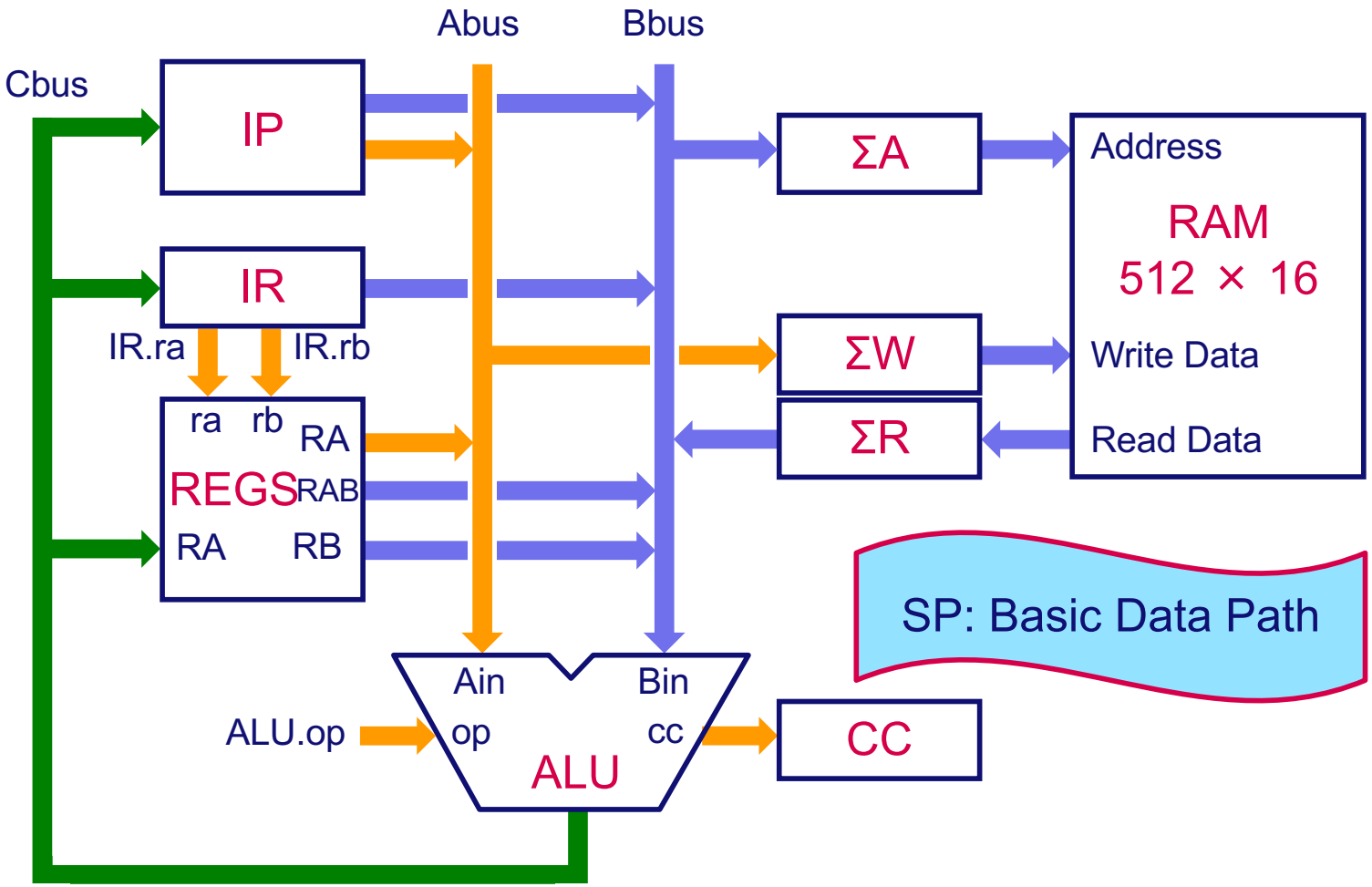
2IC30: Computer systems Instruction Execution: Speeding Up by Prefetching. Processor basics: Binary and Assembly Language; ISA instruction types.

