COL216
Computer Architecture

Designing a processor:

Datapath building blocks

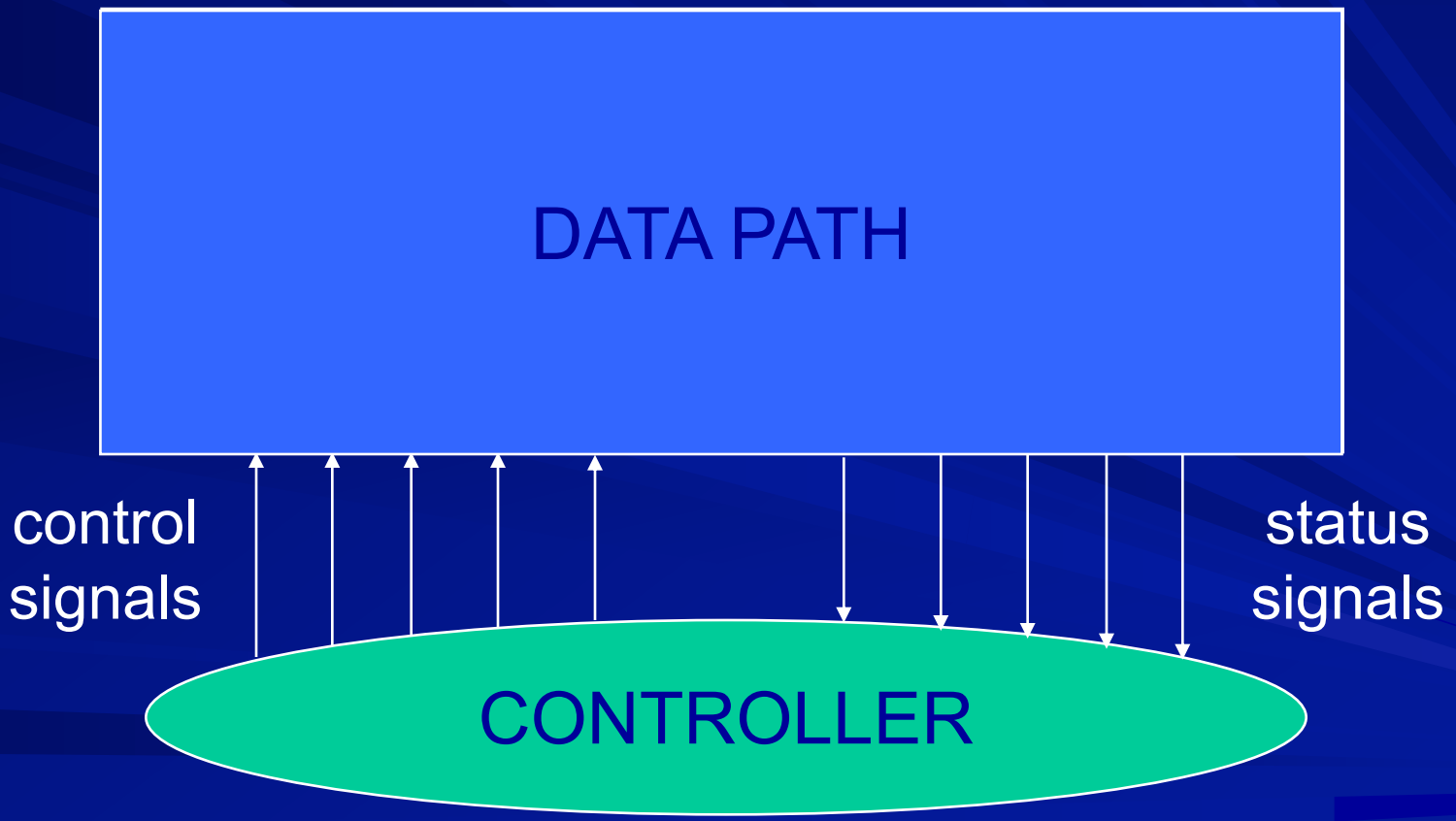
31st January, 2022

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

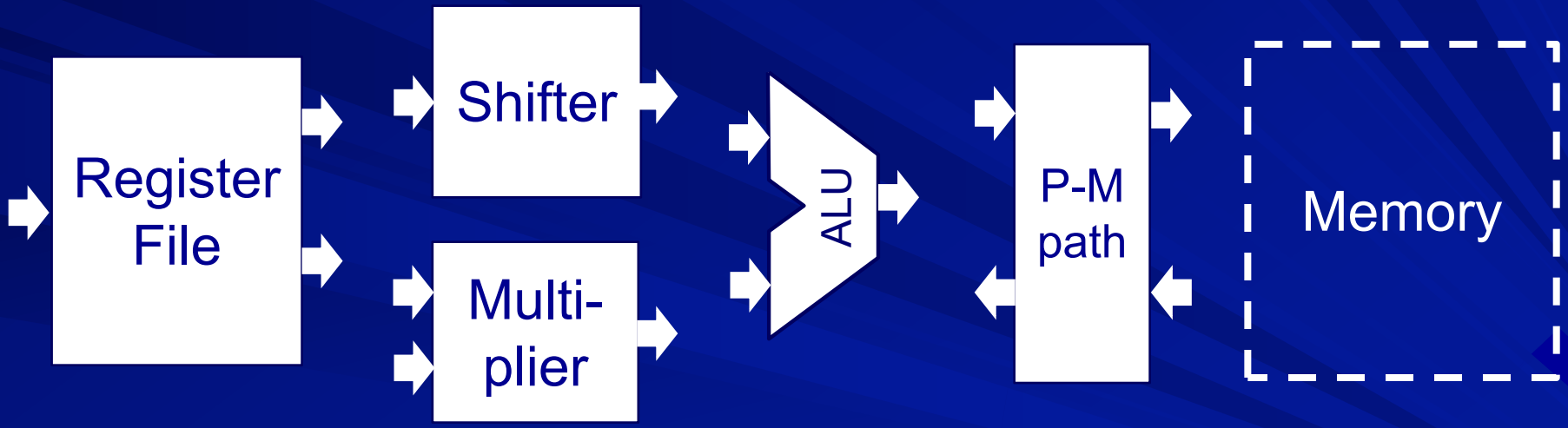
CPU datapath + controller

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 sbc	add sub	rsb	rsc
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
cmn	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

4 2 1 4 1 4 4 12

– 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

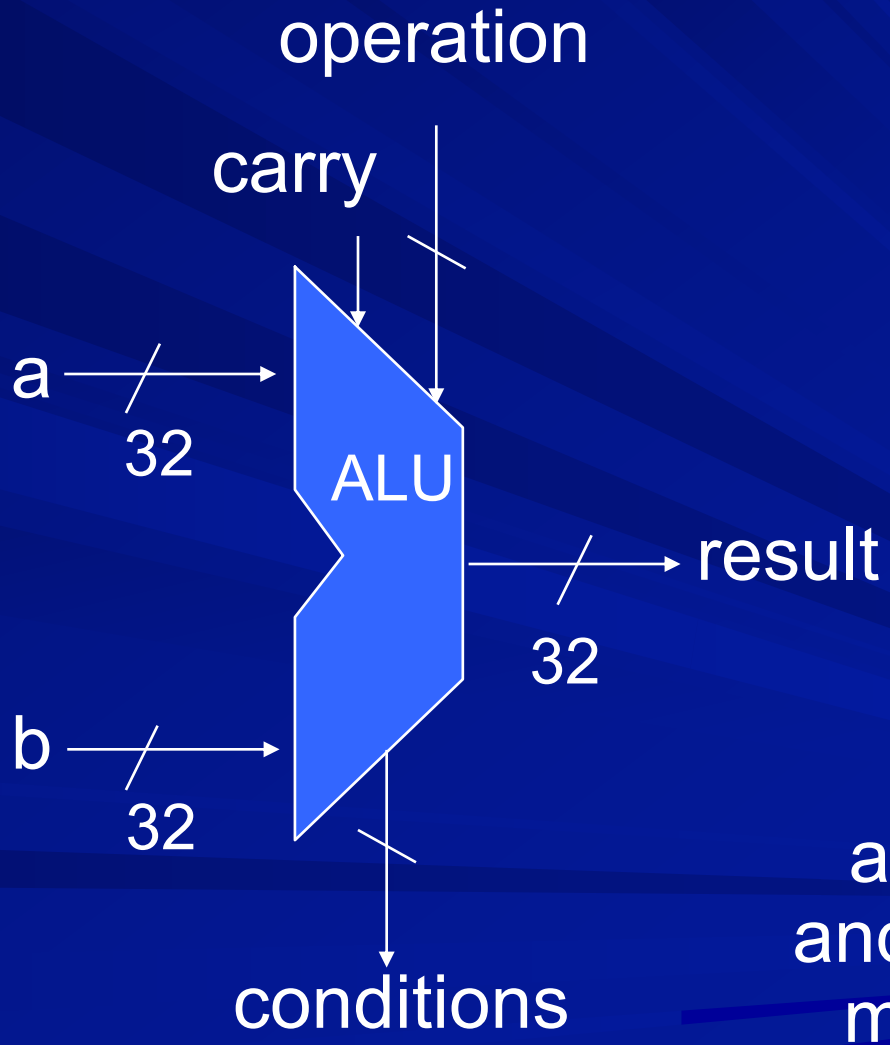
- 12 bit offset field in DT instructions

– 12 bit unsigned constant

or

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

Arithmetic and Logical operations



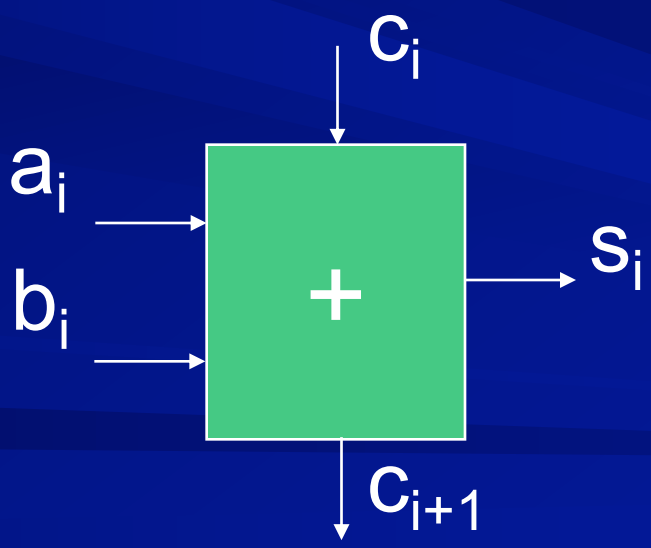
sources of operands

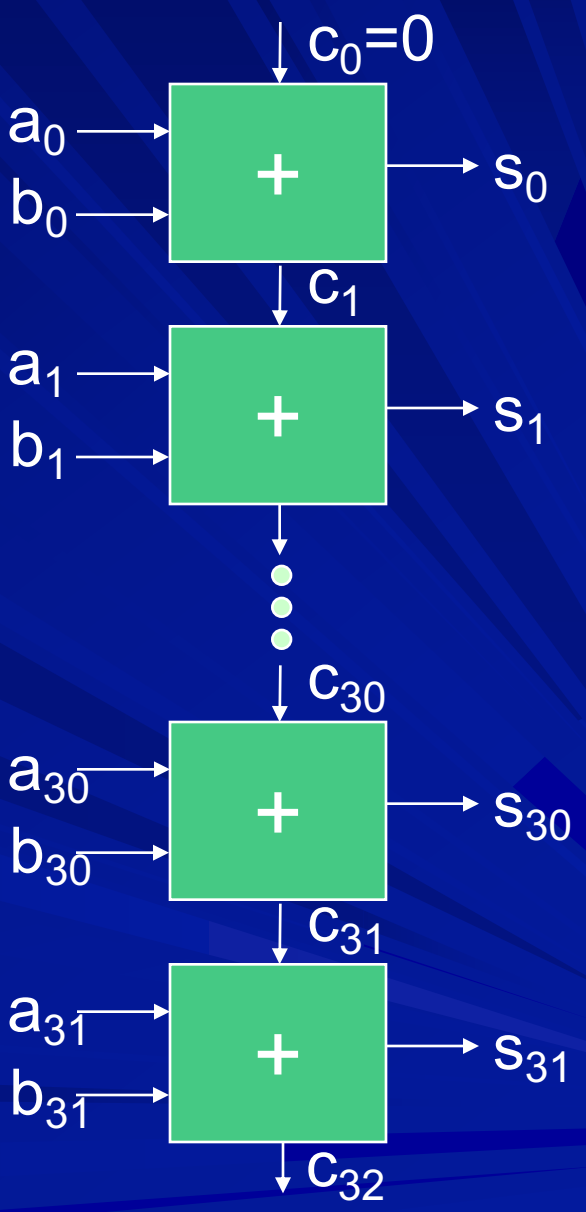
destination of result

add, sub, and, orr, eor, mov, mvn

Adder circuit

- Perform addition with carry propagation or carry look ahead

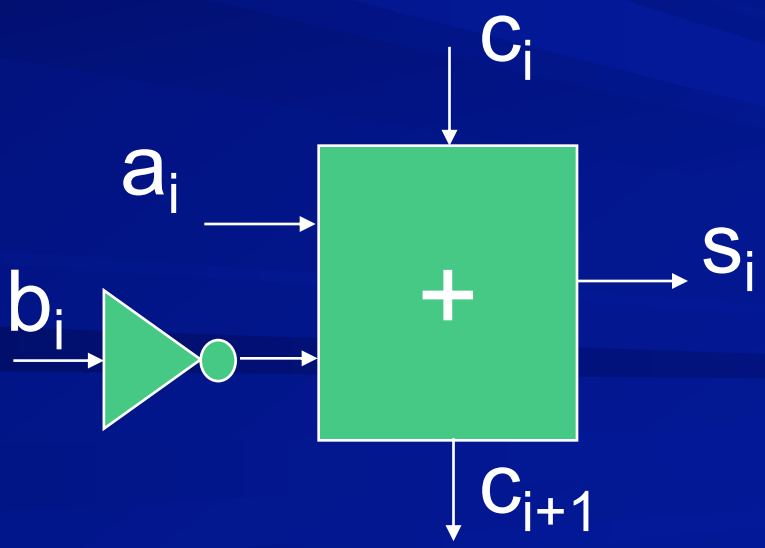


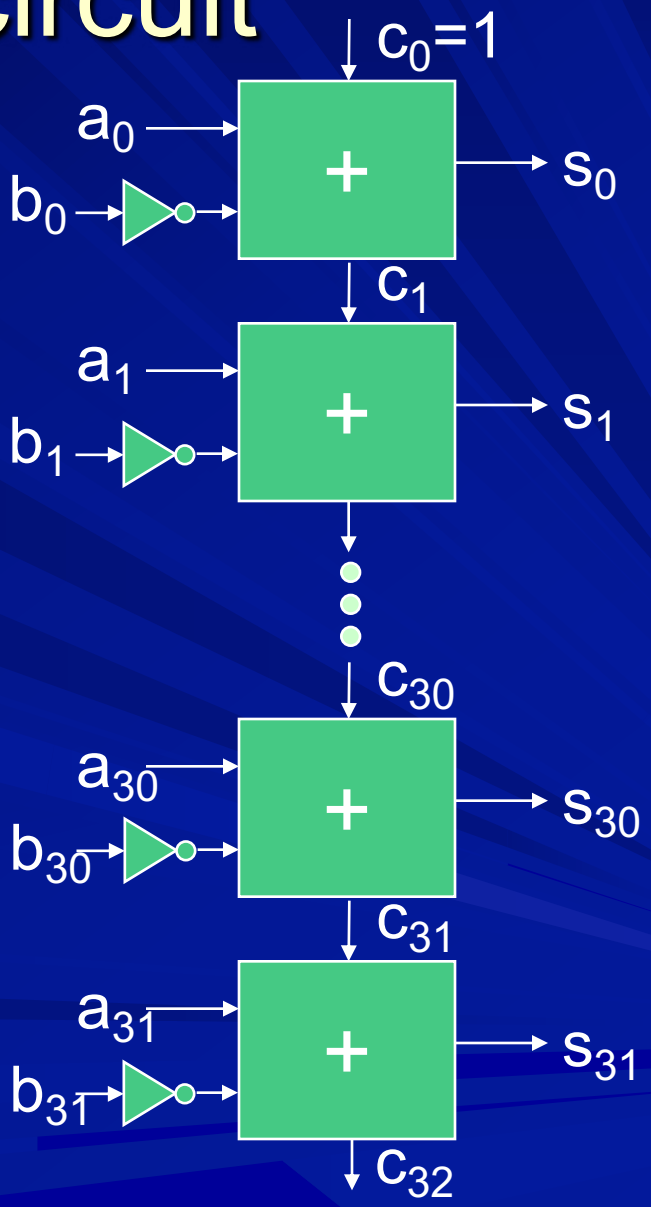


Subtraction circuit

- Use the formula

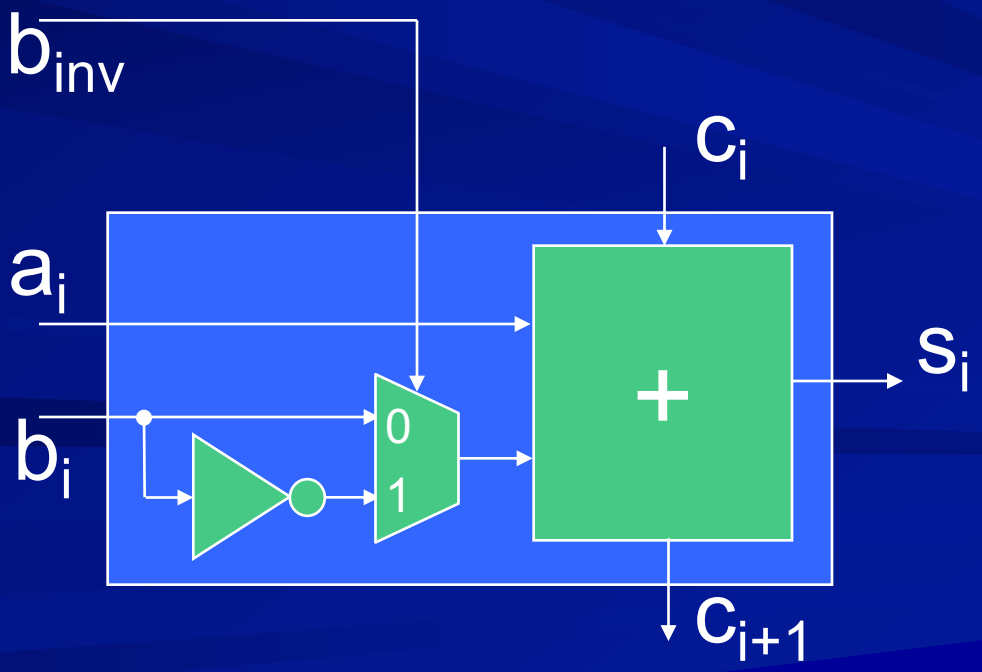
  X-Y = X+Y' +1

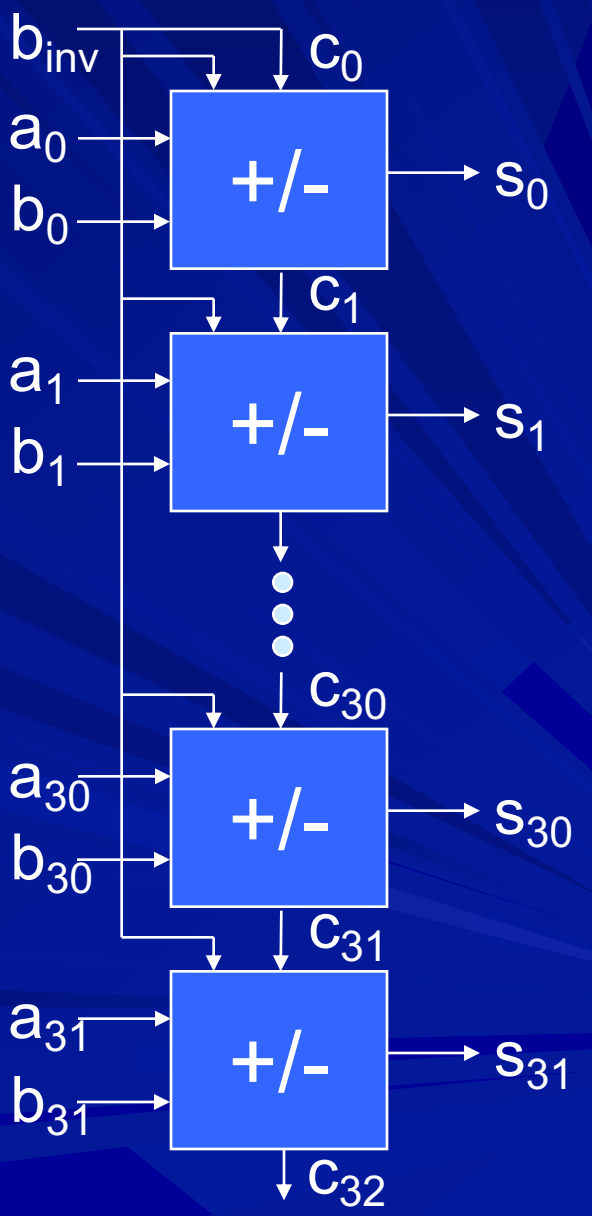




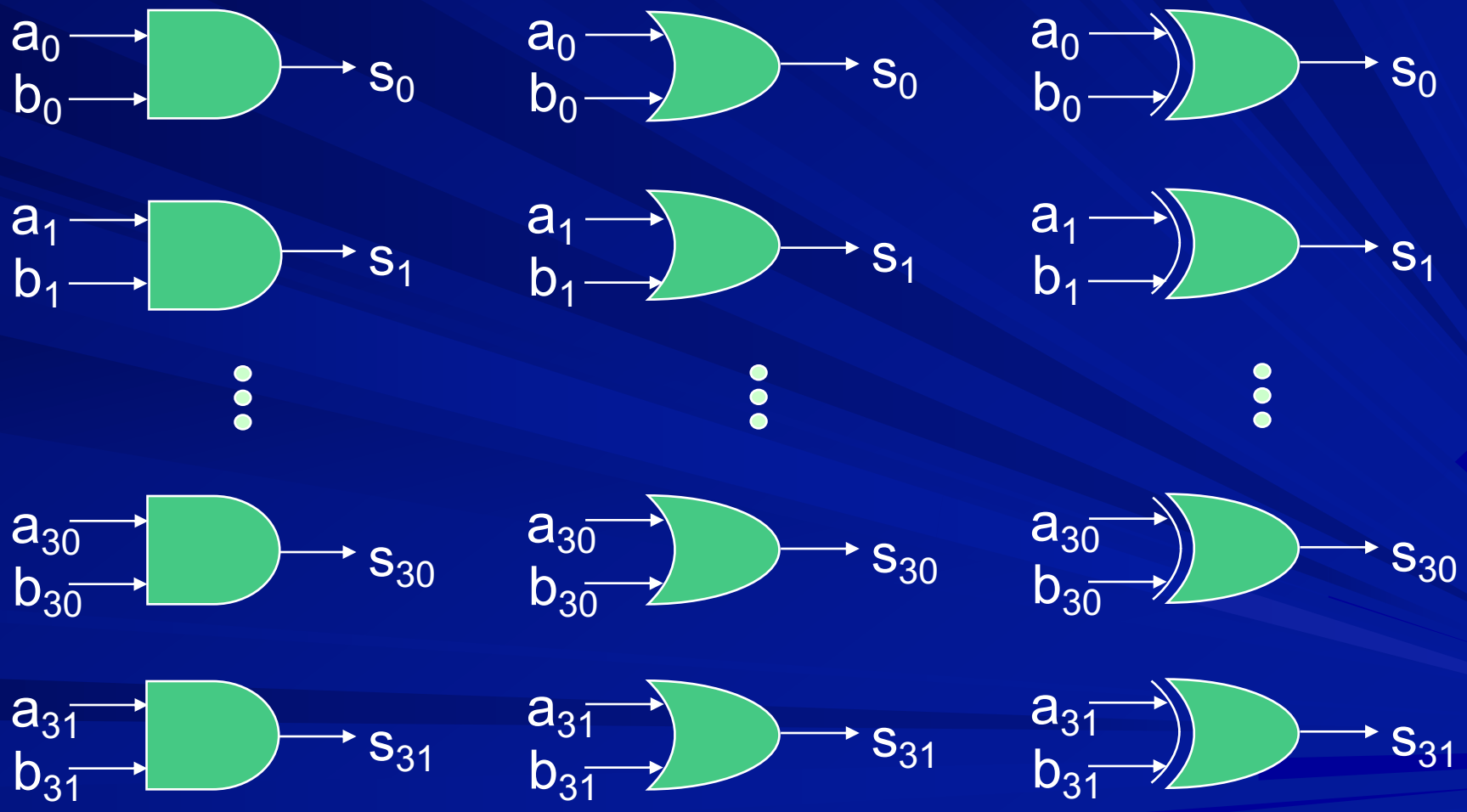
Combining addition and subtraction

- Use a multiplexer circuit

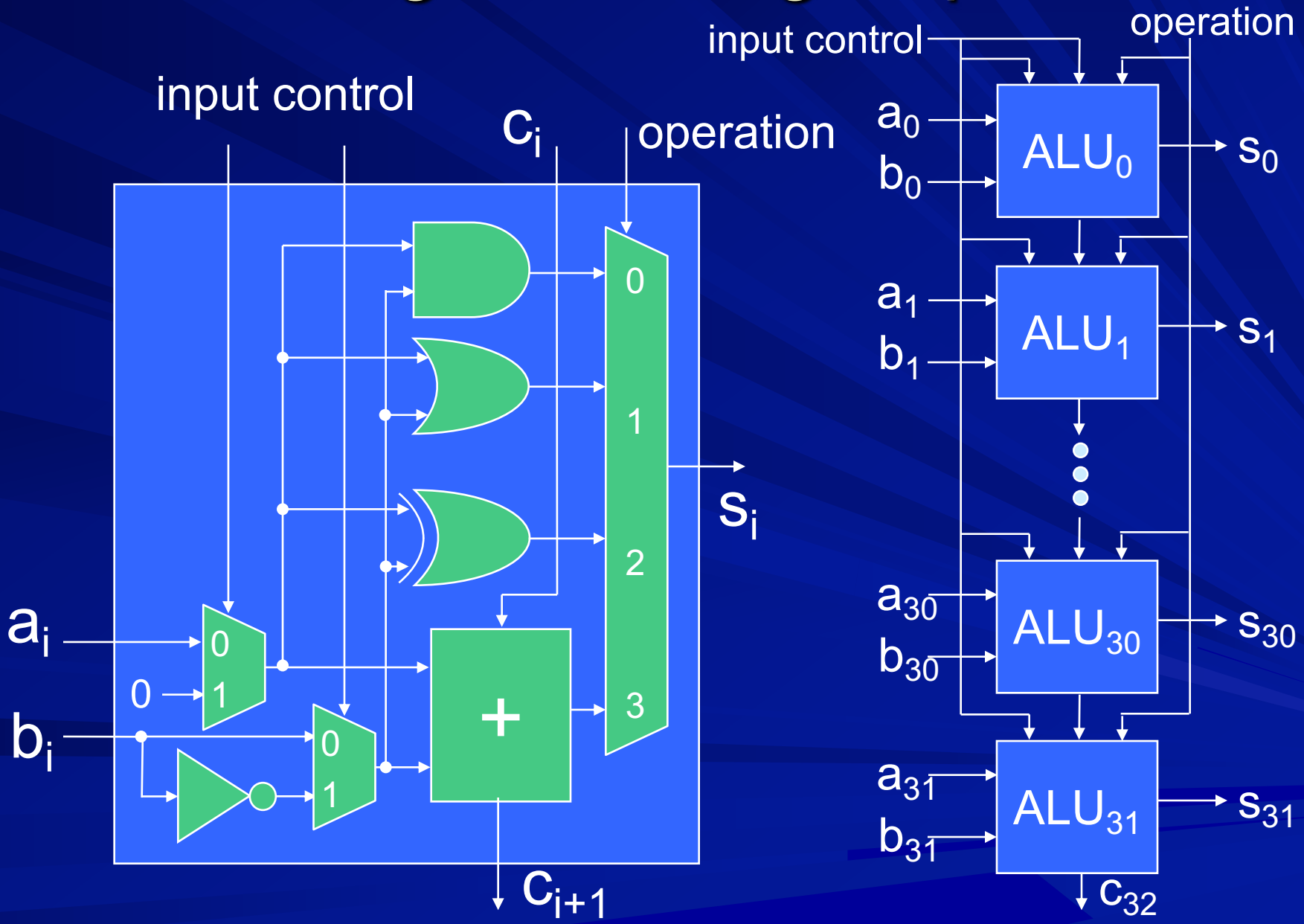




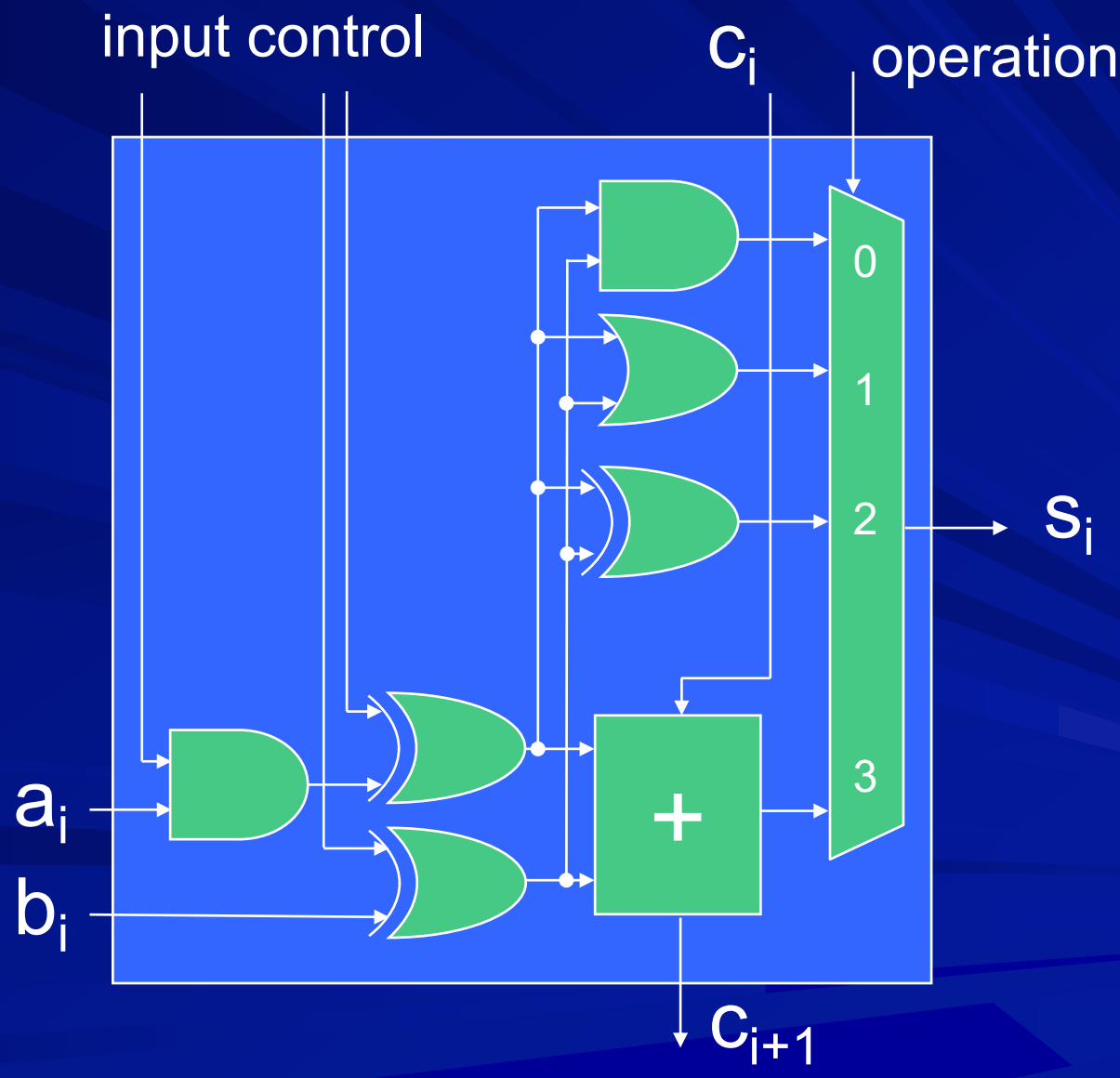
Circuit for and, or, xor



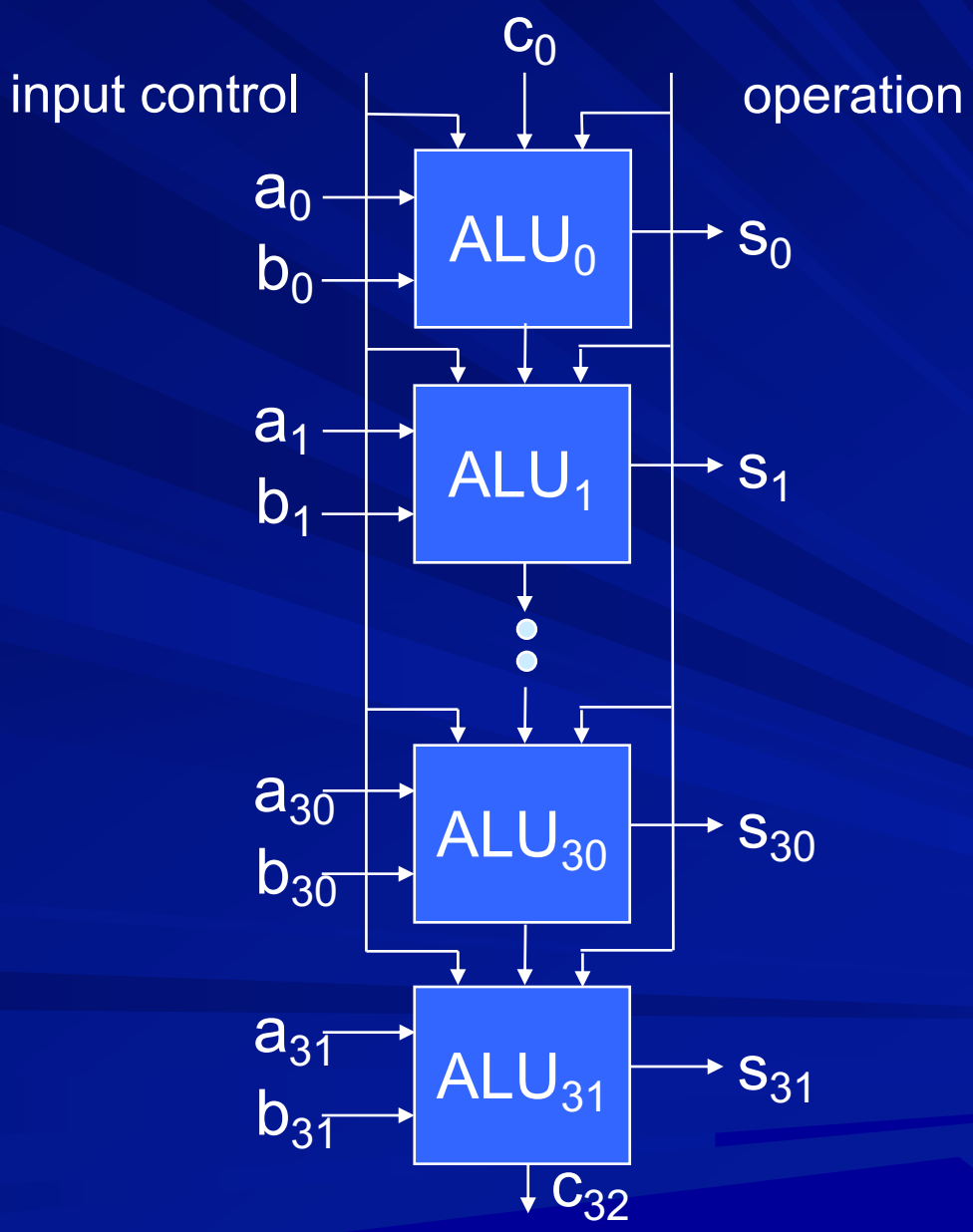
Combining arith & logic operations



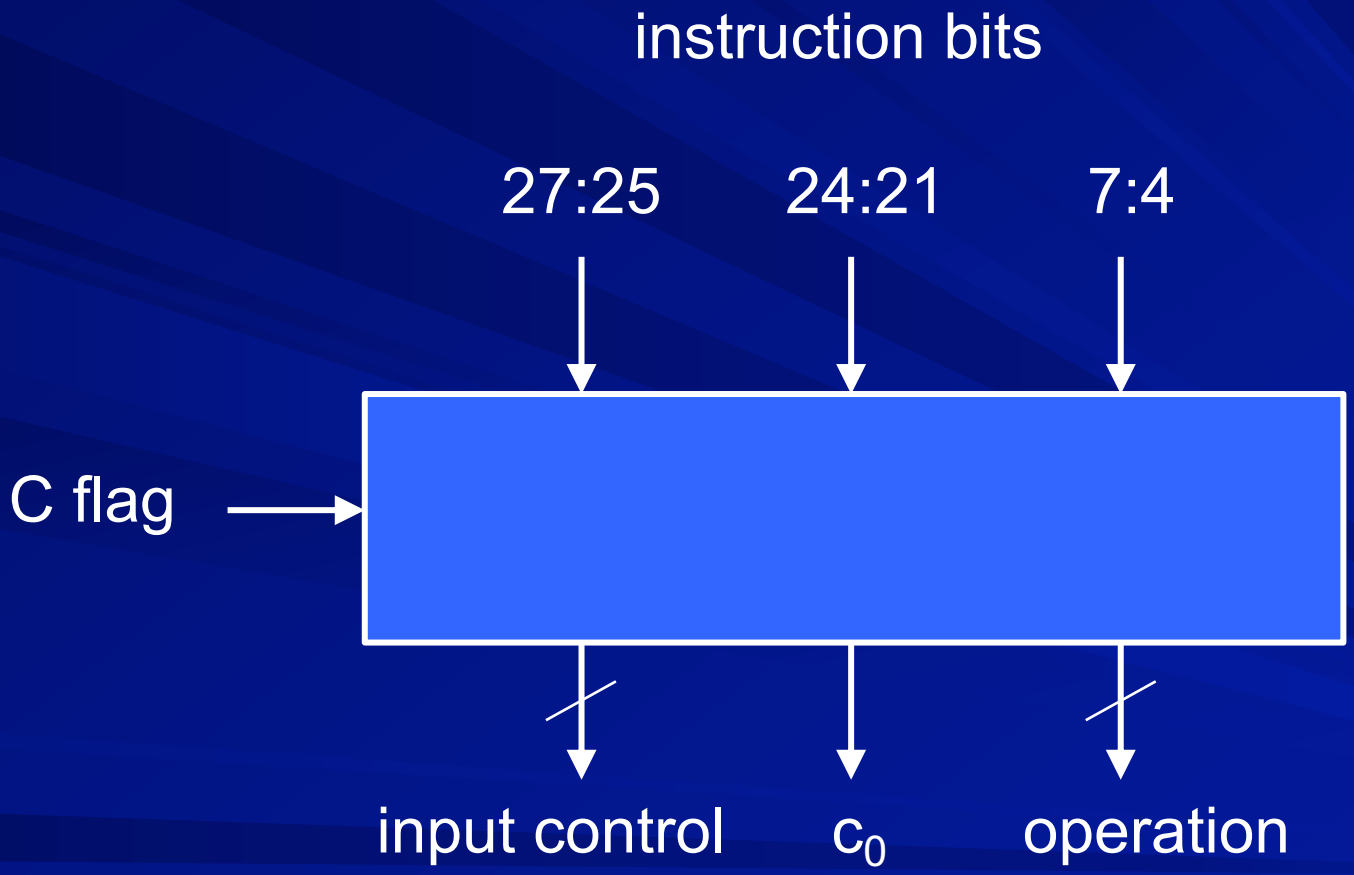
ALU Cell with reverse subtract



ALU – 32 bits

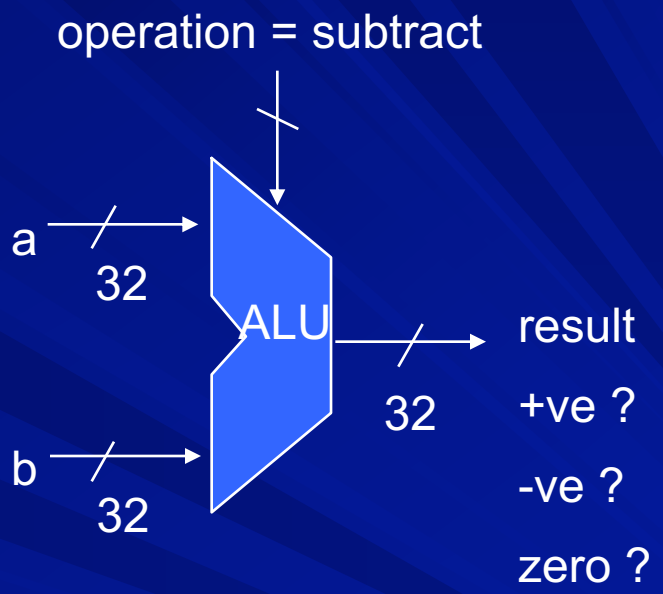

c_0
input control
operation
a_0
b_0
ALU_0
s_0
a_1
b_1
ALU_1
s_1
a_{30}
b_{30}
ALU_{30}
s_{30}
a_{31}
b_{31}
ALU_{31}
s_{31}
c_{32}

