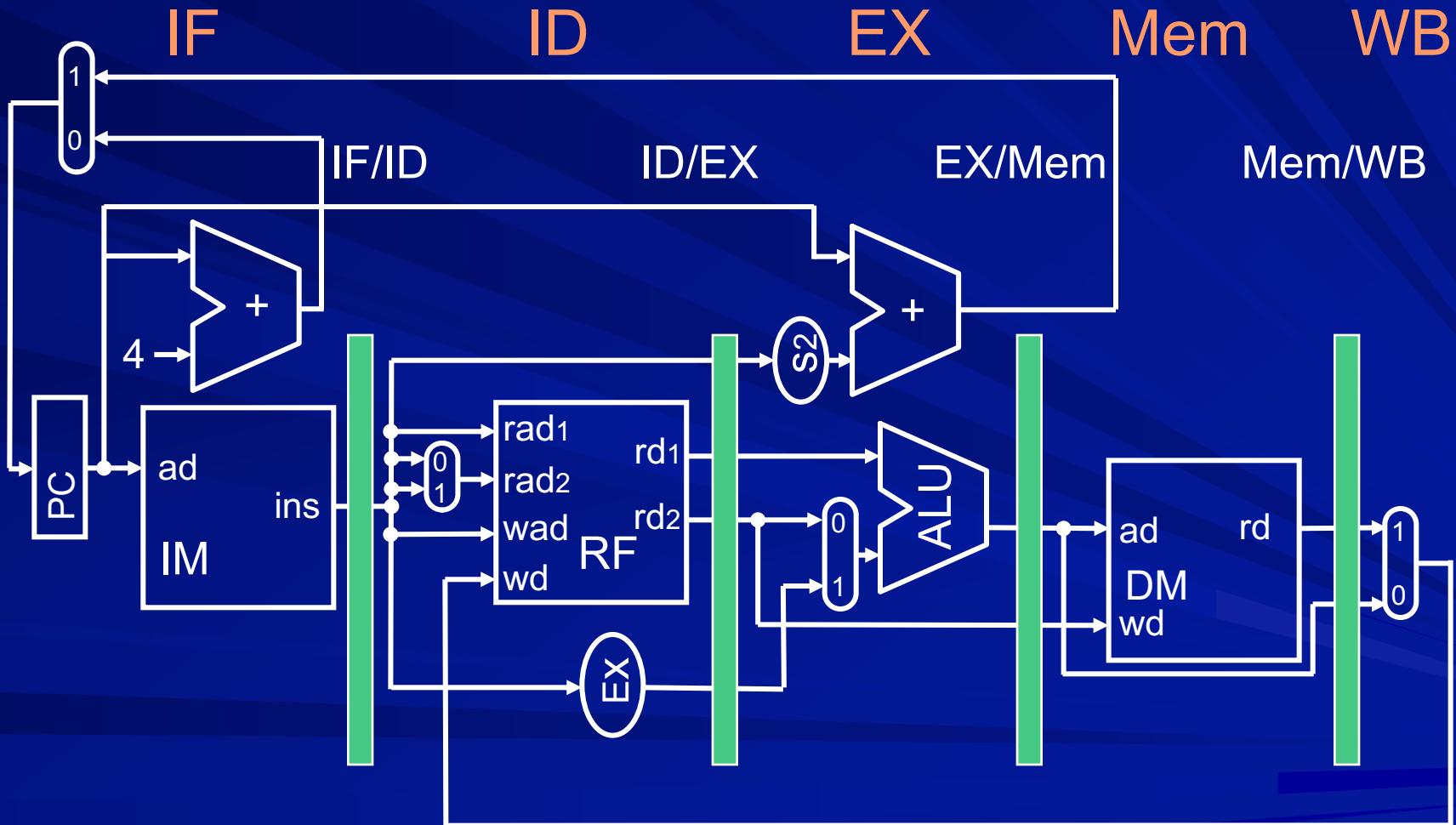
Single cycle datapath



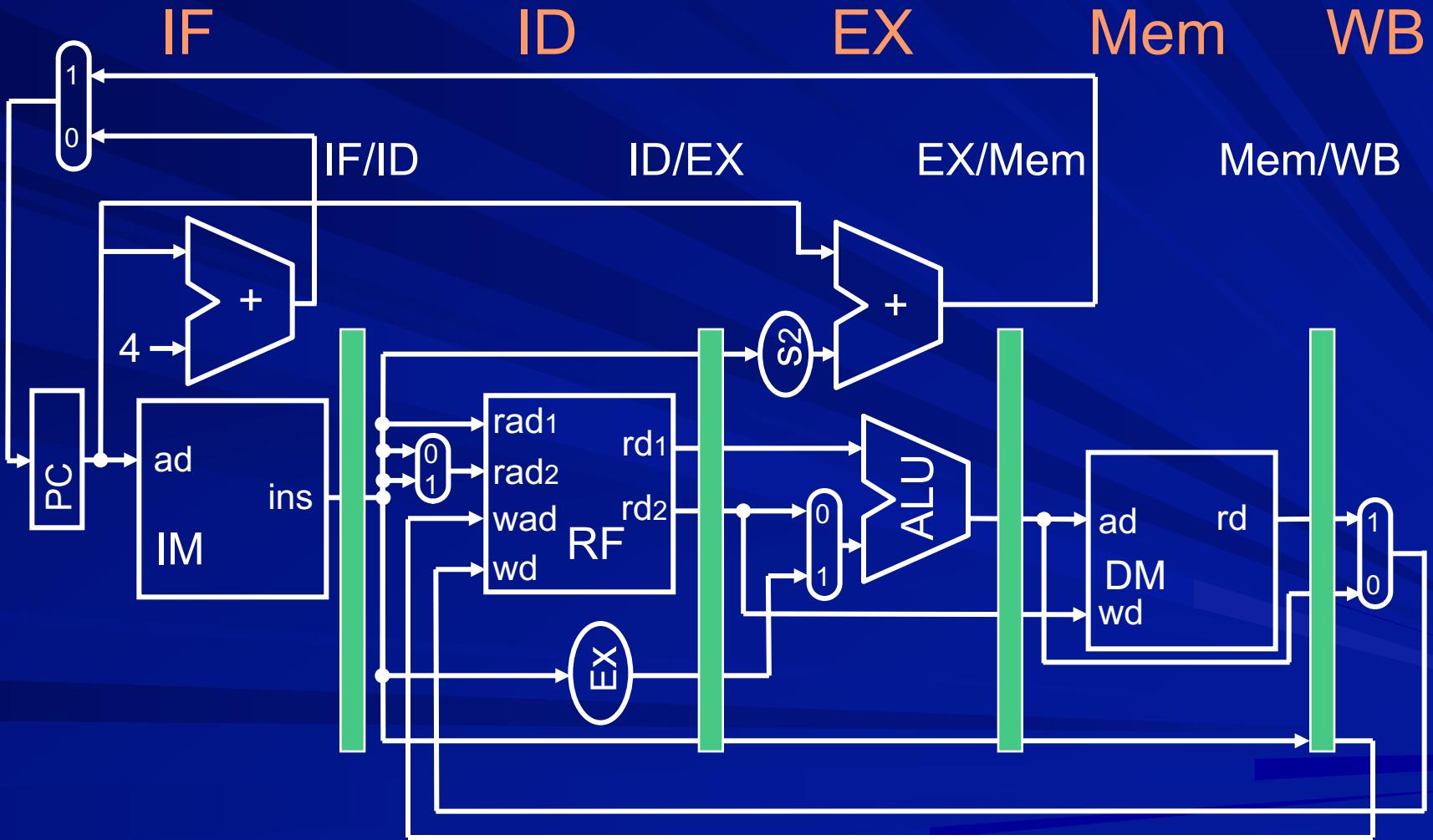
Pipelined datapath



Fetch new instruction every cycle



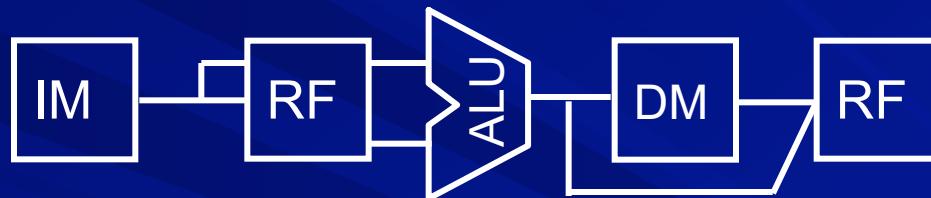
Correction for WB stage



Graphical representation

5 stage pipeline

stages

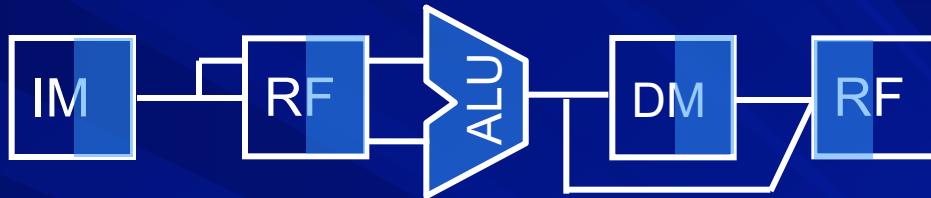


actions

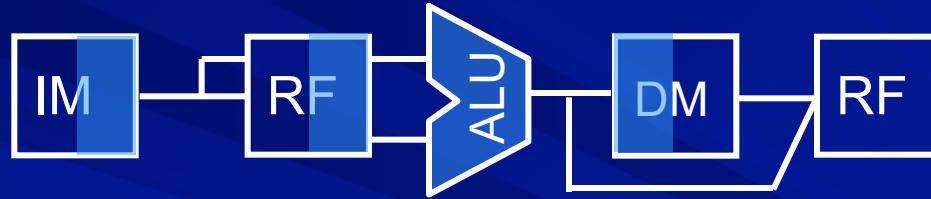


Usage of stages by instructions

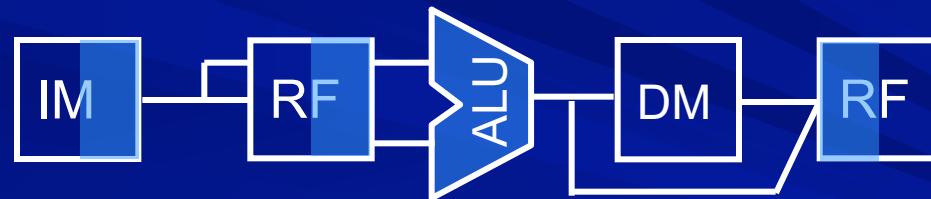
ldr



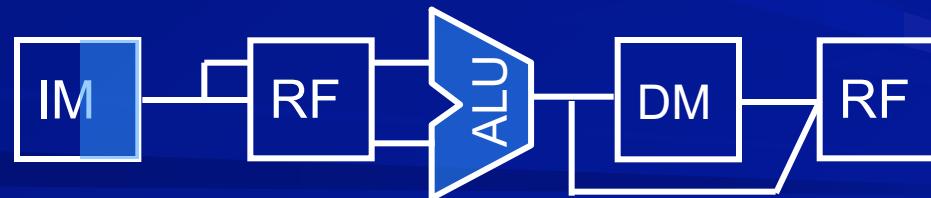
str



add



b



Representing pipelined execution