Jan Friso Groote

TU/e Technische Universiteit Eindhoven University of Technology

Where innovation starts

Simple processor: data path

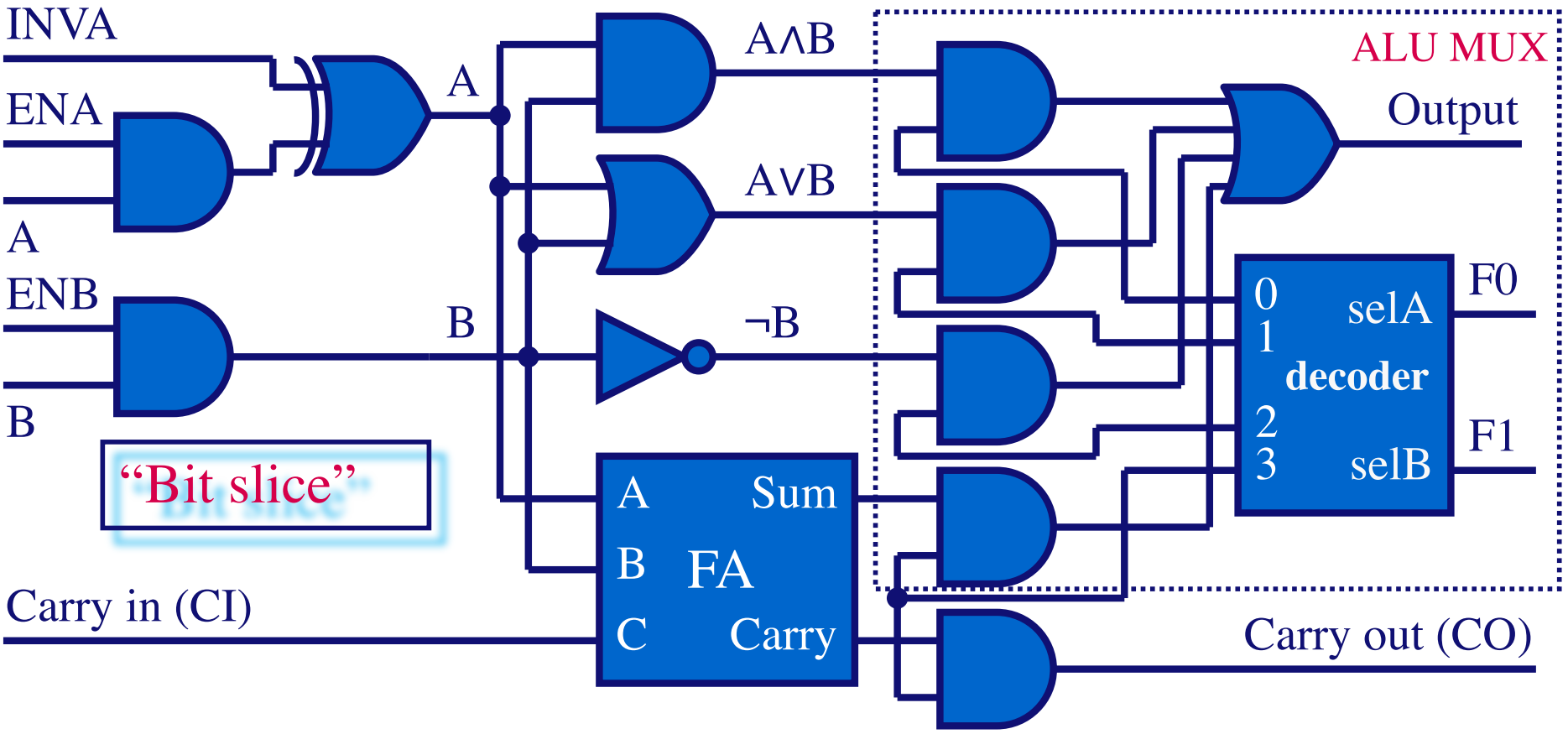


- Abus: IP or RA
- Bbus: IP, IR, ΣR, RB or RAB