ALU control inputs

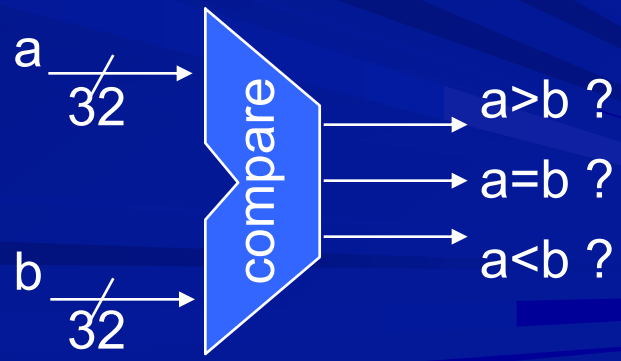


Comparing two integers

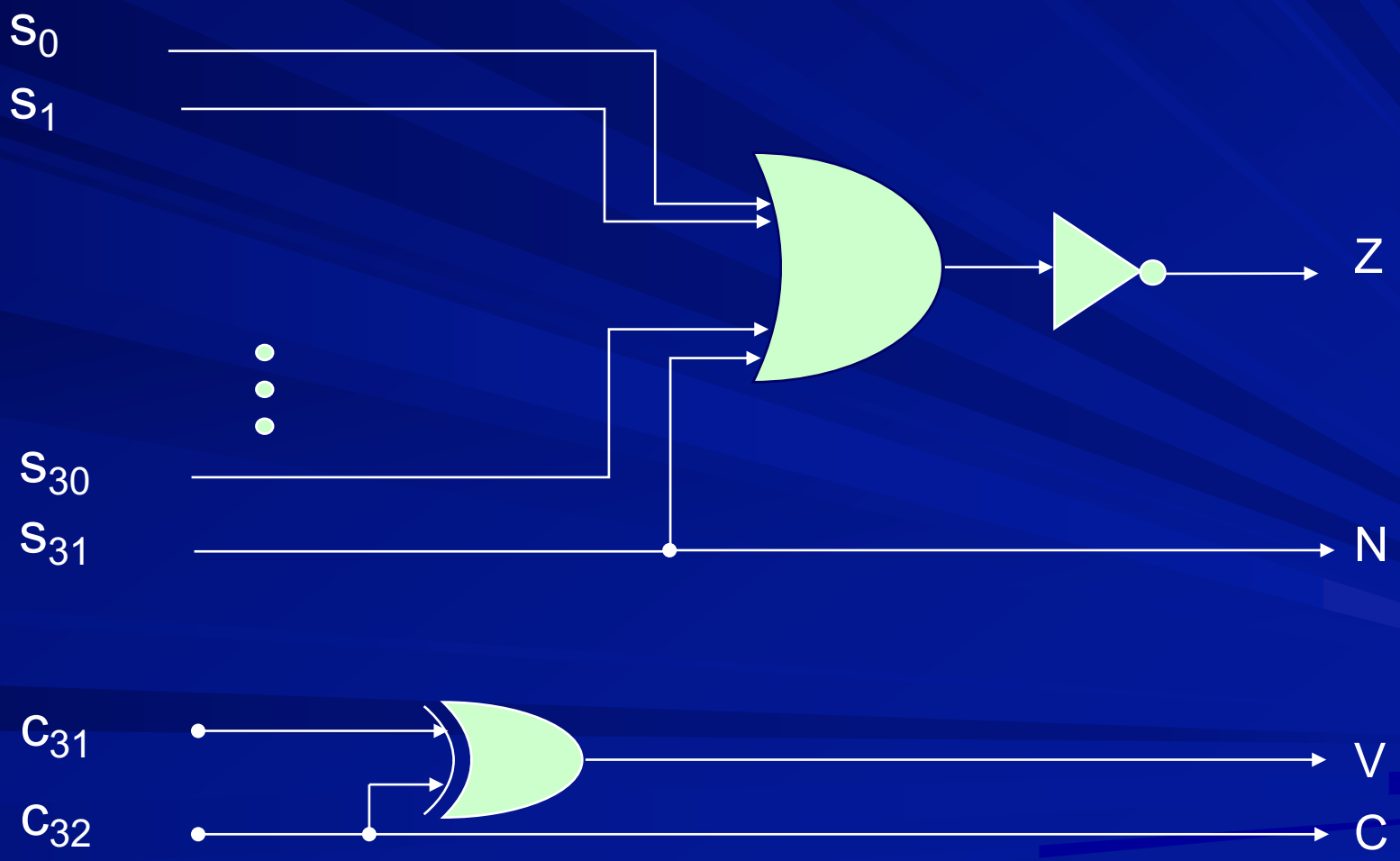
- Subtract and check the result



- Compare directly



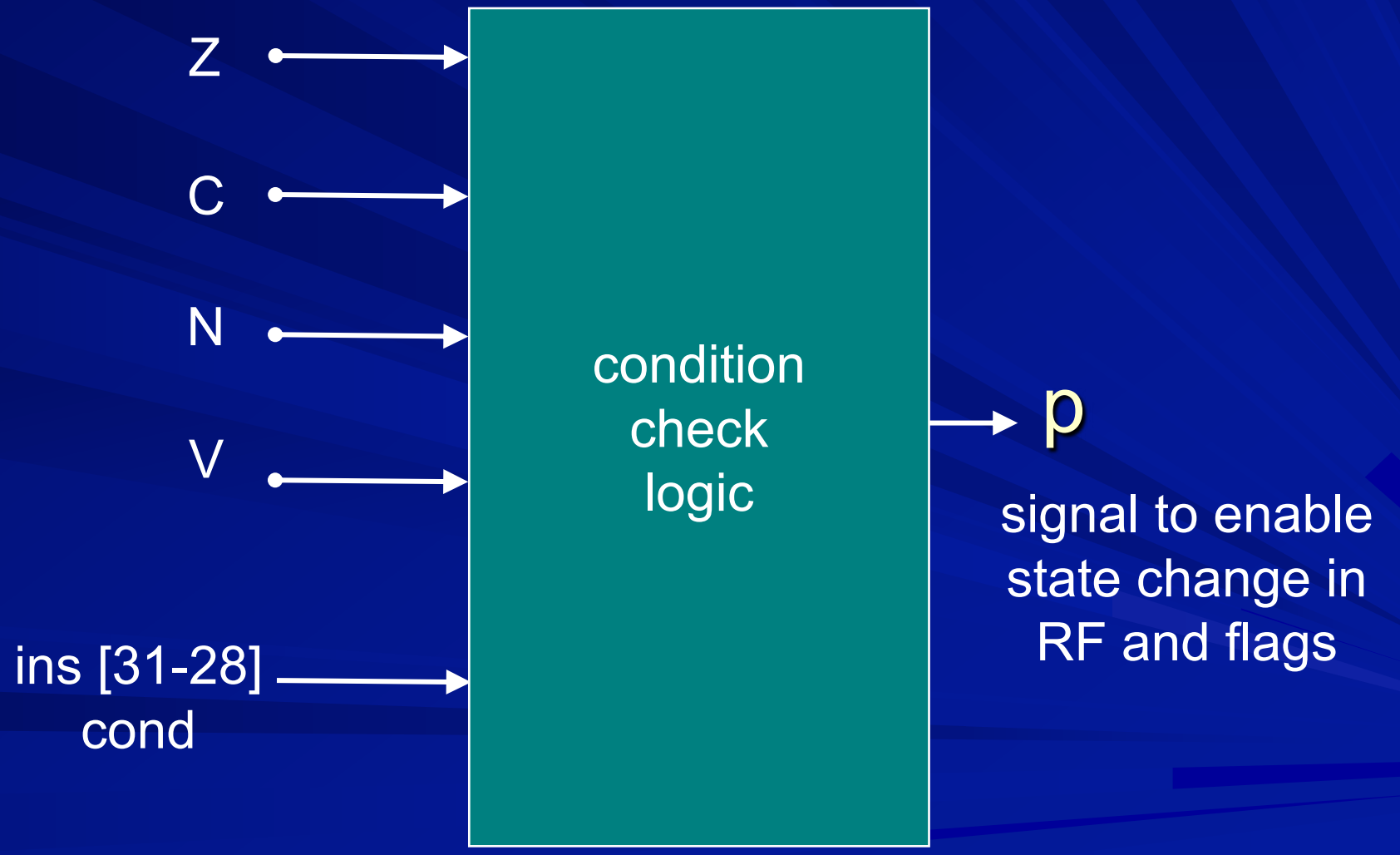
Setting flags



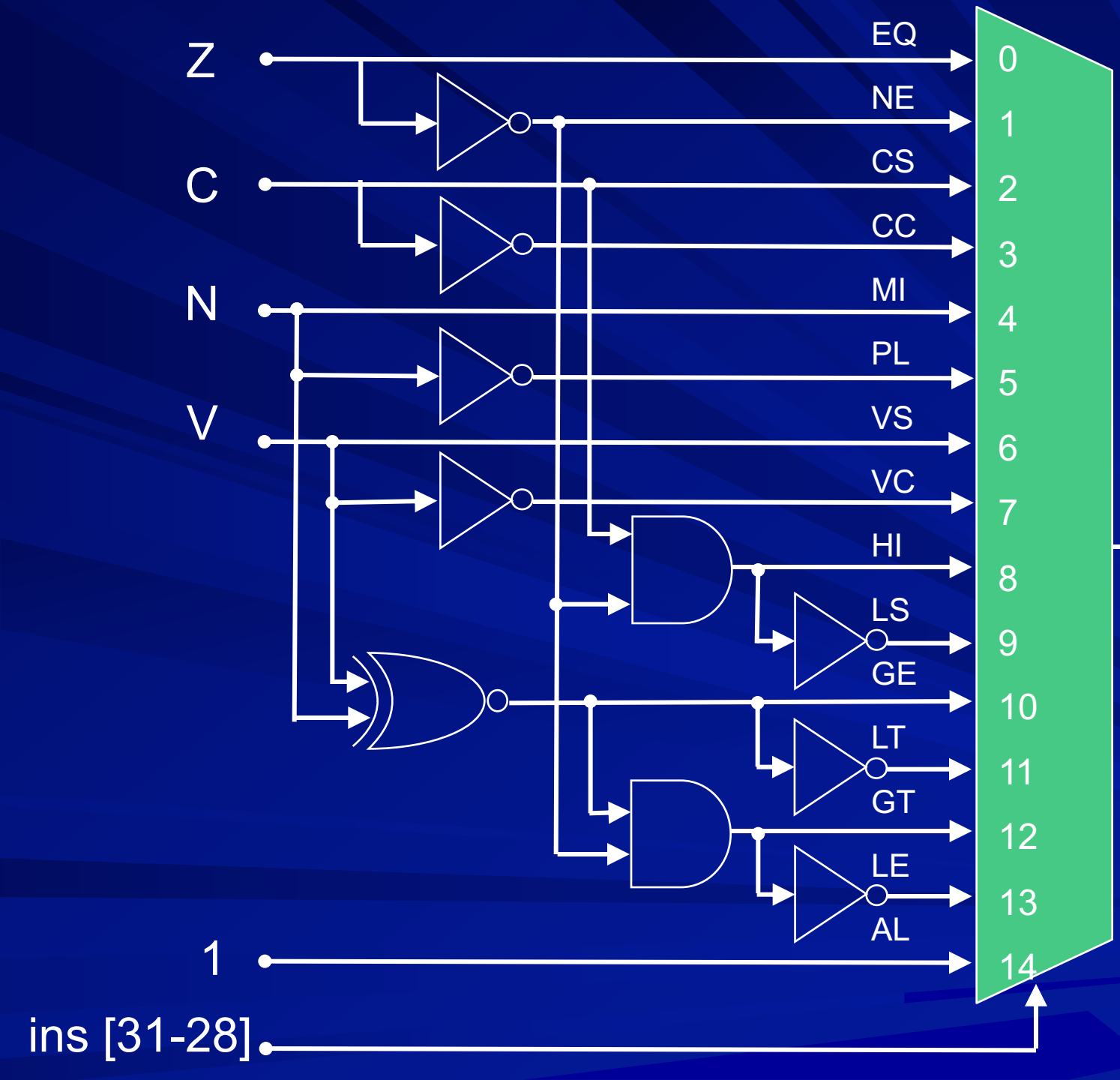
Predication

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

Predication



Predi-cation



Instructions doable by earlier ALU

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

Additional DP instructions to be done

- adc, sbc
- rsb, rsc

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

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

Op2 variants

- Rm

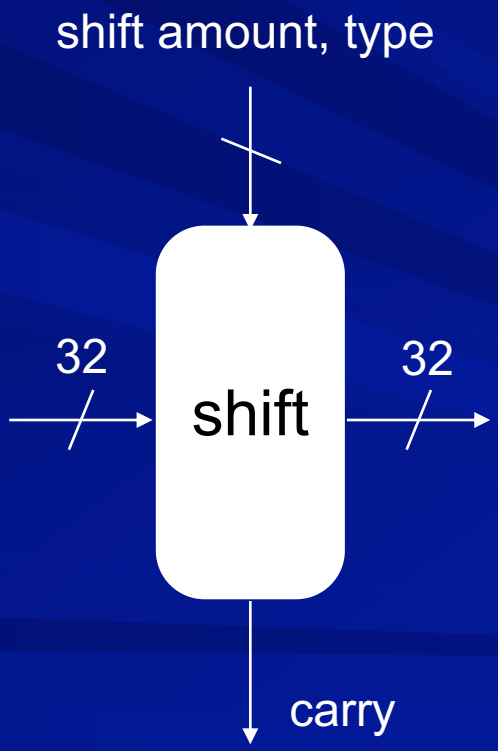
⇒Rm, <Shift type>, Rs

⇒Rm, <Shift type>, #constant

- #constant

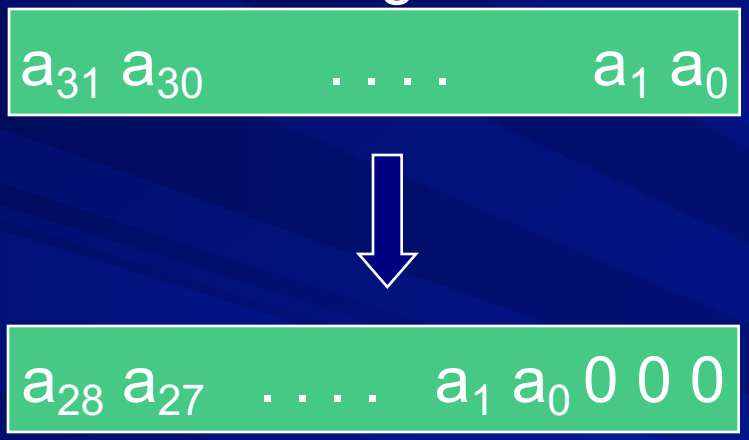
⇒#constant, ROR, constant

Shifting and rotation



Shift operations

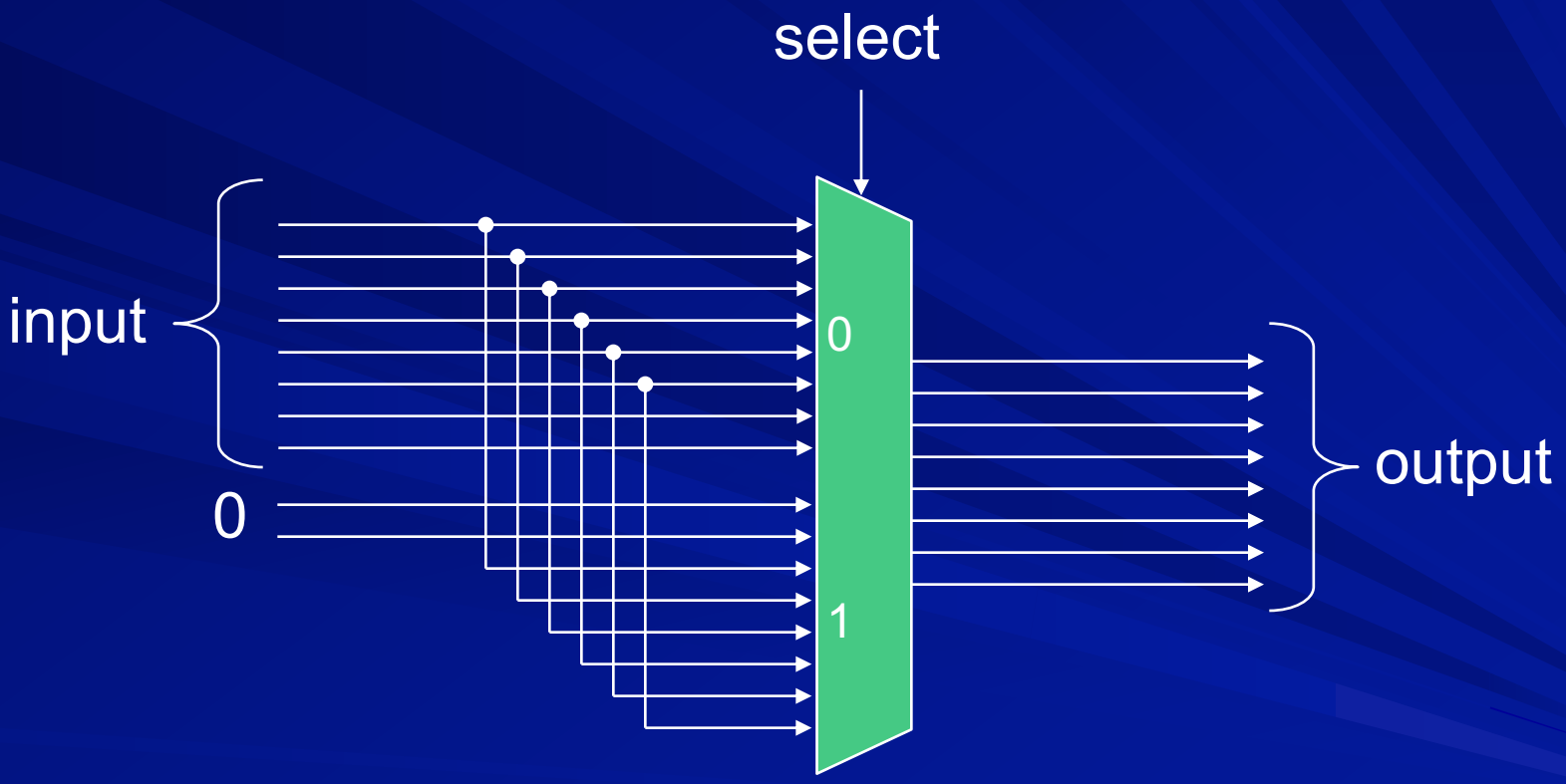
shift left logical 3 bits



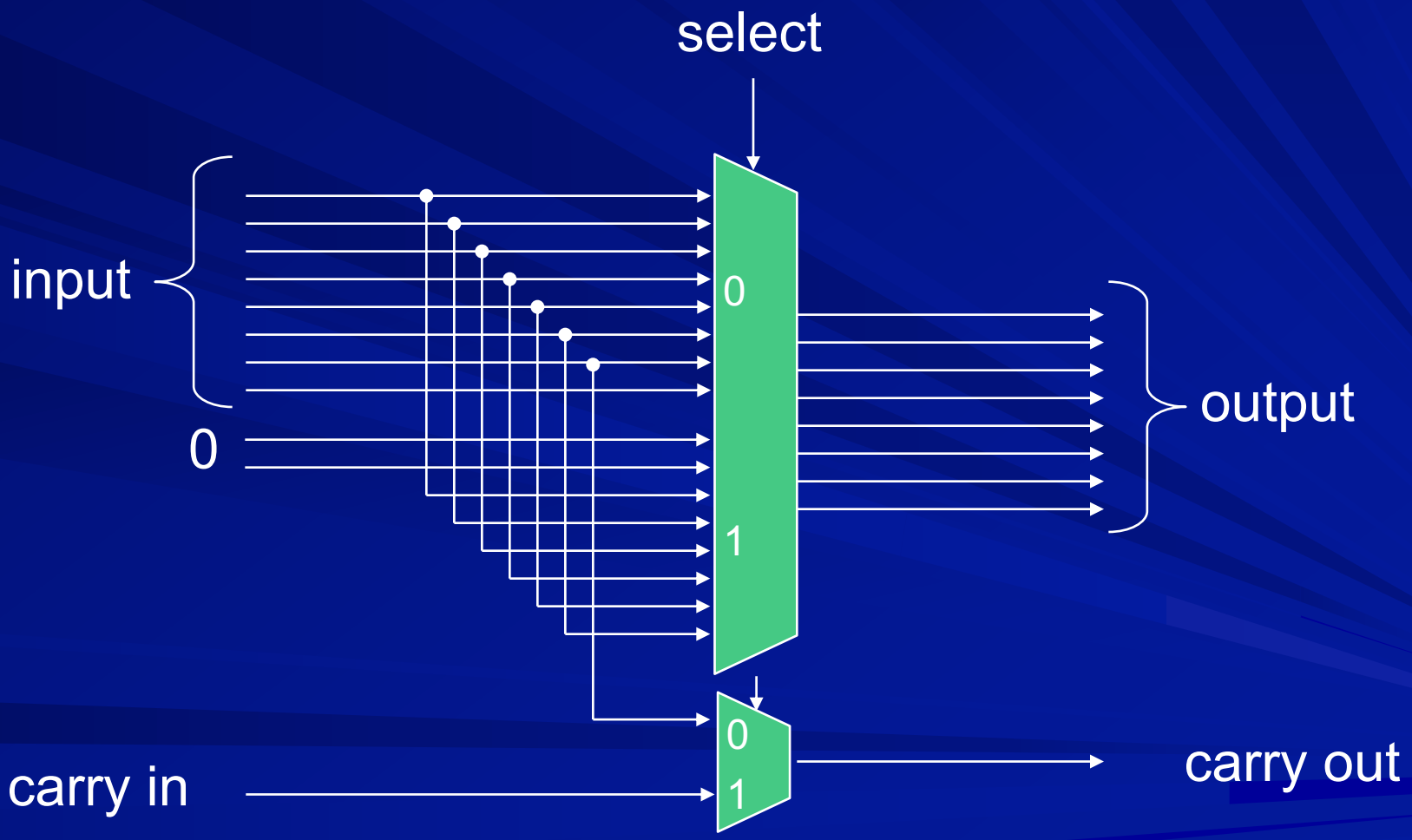
shift right logical 3 bits

shift right arithmetic 2 bits

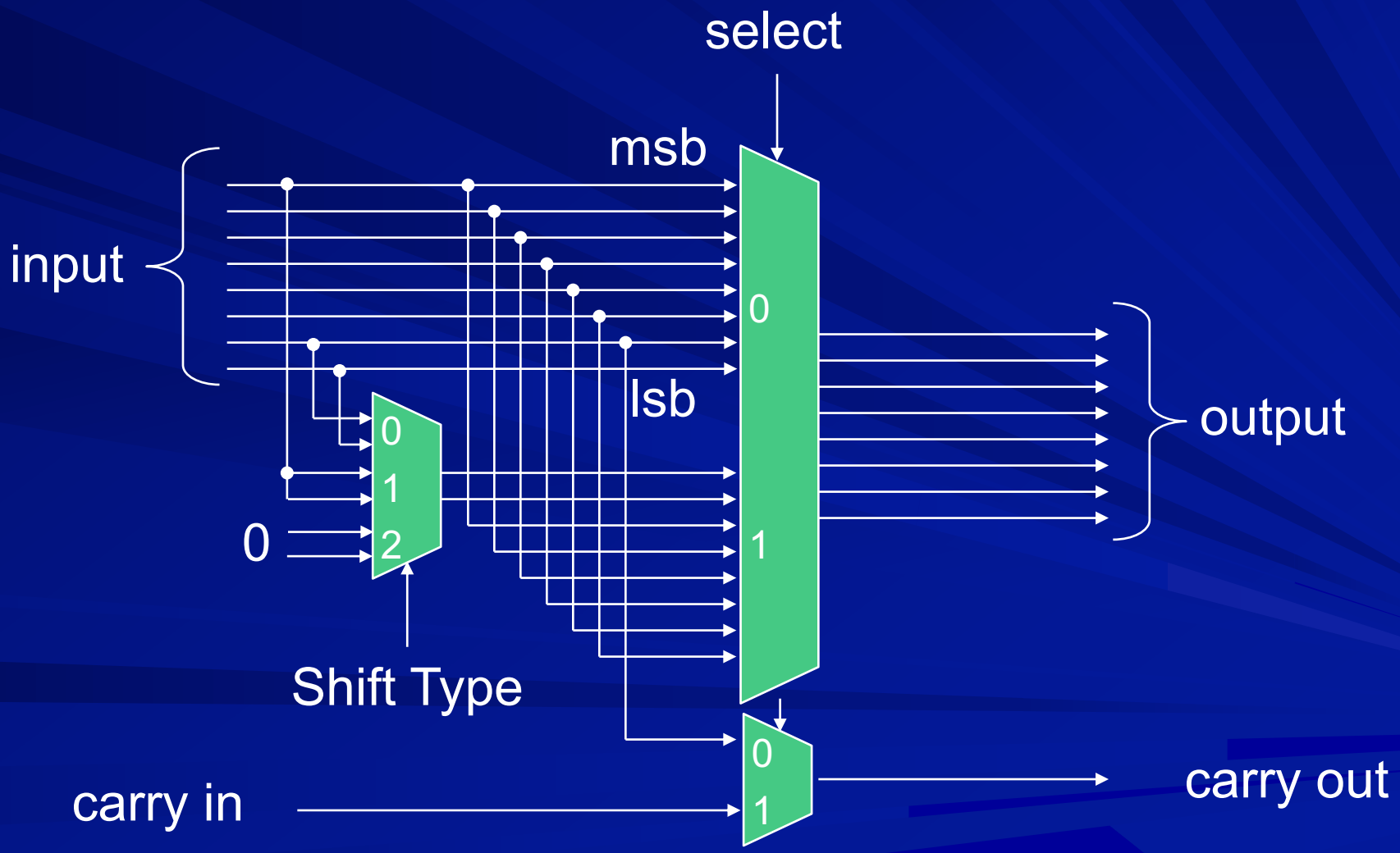
Circuit for shifting by 2 bits



Circuit for shifting by 2 bits

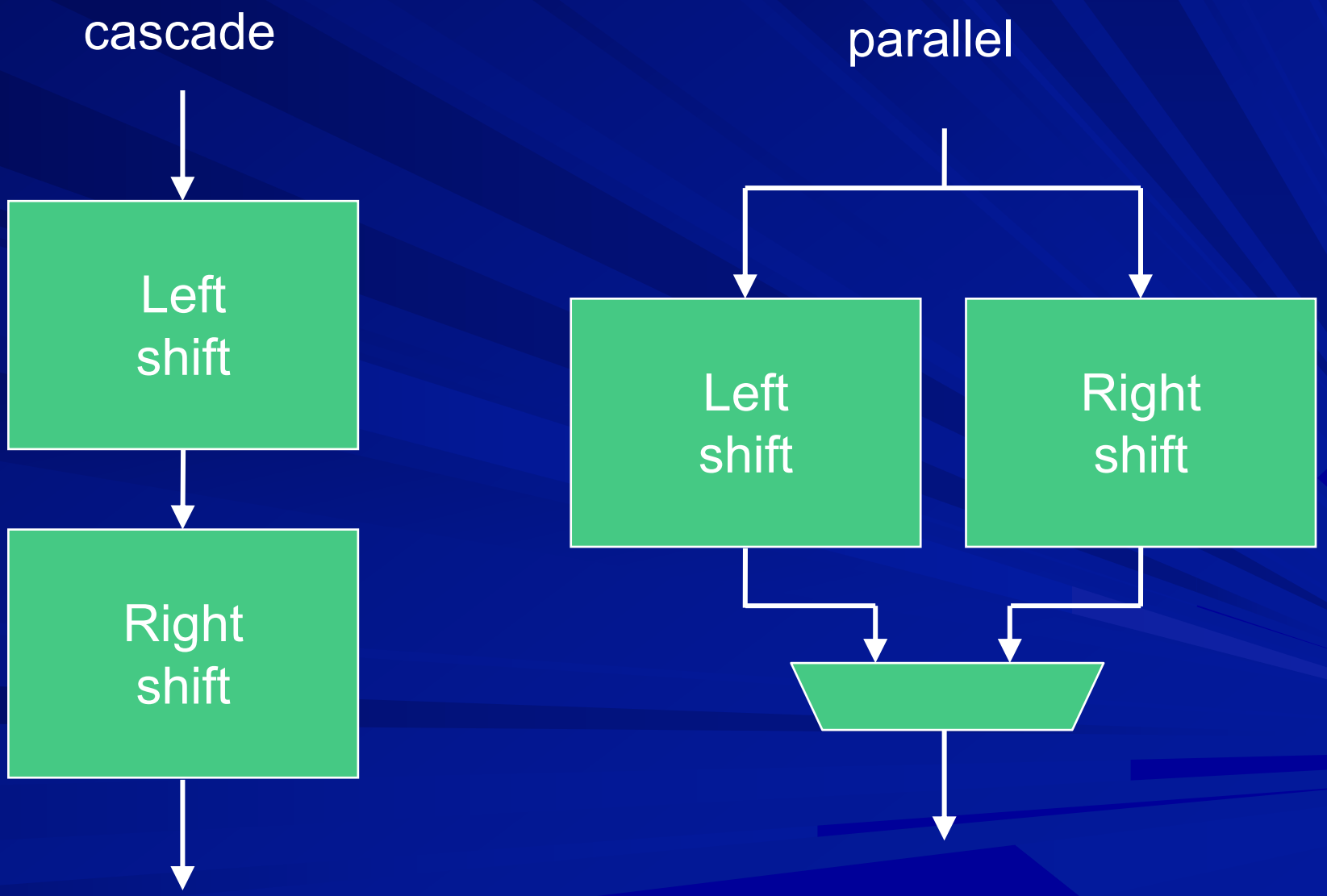


Combining ROR, ASR, LSR

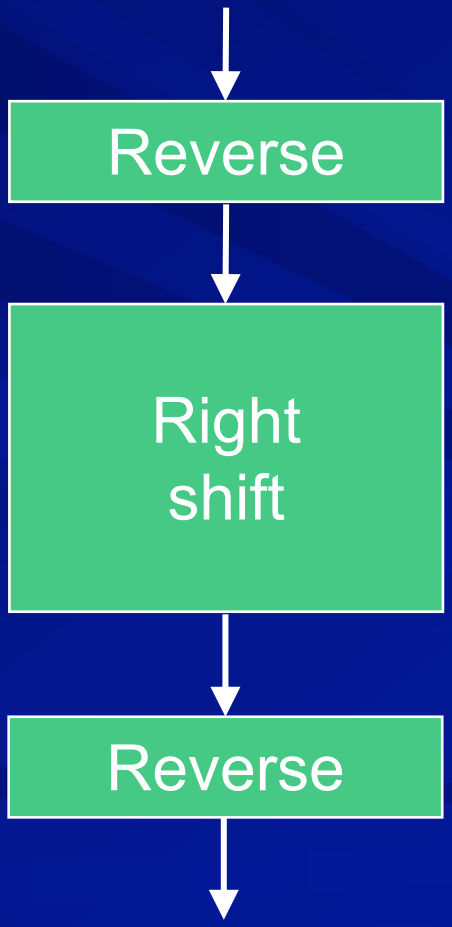


Circuit for shifting by 0 to 31 bits

Combining Left and Right Shift



Combining Left and Right Shift



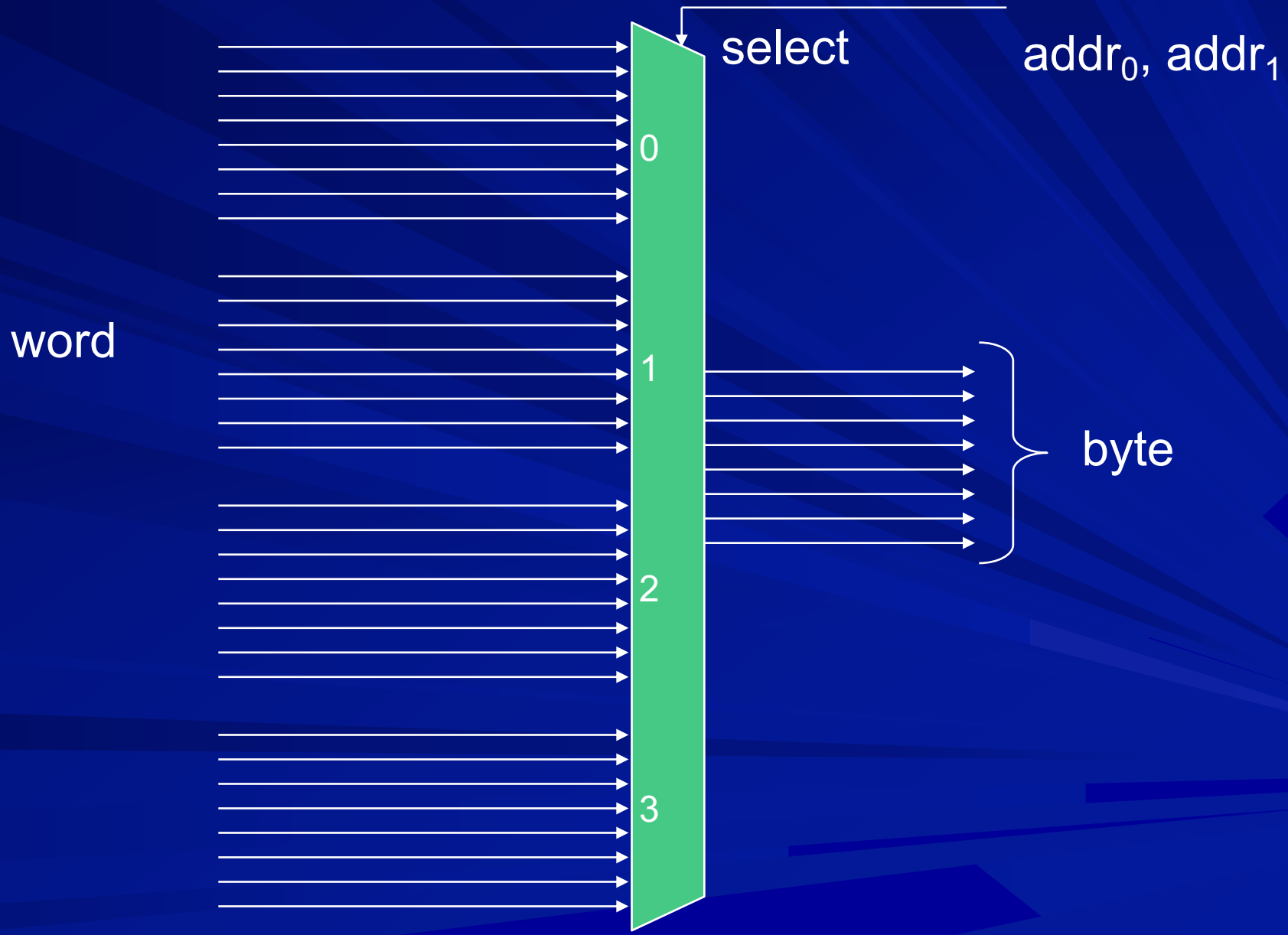
Pre/post increment/decrement

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

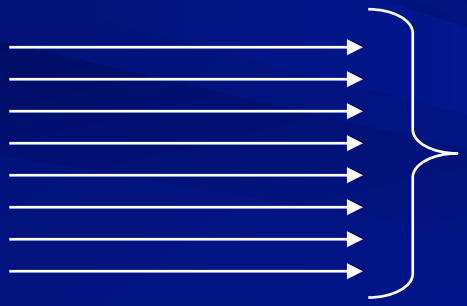
Auto increment/decrement

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

Word / half-word / byte transfer

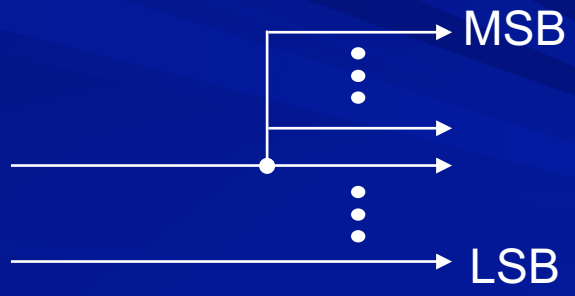


Word / half-word / byte transfer



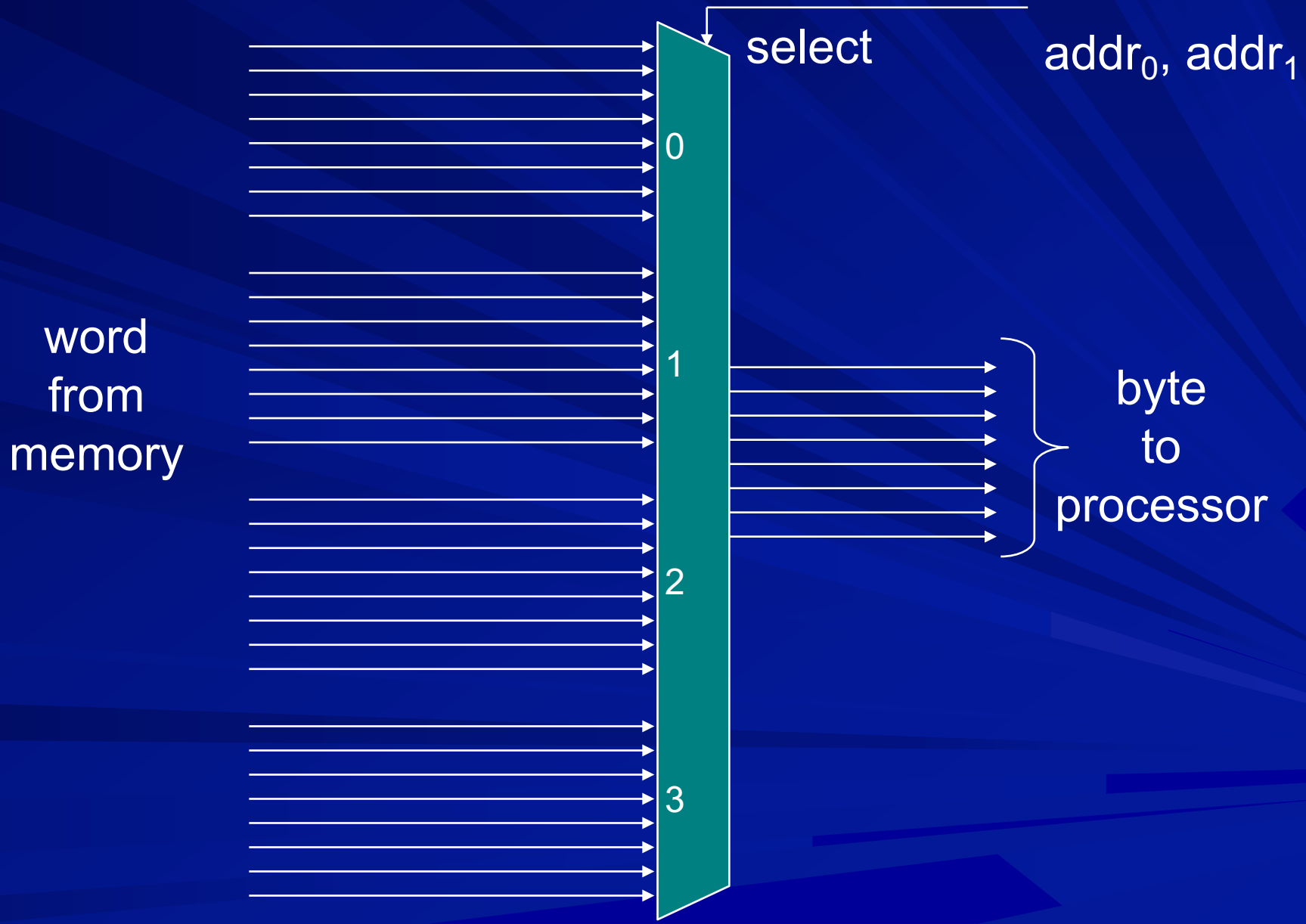
byte

zero extension

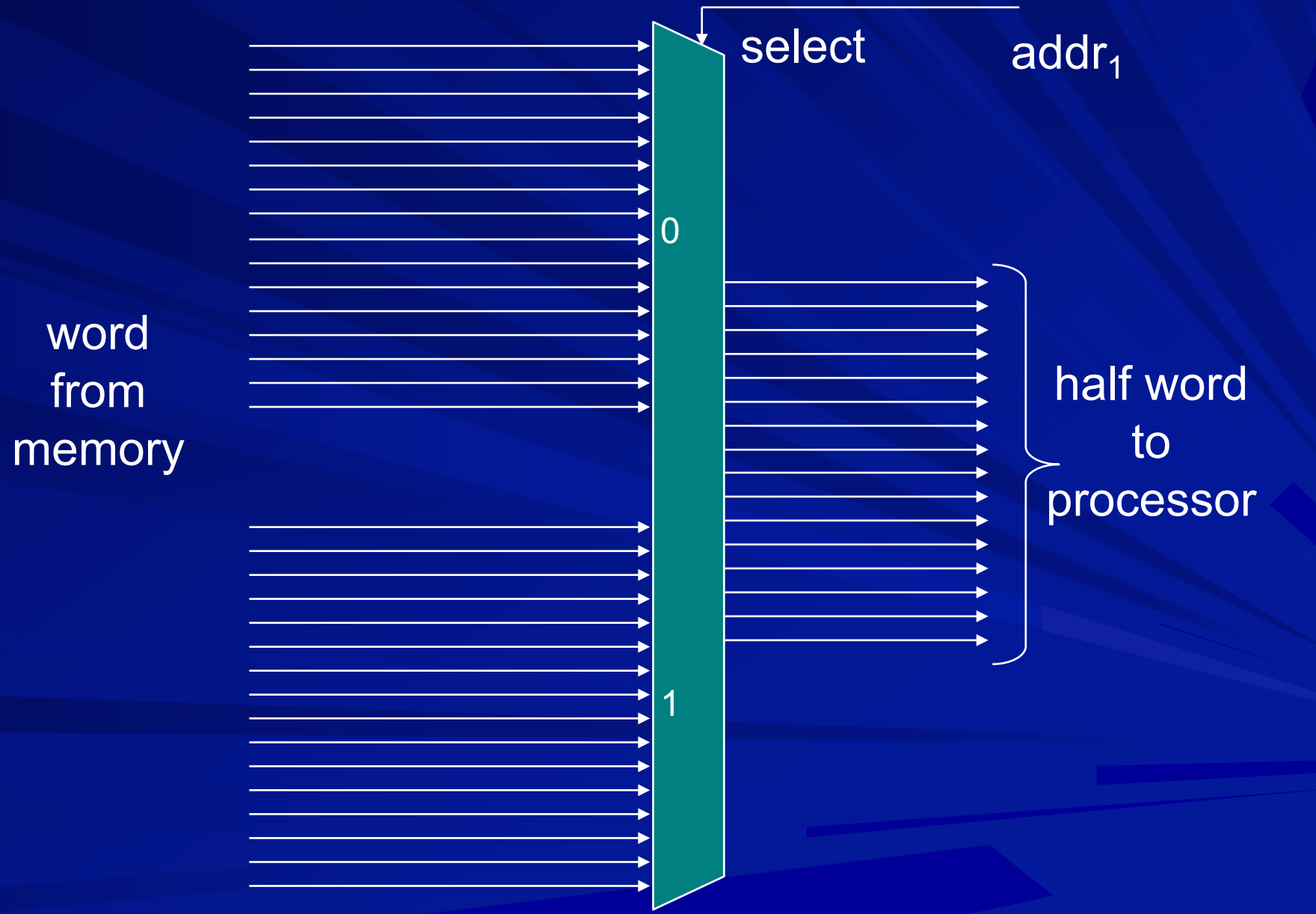


sign extension

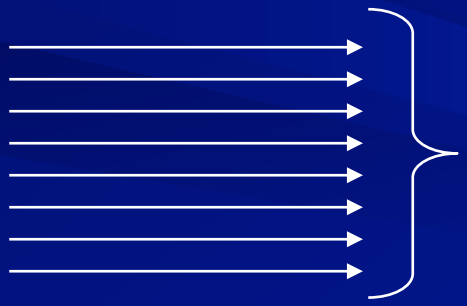
Selecting byte for ldrb / ldrsb



Selecting half word for ldrh / ldrsh

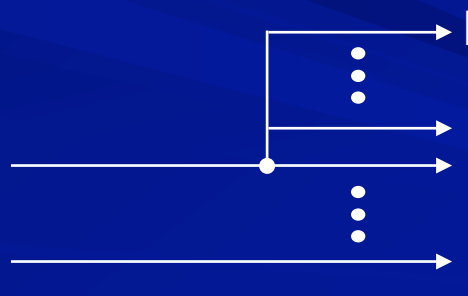


Extending from byte to word



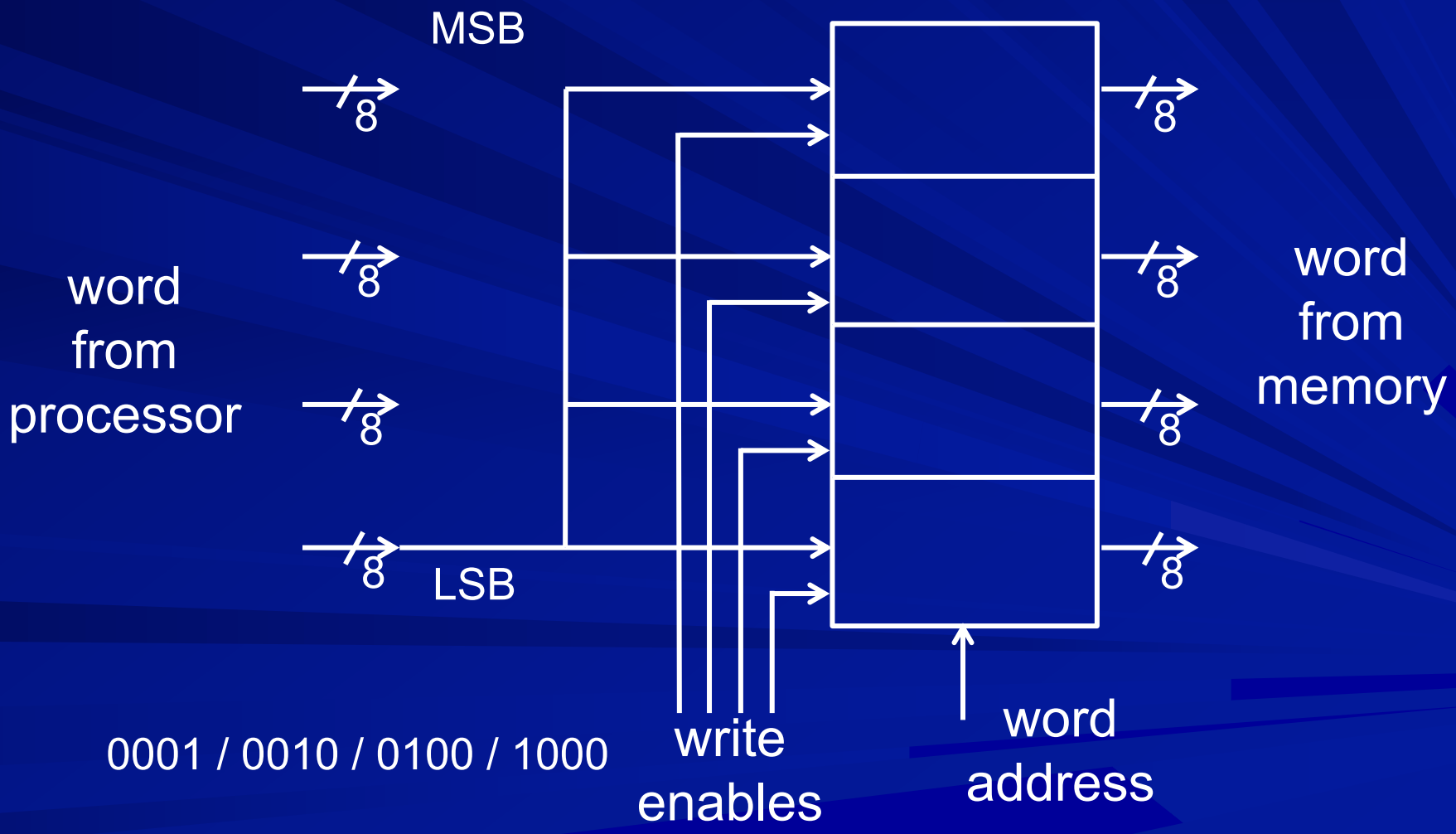
byte

zero extension

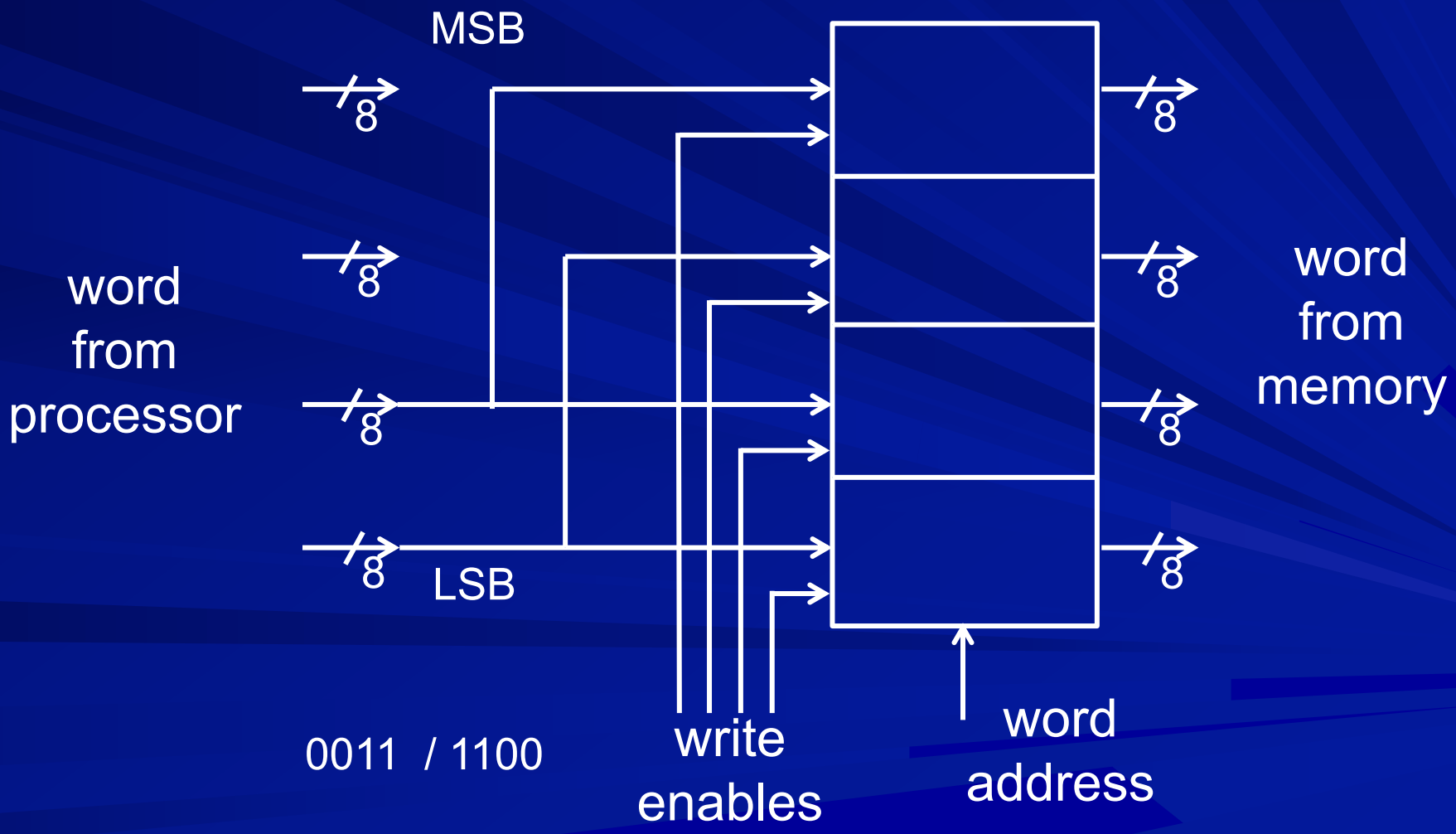


sign extension

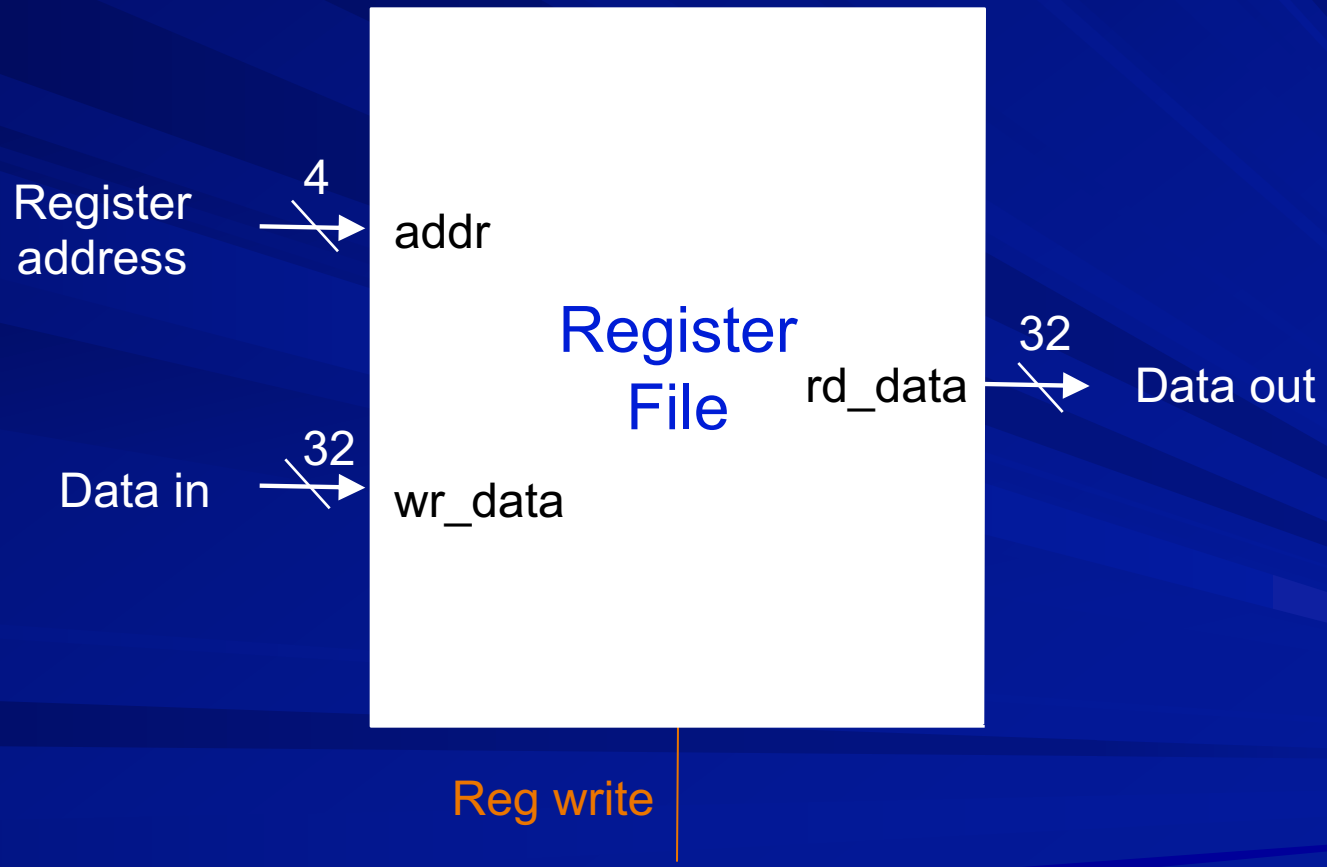
Writing a byte in memory



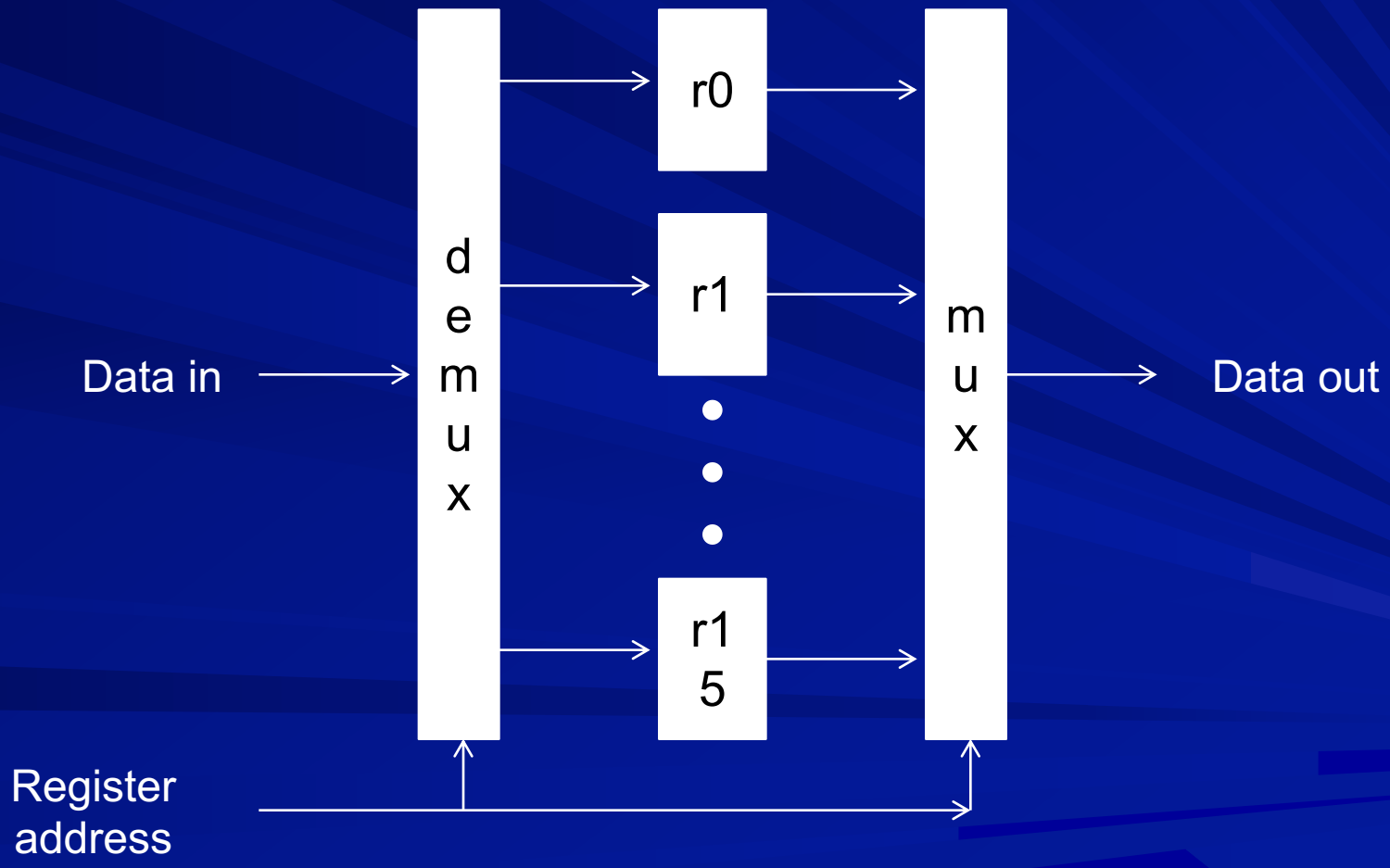