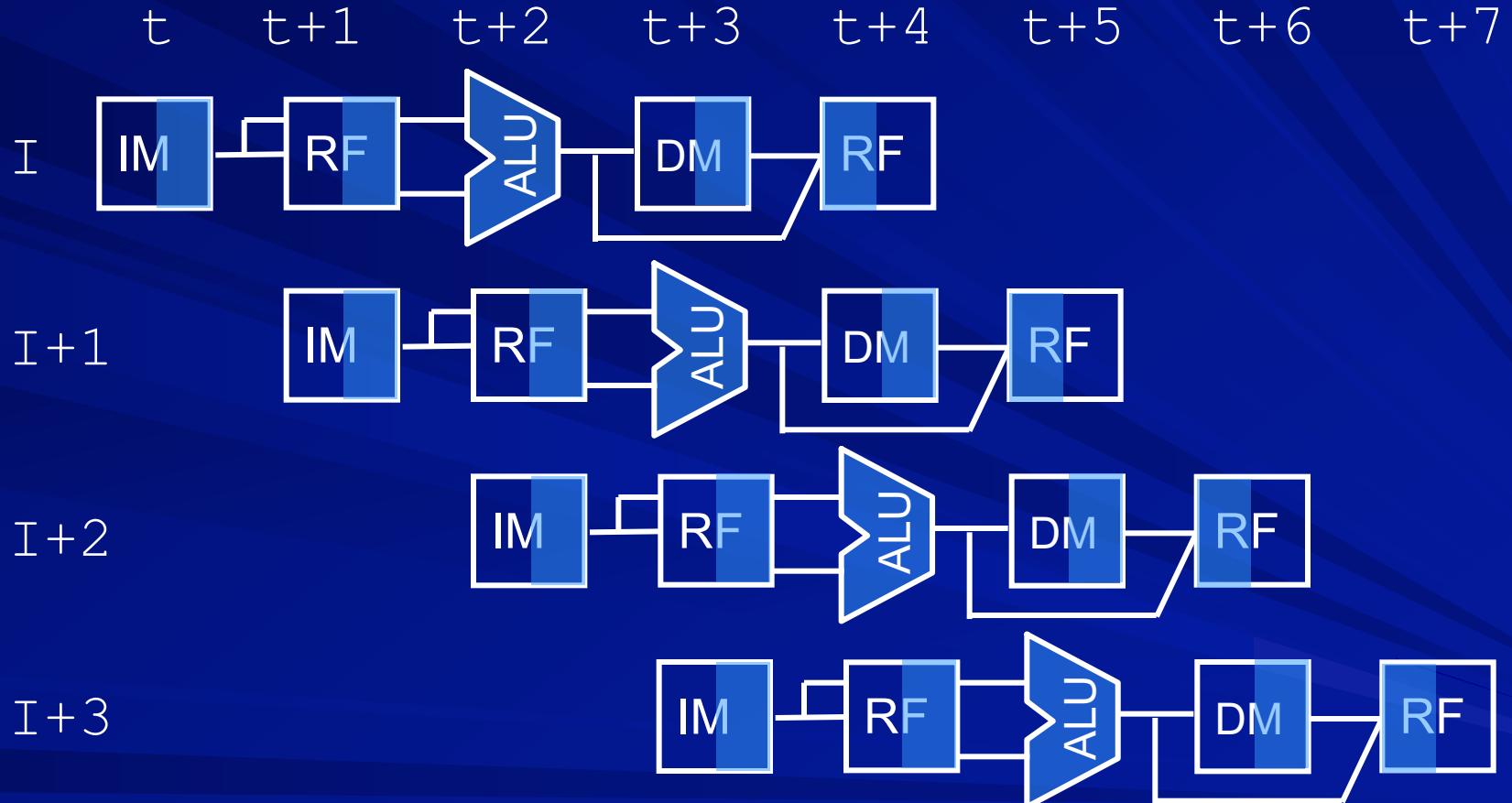
Representation I

- Horizontal axis: time
- Vertical axis: instructions

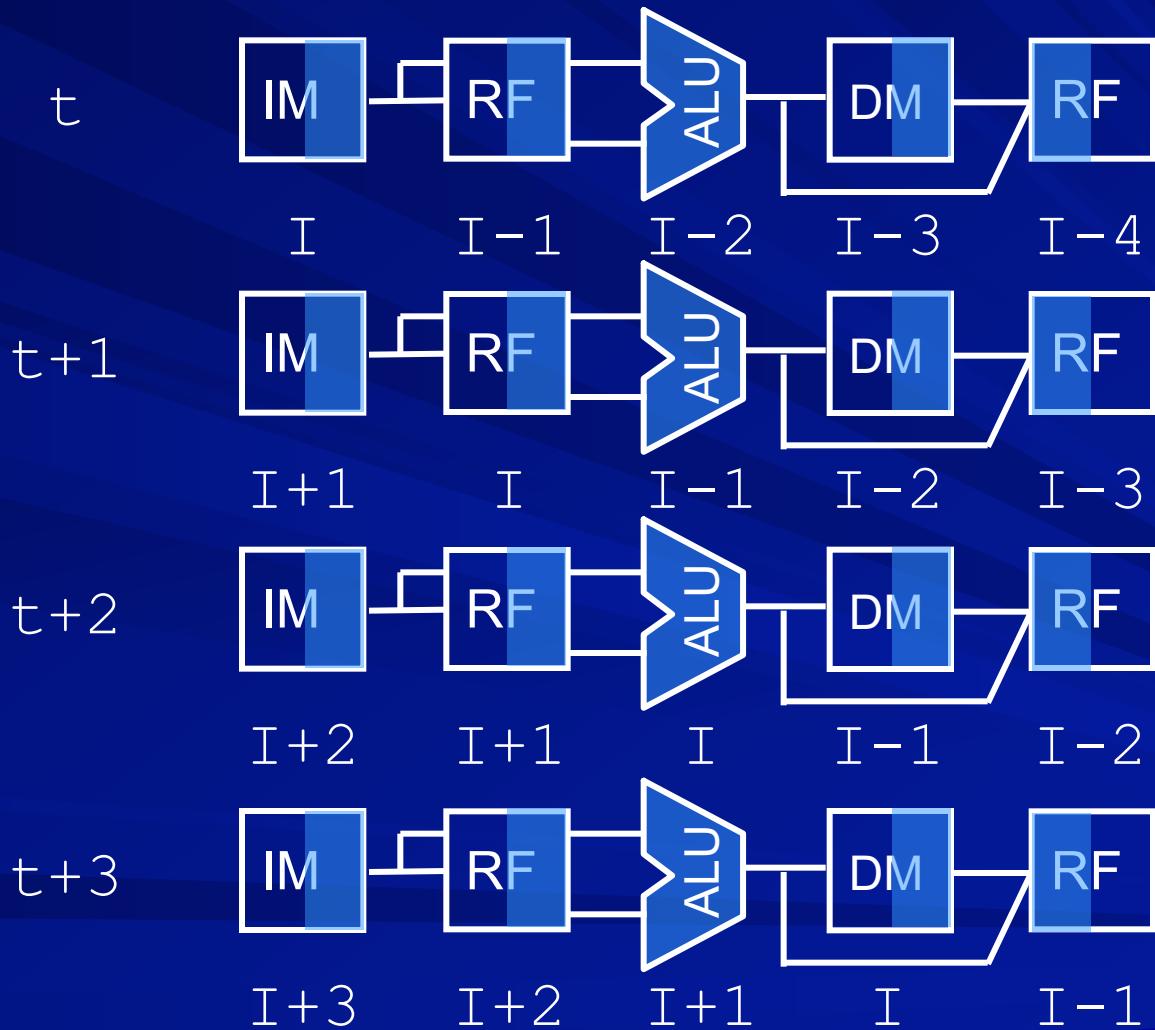
Representation II

- 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

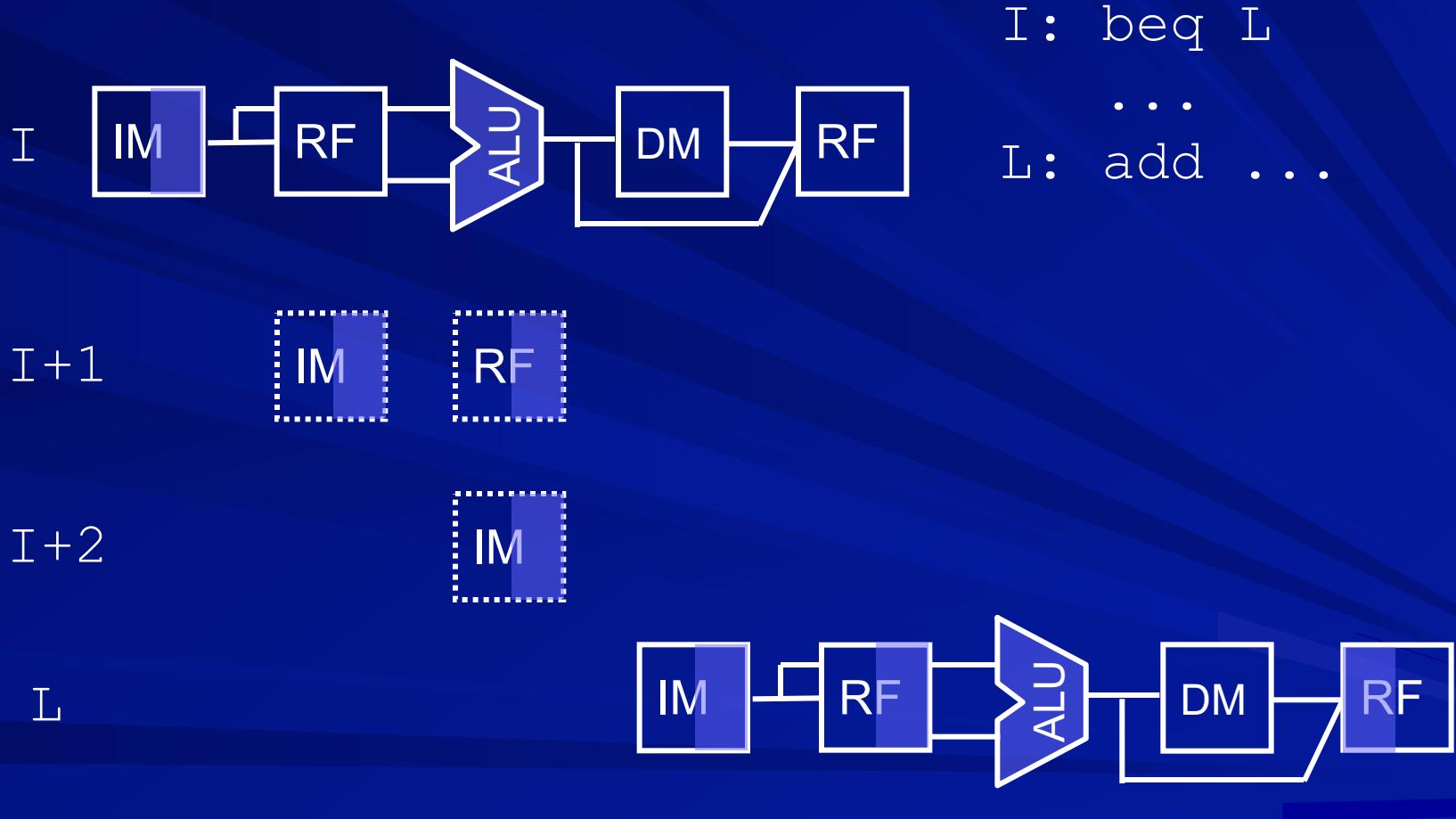
■ Control Hazards

- Decision about next instruction needs more cycles

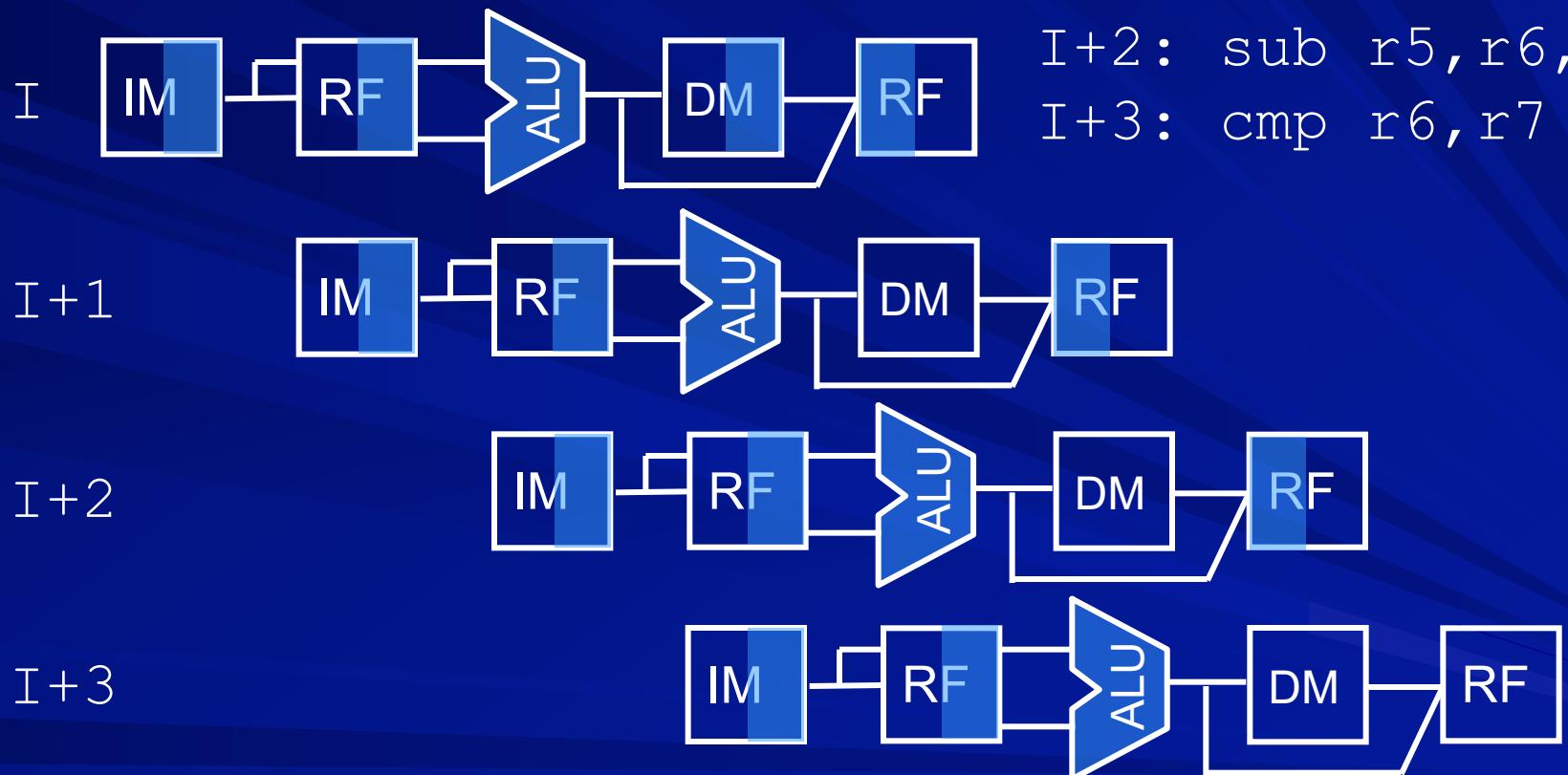
Structural hazards

- 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

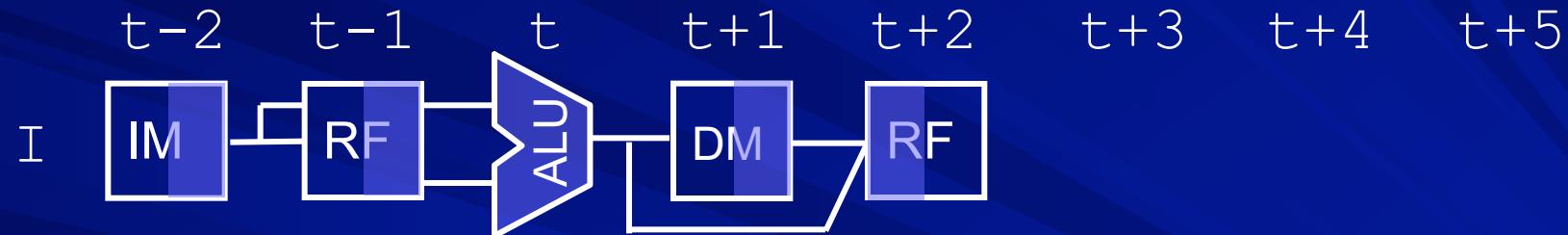


I: ldr r6, ...
I+1: add r4, r6, ...