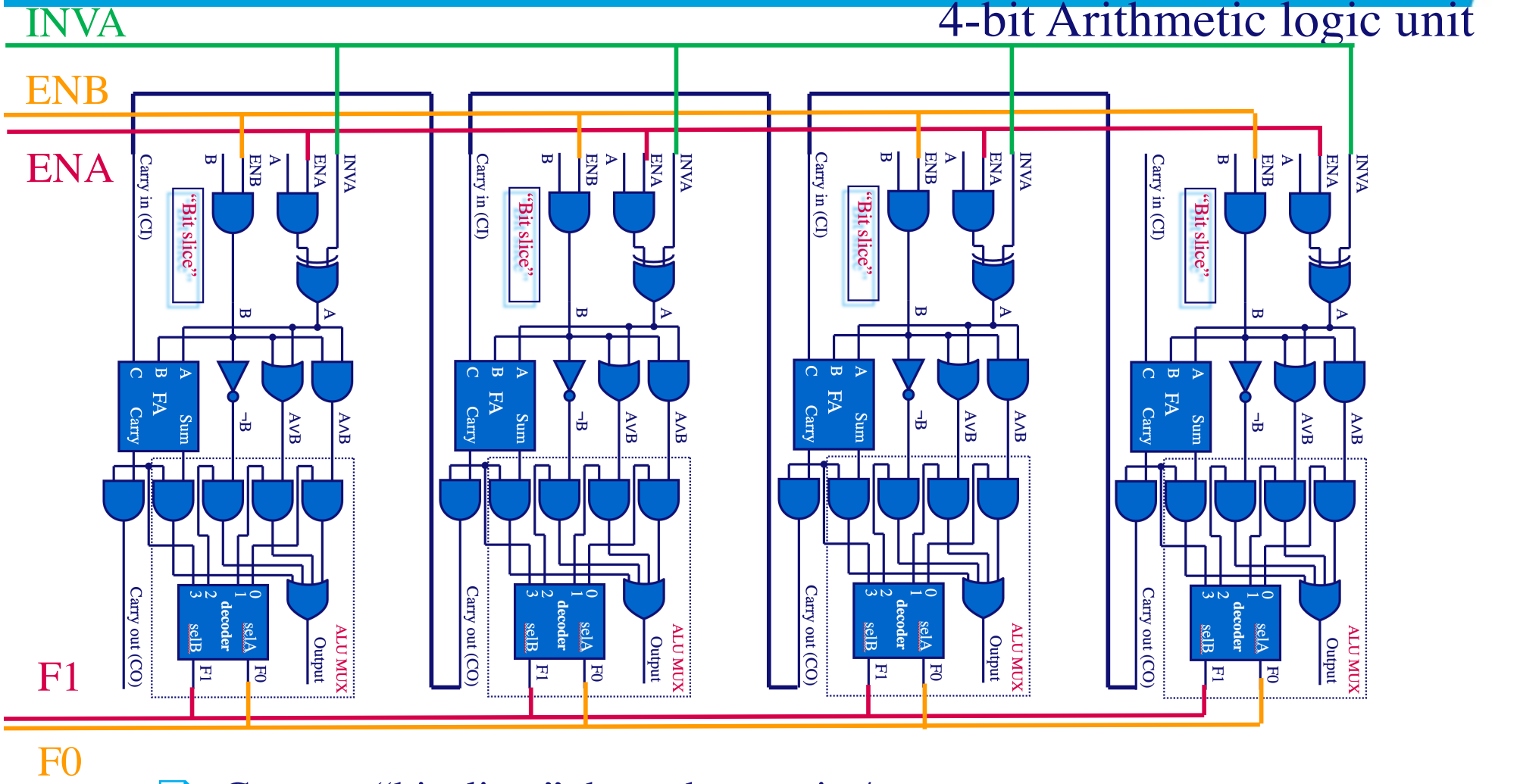
- REGS: 2 Reads and 1 Write, simultaneously, ra[1..0] and rb[1..0] select registers

The arithmetic logical unit (1)

- Hardware for one bit is doable (the ALU in the reader is different)...