Writing a half word in memory



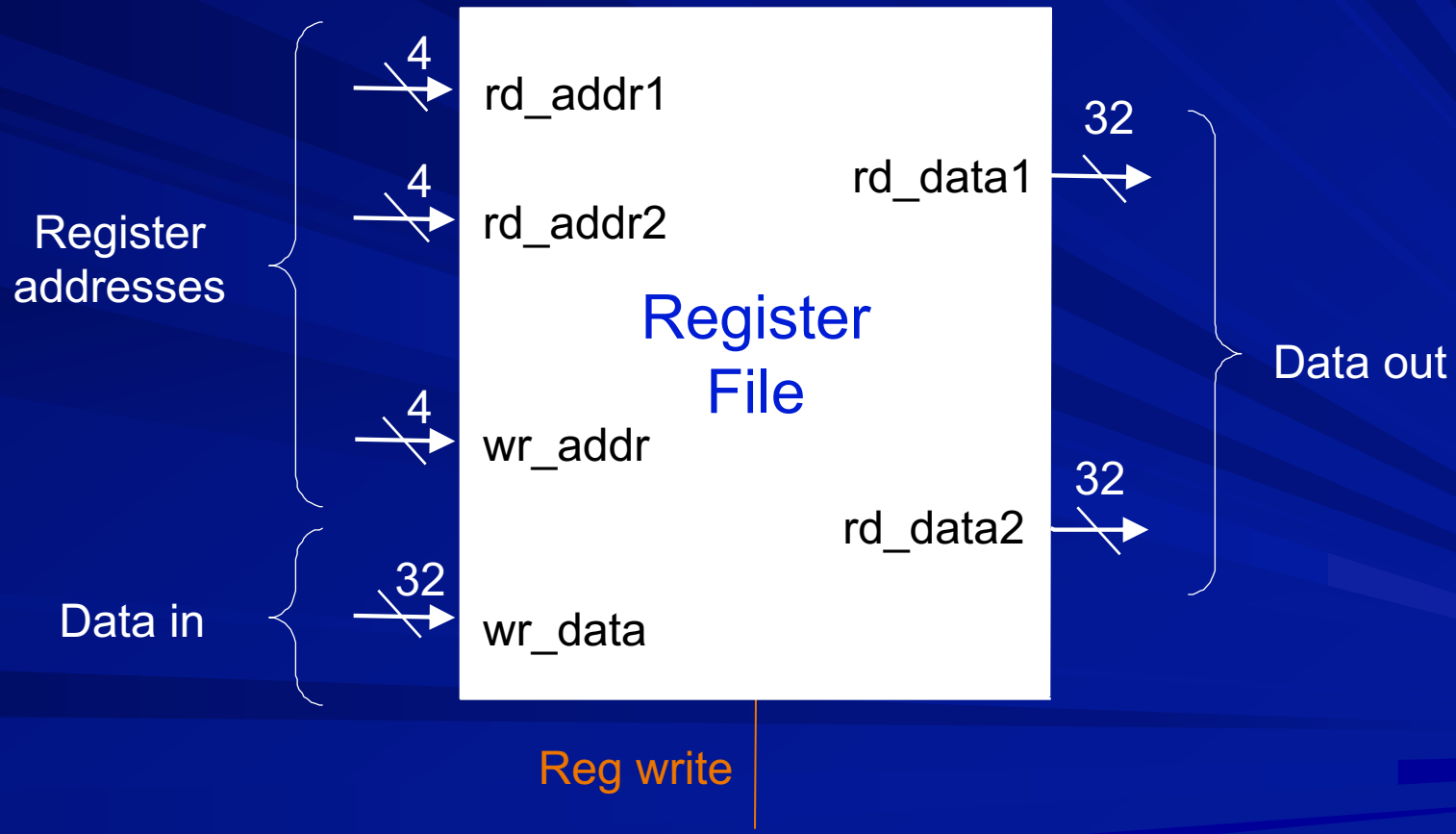
Register file



Register file



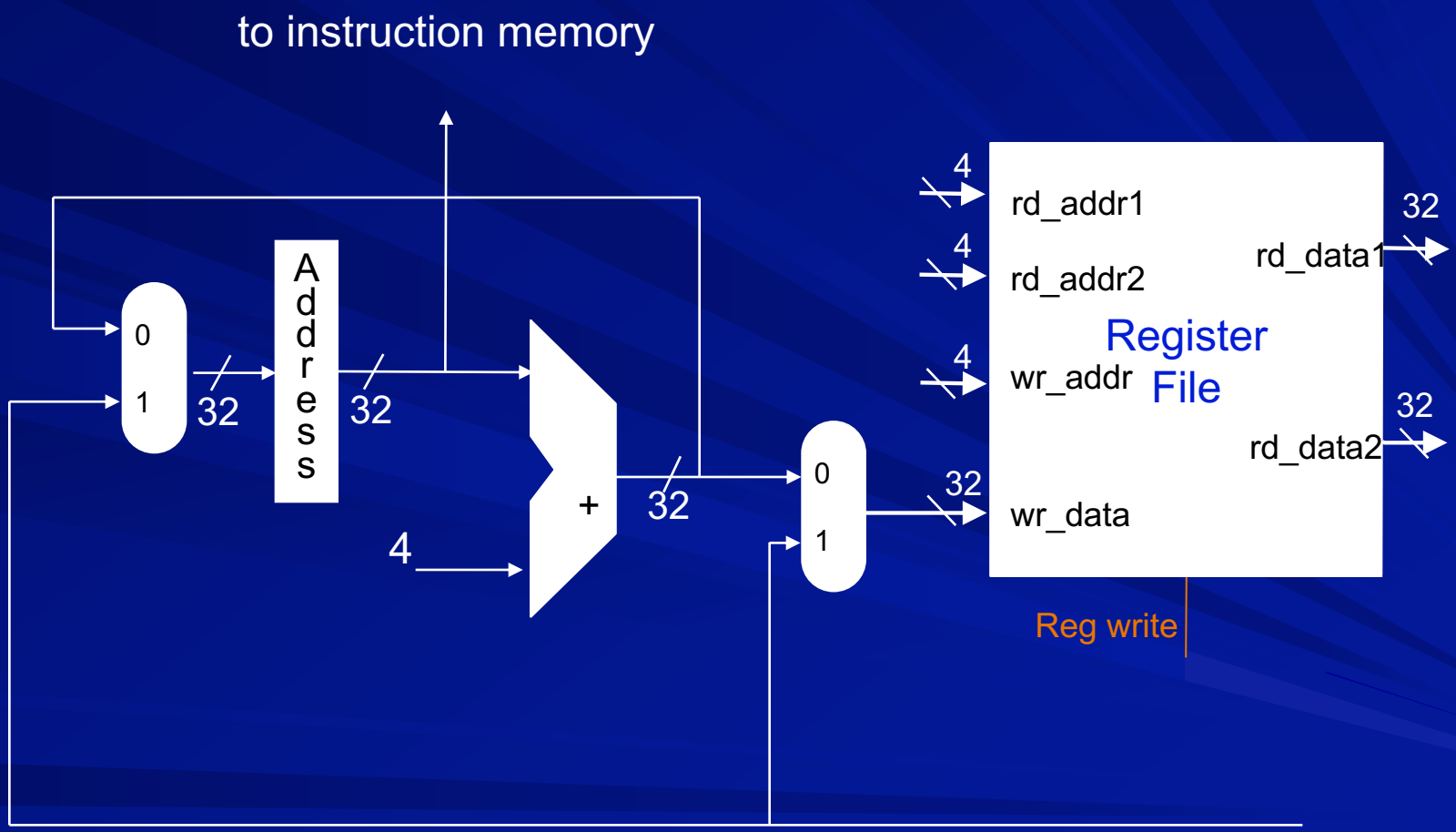
Register file (2R + 1W ports)



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



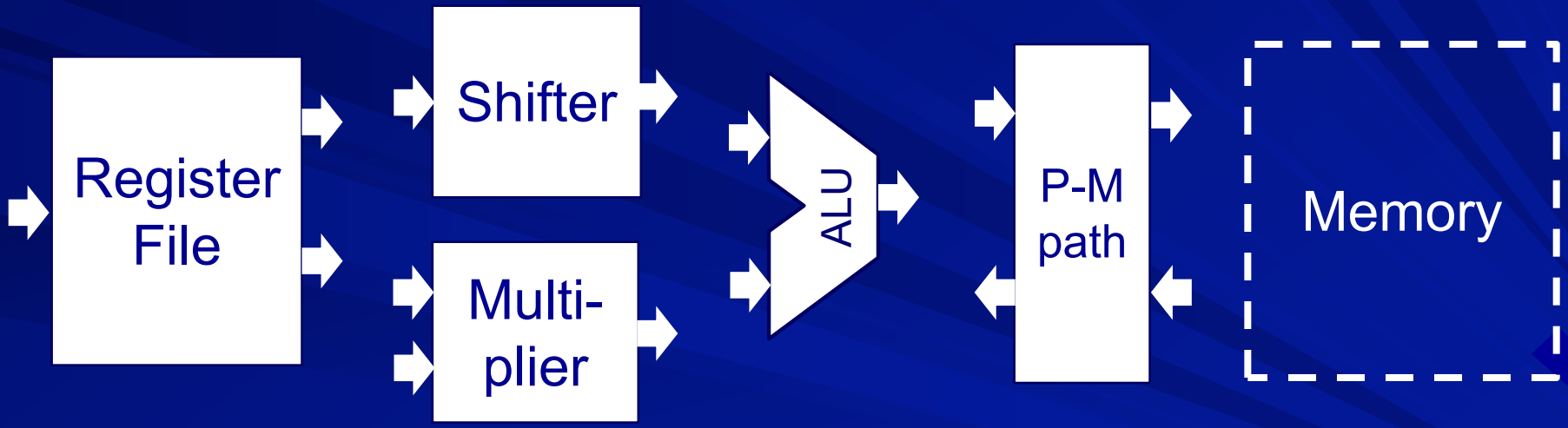
Thanks

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 Multiply
- MLA 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

$a_3\ a_2\ a_1\ a_0$ x b_0

$a_3\ a_2\ a_1\ a_0$ x b_1

$a_3\ a_2\ a_1\ a_0$ x b_2

$a_3\ a_2\ a_1\ a_0$ x b_3

4x4 unsigned

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

4x4 signed

a_3 a_3 a_3 a_3 a_2 a_1 a_0	x b_0
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
a_3 a_2 a_1 a_0	x $-b_3$

Multiplying a number by -1

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

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

Then

$$X = x_{n-1}\ x_{n-1} \ldots\ldots\ldots\ldots x_k\ 1\ 0\ .\ .\ 0\ 0$$

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

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

4x4 unsigned

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

4x4 signed

a_3 a_3 a_3 a_3 a_2 a_1 a_0	x b_0
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
$\overline{a_3}$ $\overline{a_2}$ a_1 a_0	x b_3

4x4 unsigned

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

4x4 signed

$a_3\ a_3\ a_3$	$a_3\ a_2\ a_1\ a_0$	x b_0
$a_3\ a_3\ a_3$	$a_2\ a_1\ a_0$	x b_1
$a_3\ a_3\ a_2$	$a_1\ a_0$	x b_2
$\overline{a_3}\ \overline{a_2}\ a_1$	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)

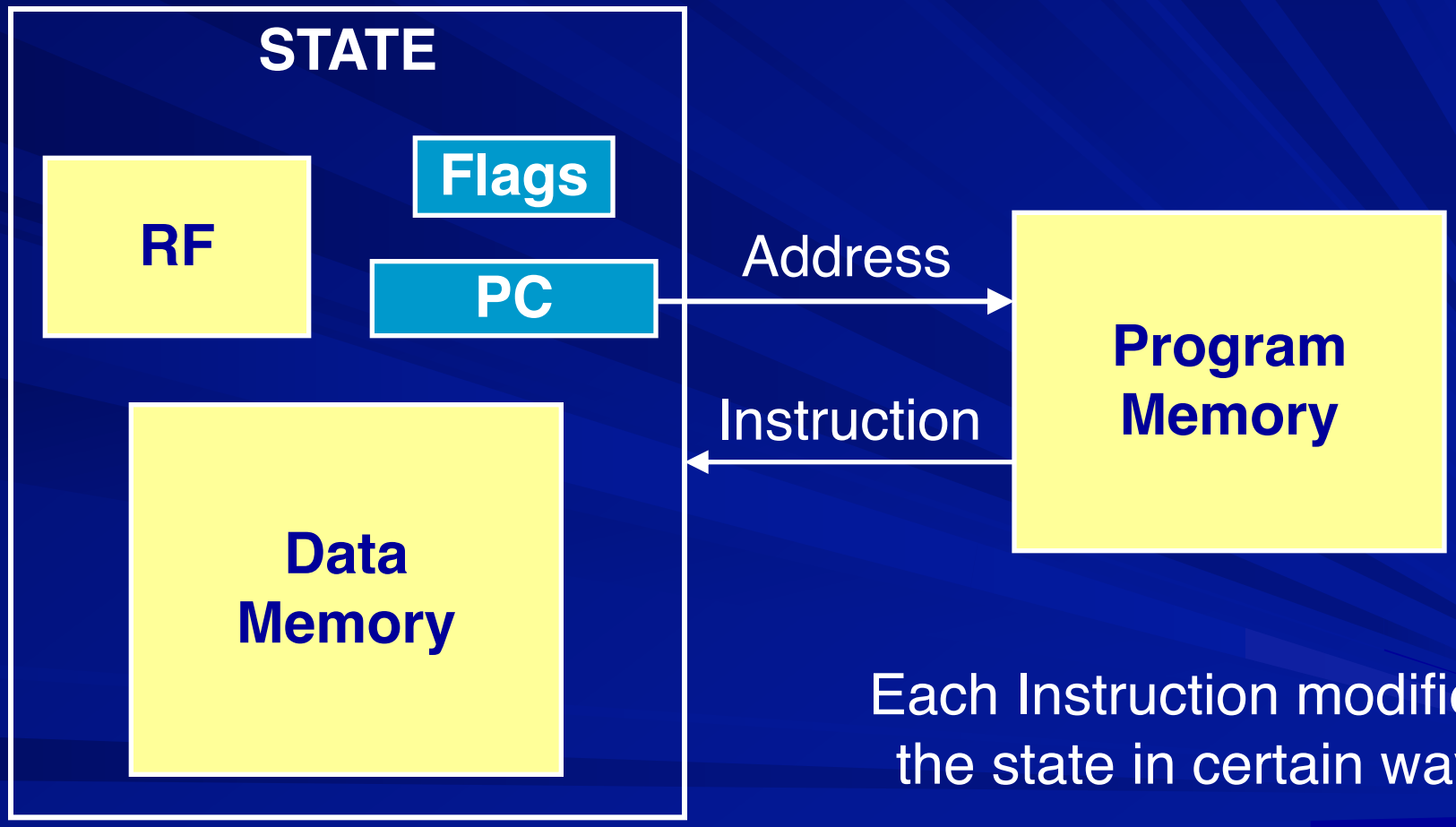
Lowest level visible to a programmer

to

Micro Architecture

High level view of hardware

The Abstract Machine



Each Instruction modifies the state in certain way

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

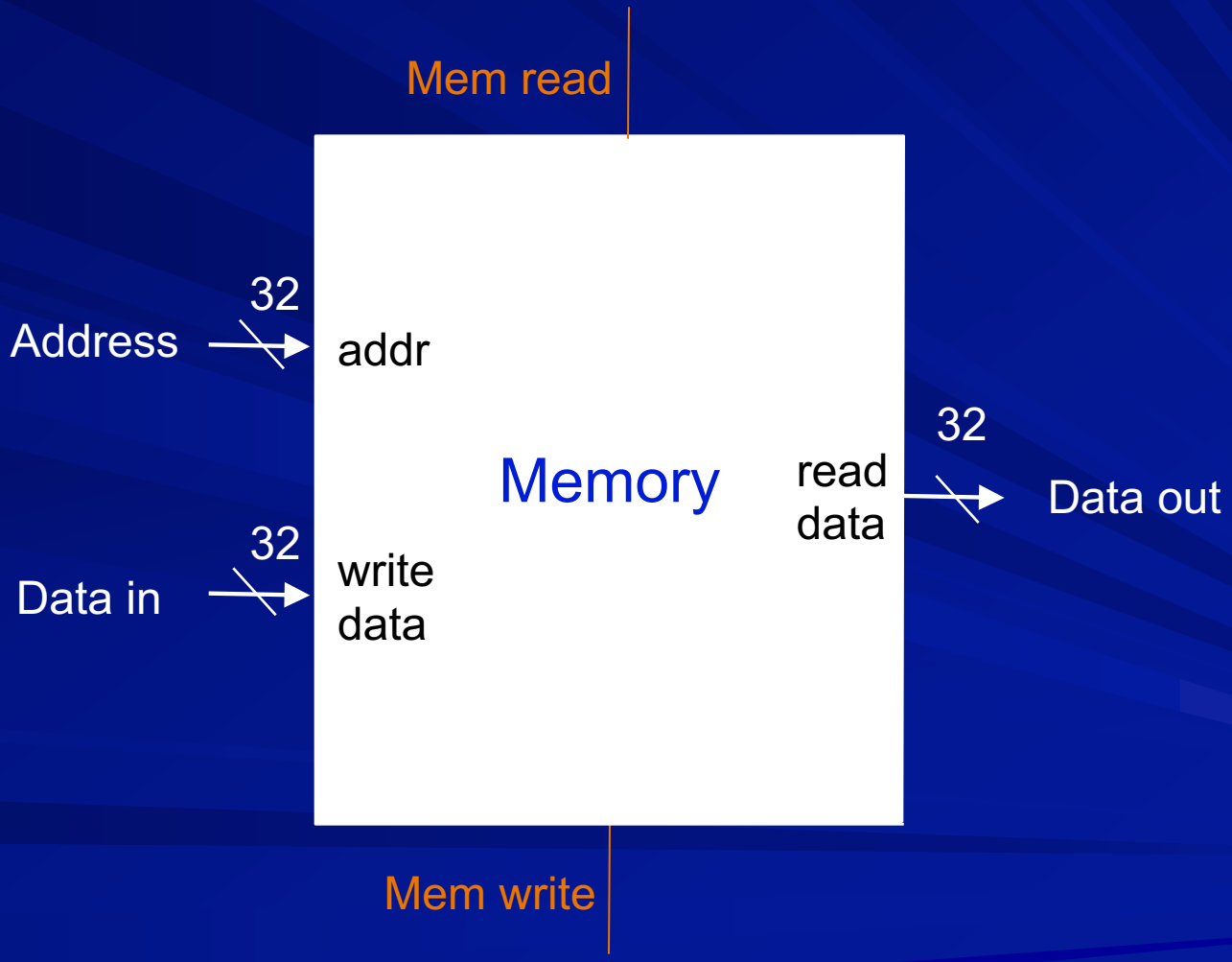
- When one instruction finishes, only then another instruction starts or execution of instructions can overlap
- Only one instruction starts at a time or multiple instructions can start concurrently
- Instructions start in strict program order or the order can be changed

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


Mem read
32
Address
addr
Memory
32
read
data
Data out
32
Data in
write
data
Mem write