I+2: sub r5, r6, ...
I+3: cmp r6, r7

Stalls due to data hazards

instruction view

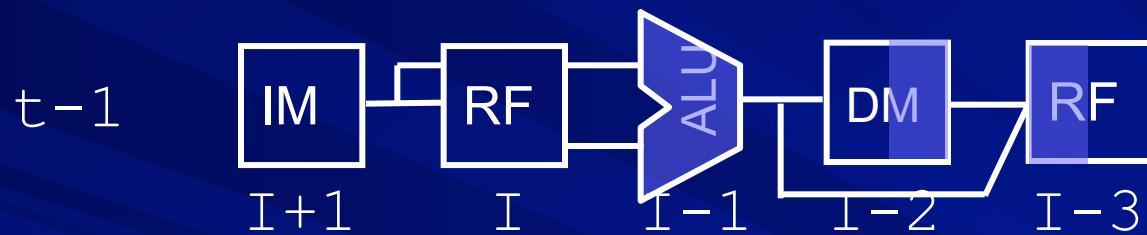
I: ldr r6, ...
I+1: add r4, r6, ...



Stalls due to data hazards

stage-wise view

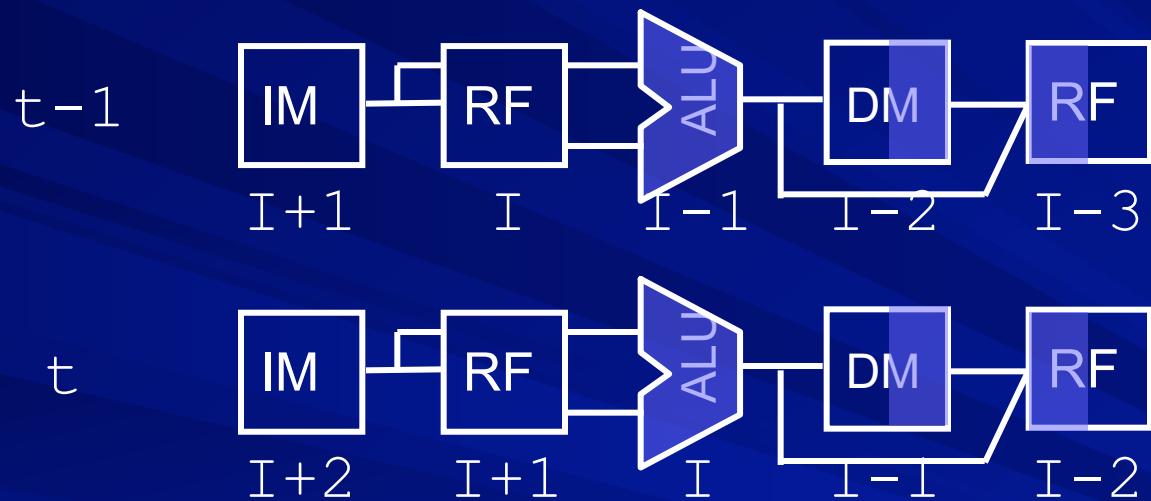
I: ldr r6, ...
I+1: add r4, r6, ...



Stalls due to data hazards

stage-wise view

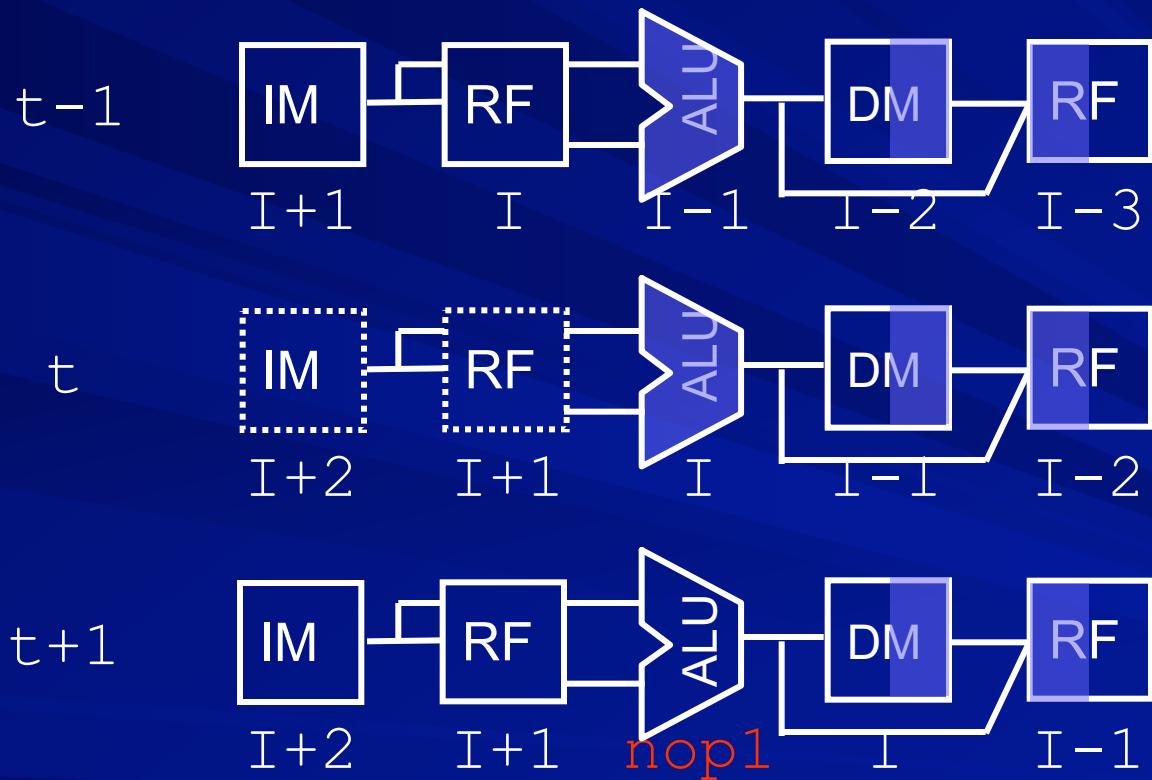
I: ldr r6, ...
I+1: add r4, r6, ...



Stalls due to data hazards

stage-wise view

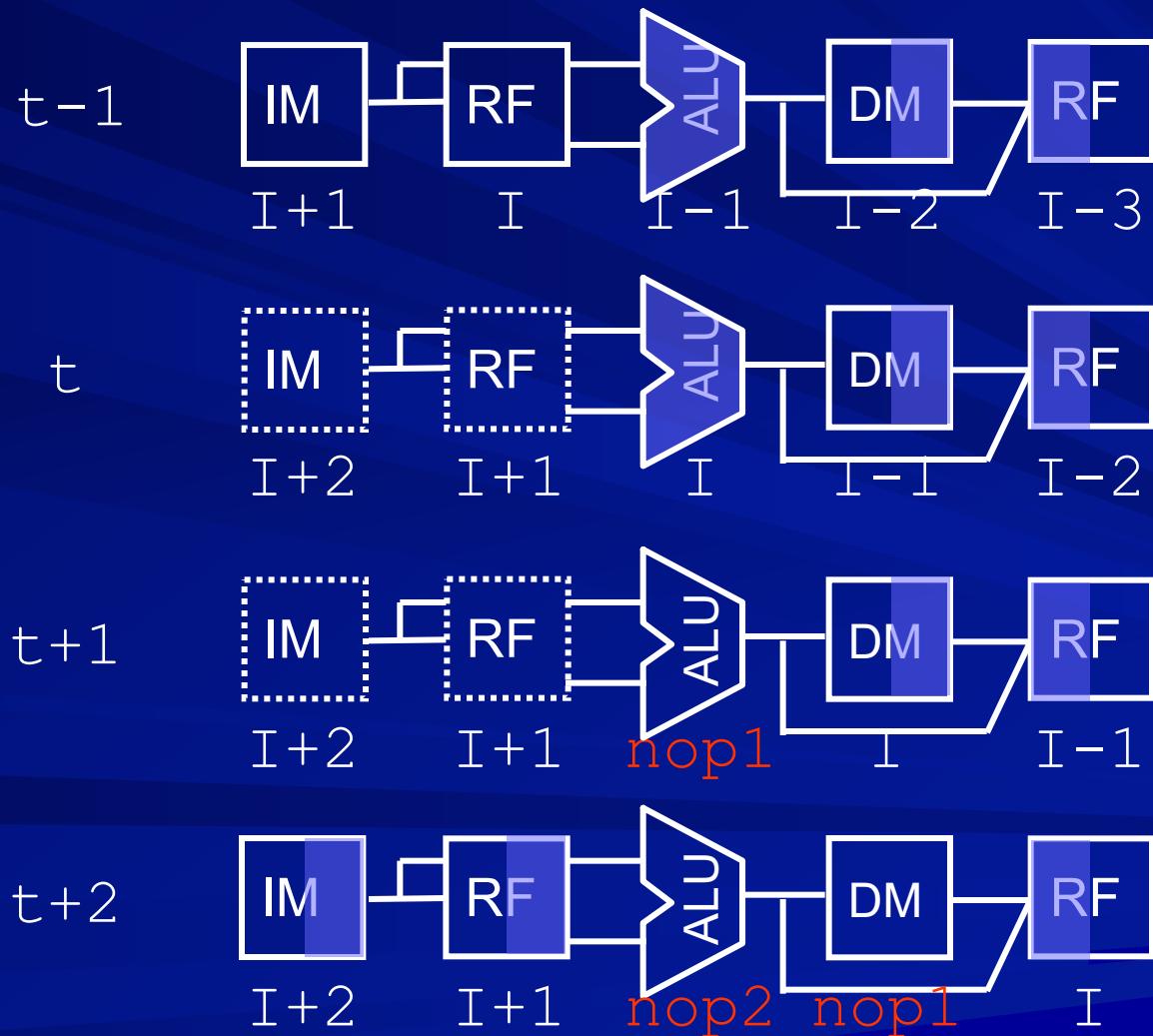
I: ldr r6, ...
I+1: add r4, r6, ...