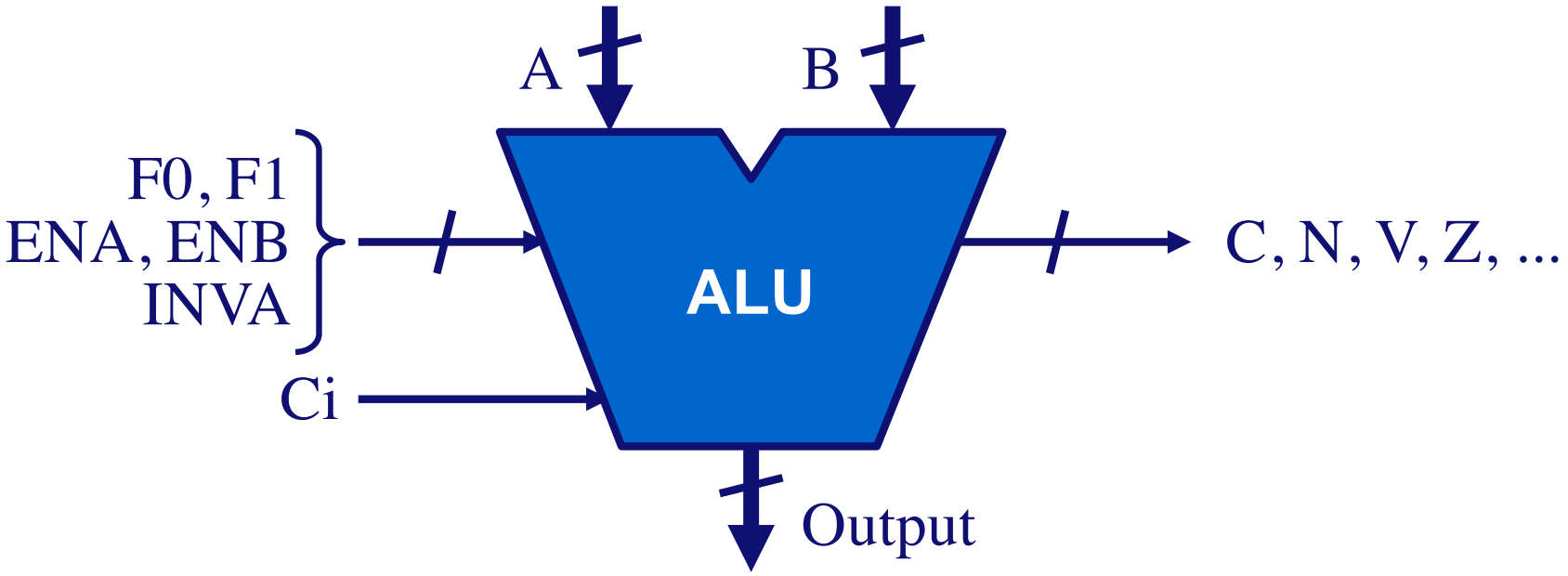
The arithmetic logical unit (2)



- Connect "bit slices" through carry in / carry out
- Control with F0, F1, ENA, ENB and INVA (equal for all slices)
- Tests: add logic for Negative, oVerflow and Zero

The arithmetic logical unit (2)

- Connect “bit slices” through carry in / carry out
- Control with F0, F1, ENA, ENB and INVA (equal for all slices)
- Tests: add logic for Negative, oVerflow and Zero