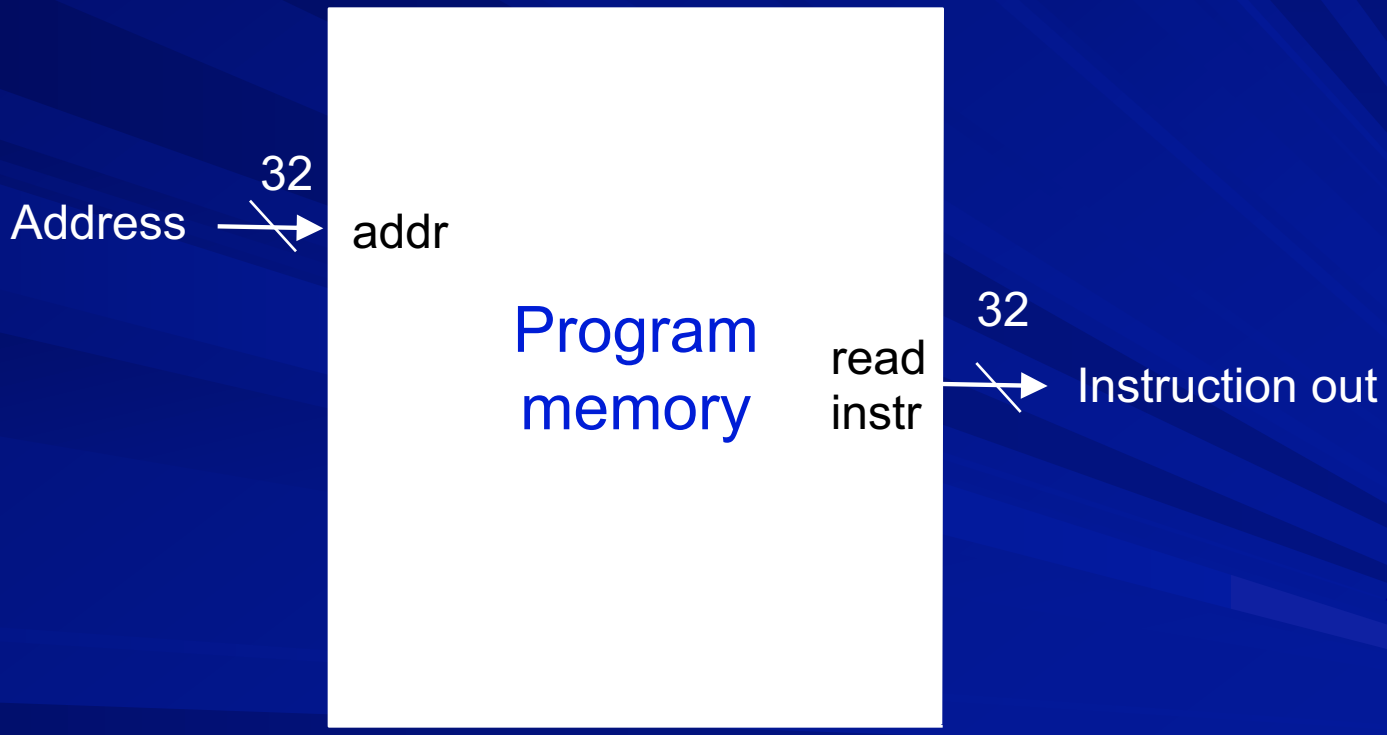
Program and data memory

- Separation of program and data memory

  allows

- accessing instruction and data within same cycle

Program memory

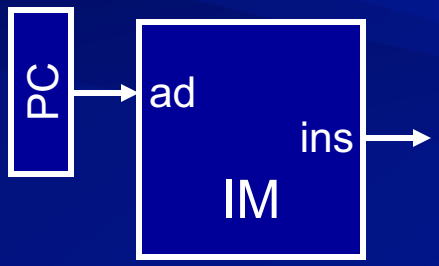


Building datapath

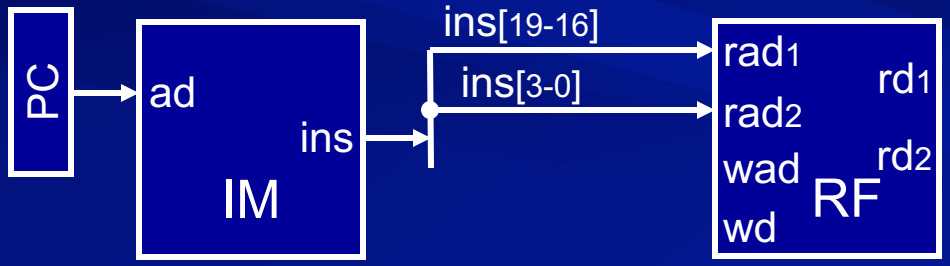
Actions for DP instructions

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

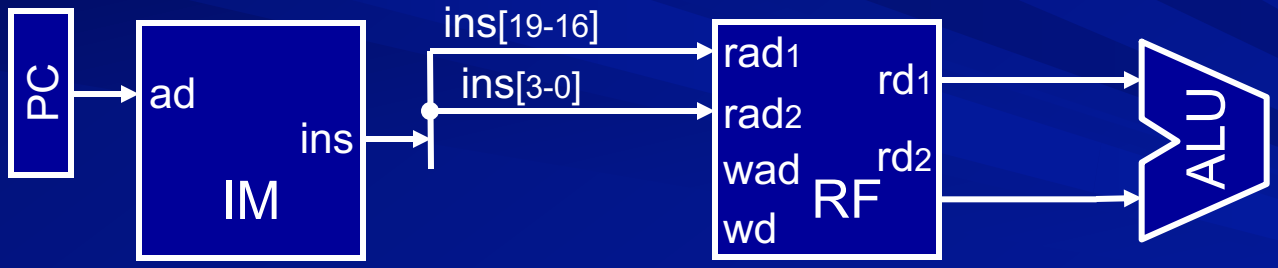
Fetching instruction



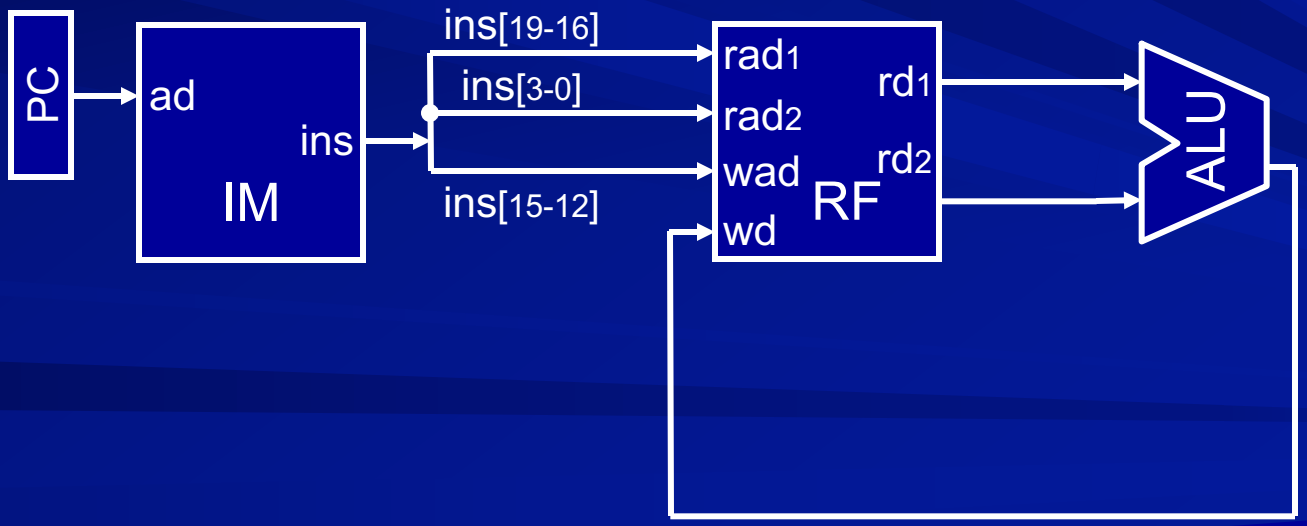
Accessing RF



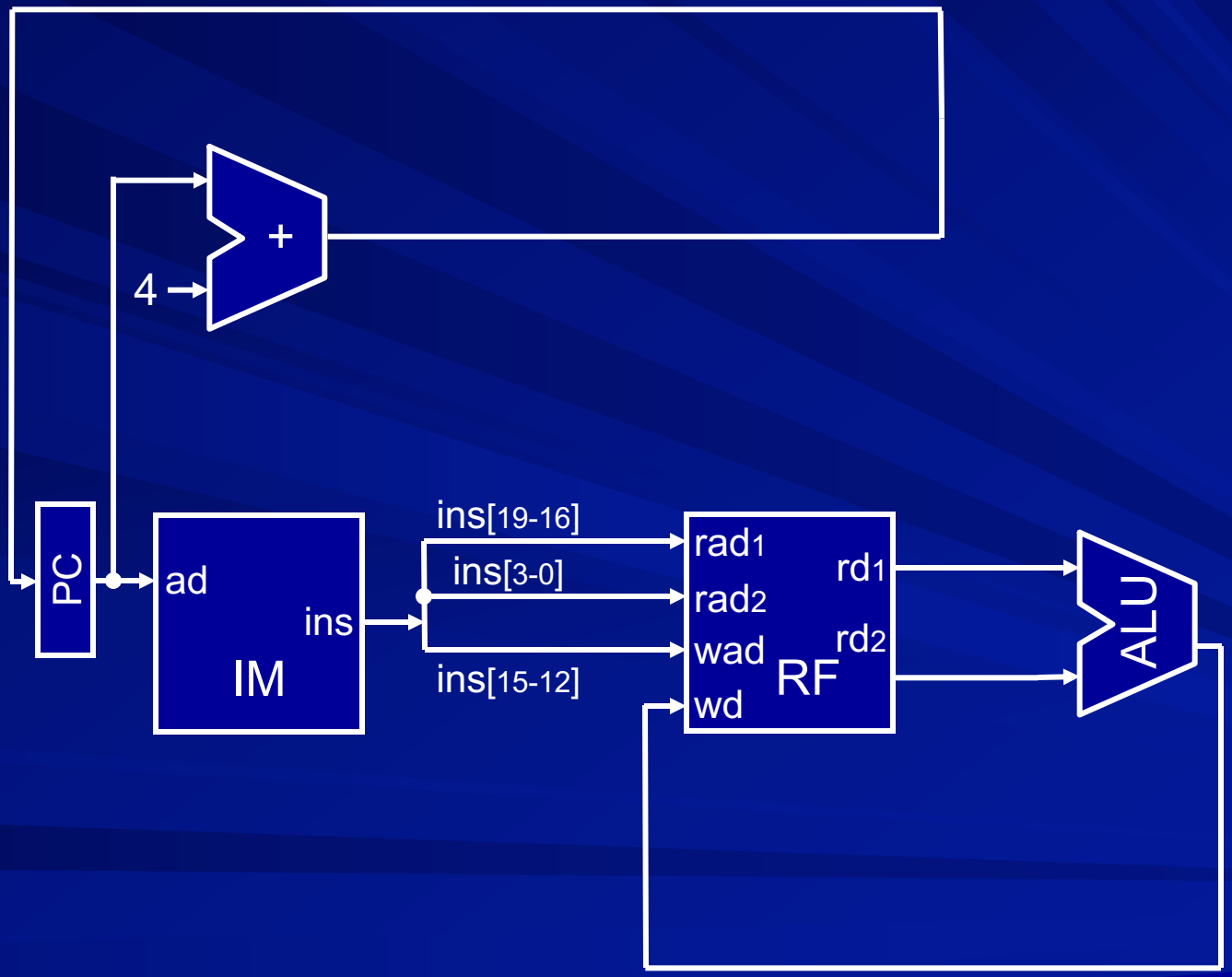
Passing operands to ALU



Passing the result to RF



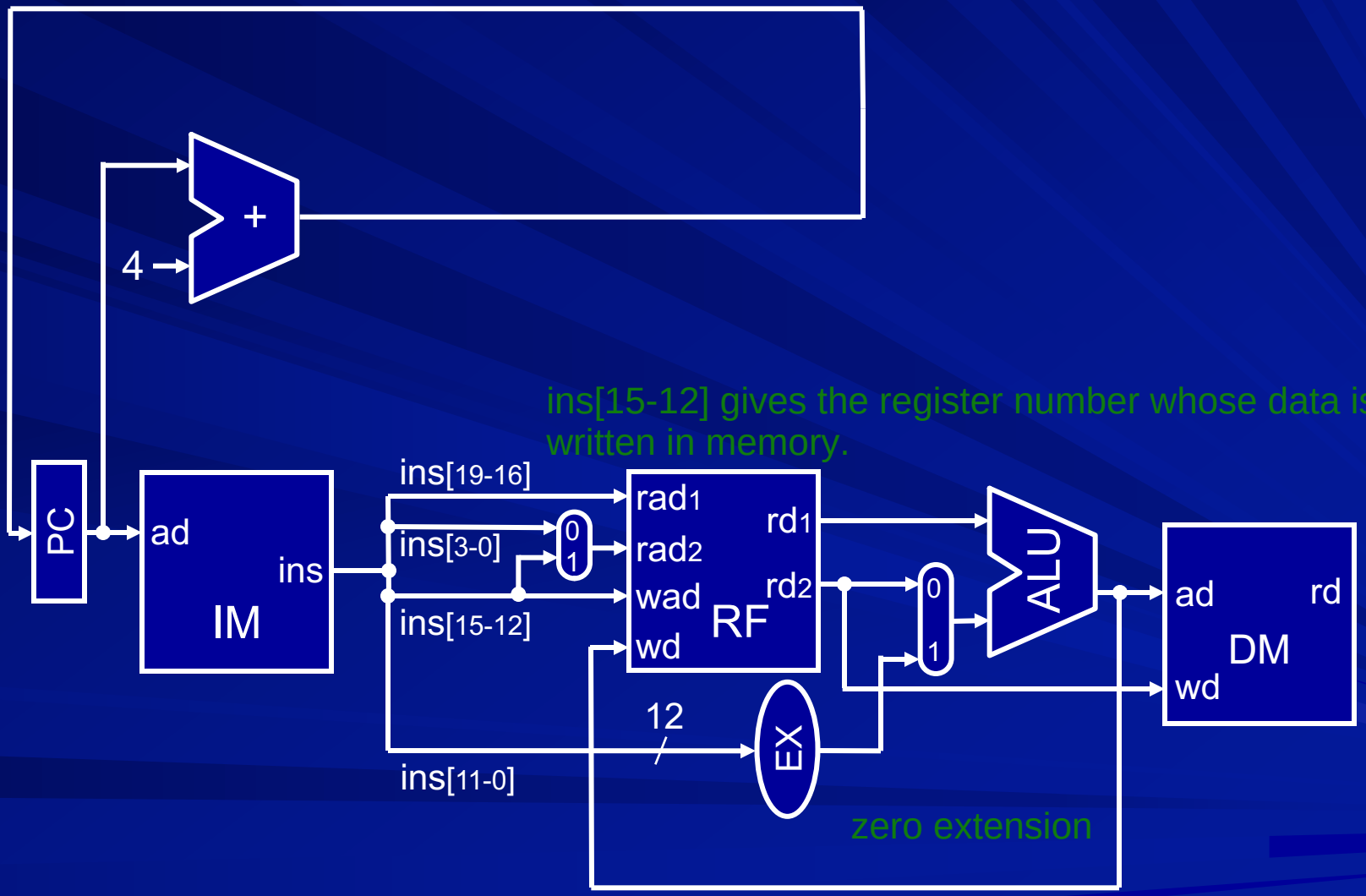
Incrementing PC



Actions for str instructions

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

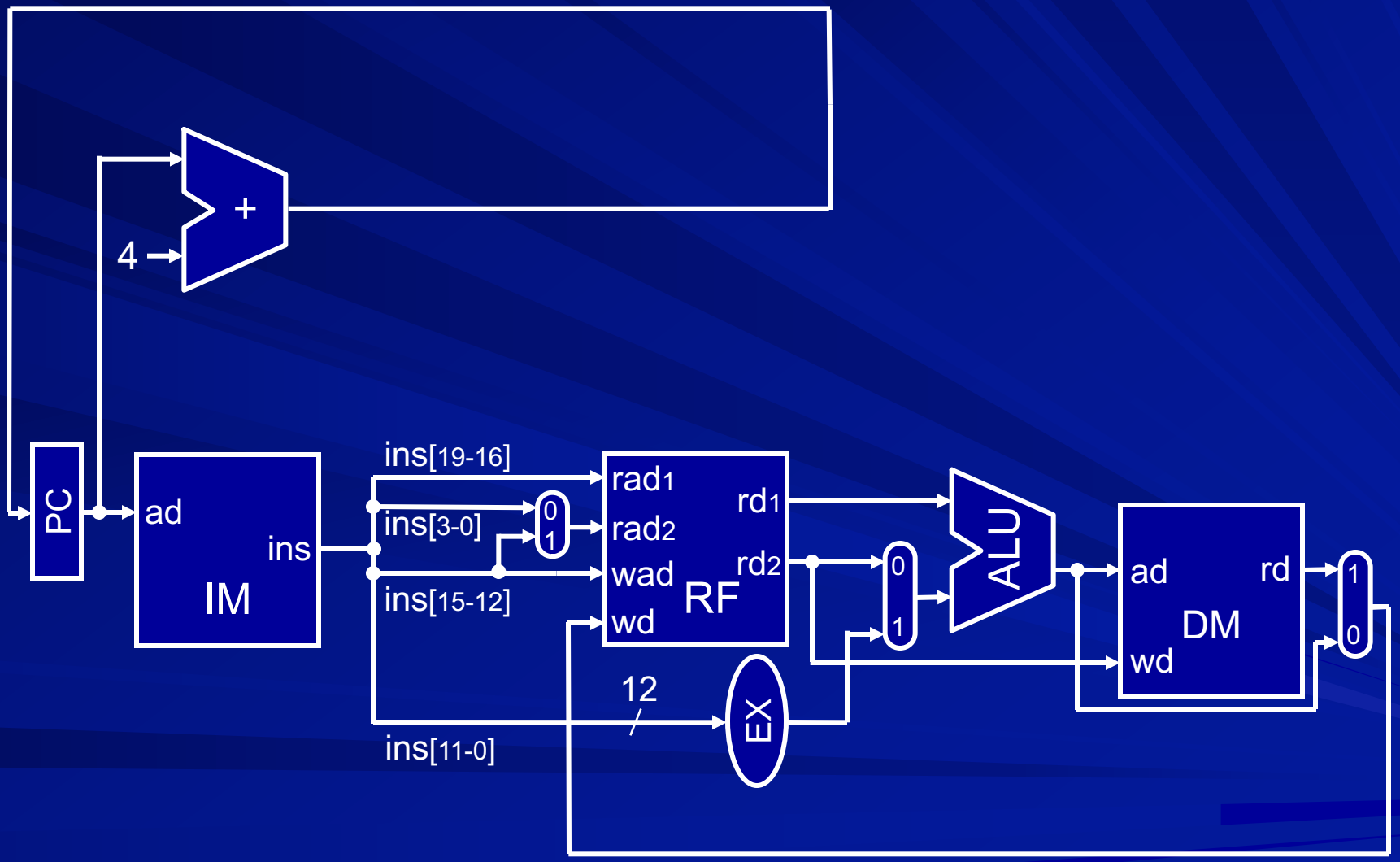
Adding “str” instruction



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

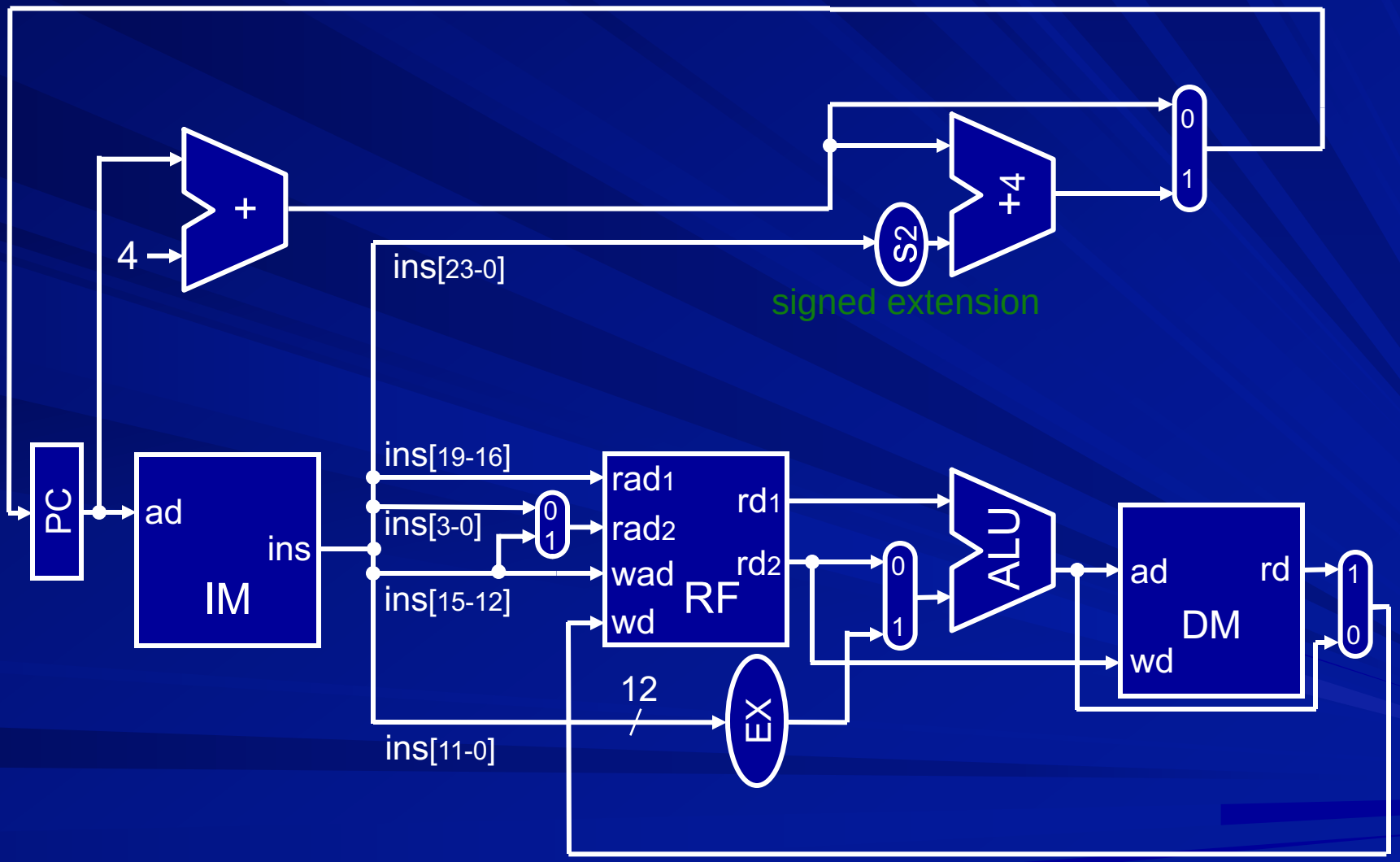


Actions for branch instruction

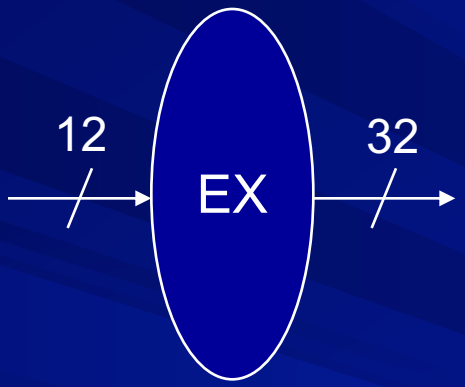
- fetch instruction

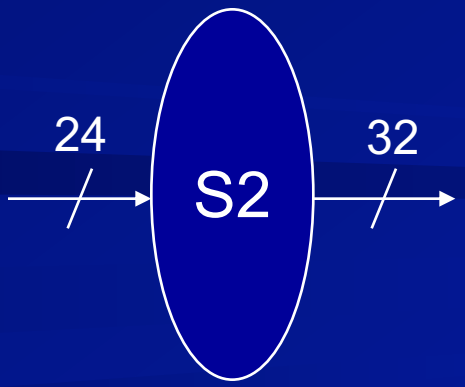
- compute target address
- transfer address to pc

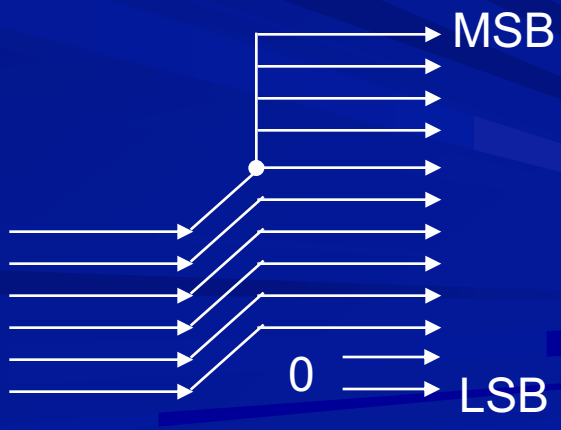
Adding “b” instruction



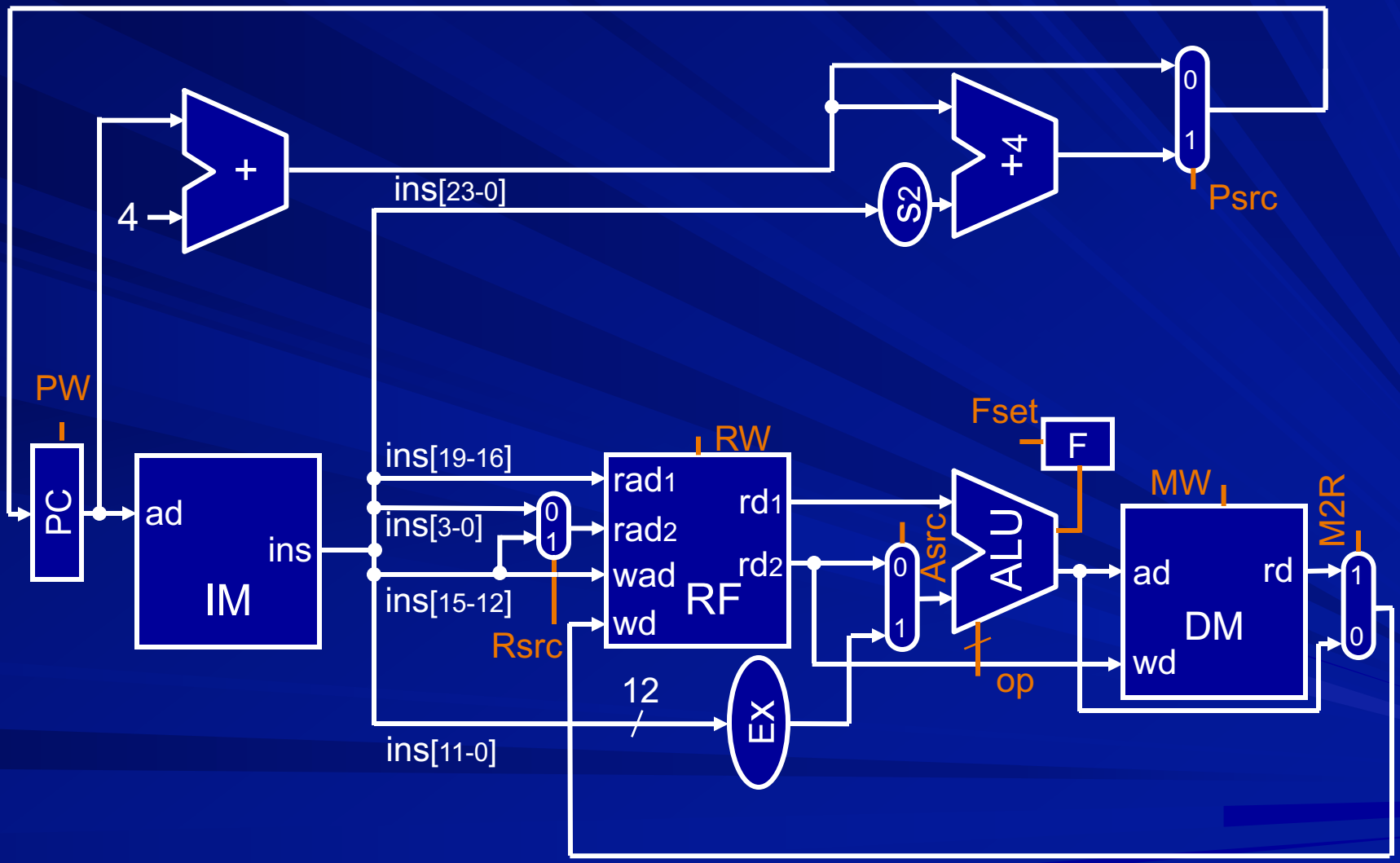
Extending offsets







Single cycle Datapath



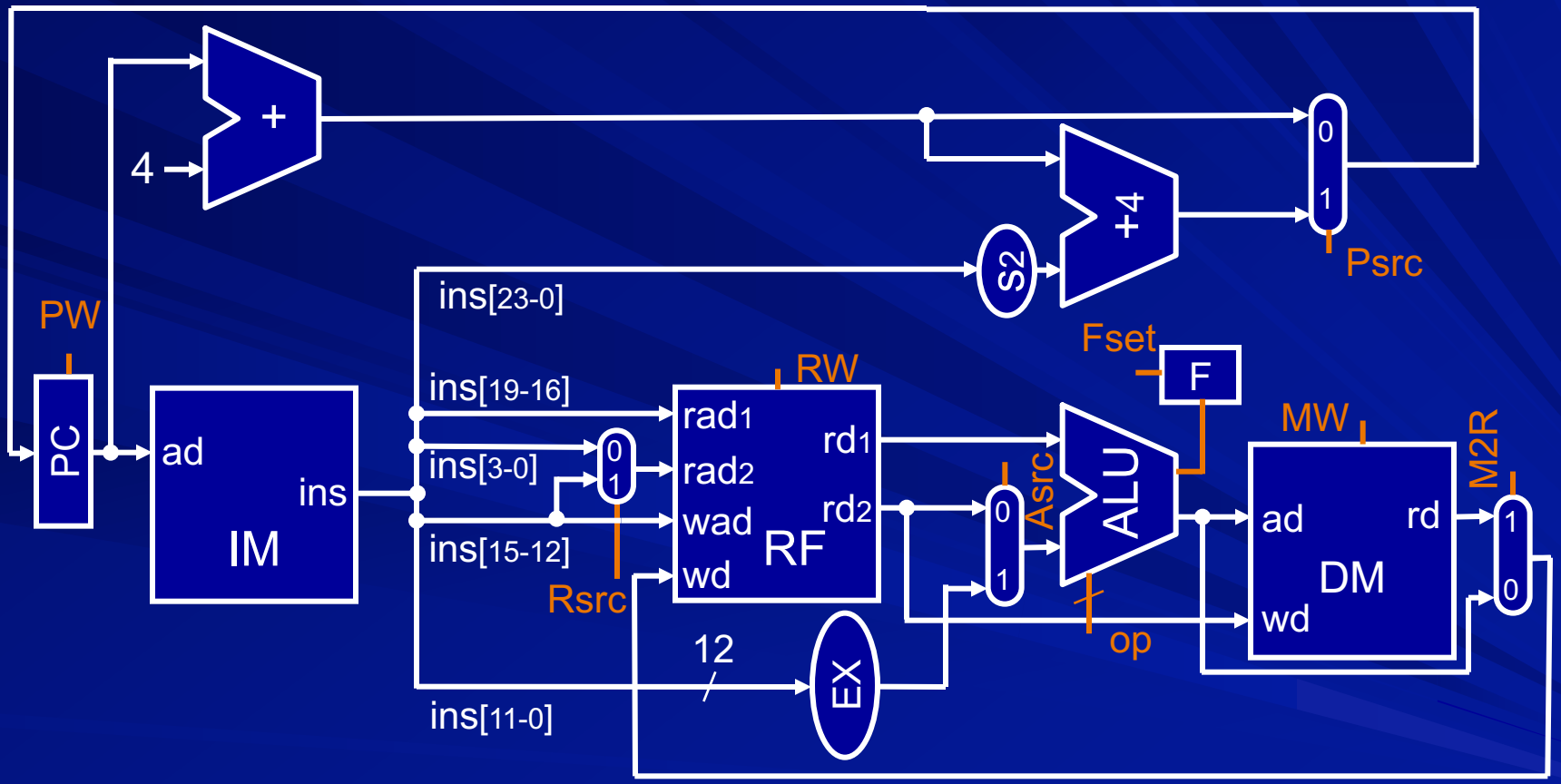
Problems with single cycle design

- 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

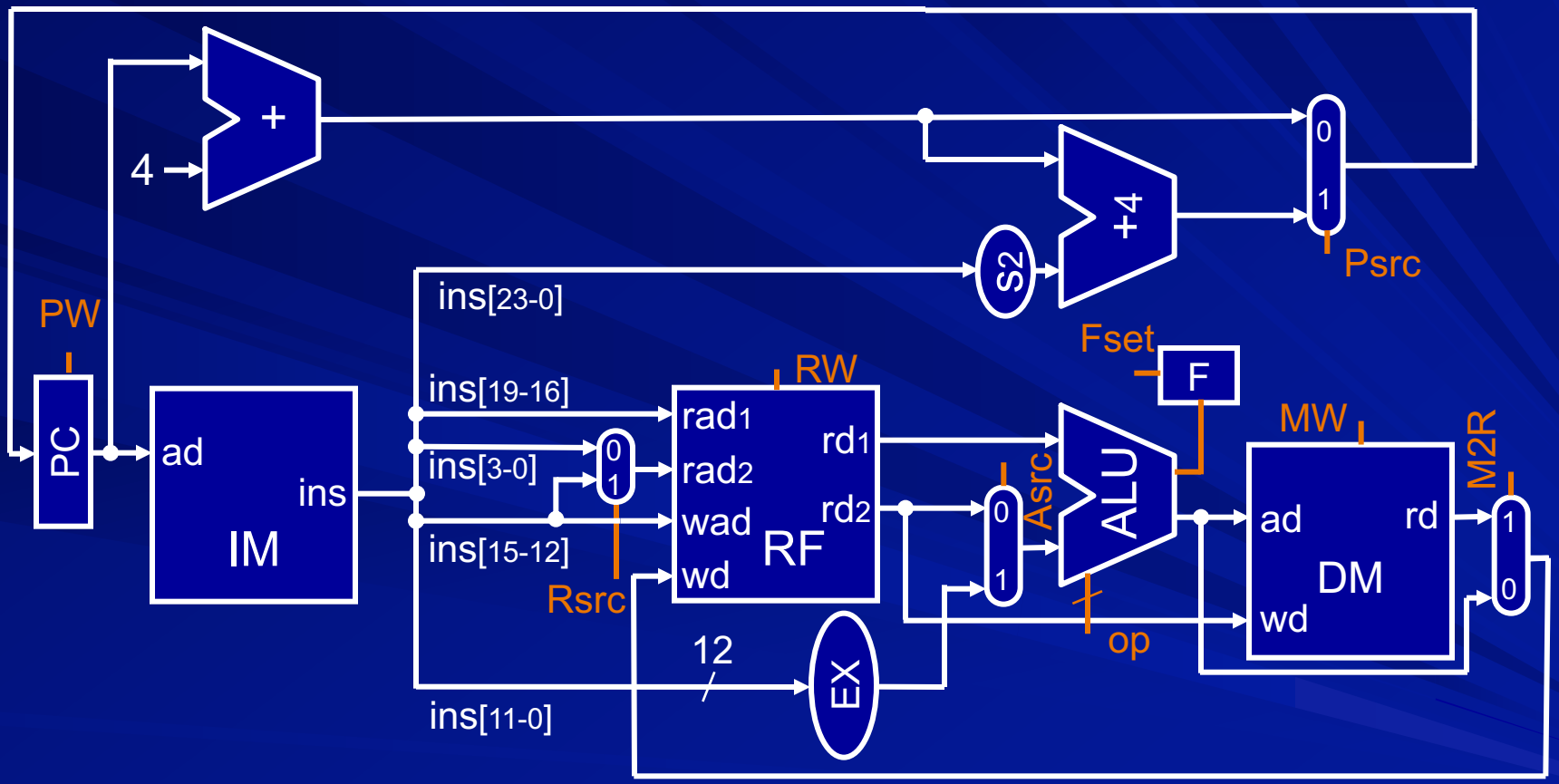
	Component delays
Register	0
Adder	t_+
ALU	t_A
Multiplexer	0
Register file	t_R
Program memory	t_I
Data memory	t_M
Bit manipulation components	0

Delay for {add, sub, …}



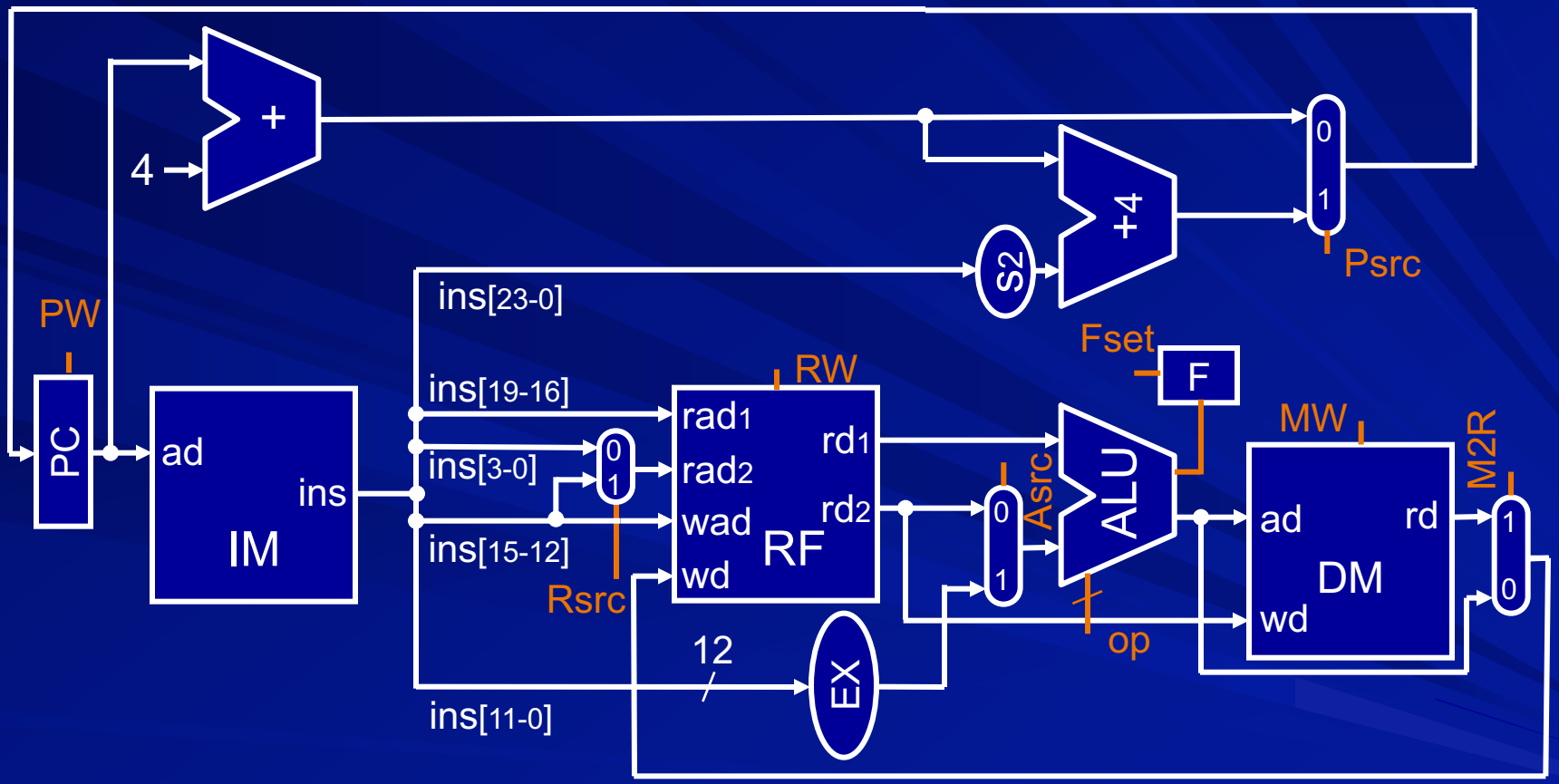
$$\max\left\{\begin{array}{c} t_{+} \\ t_I + t_R + t_A + t_R \end{array}\right.$$