Stalls due to data hazards

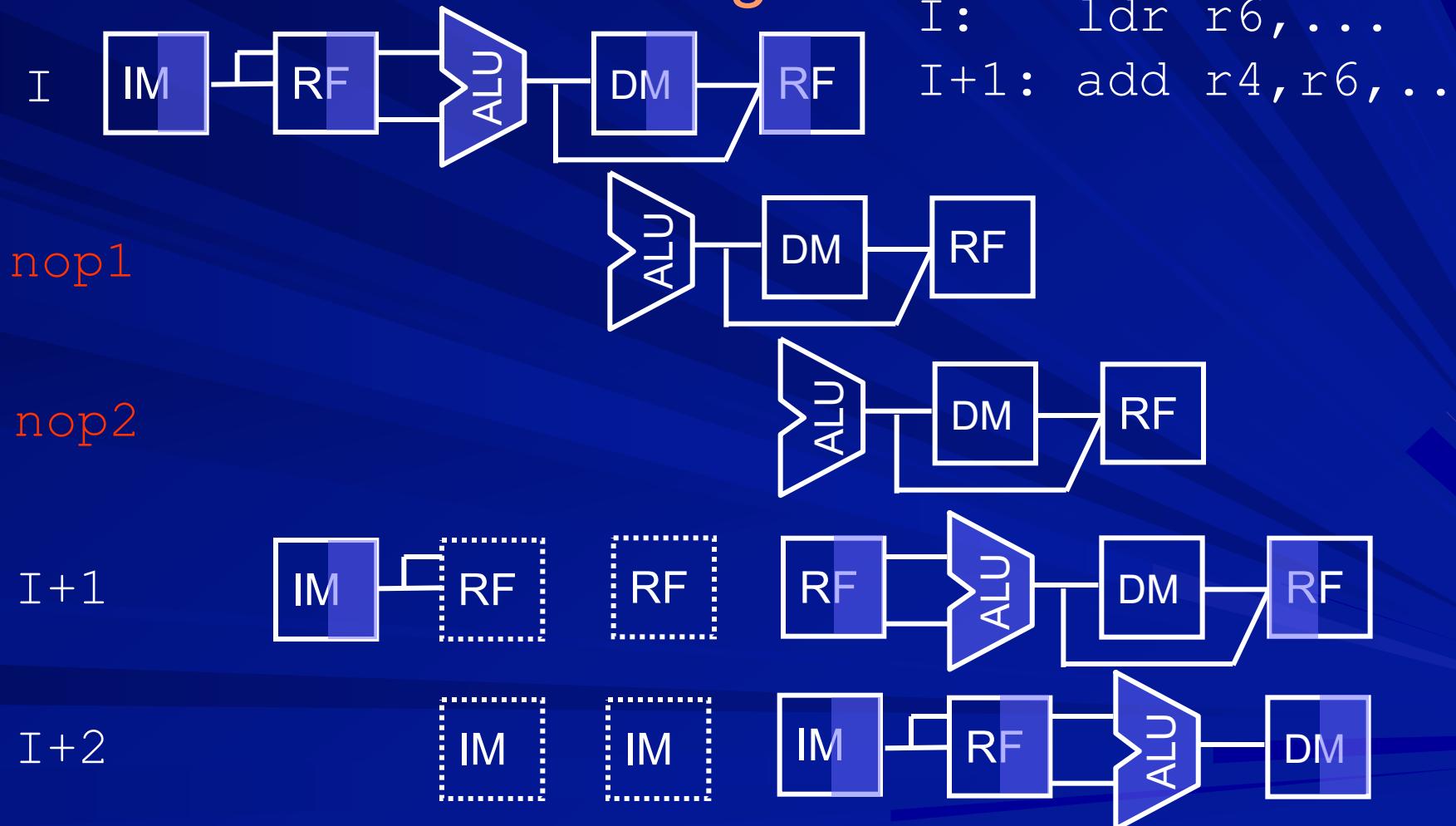
stage-wise view

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



Stalls due to data hazards

instruction view again



Handling hazards

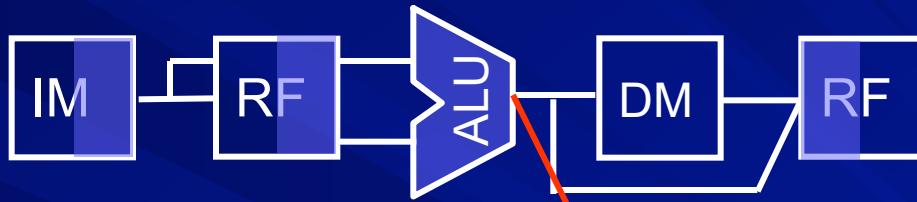
- Data hazards
 - detect instructions with data dependence
 - introduce nop instructions (bubbles) in the pipeline
 - more complex: data forwarding
- Control hazards
 - detect branch instructions
 - flush inline instructions if branching occurs
 - more complex: branch prediction

Are there software solutions?

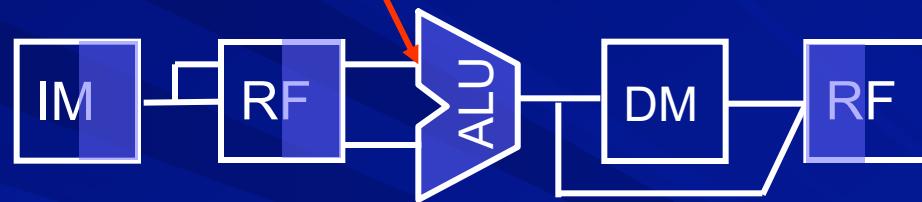
- Separate dependent instructions by reordering code
- Insert nop instructions in worst case
- Treat branches as delayed branches and insert suitable instructions in delay slots

Data forwarding path P1

I:add r6,...

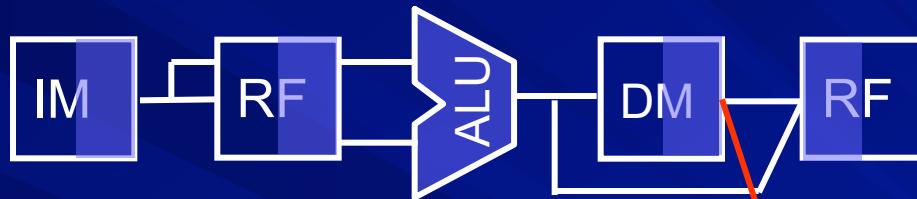


I+1:add r4, r6,..

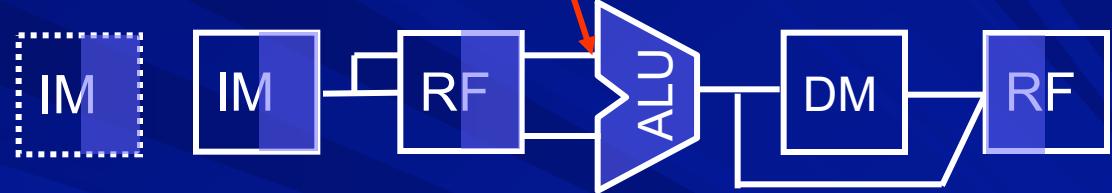


Data forwarding path P2

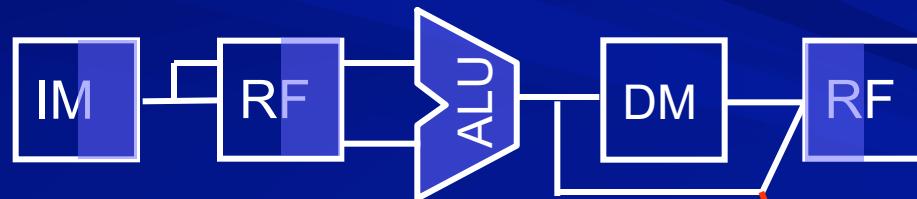
I:ldr r6,..



I+1:add r4,r6,..



I:add r6,..

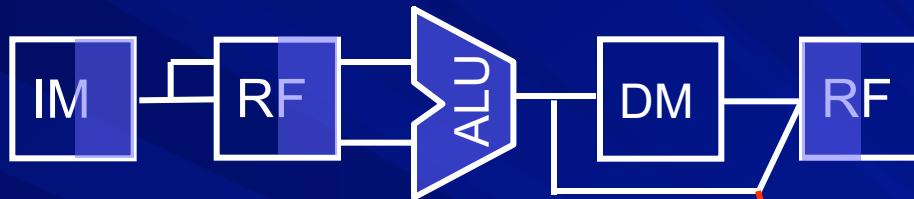


I+2:add r4,r6,..

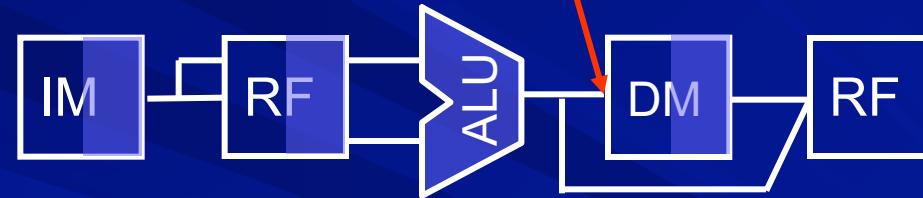


Data forwarding path P3

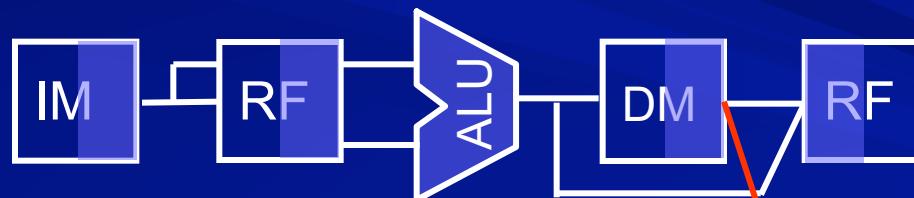
I:add r6,...



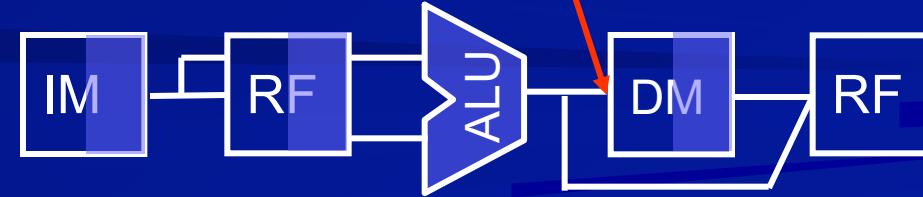
I+1:str r6,...



I:ldr r6,...



I+1:str r6,...