Delay for {cmp, tst, …}



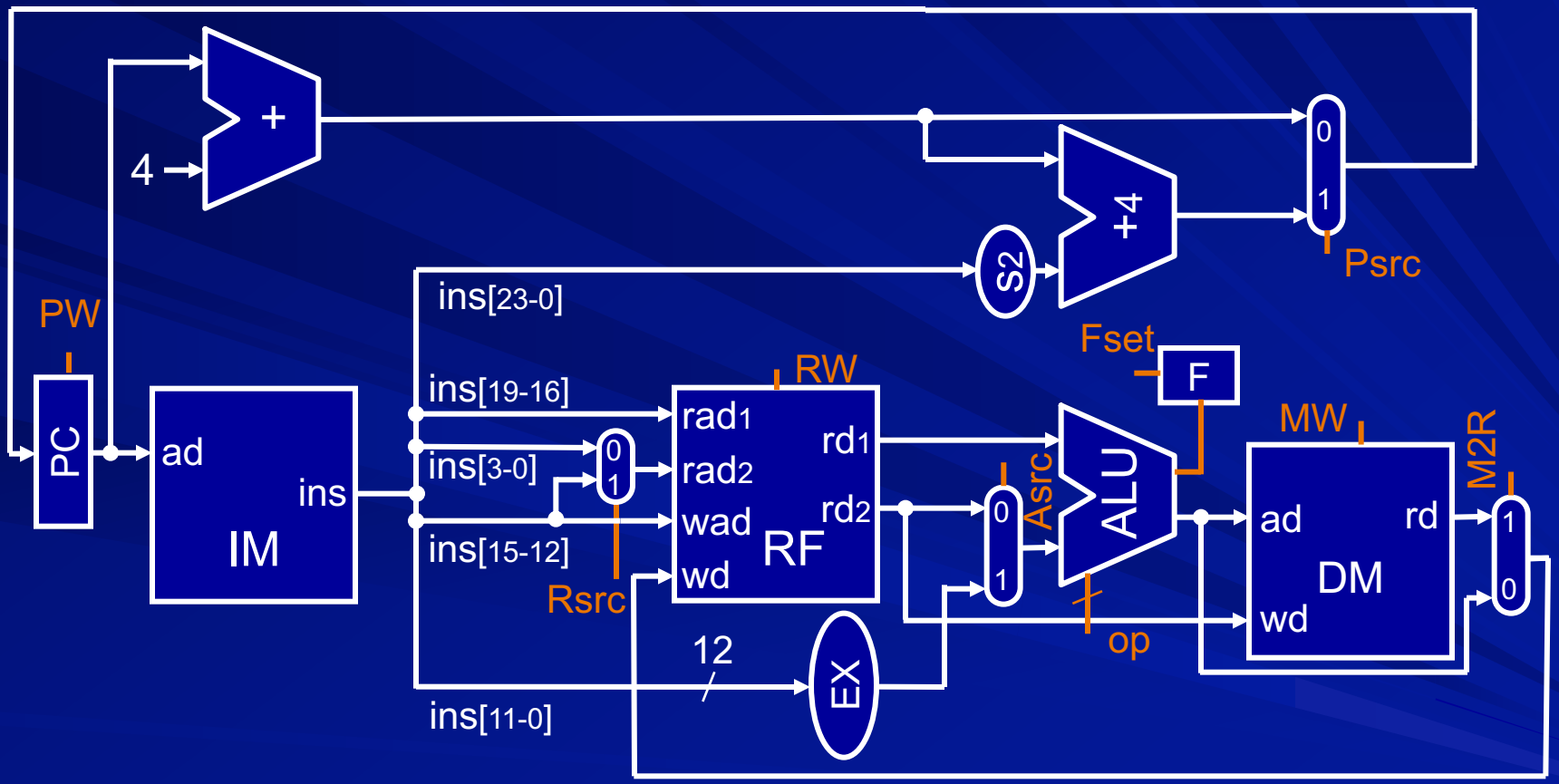
$$\max\left\{\begin{array}{c} t_{+} \\ t_I + t_R + t_A \end{array}\right.$$

Delay for {str}



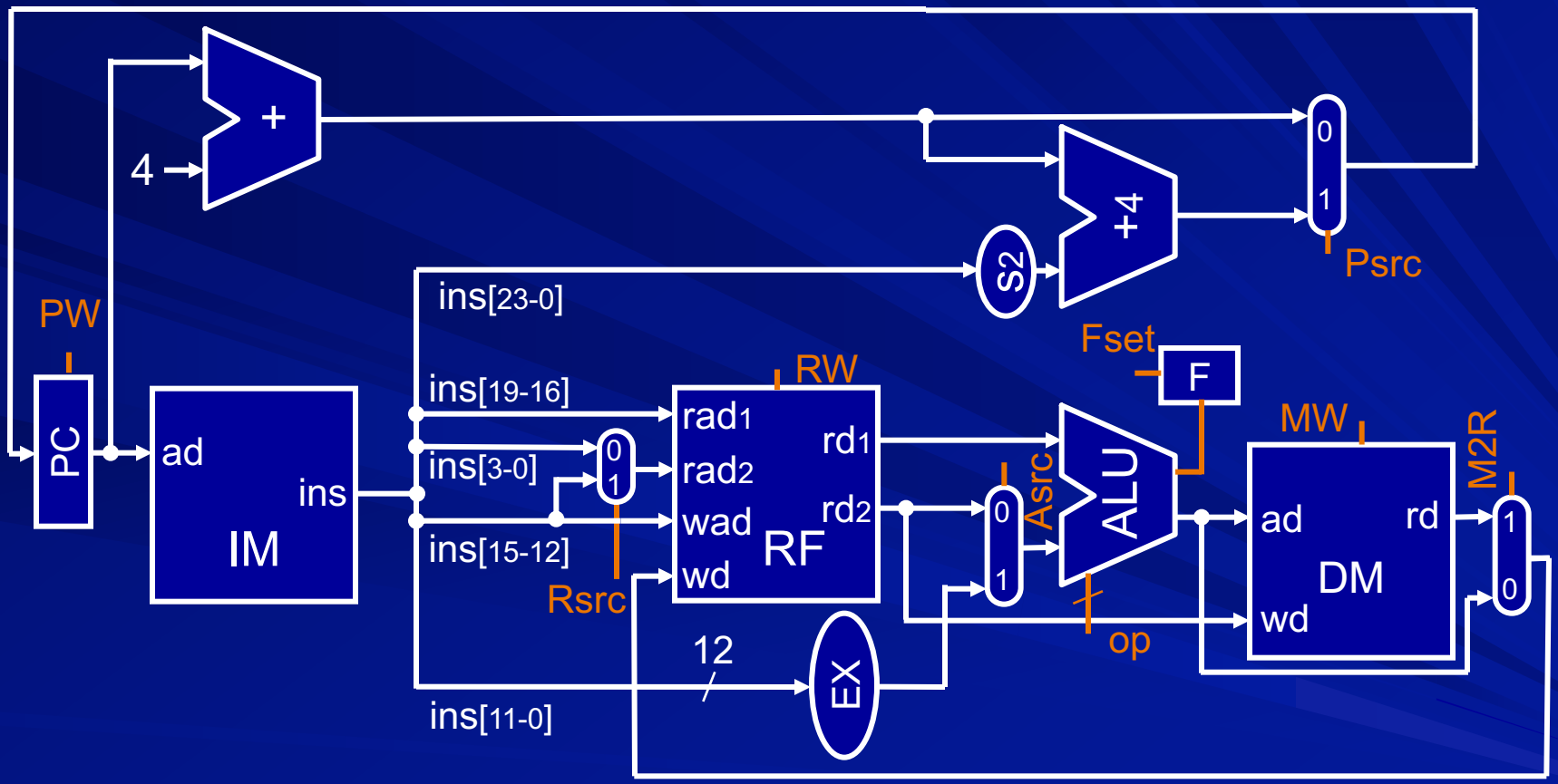
$$\max\begin{cases} t_{+} \\ t_I + t_R + t_A + t_M \end{cases}$$

Delay for {ldr}



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

Delay for {b}



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

Overall clock period

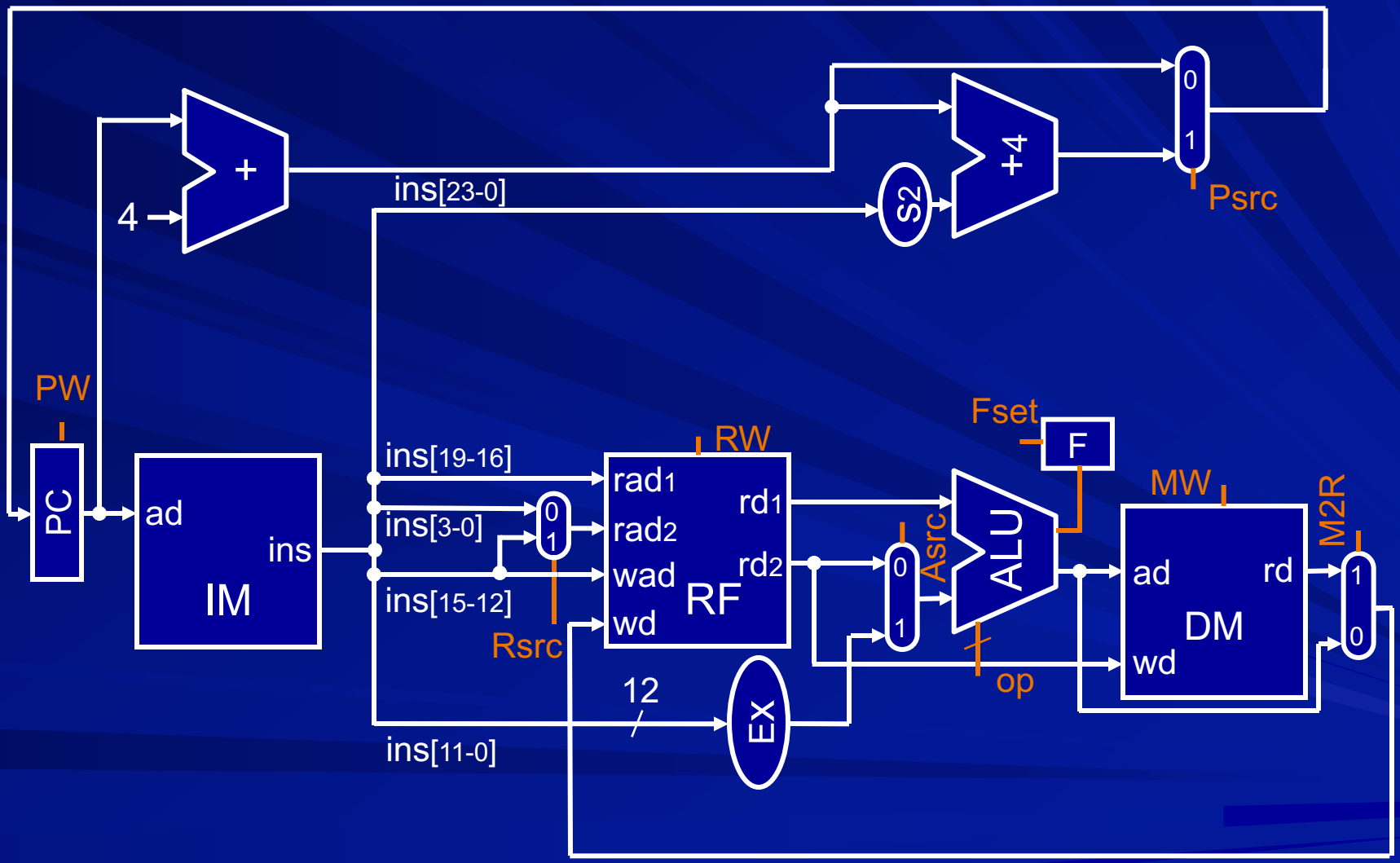
$$\max \begin{cases} t_+, & t_I + t_R + t_A + t_R \\ t_+, & t_I + t_R + t_A \\ t_+, & t_I + t_R + t_A + t_M \\ t_+, & t_I + t_R + t_A + t_M + t_R \\ t_I + t_+, & t_+ + t_+ \end{cases} ?$$

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

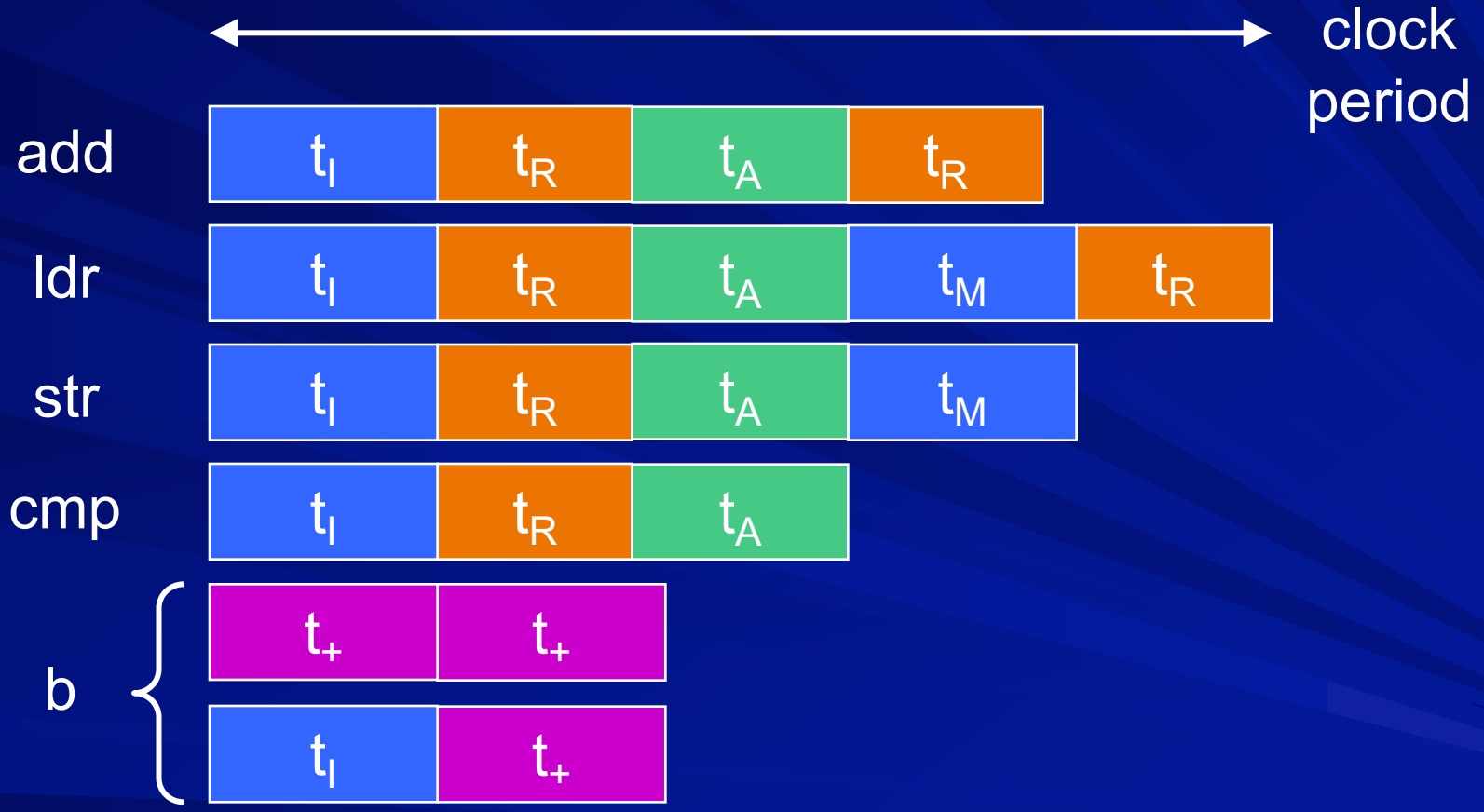
Analyzing performance

	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

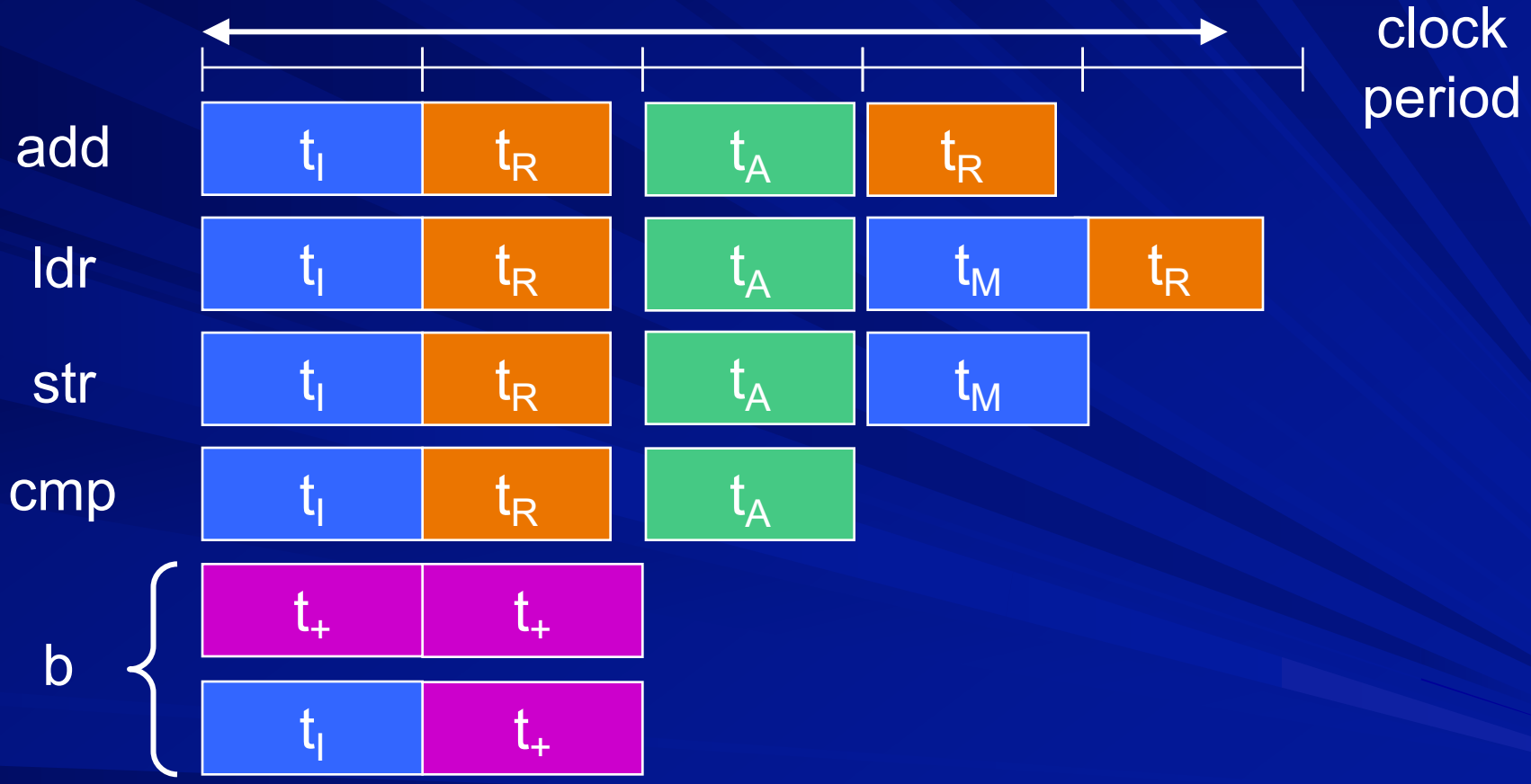
Overall clock period

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

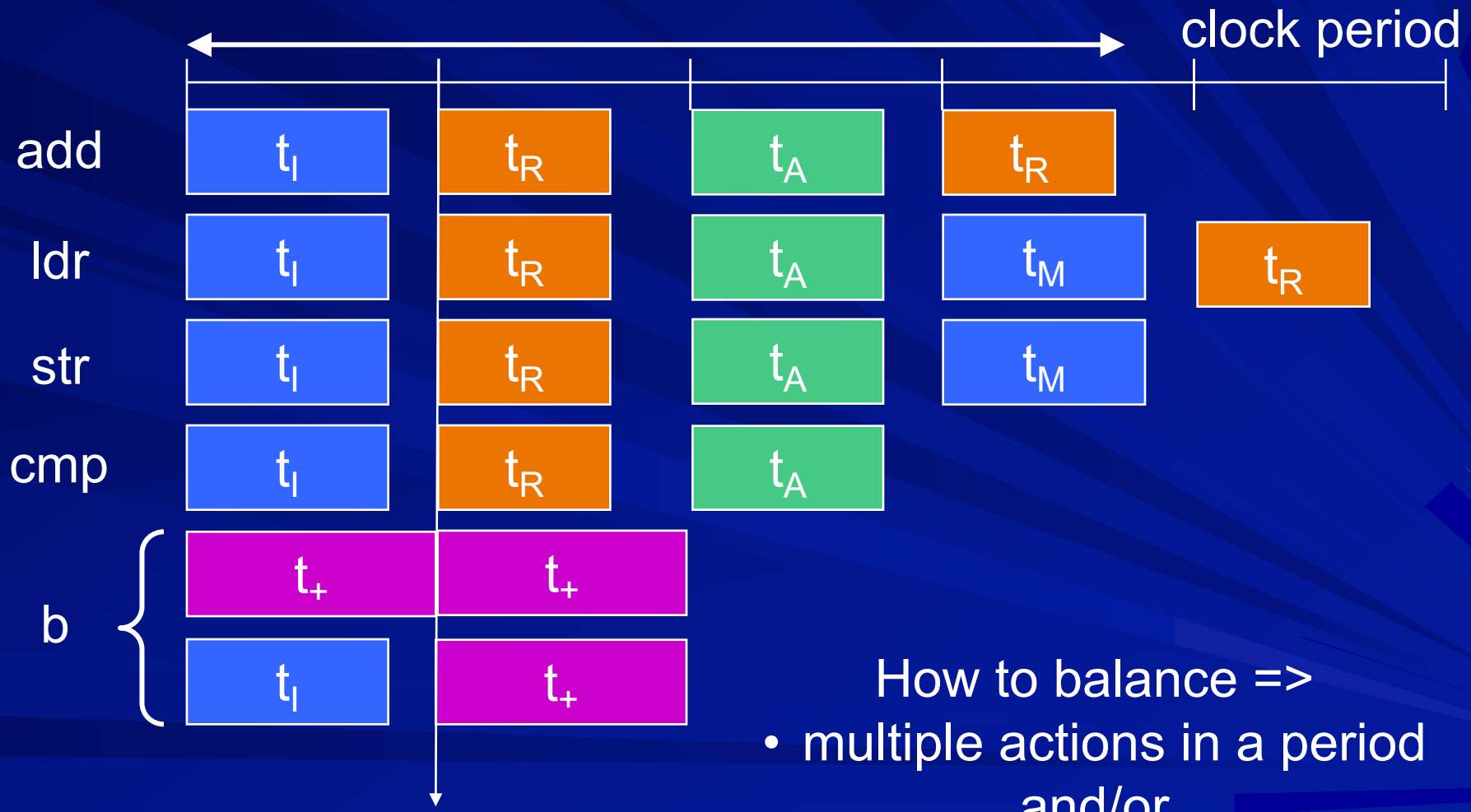
Clock period in single cycle design



Split the cycle into multiple cycles



Unbalanced delays



How to balance =>

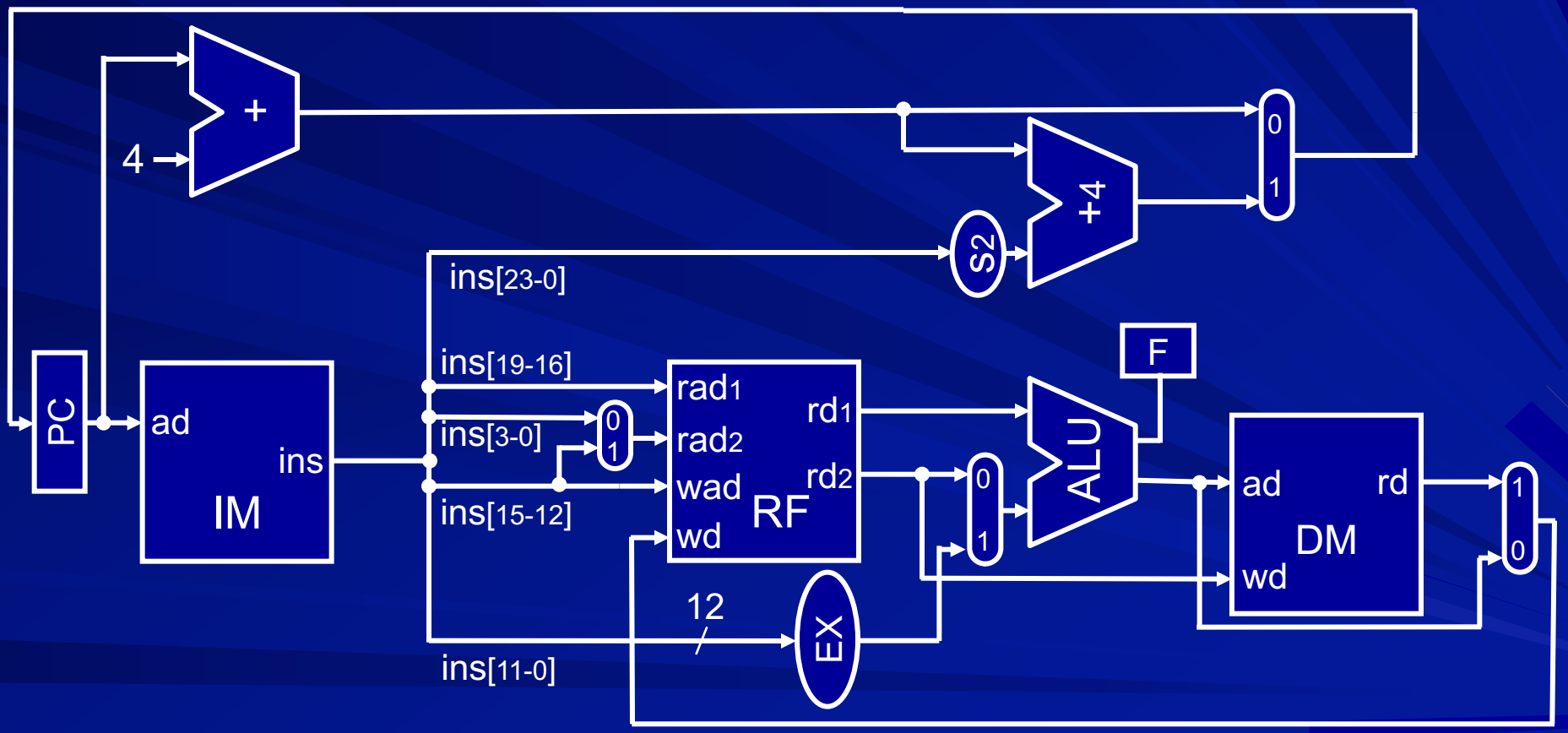
- multiple actions in a period and/or
- multiple periods for an action

Improving resource utilization

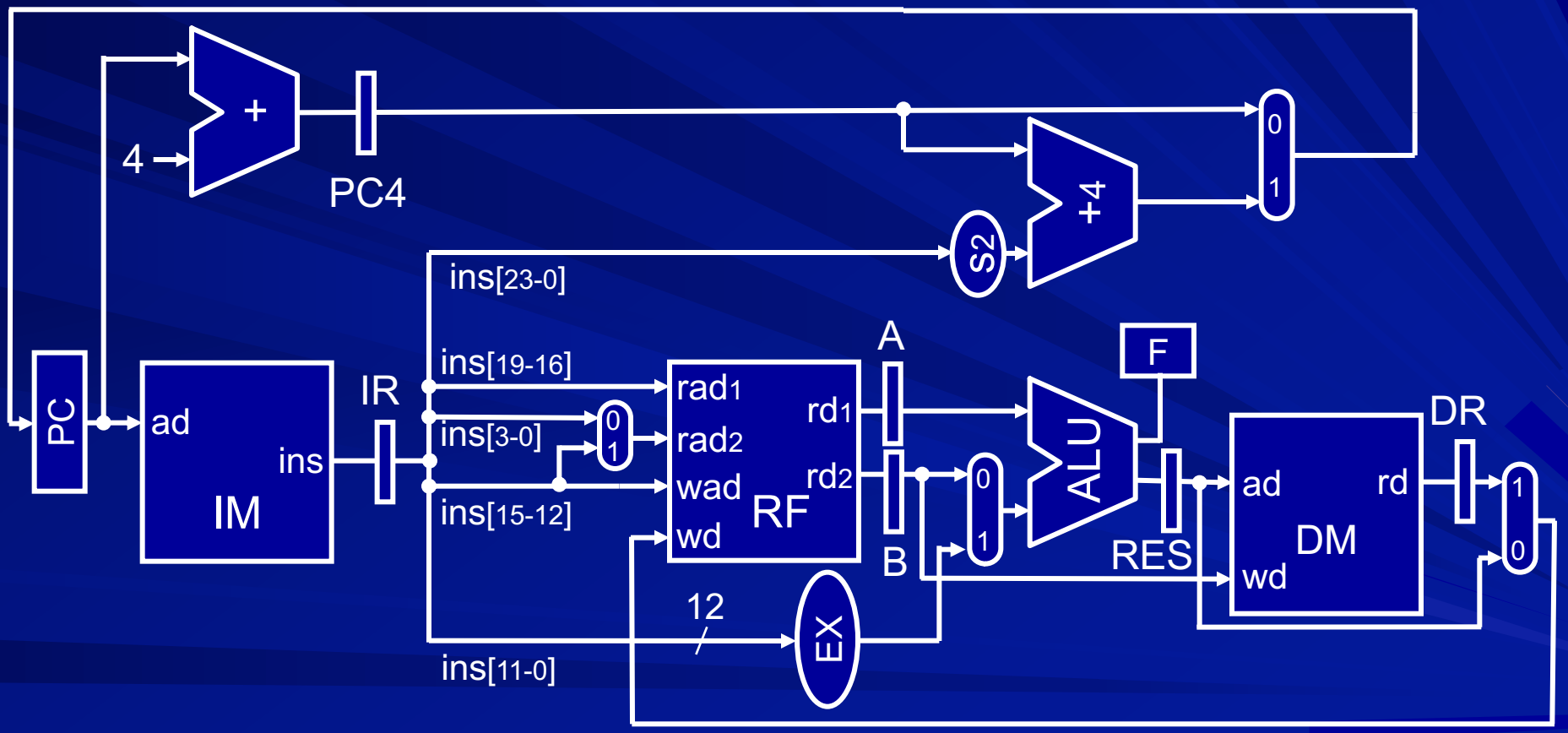
- Can we eliminate two adders?
- How to share (or reuse) a resource (say ALU) in different clock cycles?

- Store results in registers.
- Of course, more multiplexing may be required!
- Resources in this design: RF, ALU, MEM.

Single Cycle Datapath

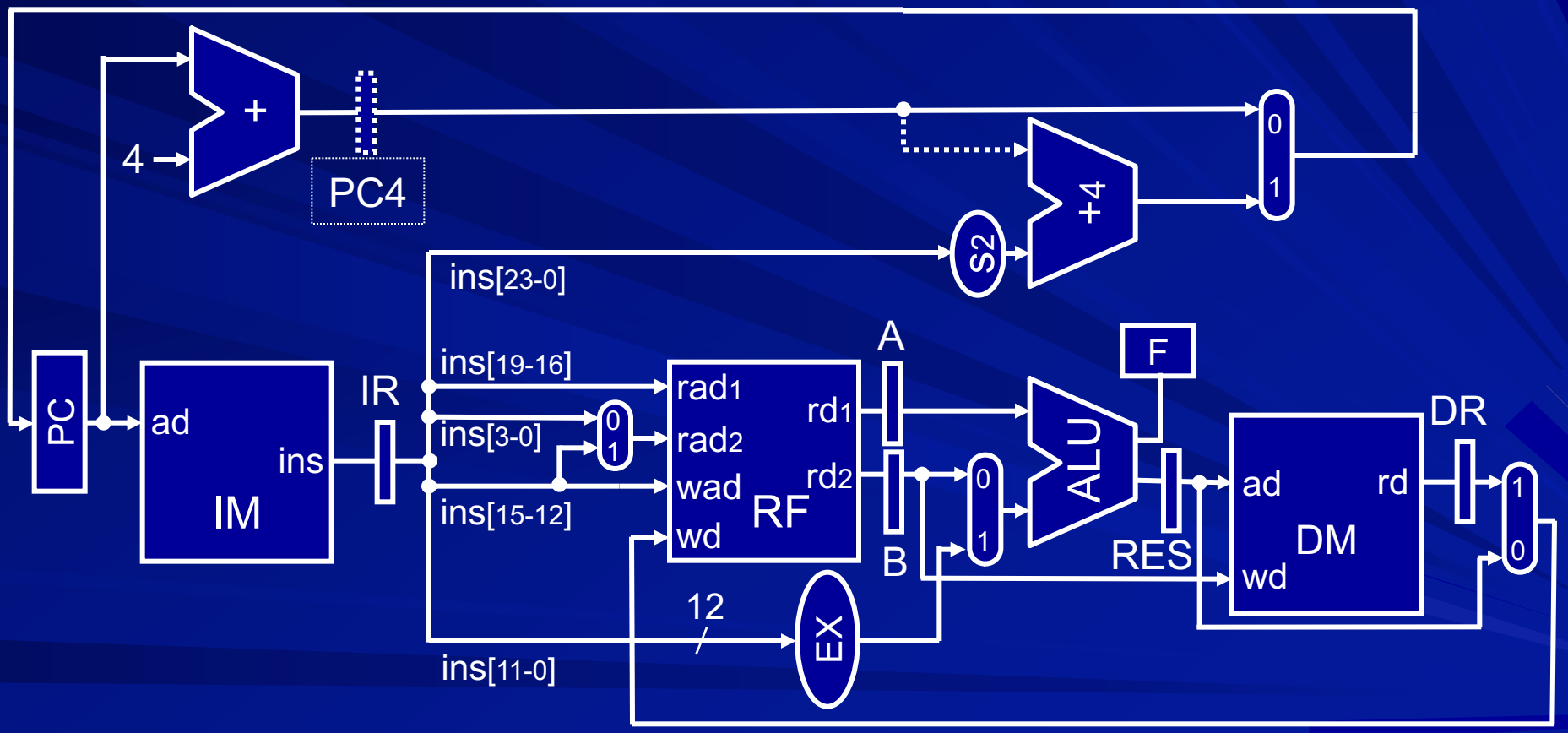


Introducing registers

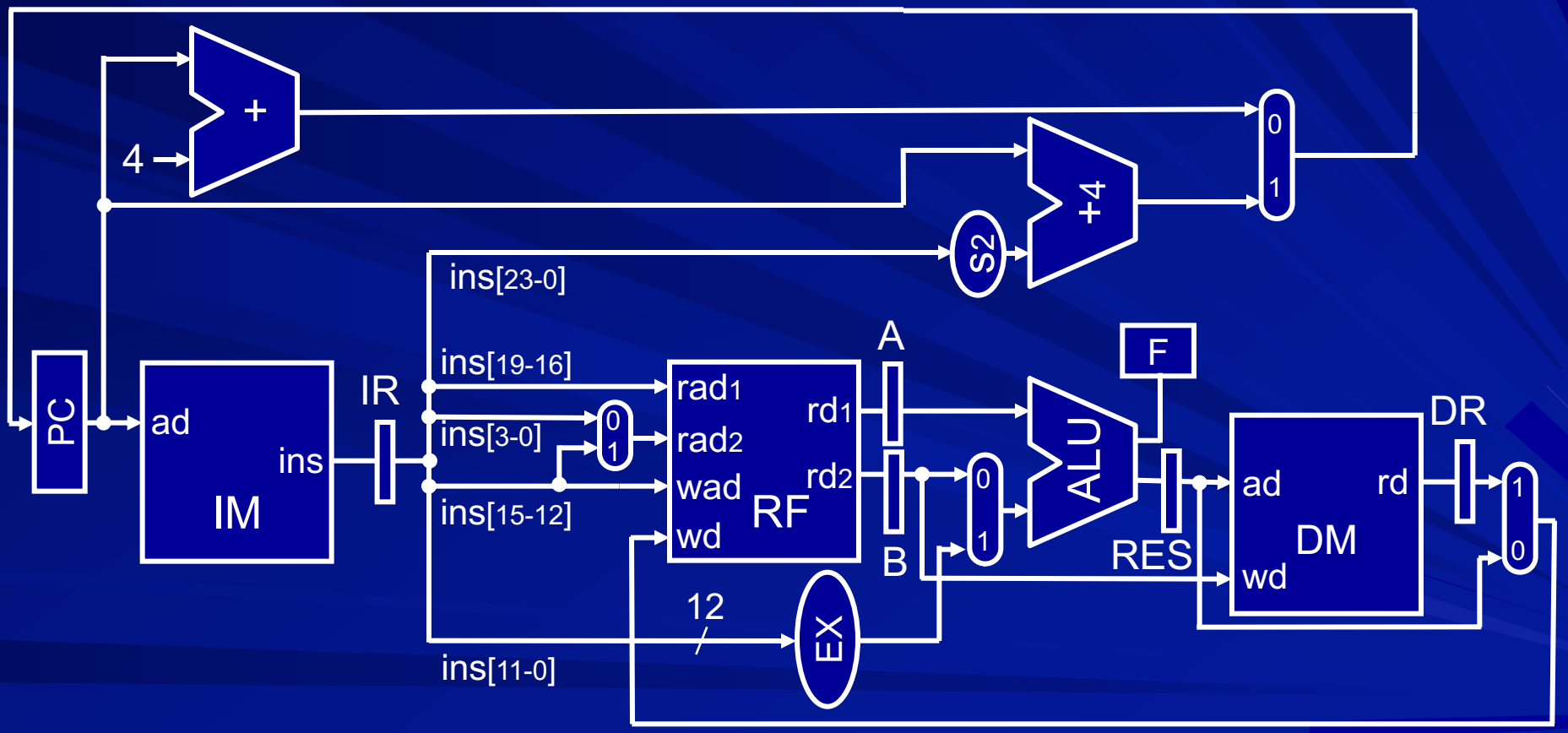


PC4 can be eliminated

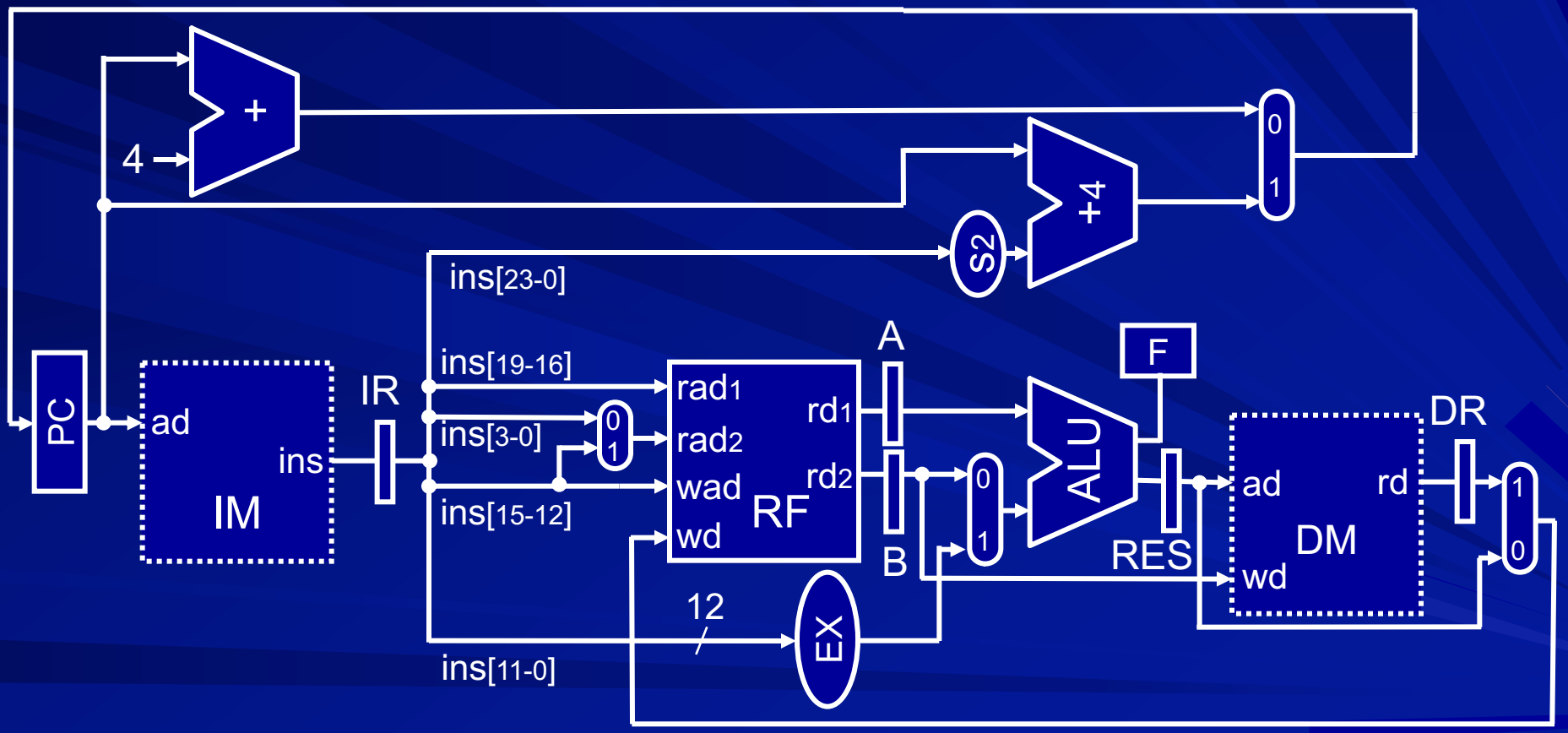
Store PC + 4 in PC itself



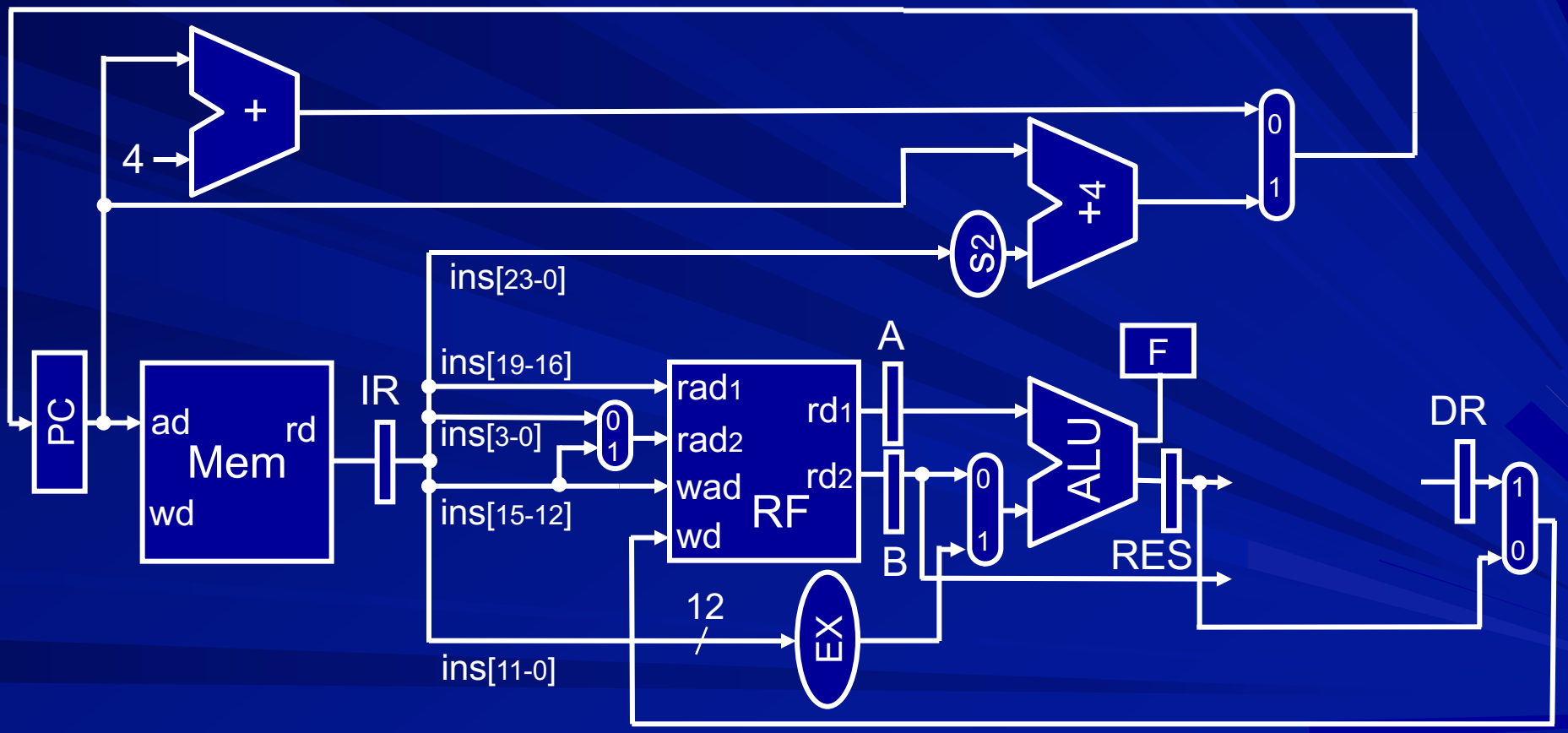
PC4 can be eliminated



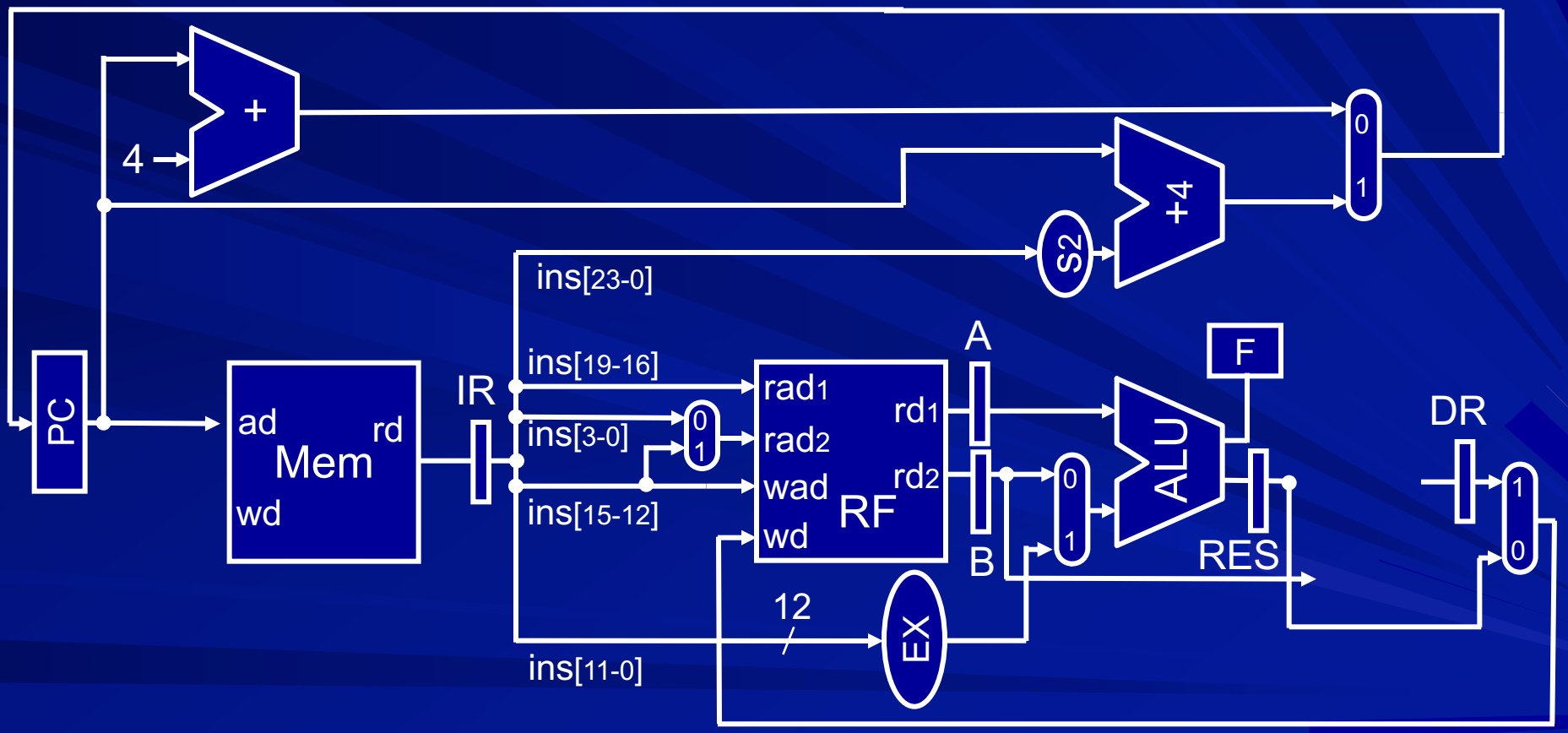
Merge IM and DM



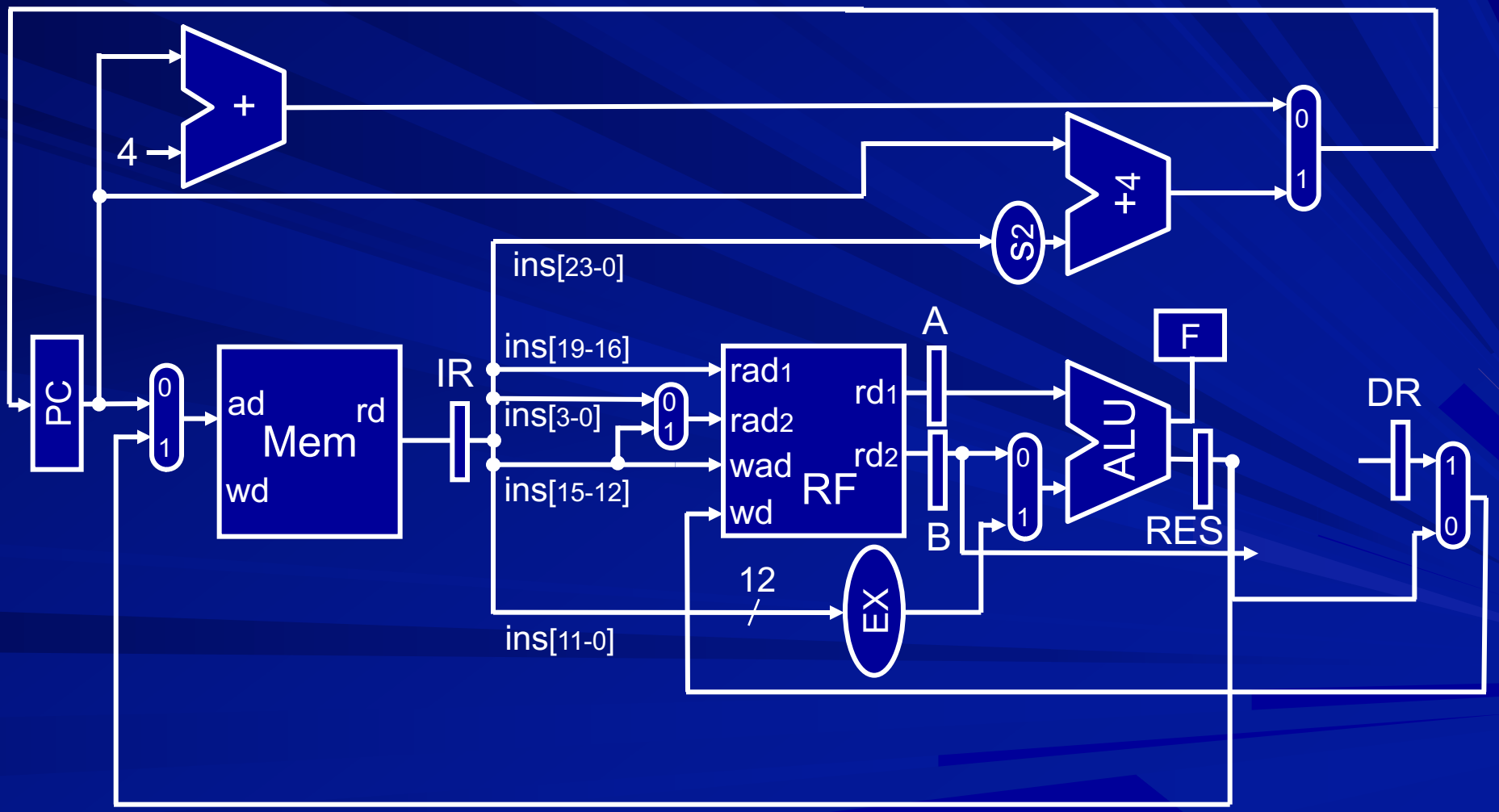
Merge IM and DM



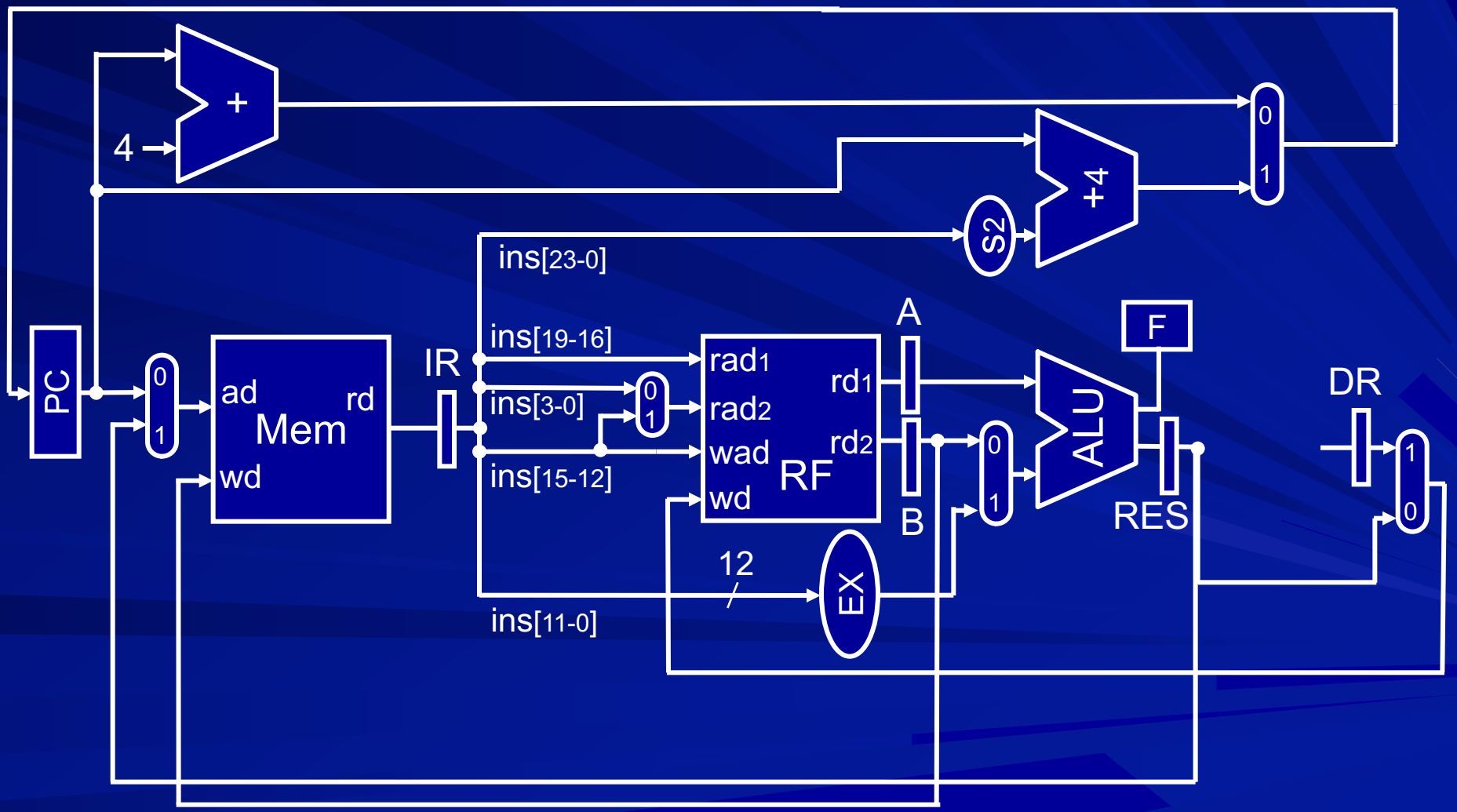
Merge IM and DM



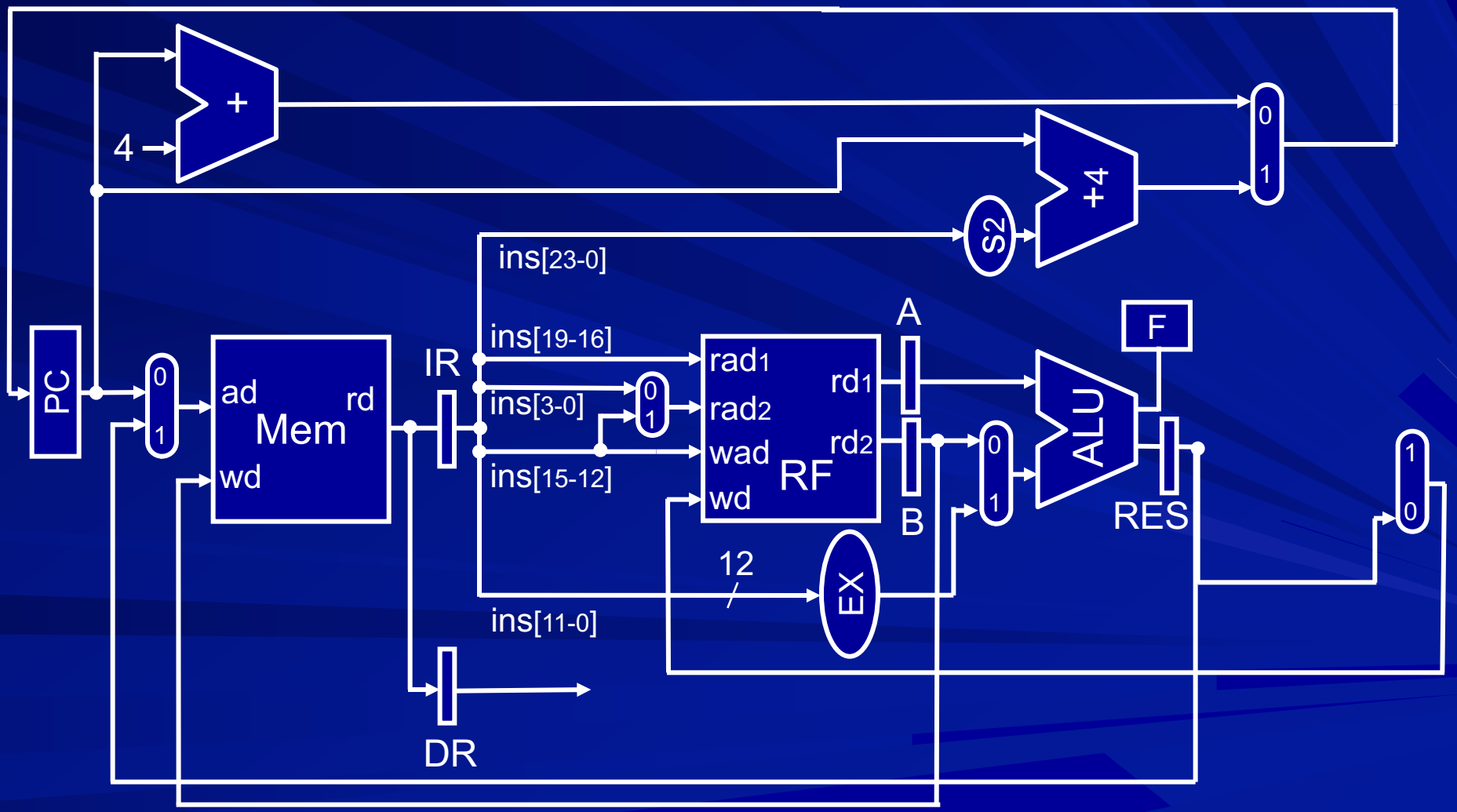
Merge IM and DM



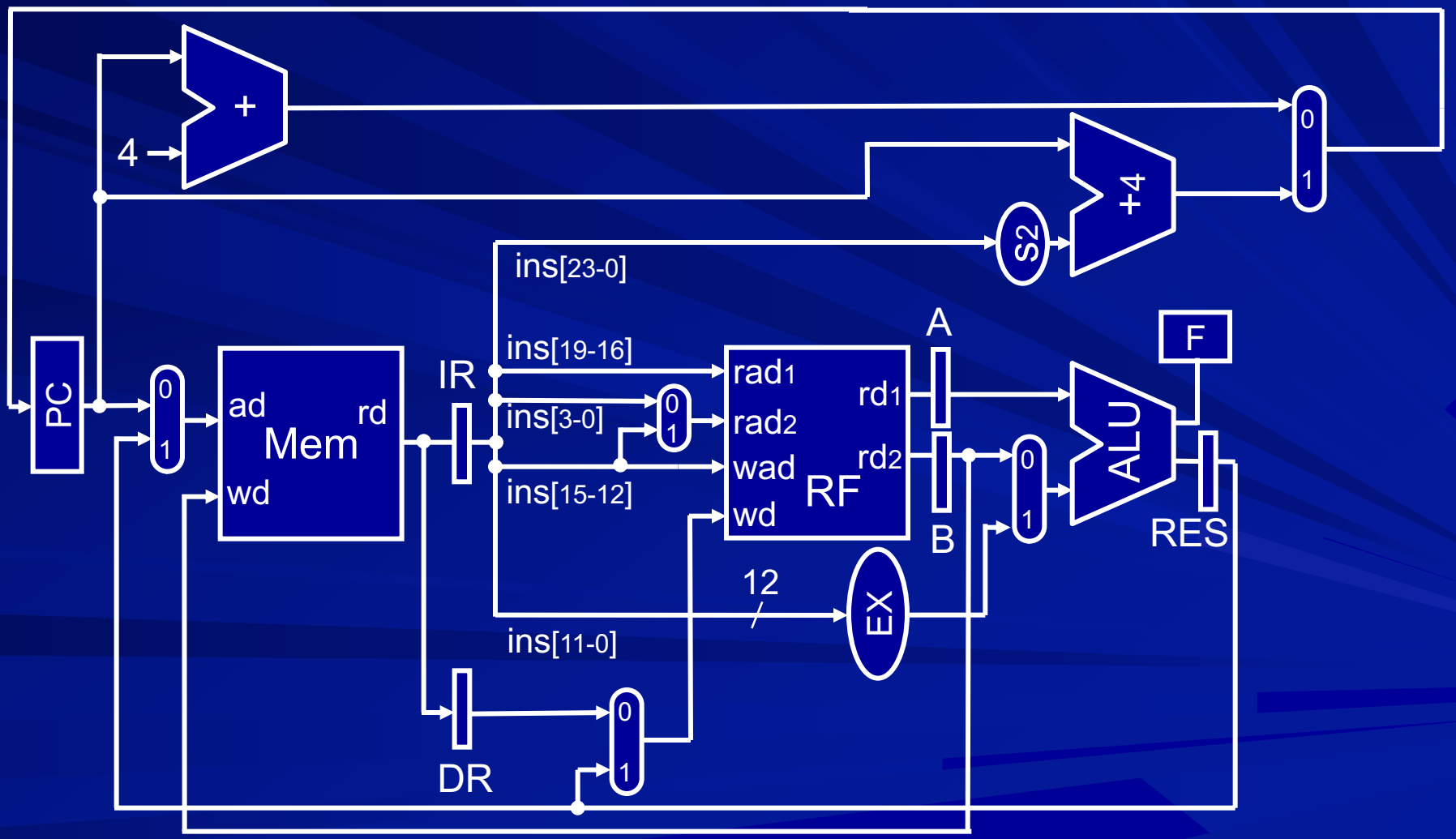
Merge IM and DM



Merge IM and DM

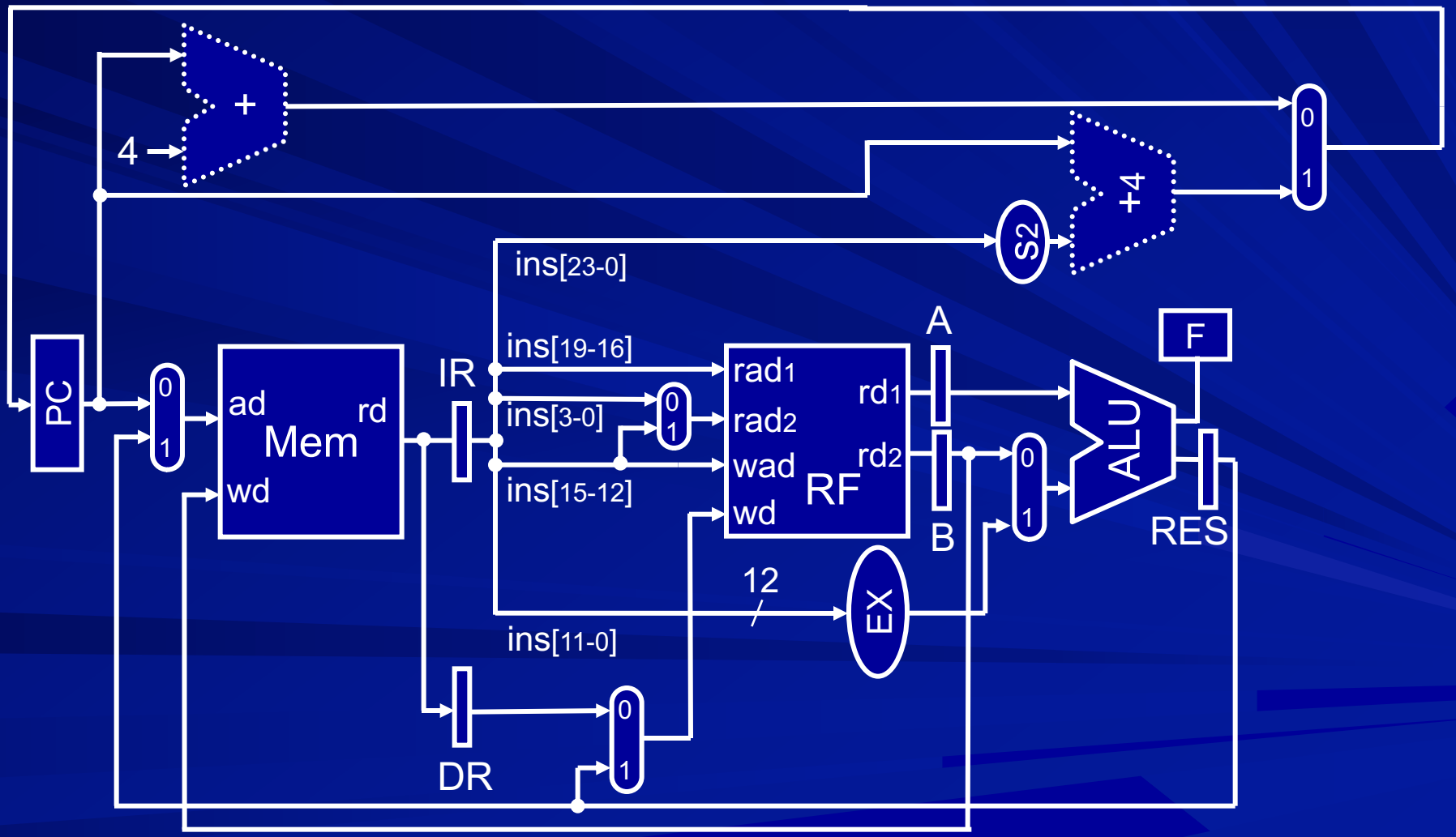


Merge IM and DM

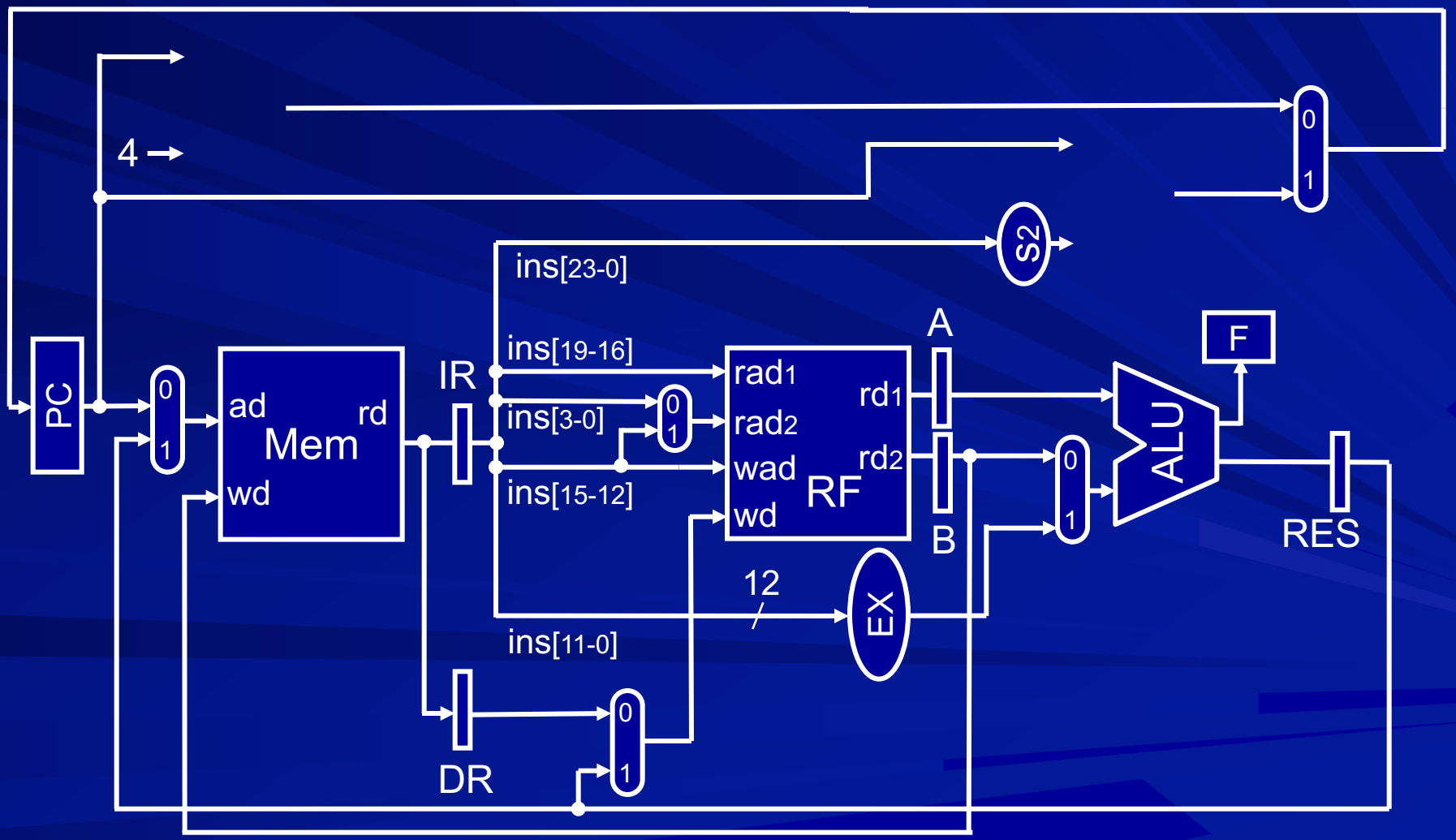


Eliminate adders

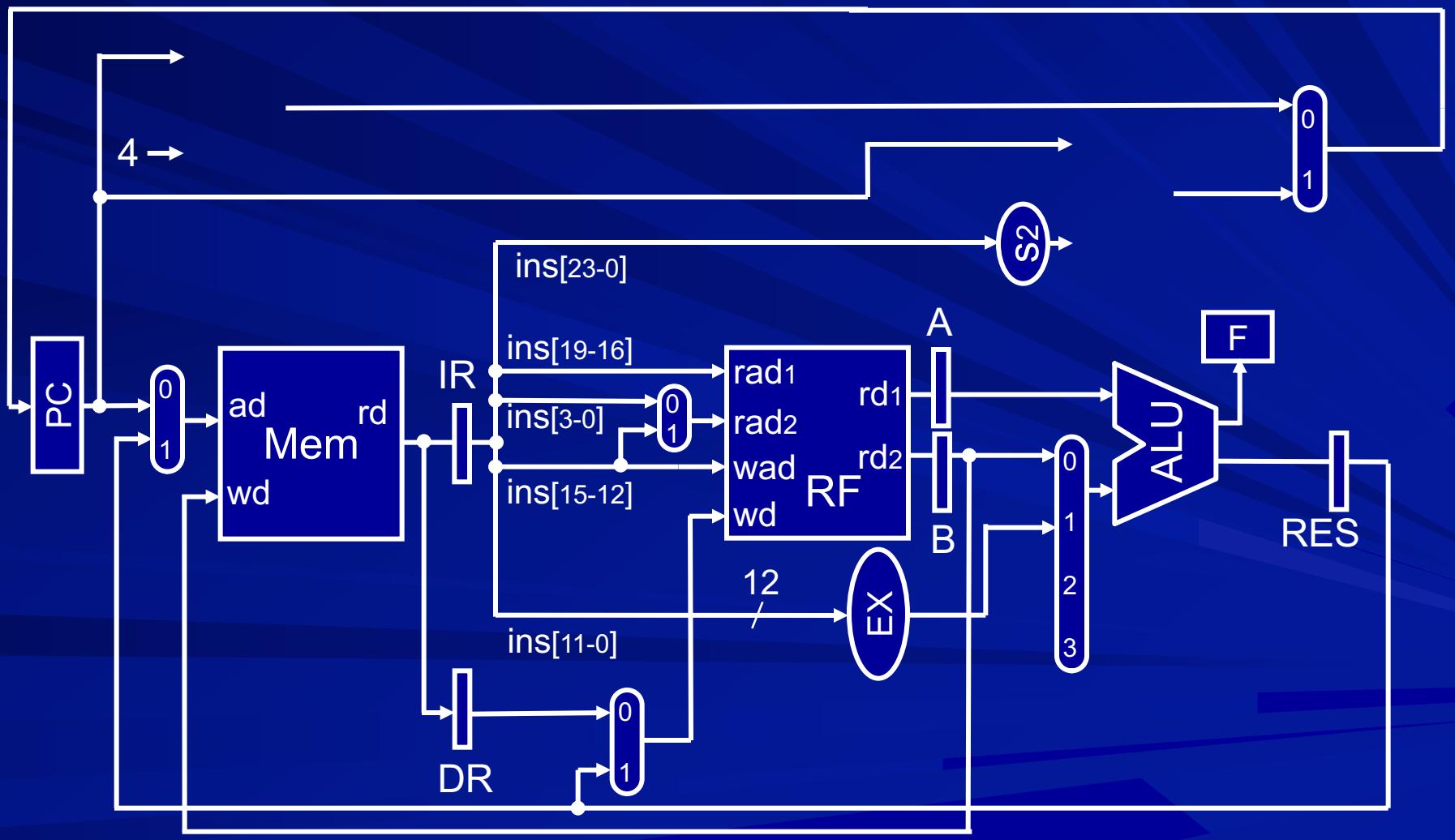
Use ALU



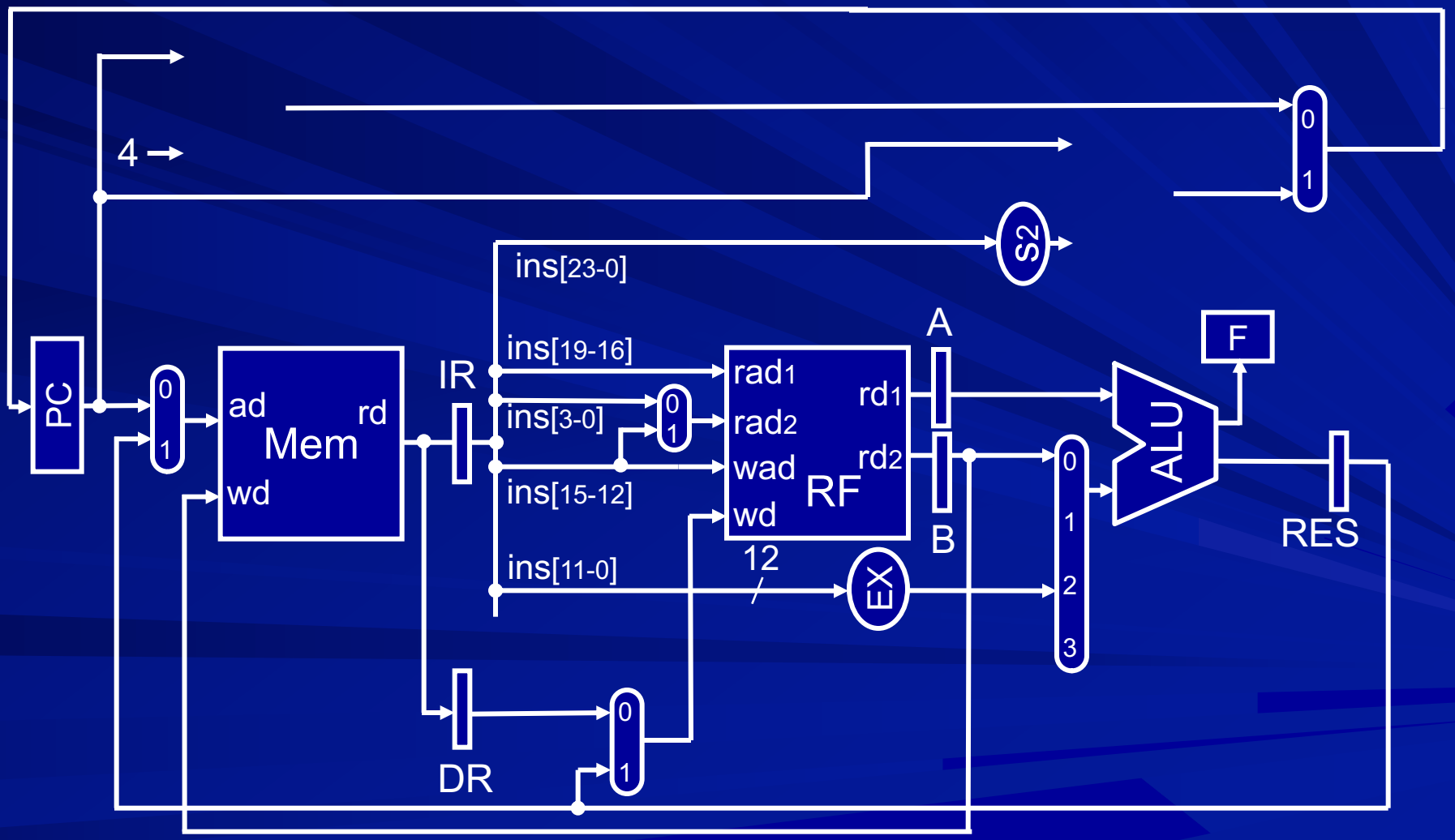
Eliminate adders



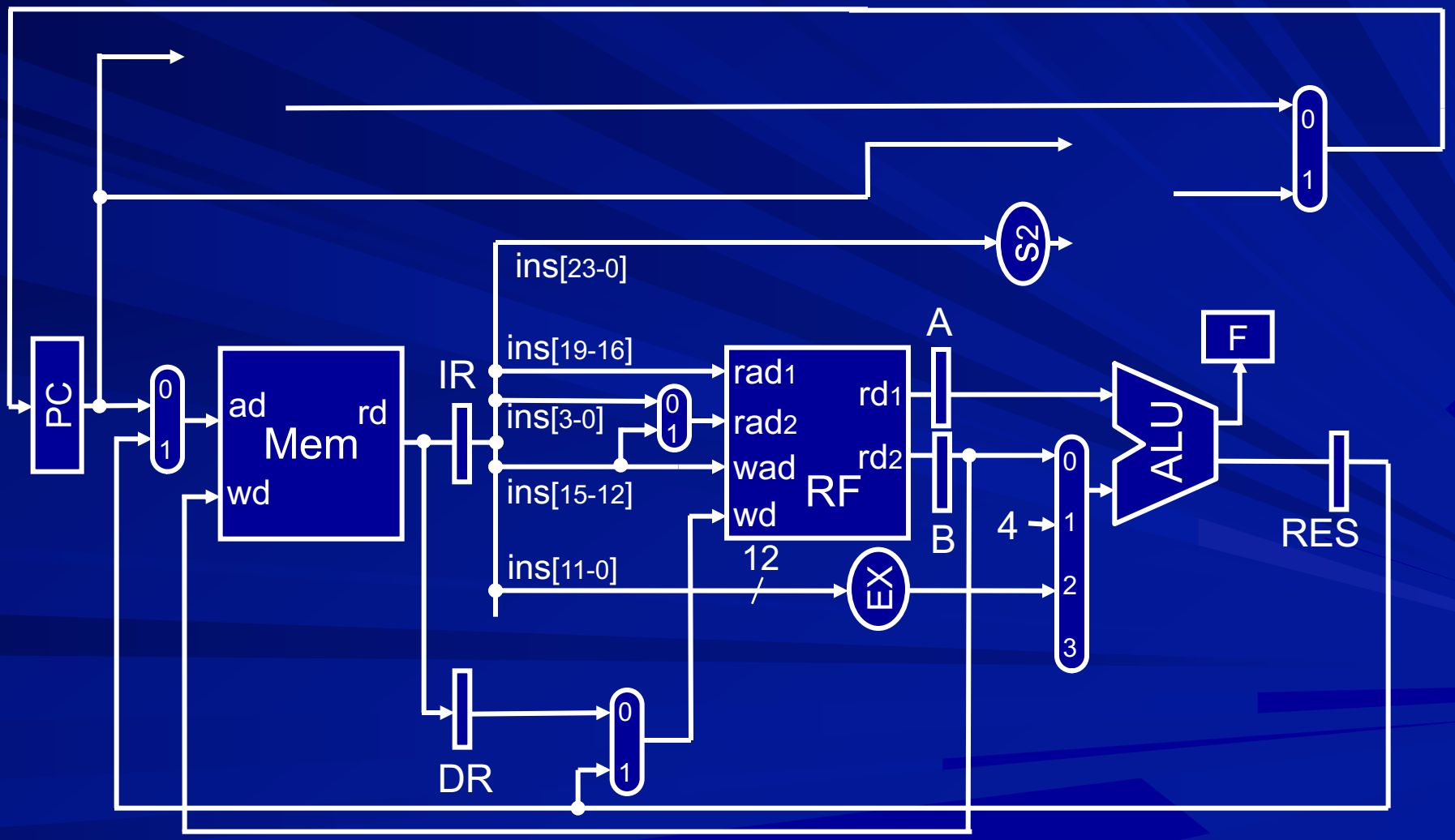