Thanks

COL216

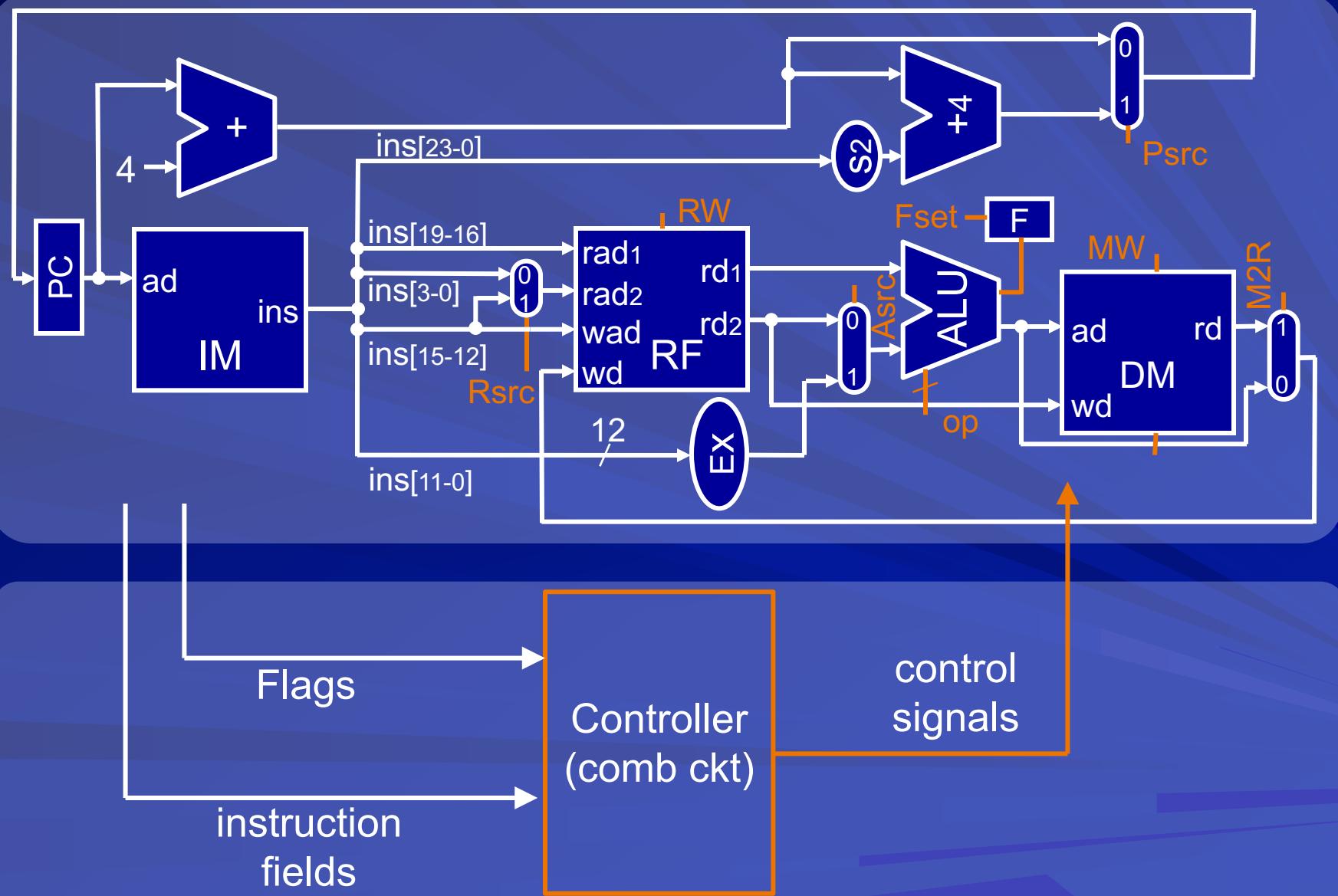
Computer Architecture

Pipelined Processor design –

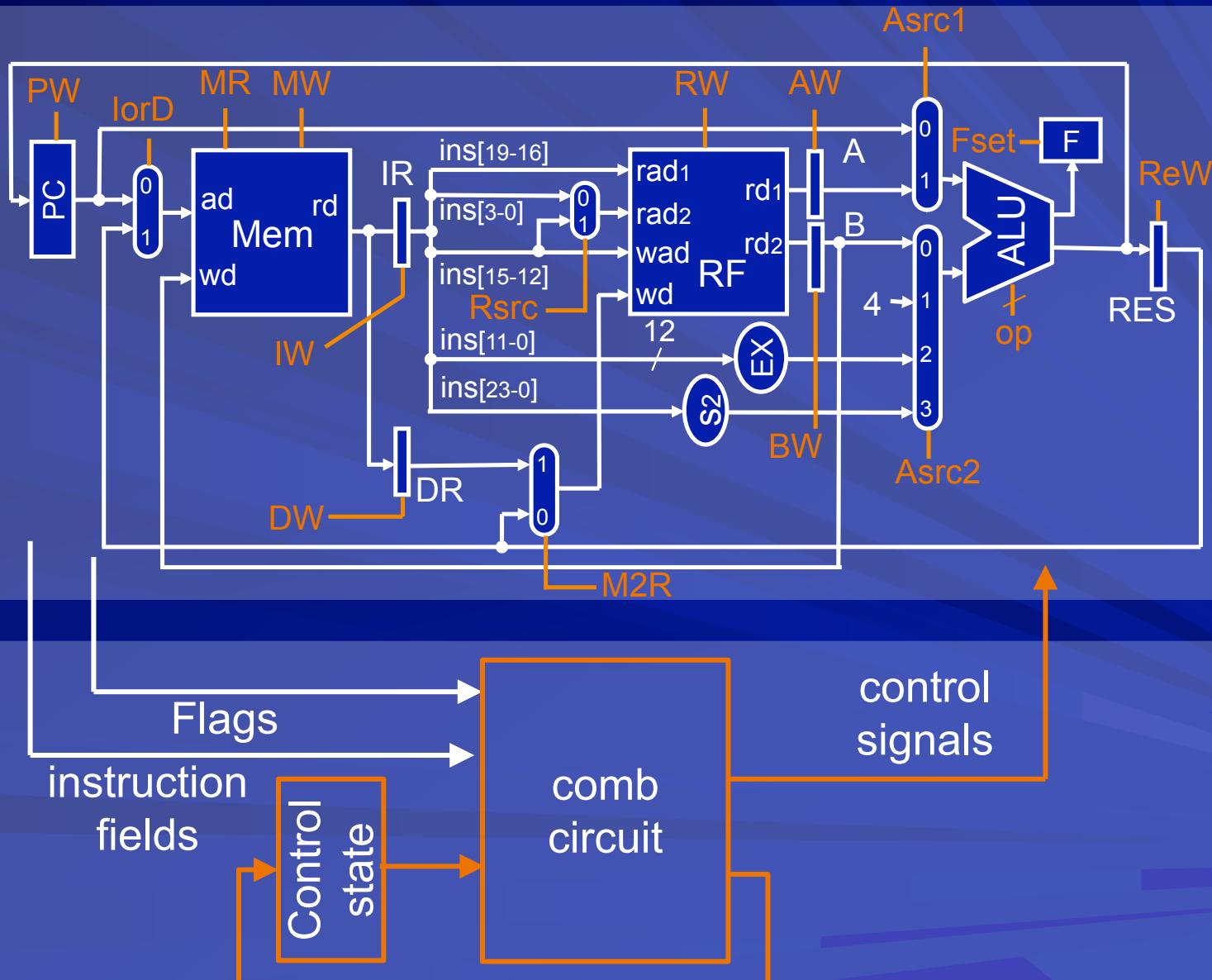
Controller, Data forwarding

10th February 2022

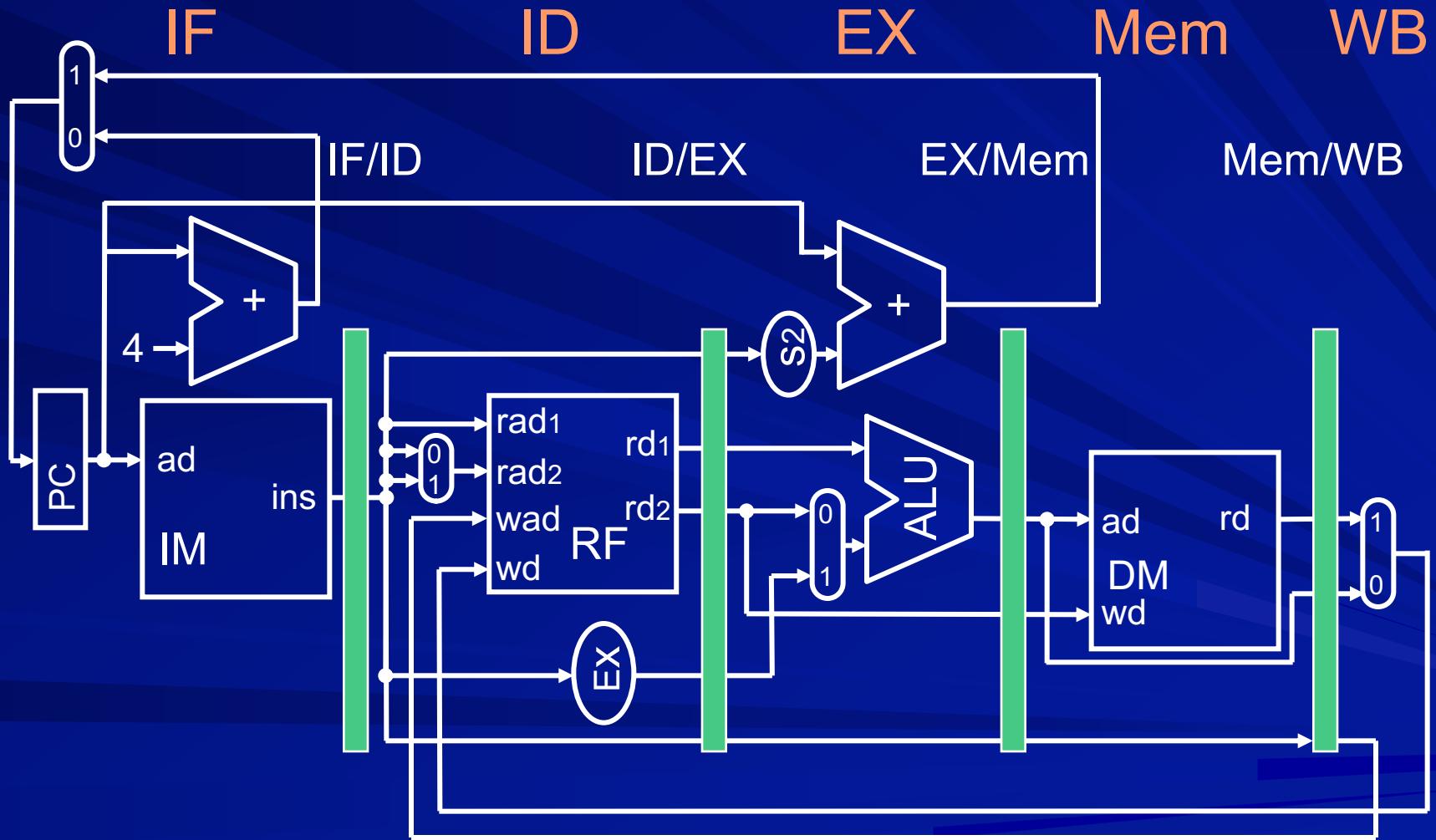
Single Cycle Datapath + Controller



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

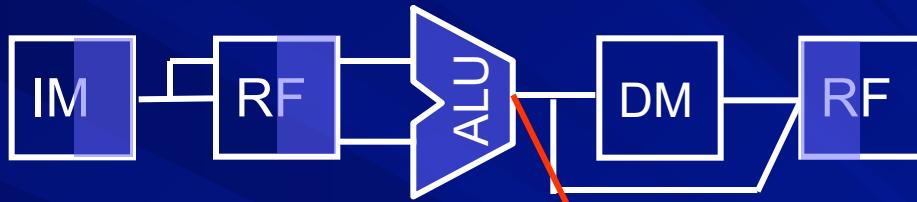
Detecting data hazard

Condition to be checked:

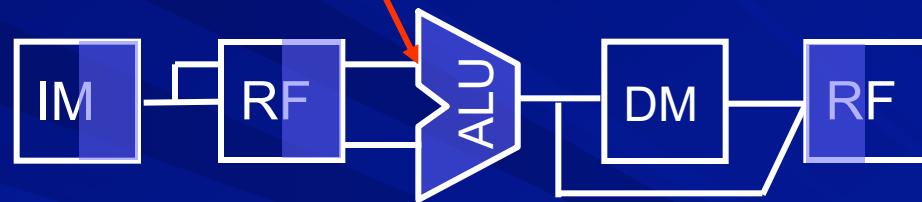
Instruction in RF stage reads from a register in which instruction in ALU stage or DM stage is going to write

Data forwarding path P1

I:add r6,...

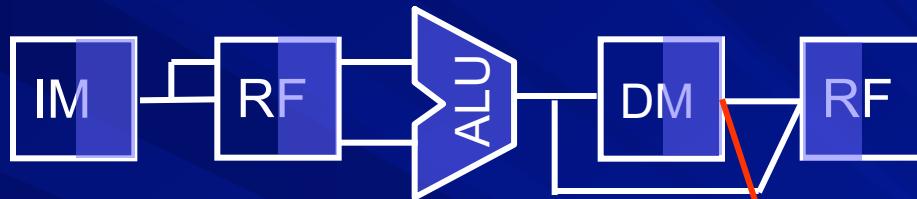


I+1:add r4, r6,..