Eliminate adders



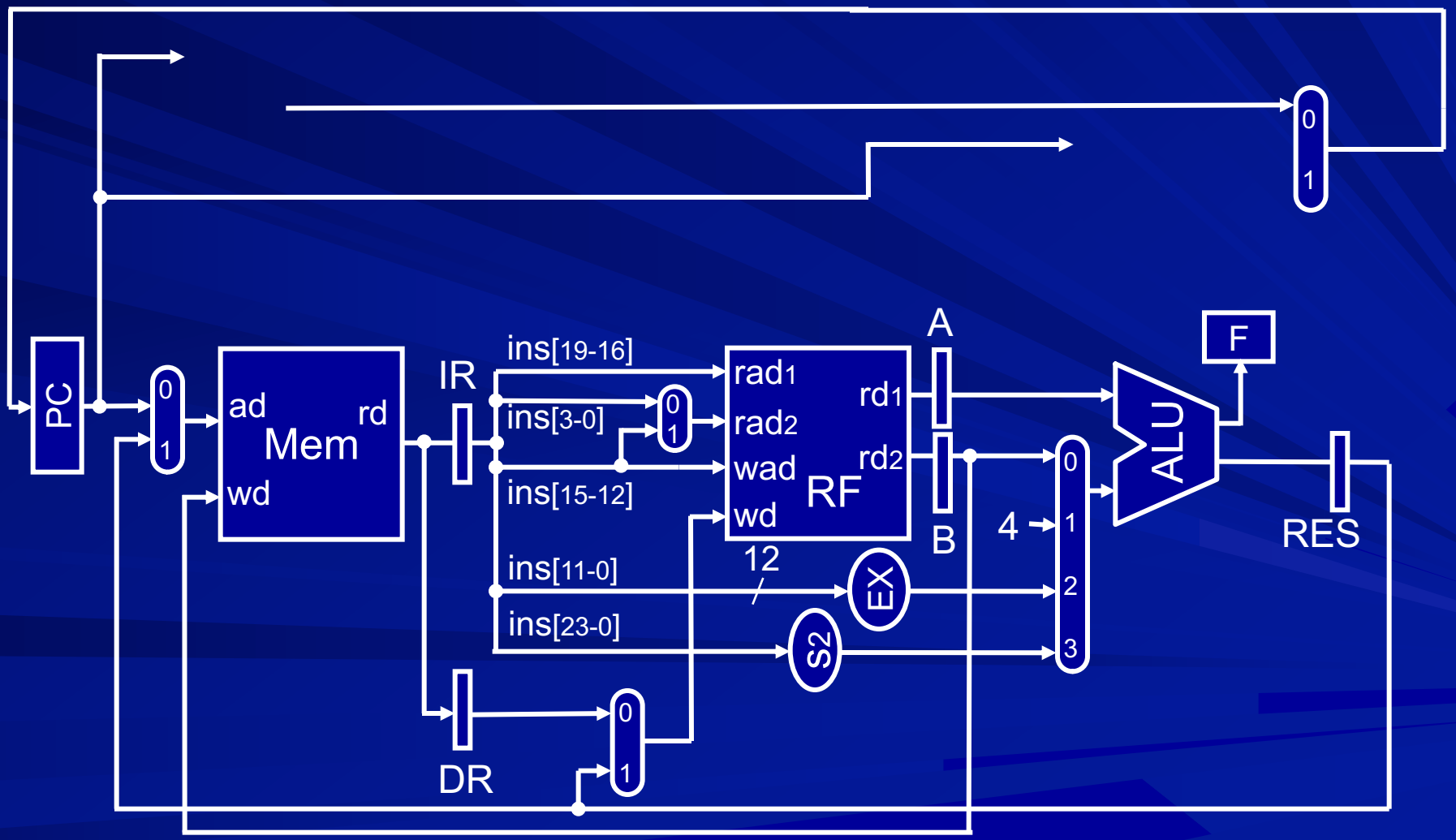
Eliminate adders



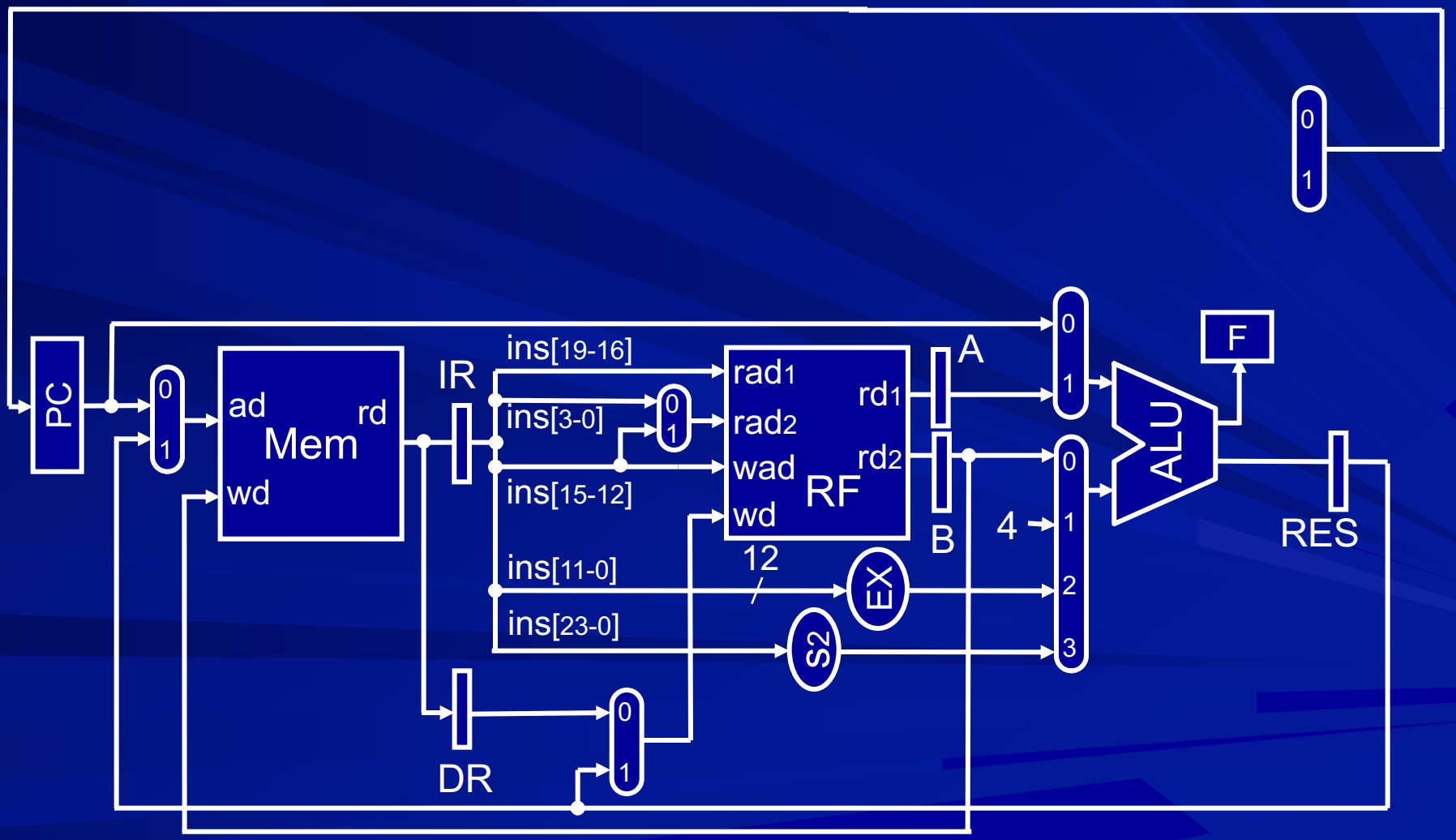
Eliminate adders



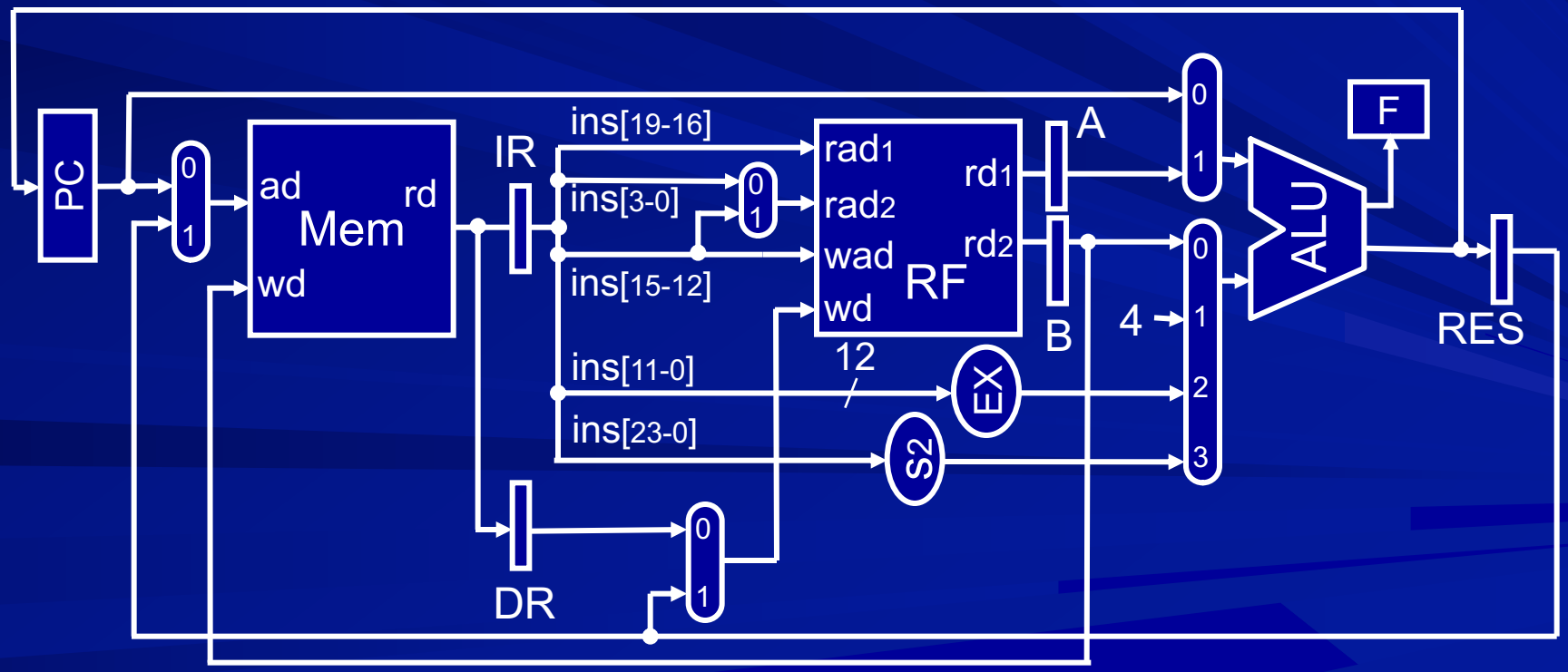
Eliminate adders



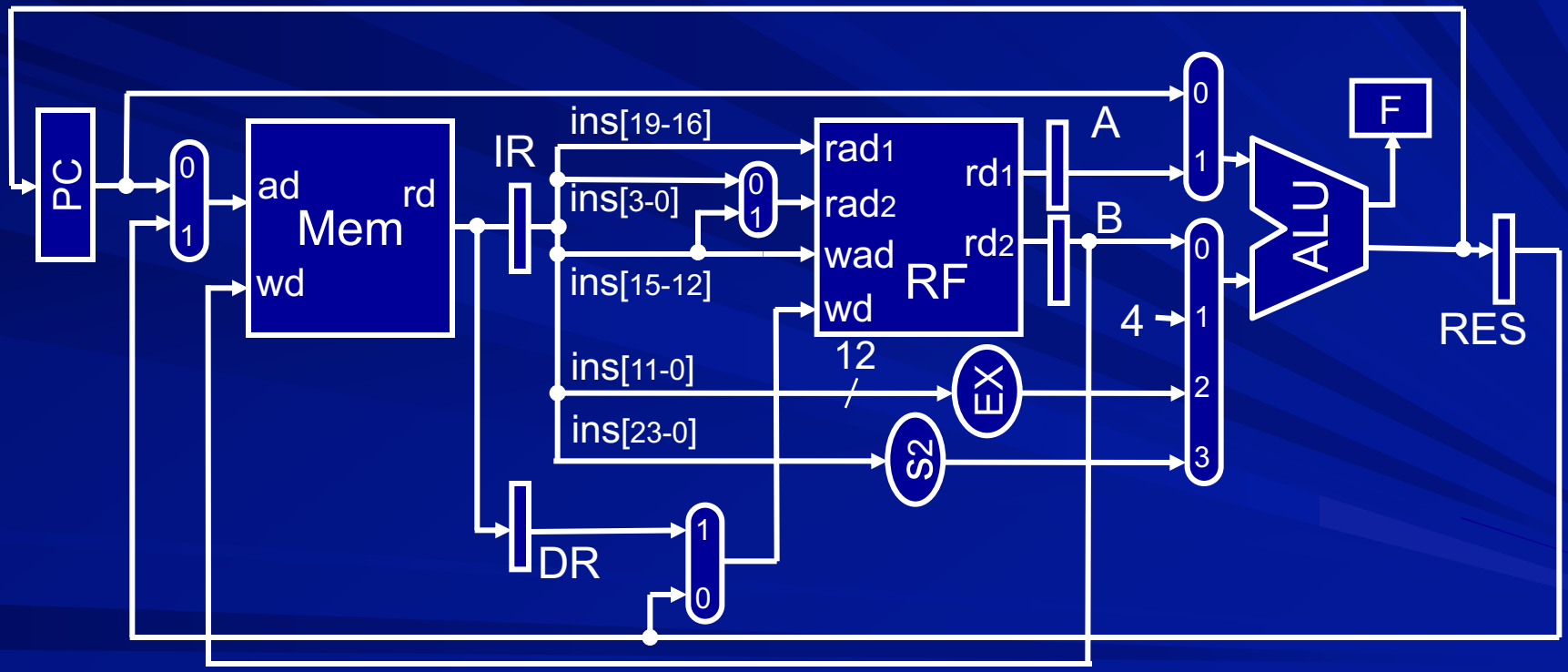
Eliminate adders



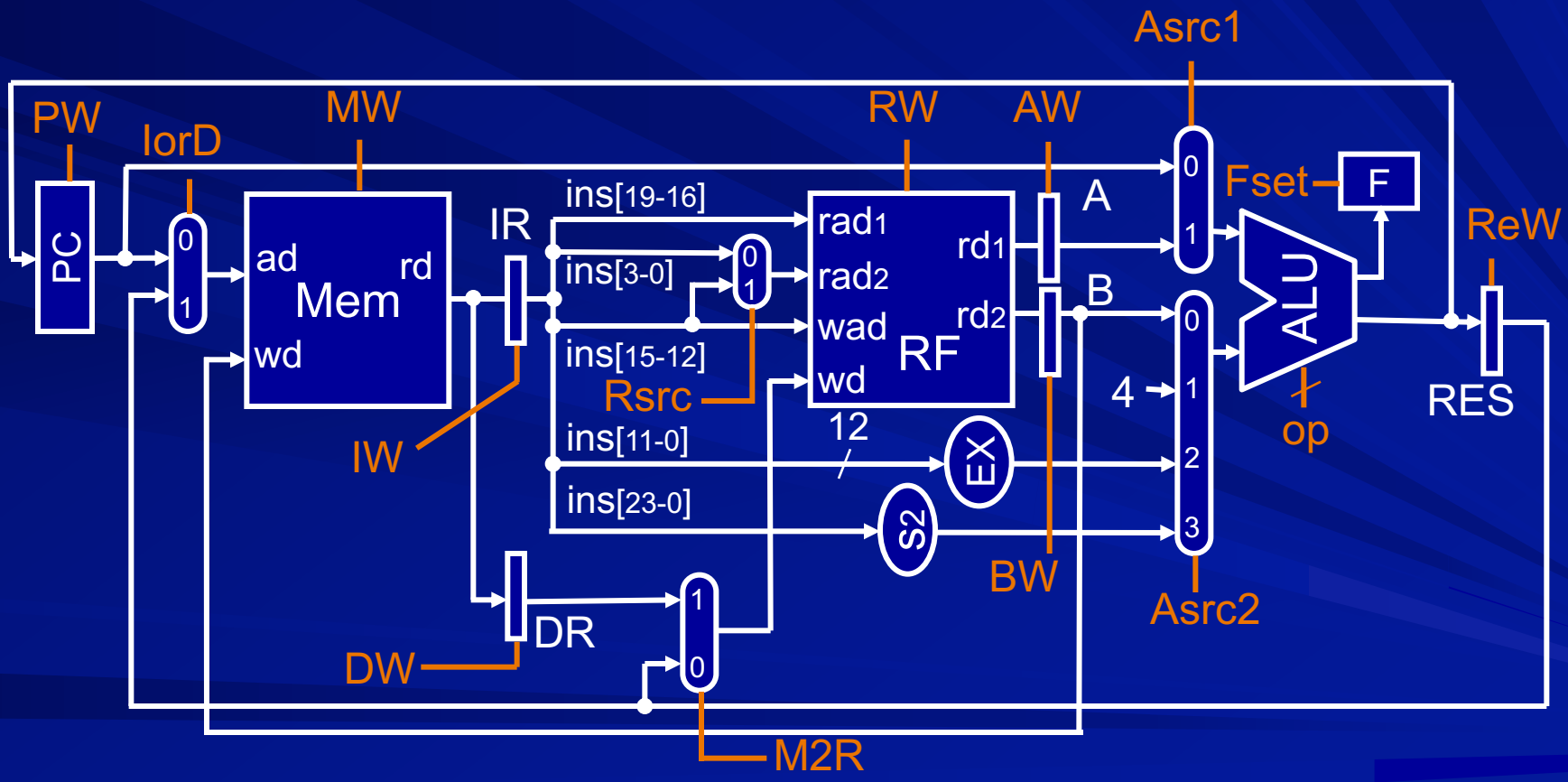
Eliminate adders



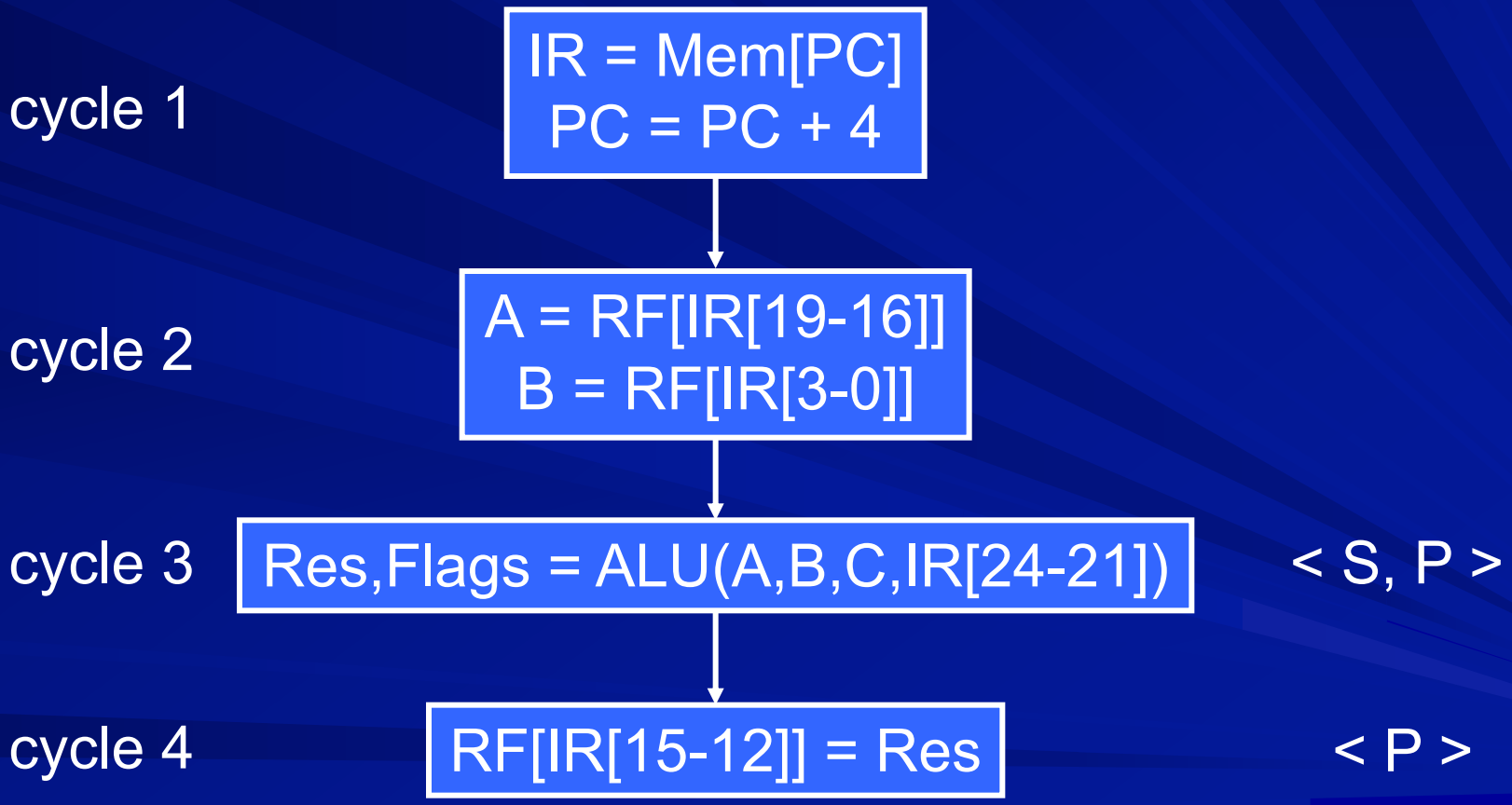
Multi-cycle Datapath



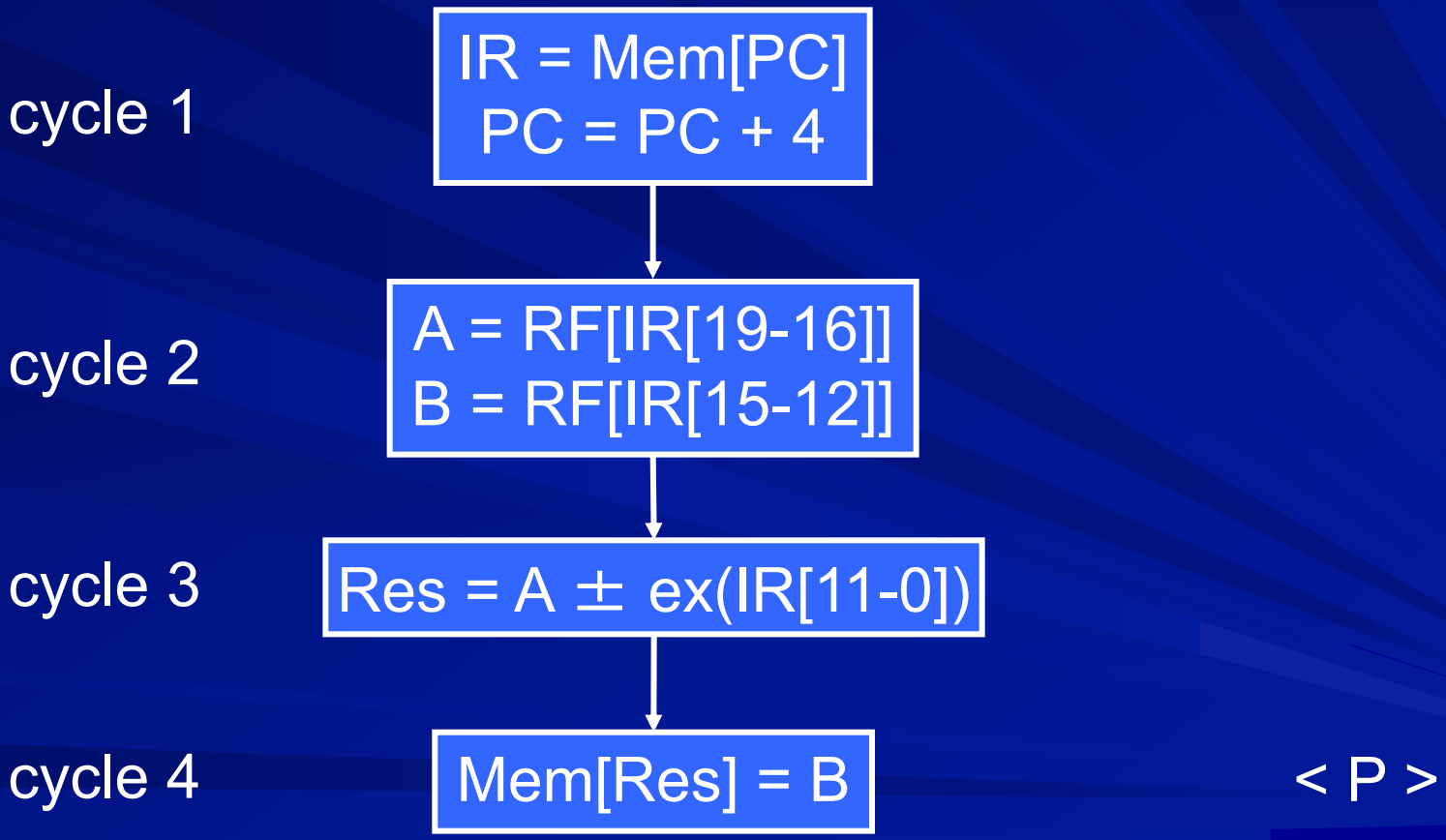
Control signals in multi-cycle datapath



Break Instruction Execution into Cycles: DP instructions

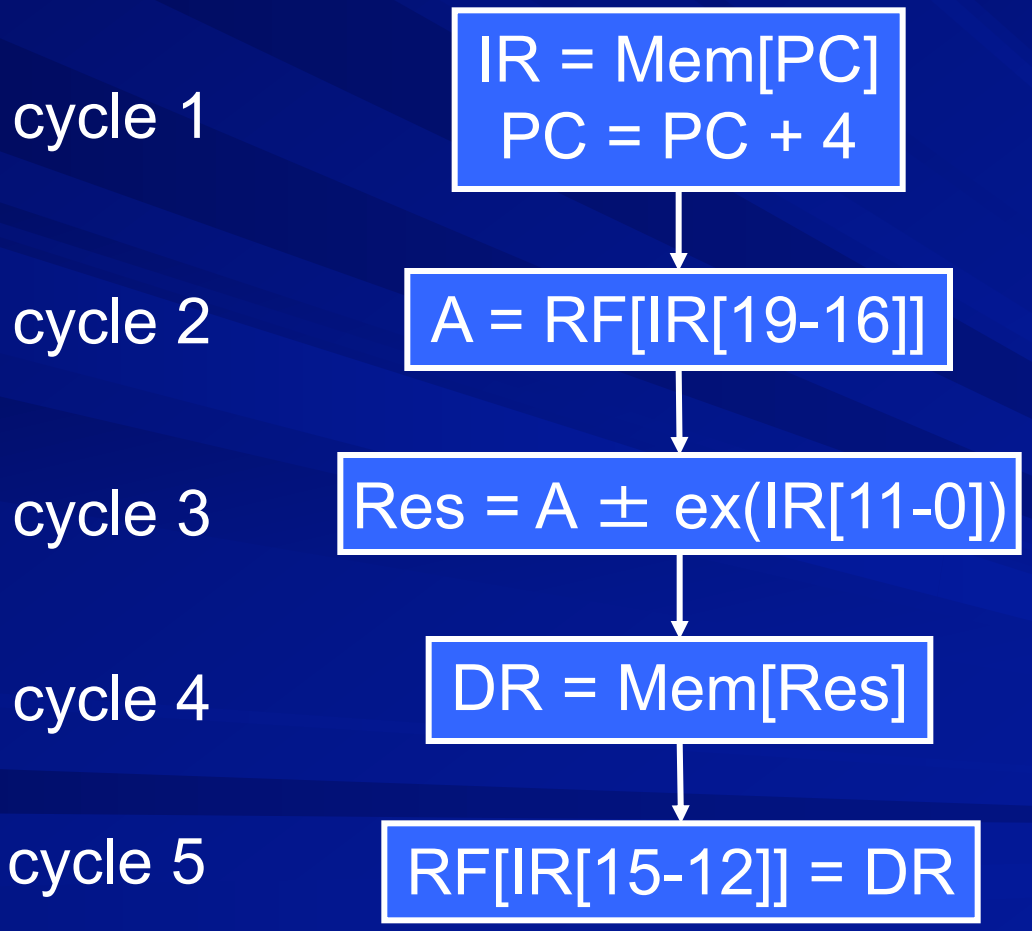


Break Instruction Execution into Cycles: str instruction



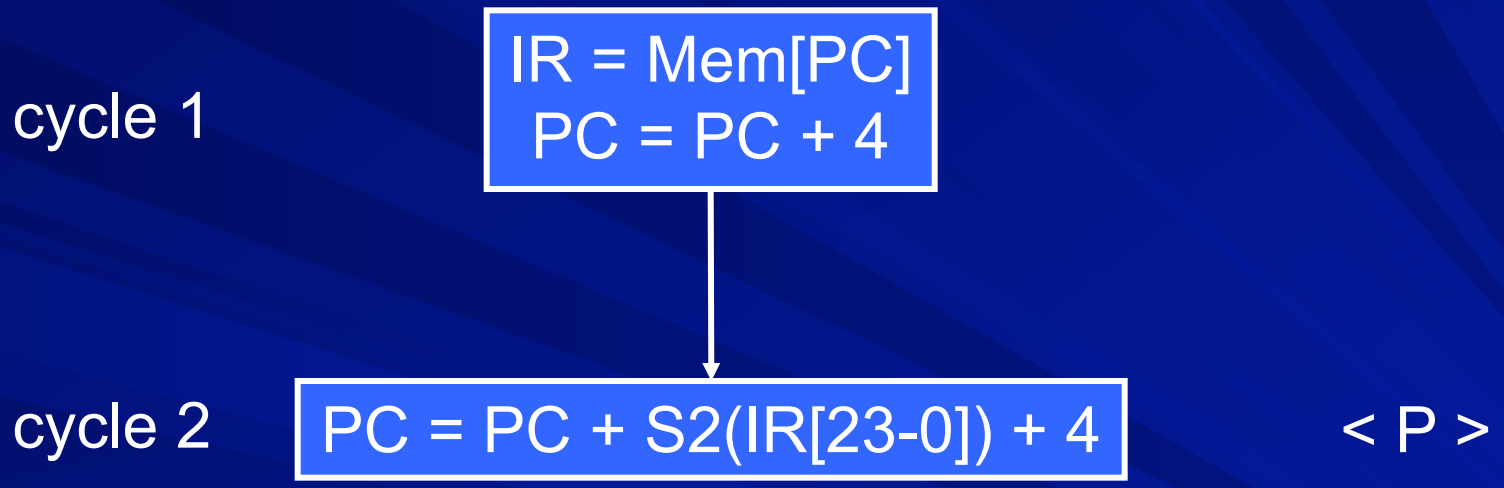
< P >

Break Instruction Execution into Cycles: ldr instruction

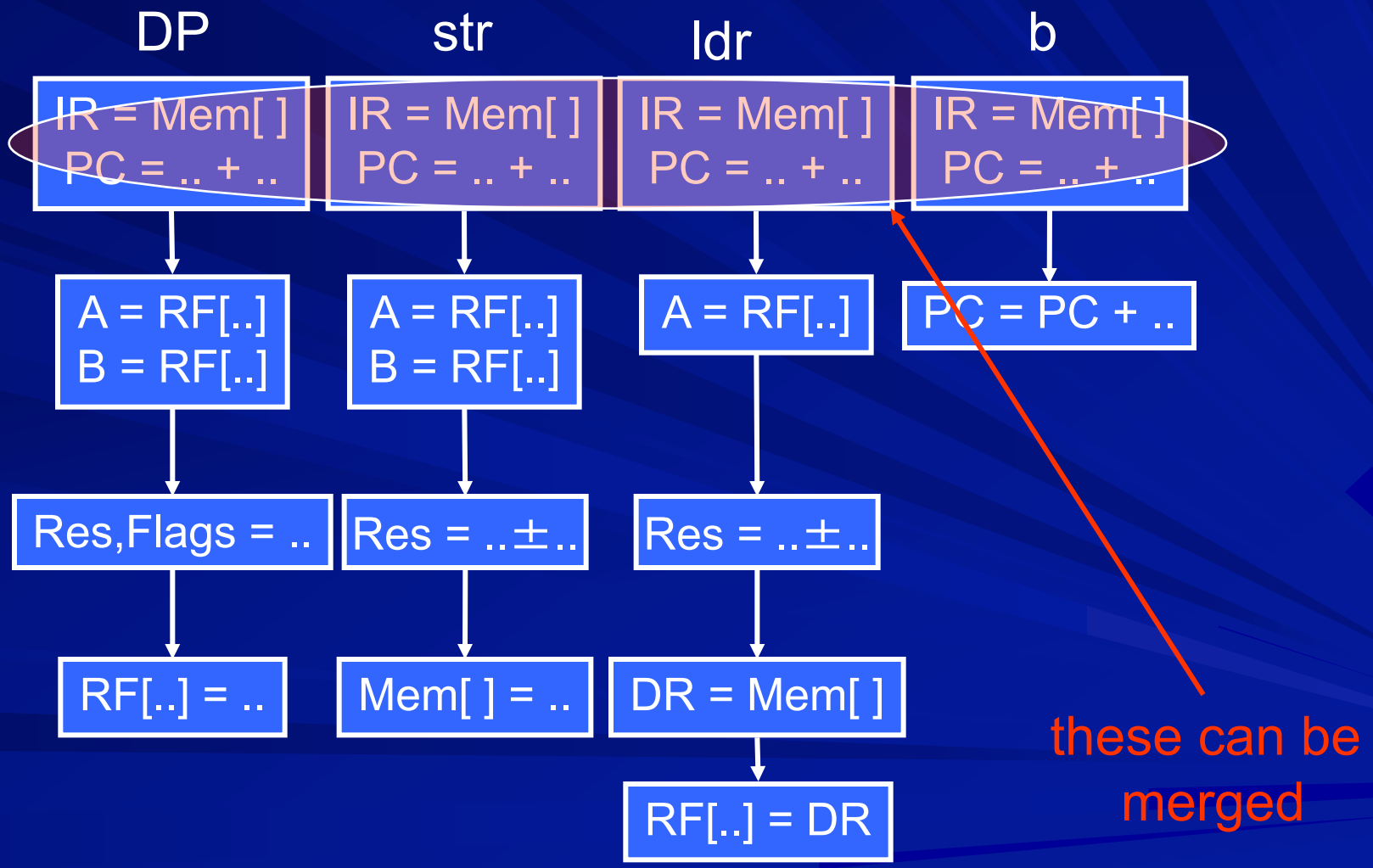


< P >

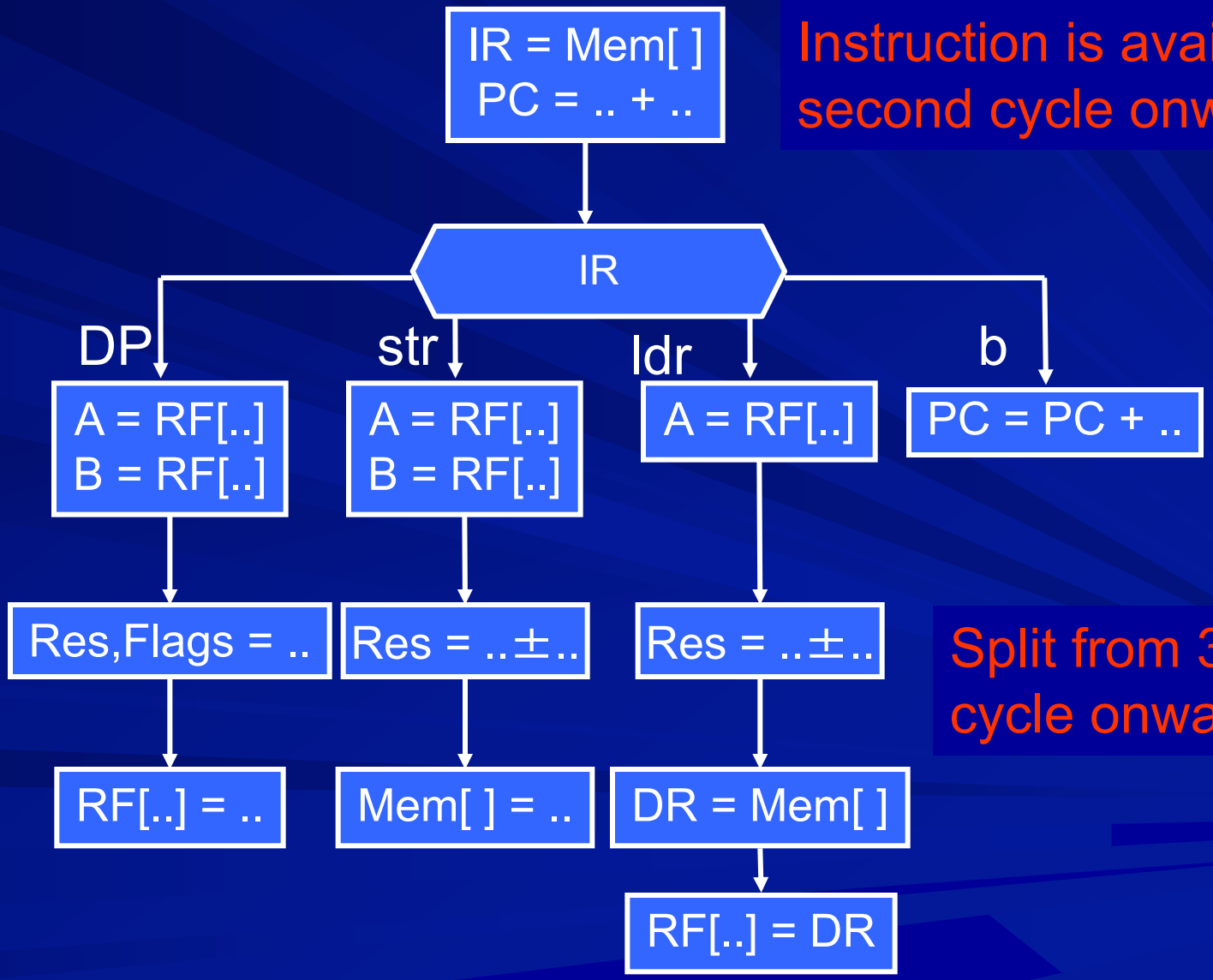
Break Instruction Execution into Cycles: b instruction