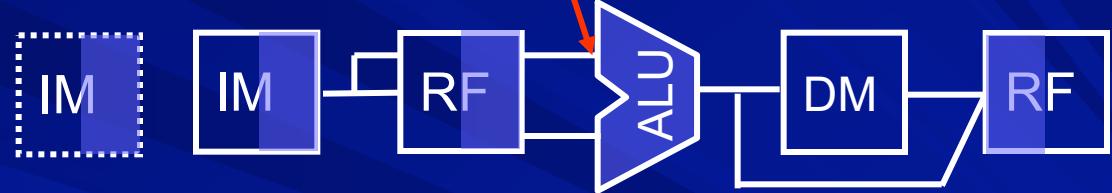


Data forwarding path P2

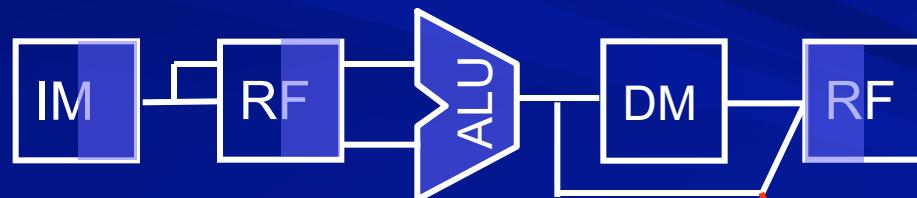
I:ldr r6,..



I+1:add r4,r6,..



I:add r6,..

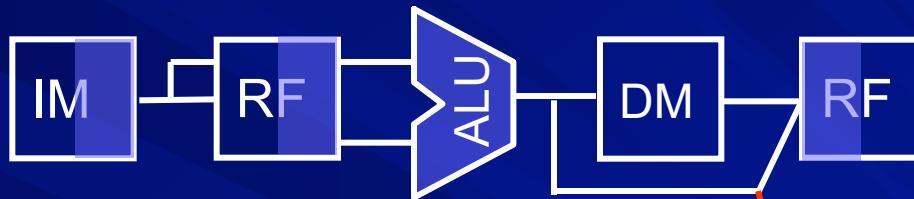


I+2:add r4,r6,..

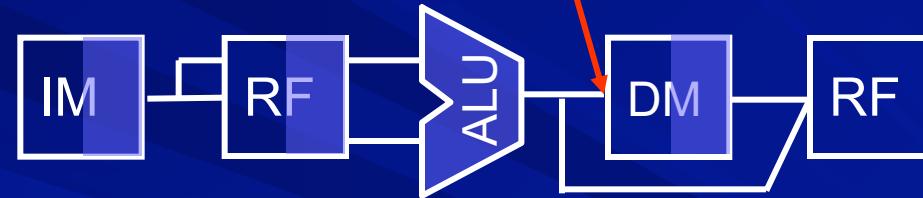


Data forwarding path P3

I:add r6,...



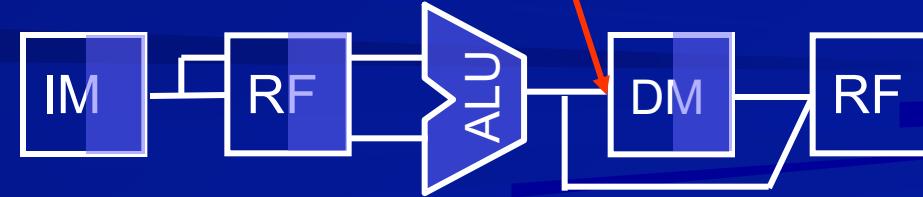
I+1:str r6,...



I:ldr r6,...



I+1:str r6,...



Data forwarding path list

■ P1

from ALU out

to ALU in1/2

(EX/DM register)

■ P2

from DM/ALU out

to ALU in1/2

(DM/WB register)

■ P3

from DM/ALU out

to DM in

(DM/WB register)

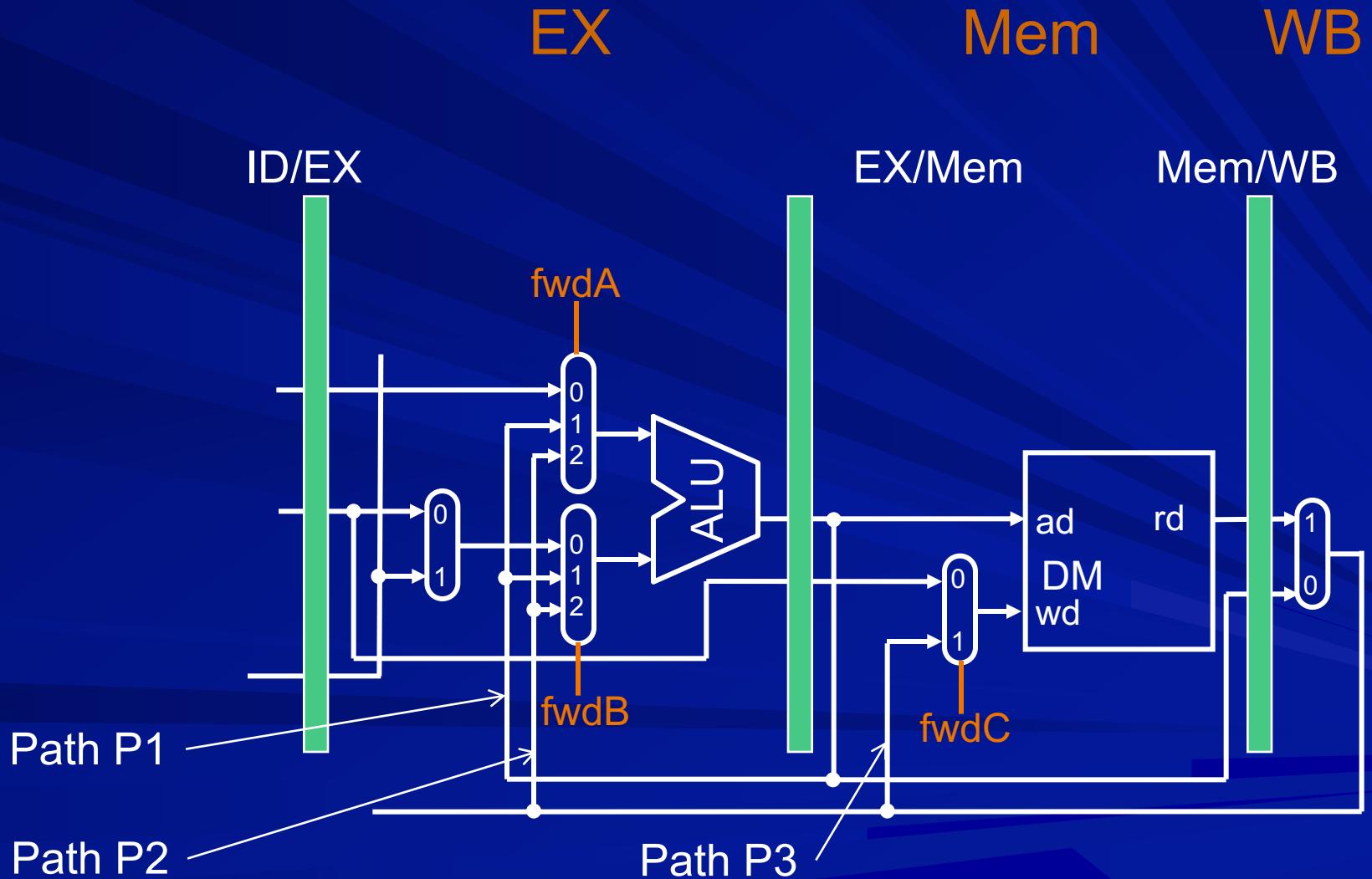
Dependence check logic

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



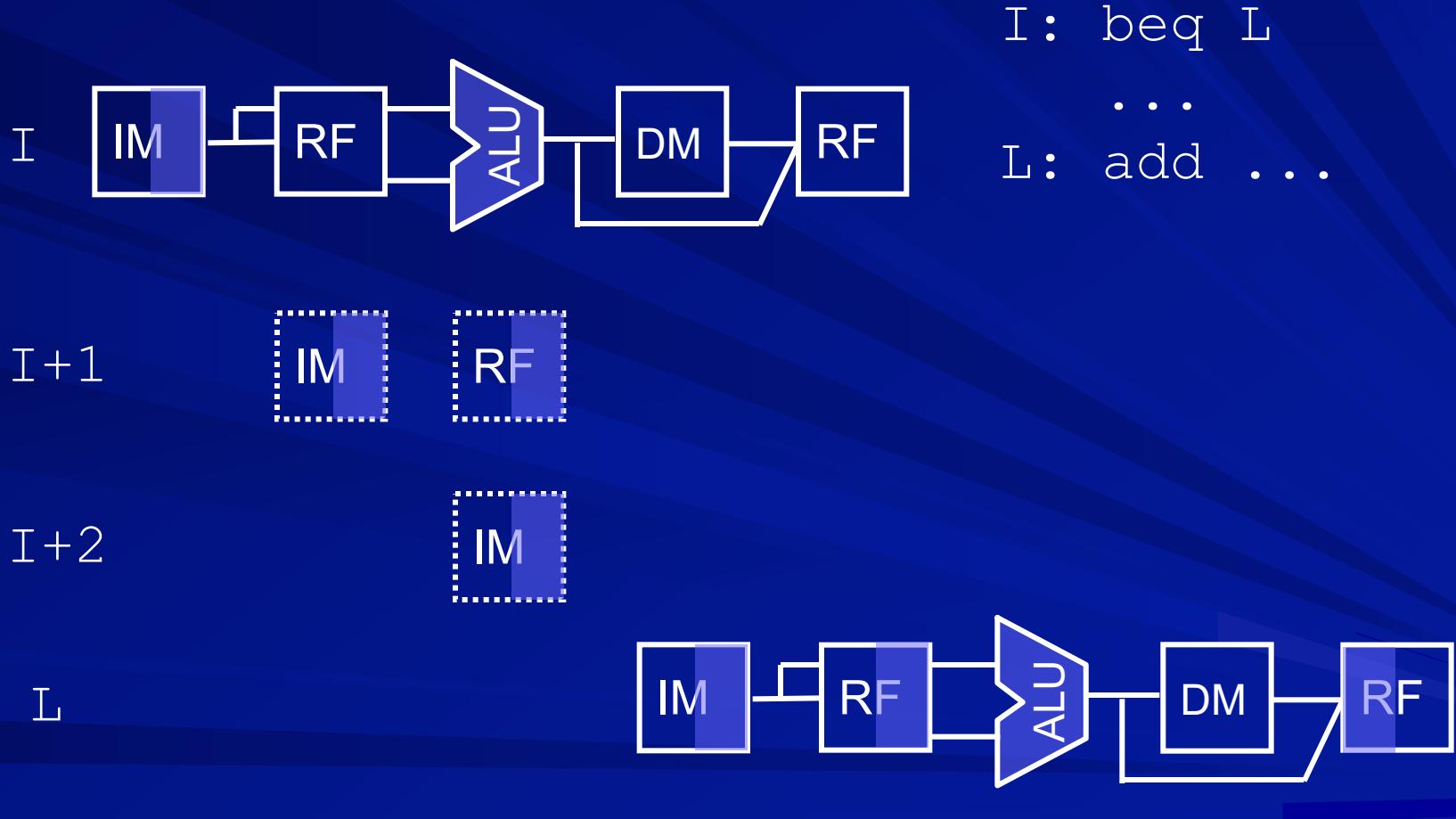
Executing branch instructions

- 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

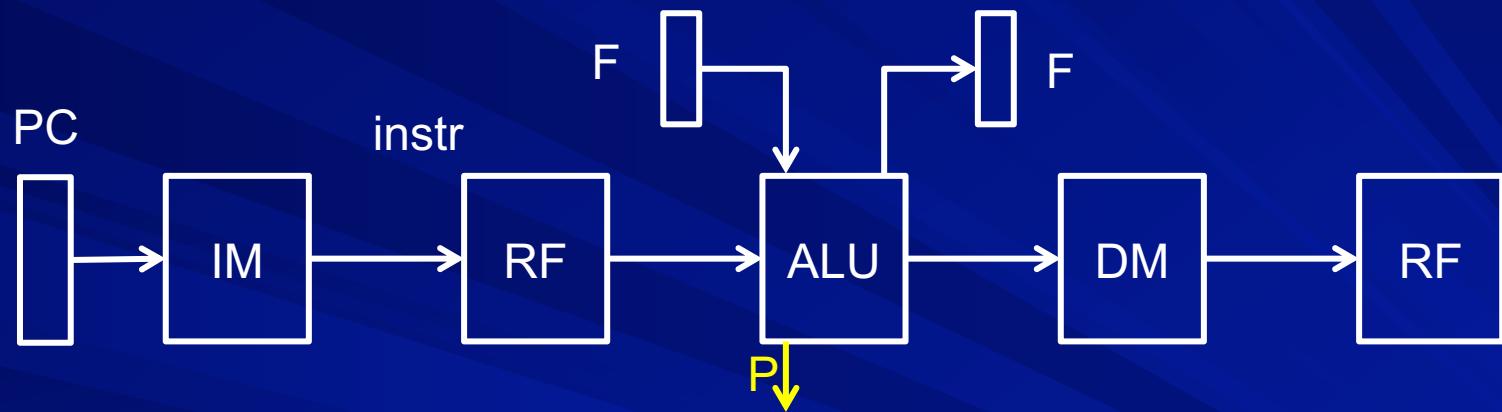
- Flush the inline instructions
- Freeze (stall) the inline instructions
- Allow the inline instructions to continue

Stalls due to control hazards



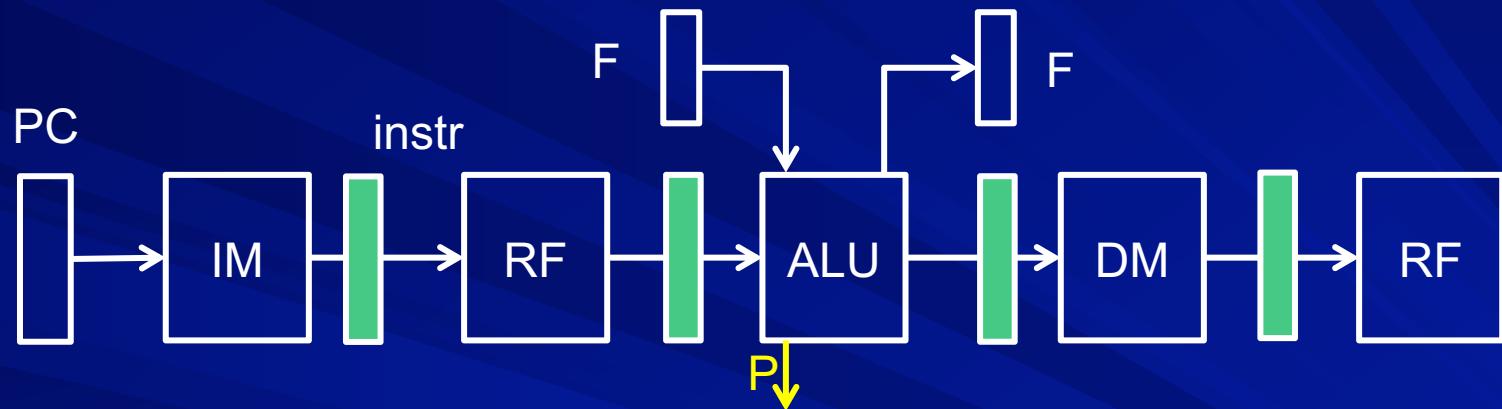
Controller design

Single cycle datapath



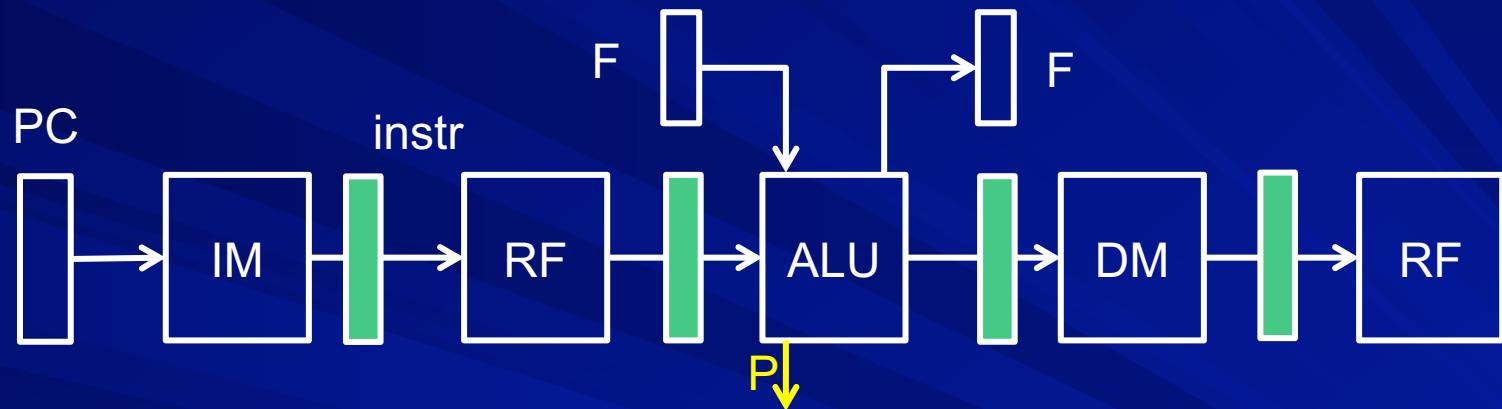
Multi-cycle datapath

Resource sharing possible across cycles

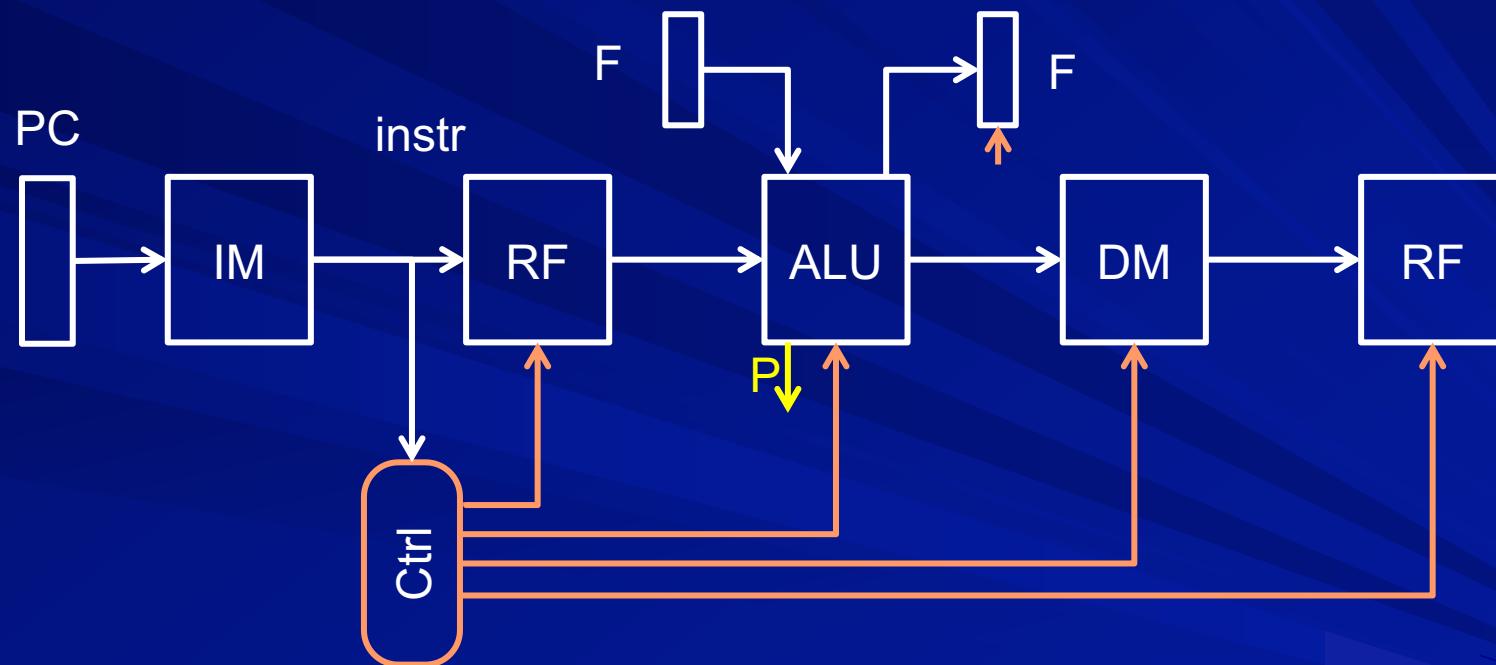


Pipelined datapath

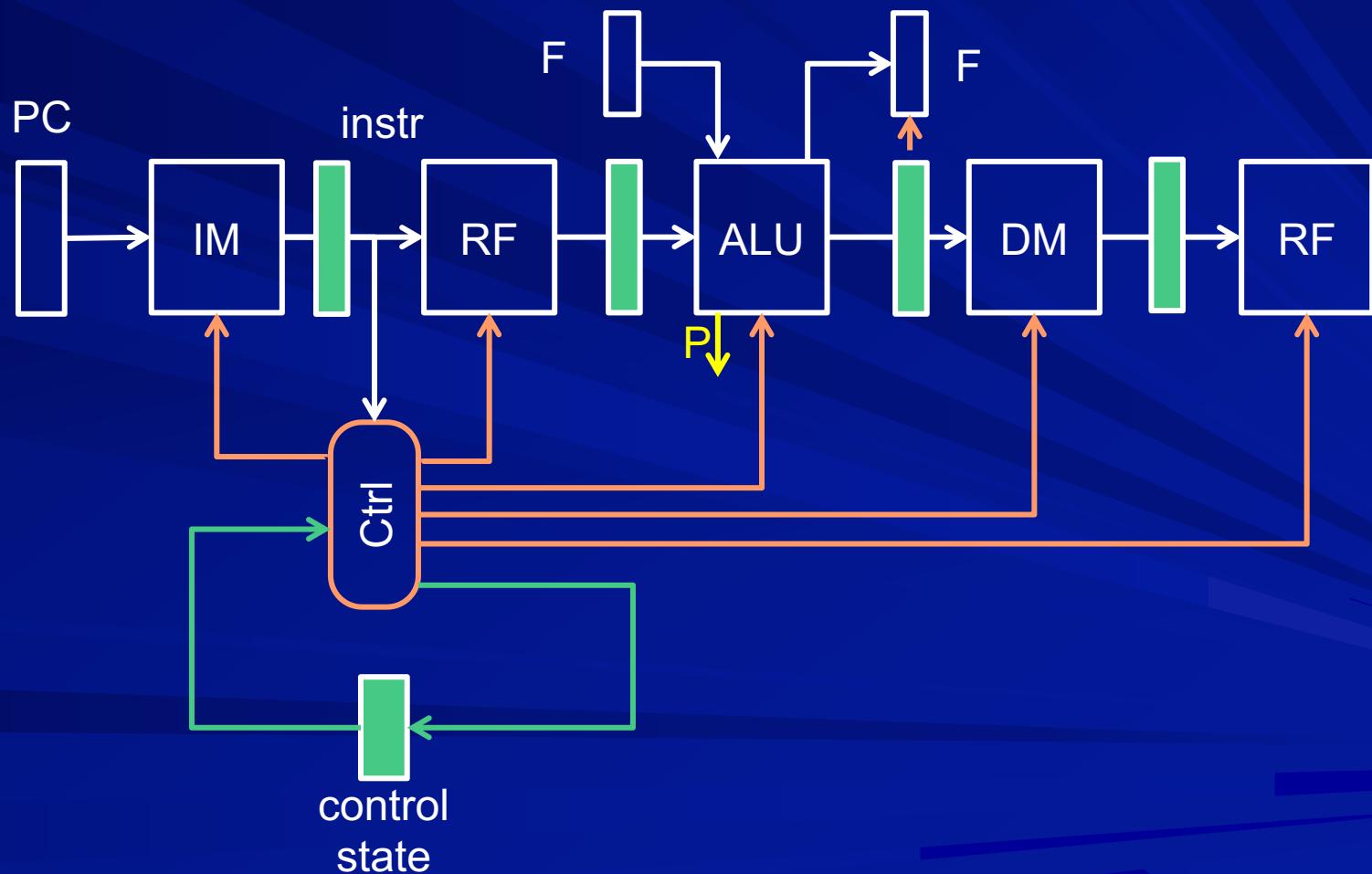
Resource sharing leads to hazards