Put cycle sequences together



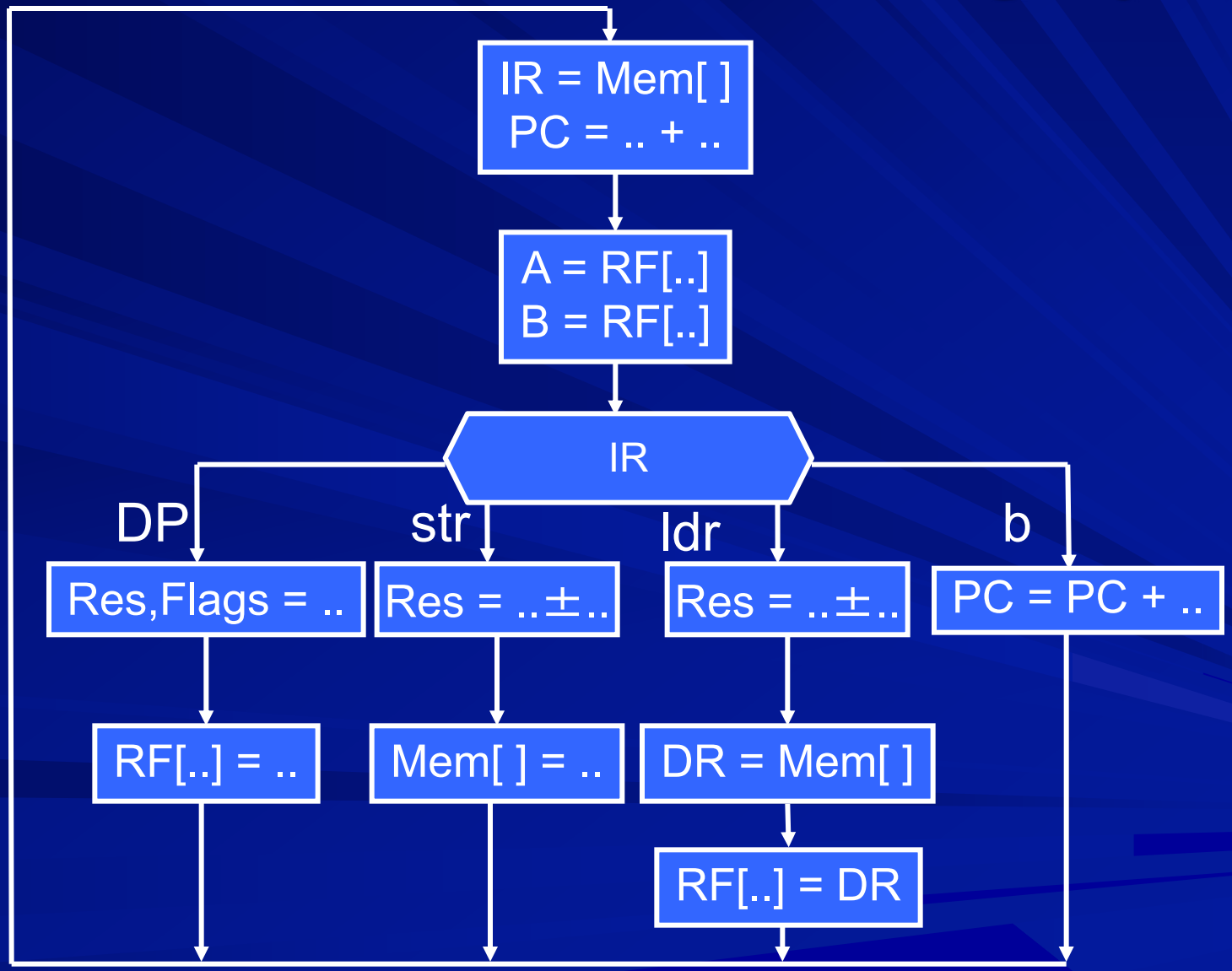
After merging fetch cycle



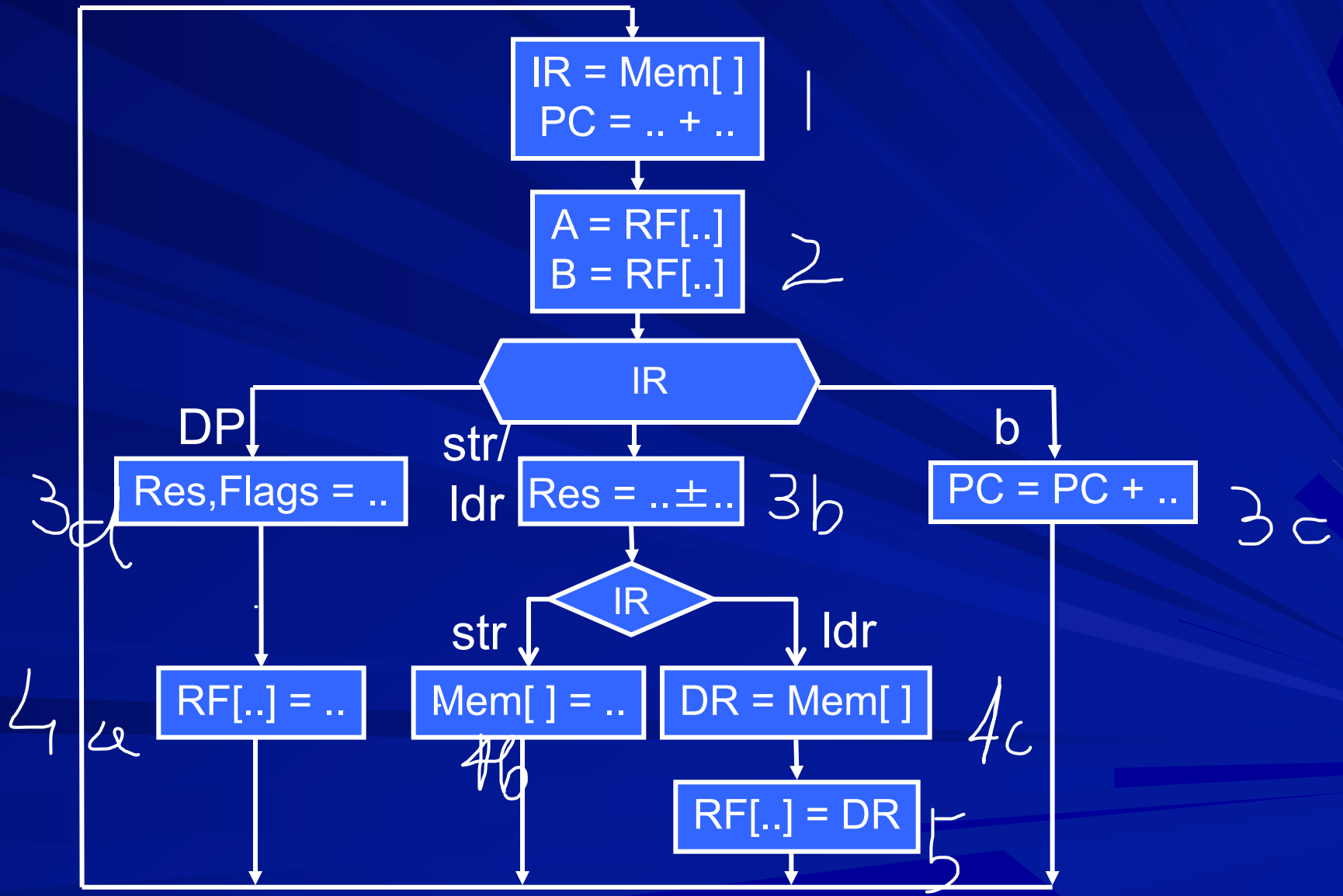
Instruction is available second cycle onwards.

Split from 3rd cycle onwards.

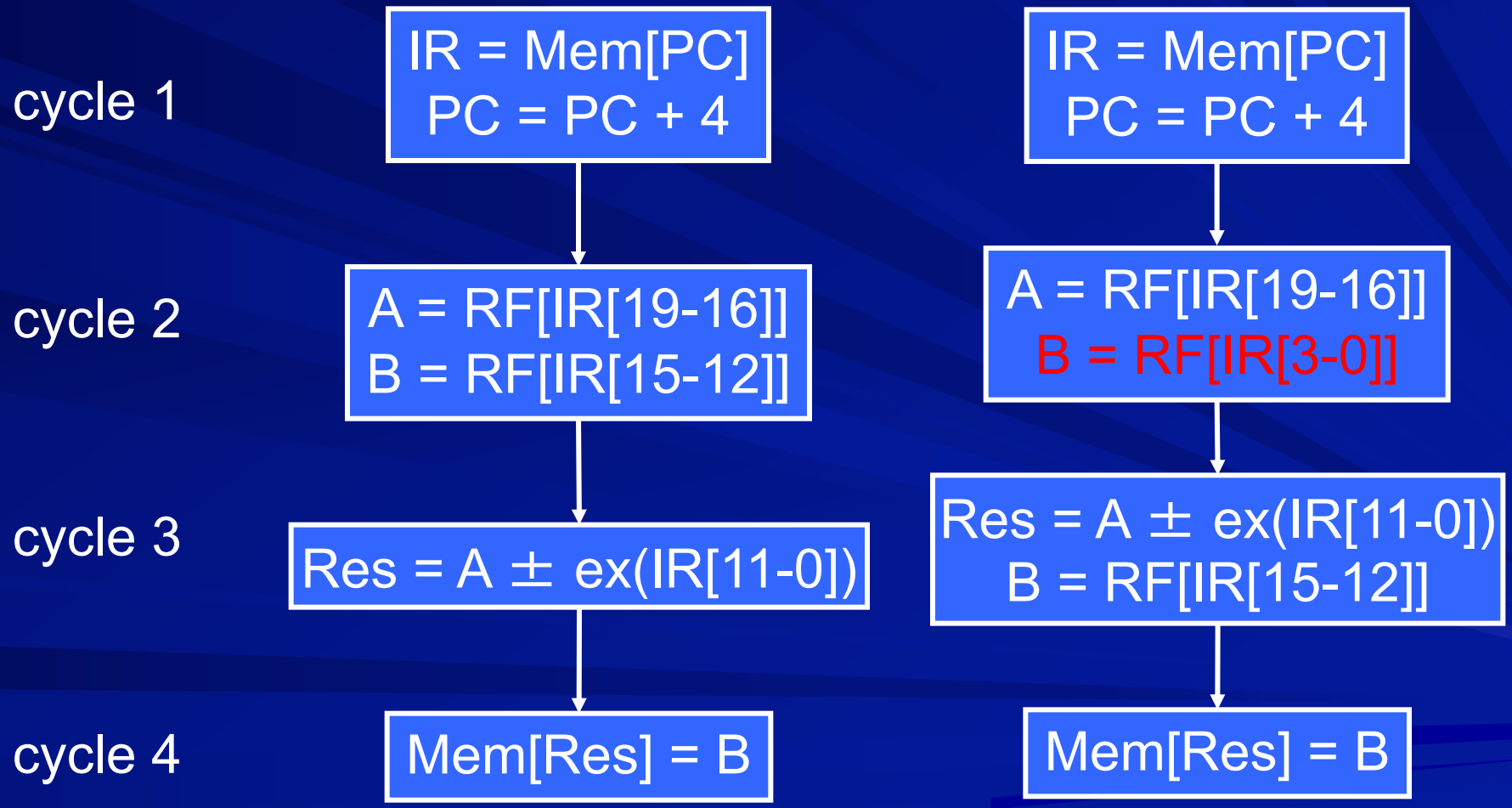
With a common decoding cycle



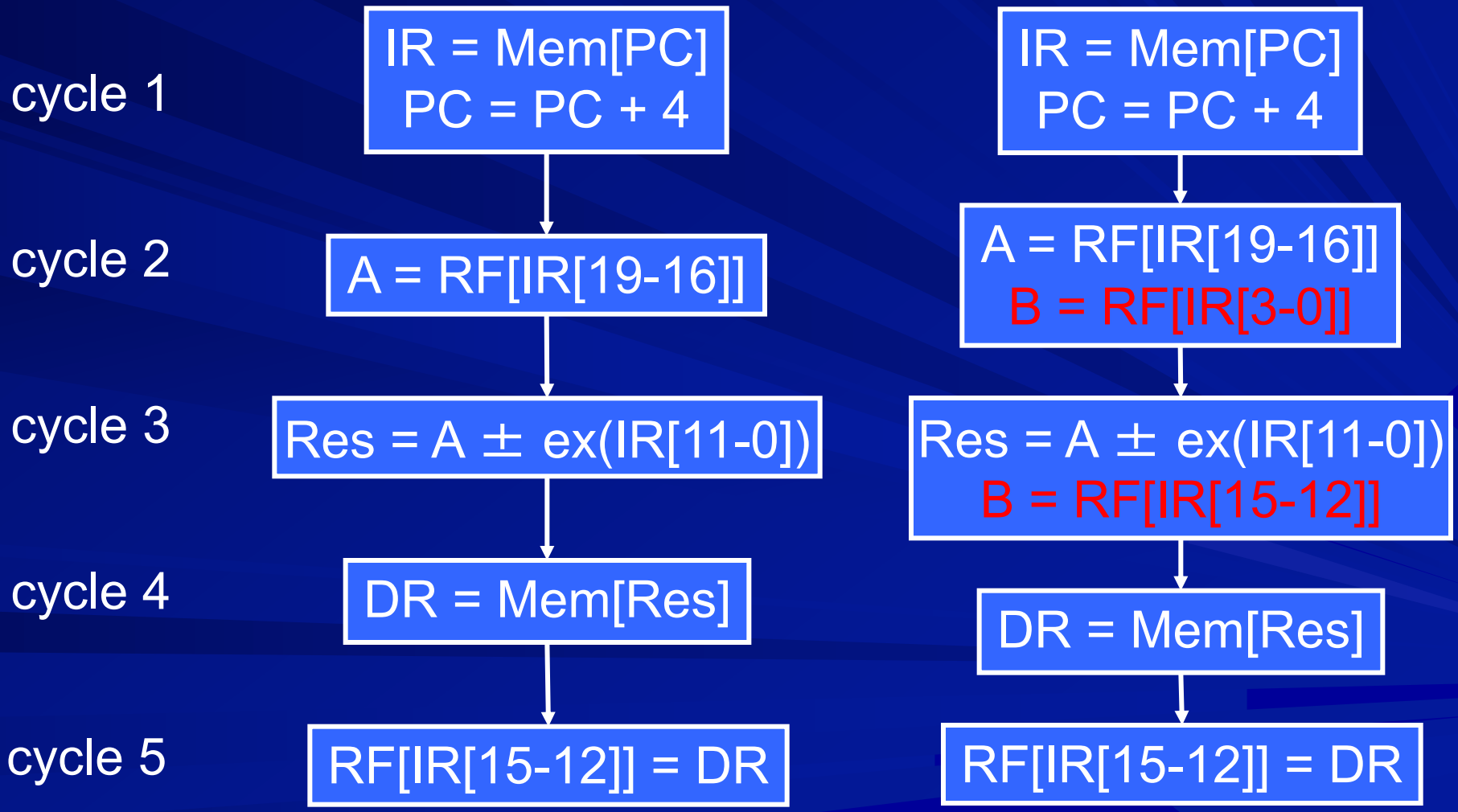
ldr, str can split after third cycle



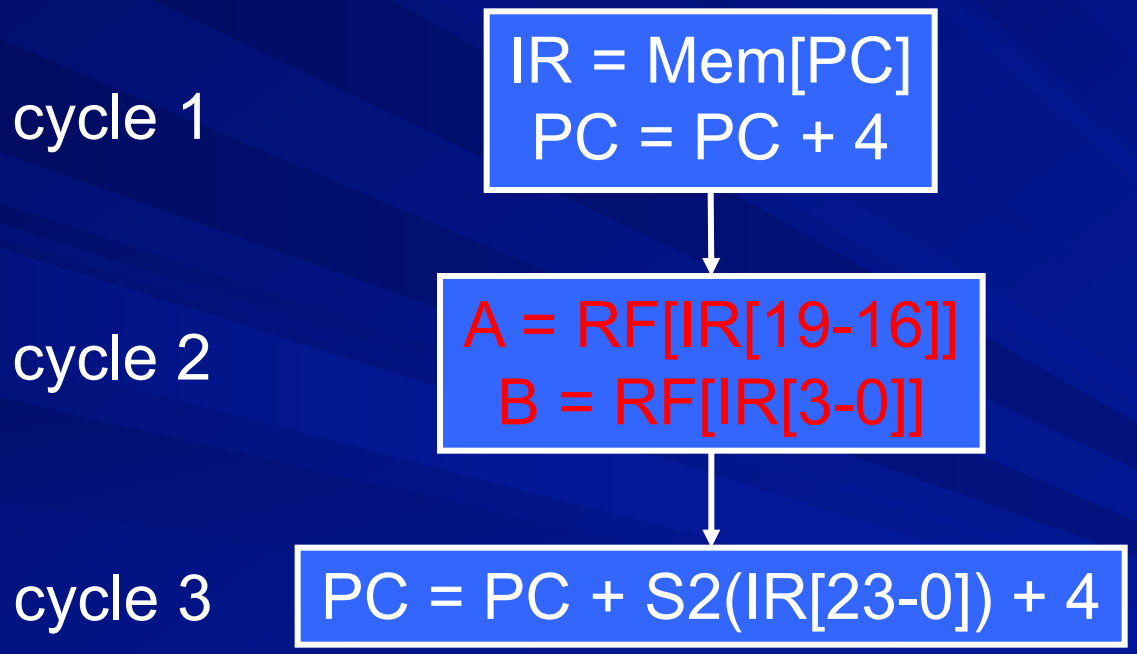
Modified str actions



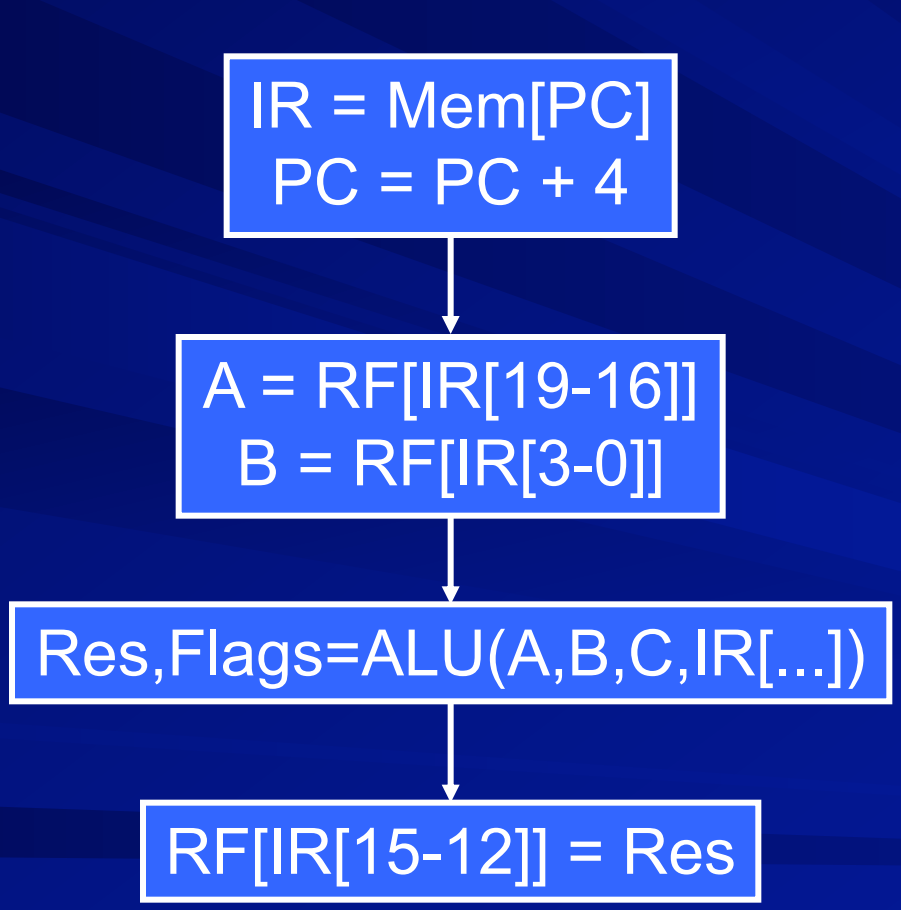
Modified ldr actions

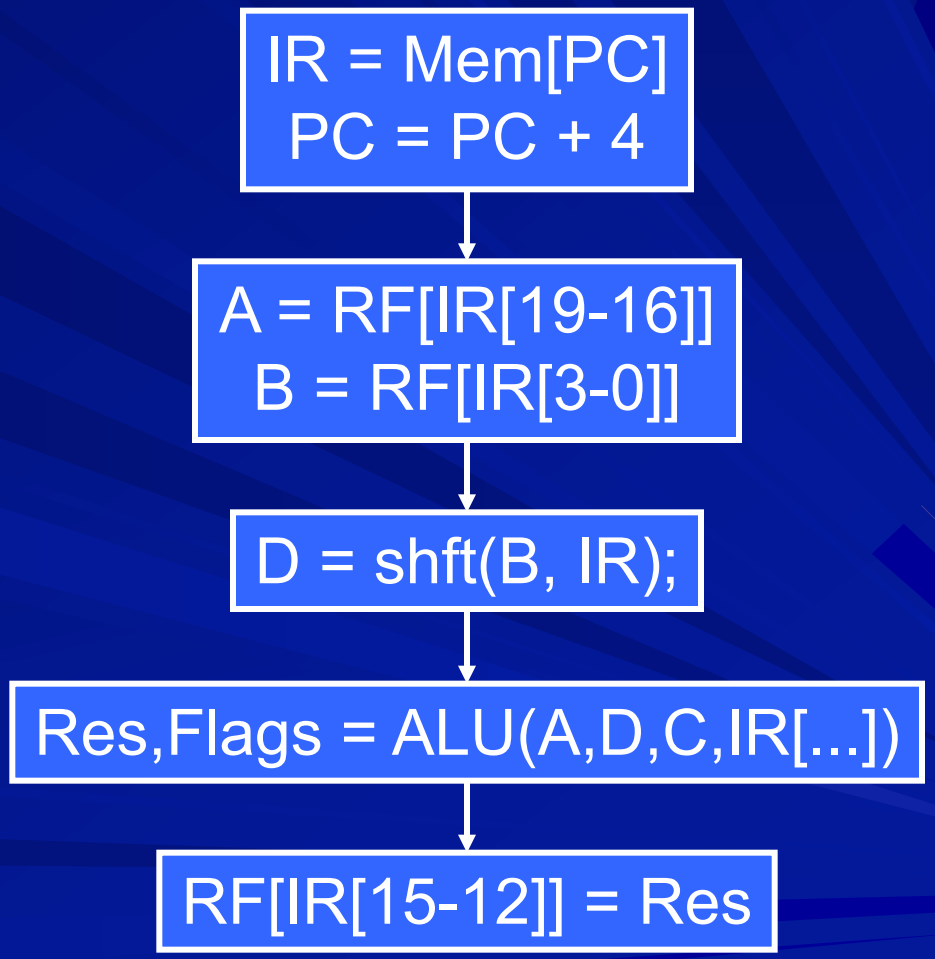


Modified b actions



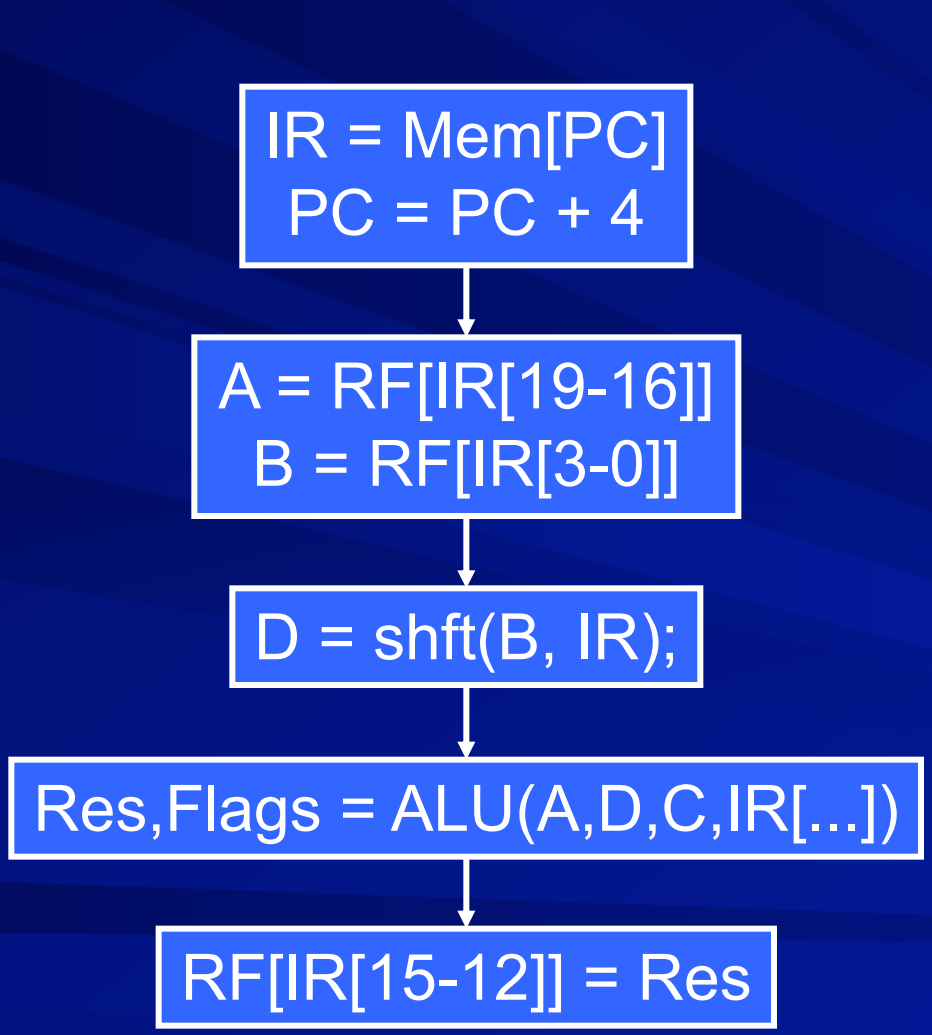
Other forms of DP instructions



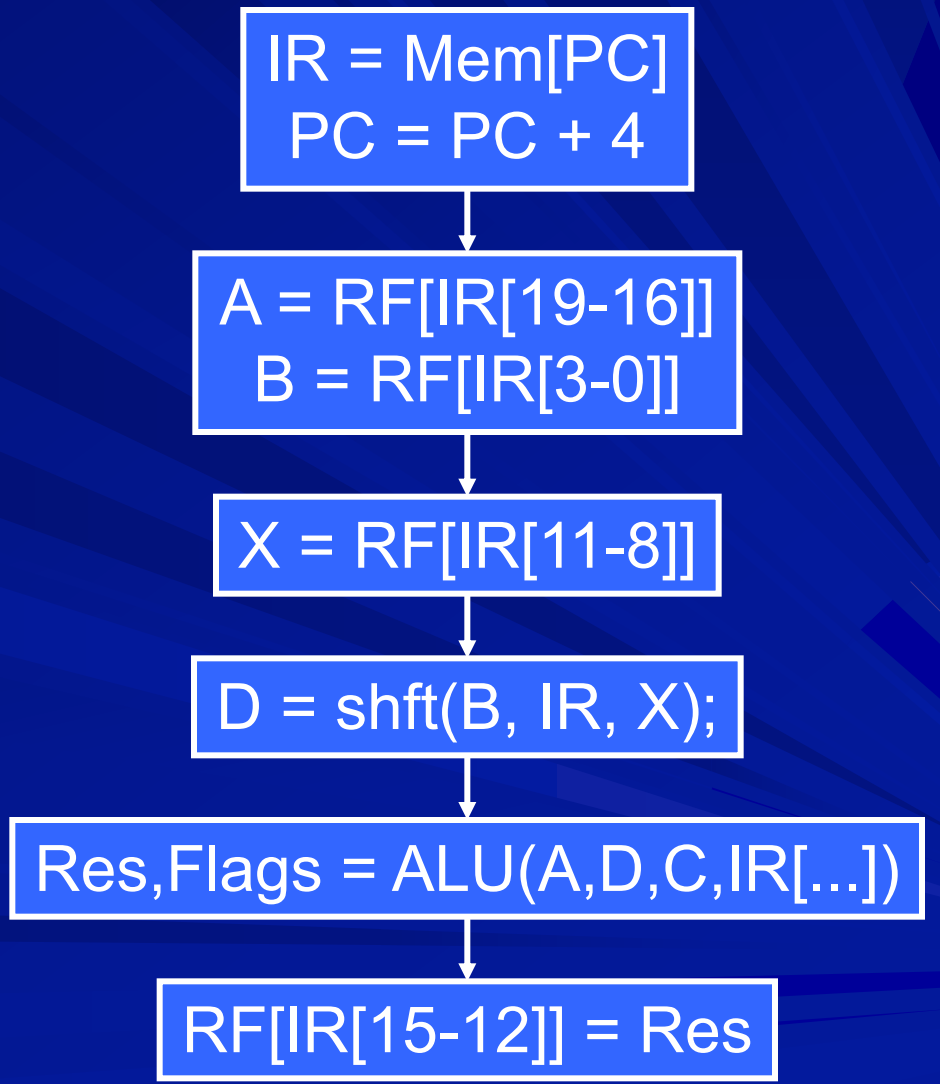


constant shift

Other forms of DP instructions

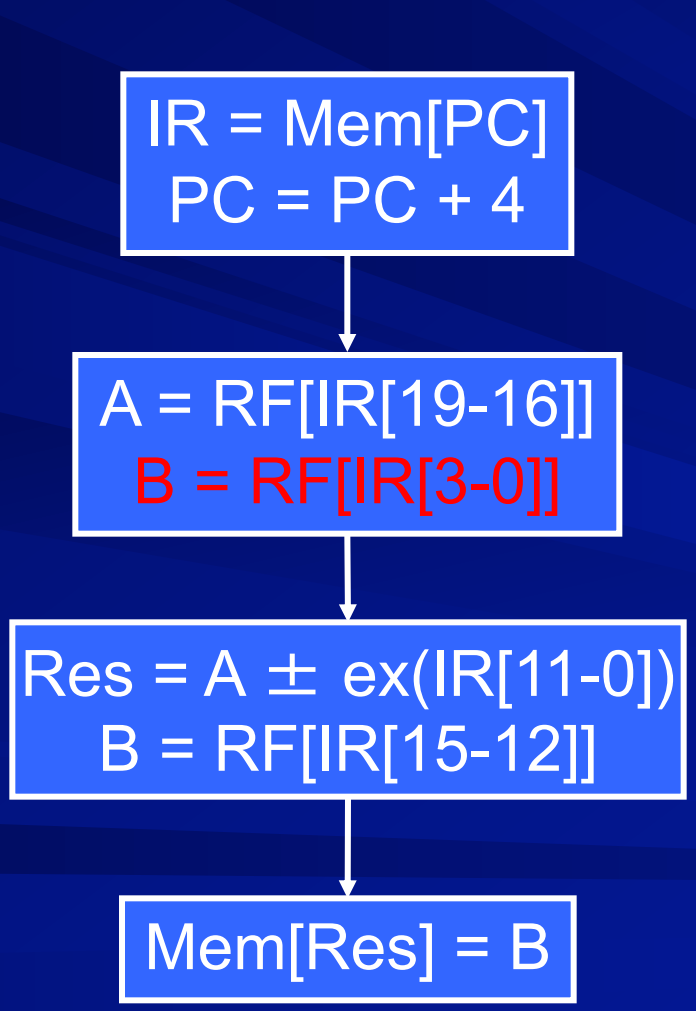


constant shift

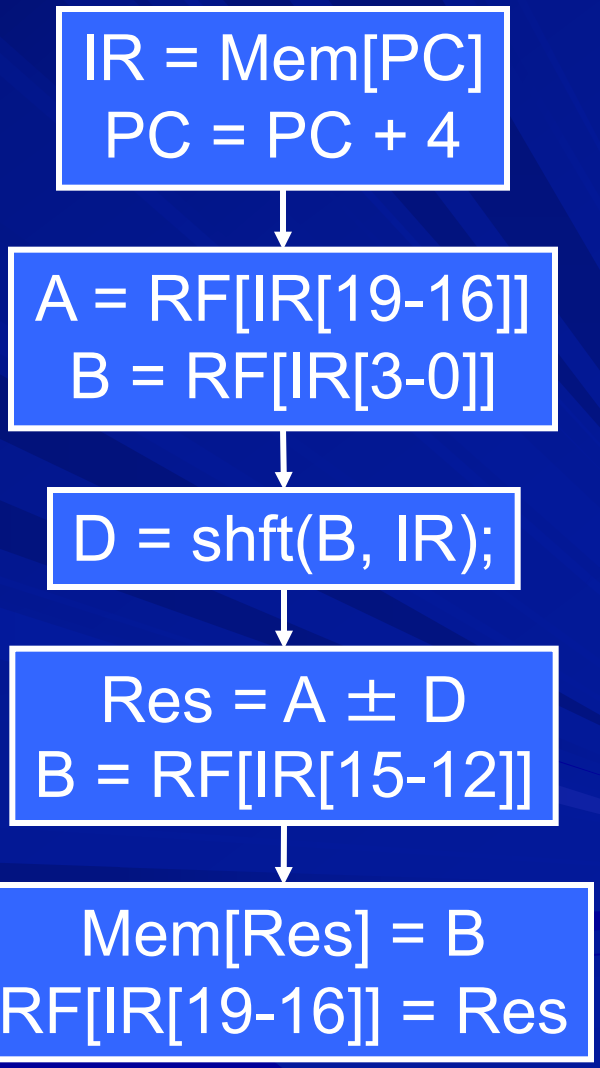


register specified shift

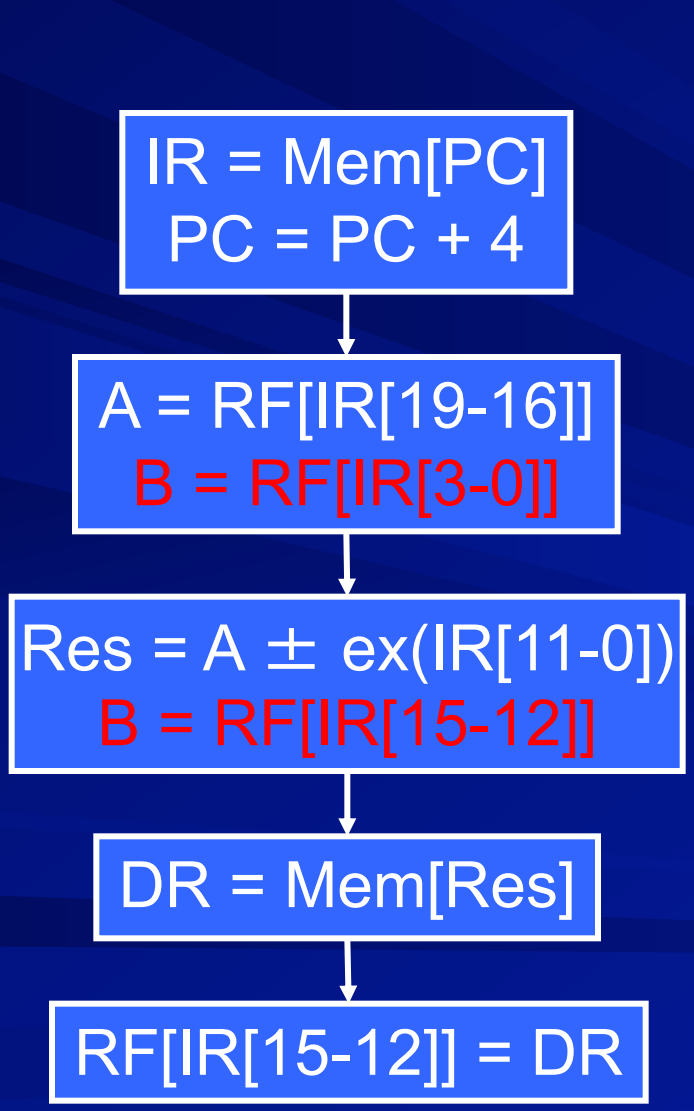
Other forms of str instruction



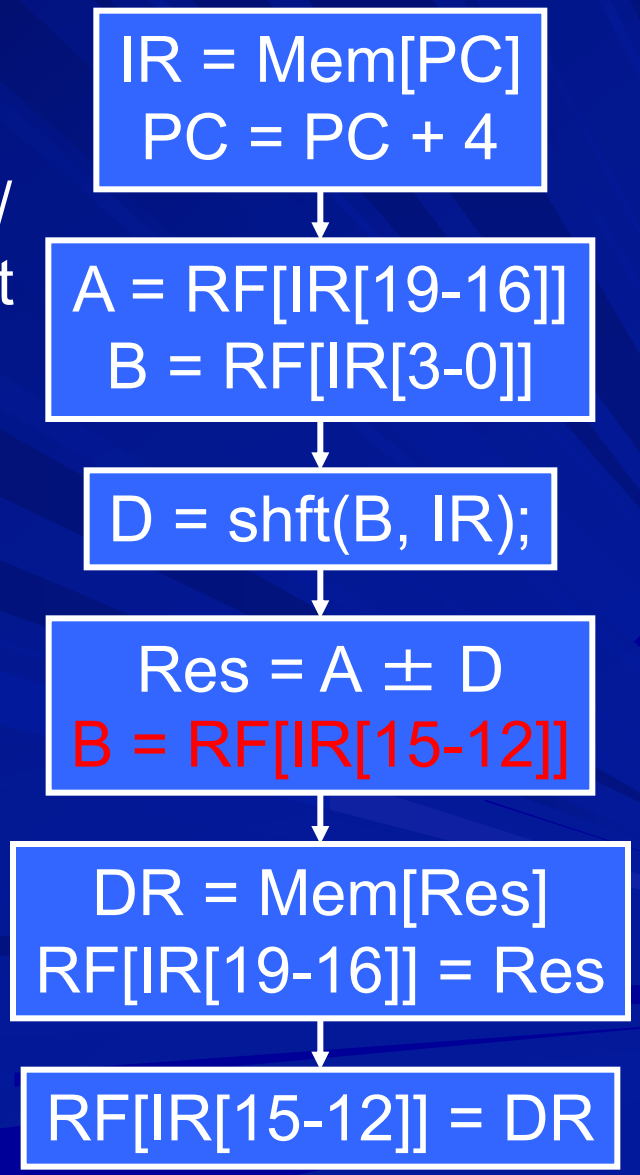
auto (pre) increment/ decrement register offset



Other forms of ldr instruction



auto (pre) increment/ decrement register offset



Thanks