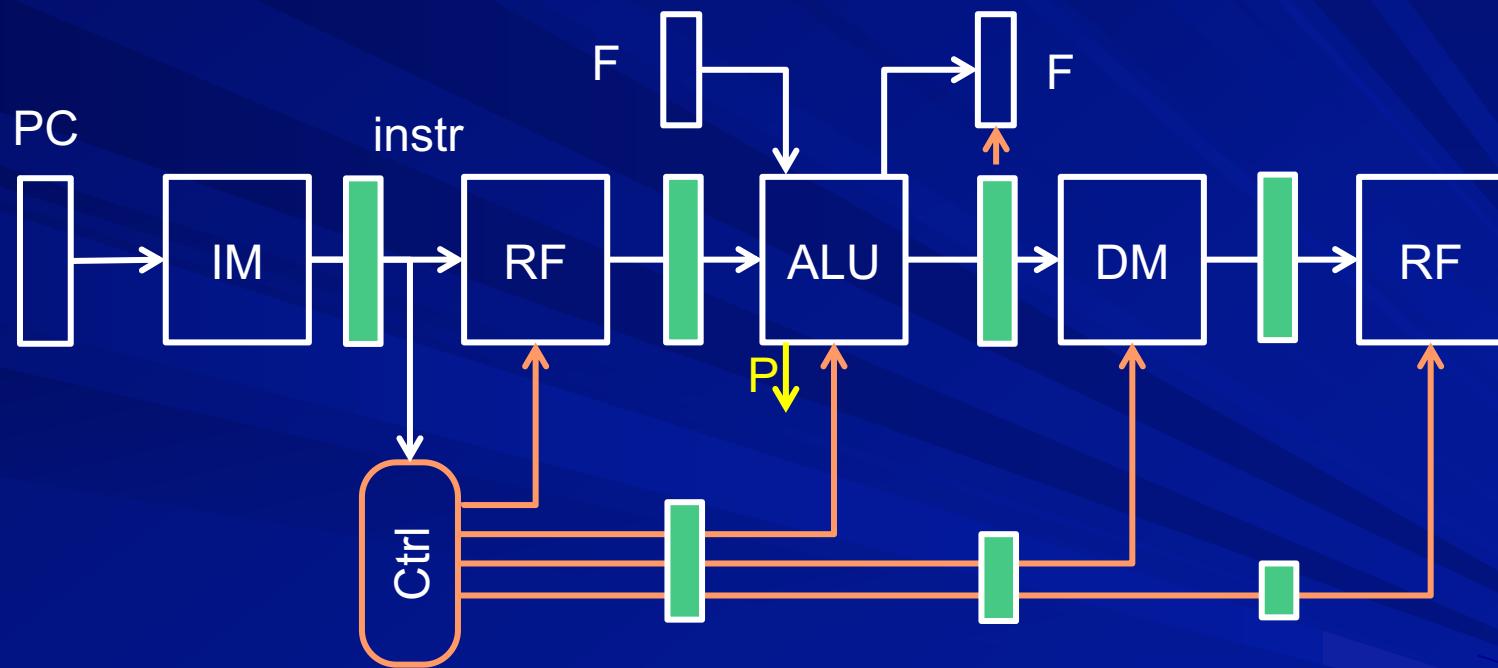
Controller for single cycle DP



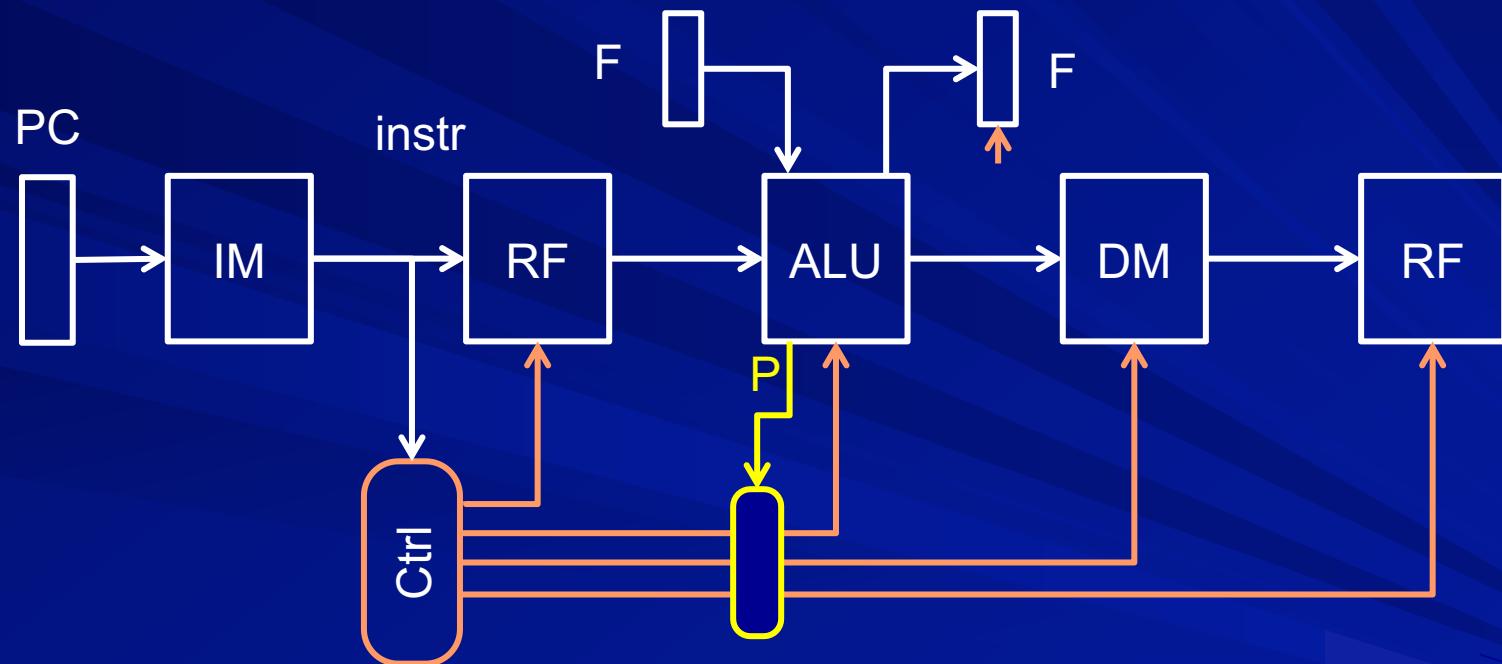
Controller for multi-cycle DP



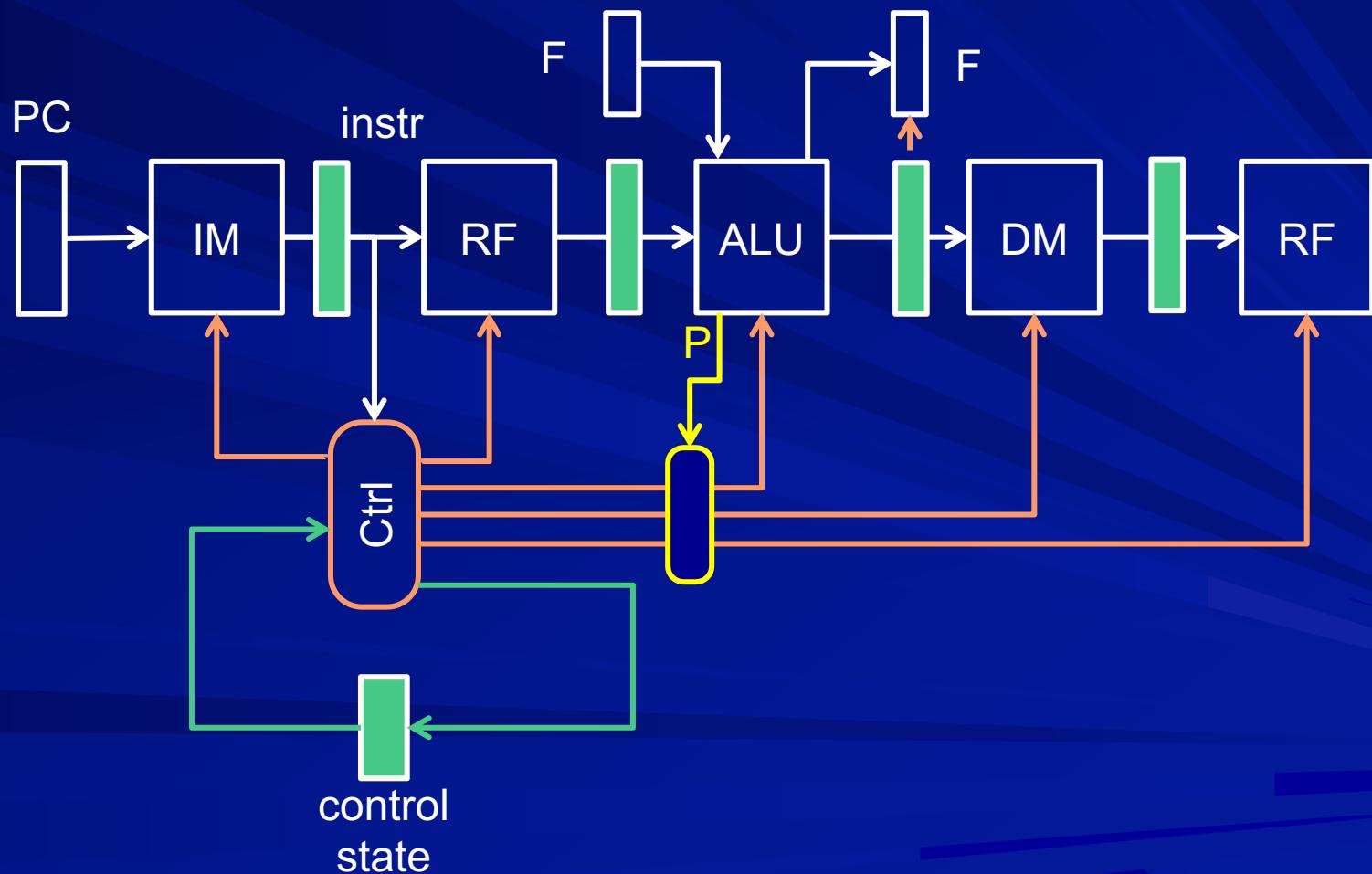
Controller for pipelined DP



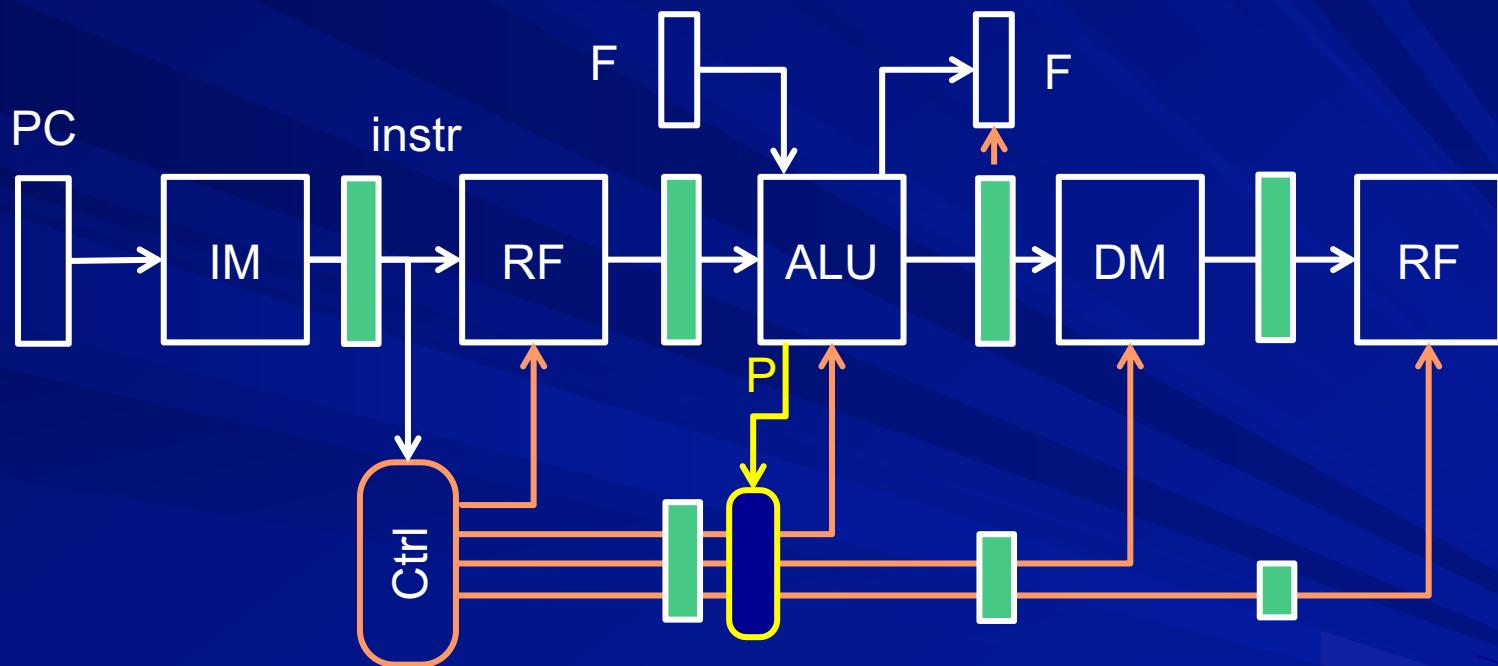
Controller for single cycle DP



Controller for multi-cycle DP



Controller for pipelined DP



Thanks