Functions of the example ALU

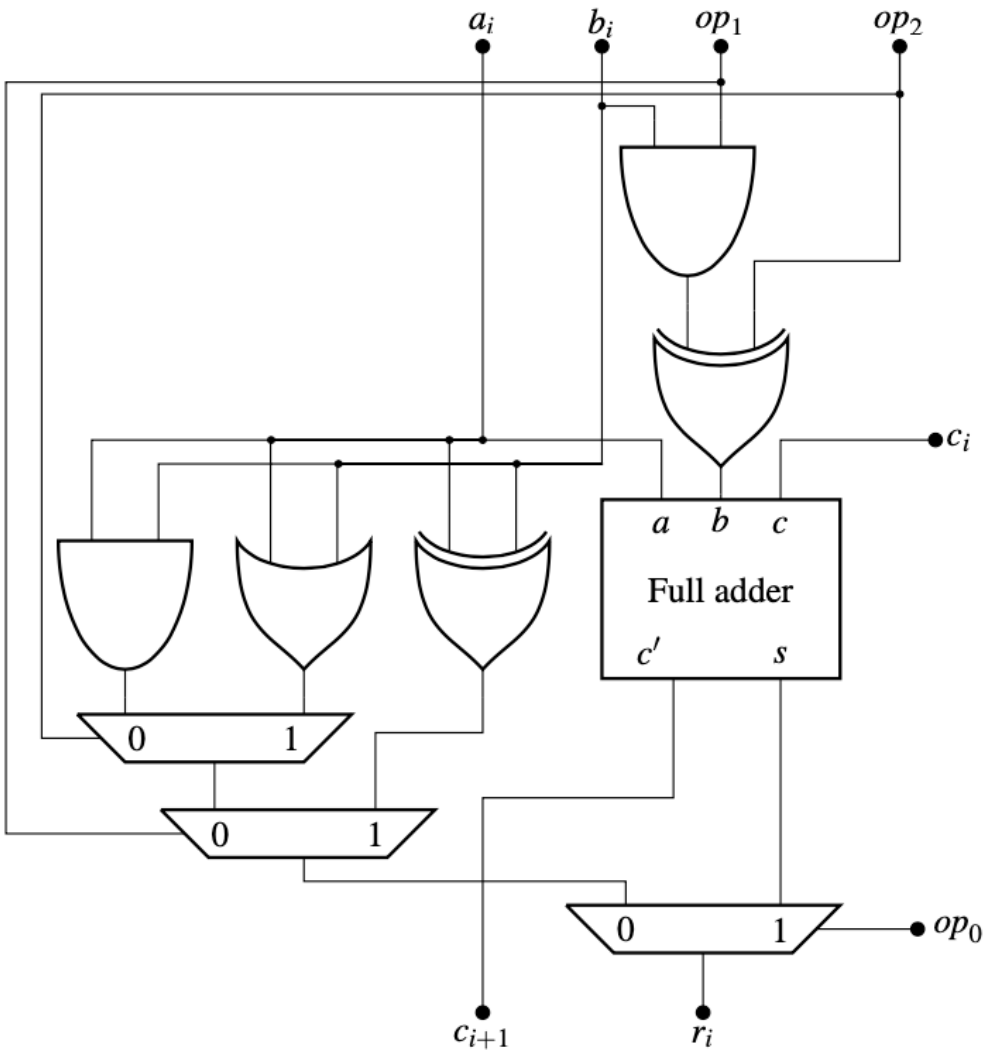
- Five control signals and Ci: $2^6 = 64$ functions ?
  - Well yes, but some of those are useless…
- Pass values and generate constants:

  A, B, 0 (zero), 1 (one), −1 (minus one)
- Mathematical functions:

  A+B, A+B+1, B−A, B−A−1,
  0−A (NEG), A+1 (INC), B+1, B−1 (DEC)
- Logical functions:

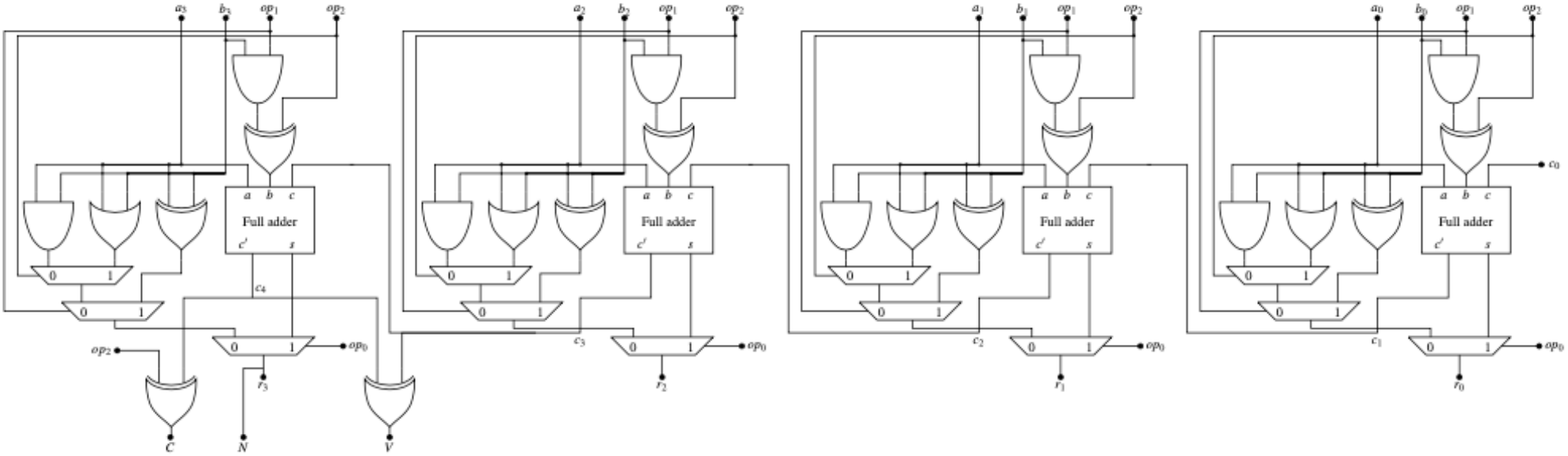
  A AND B, A OR B, NOT A, NOT B

The ALU from the book.



The ALU in the book is slightly different.

The 4-bit ALU from the book.



Simple processor: ALU functions

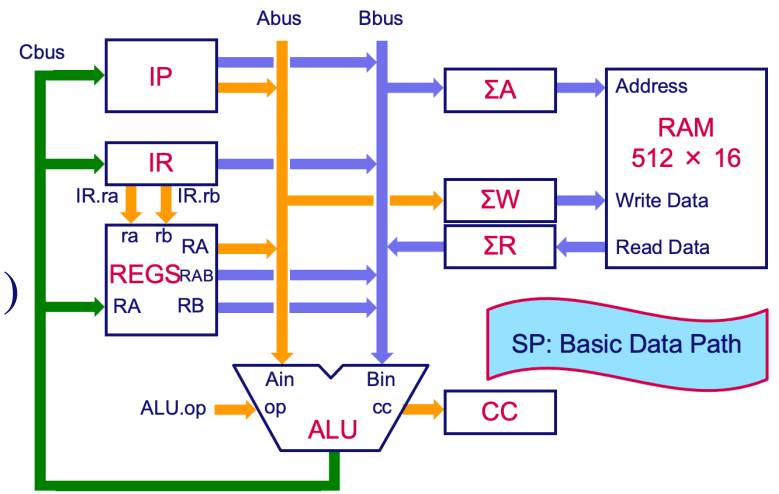
ALU functions for our simple processor (see the reader)

loadA:	A	
loadB:	B	
inc:	A + 1	
dec:	A – 1	(not with bitslice ALU)
add:	A + B	
sub:	A – B	(not with bitslice ALU)
and:	A ∧ B	
or:	A ∨ B	
xor:	A $\oplus$ B	(not with bitslice ALU)

Conductor: instruction execution (ADD, direct)

Instruction: ADD RA RB

Required action: RA ← RA + RB



- 0 ΣA, IP ← IP(Bbus), IP(Abus)+1 (ALU: "A+1")
- 1 ΣR ← RAM [ΣA]
- 2 IR ← ΣR (ALU: "B")
- 3 RA, CC ← RA + RB (ALU: "A+B"), ALU.cc

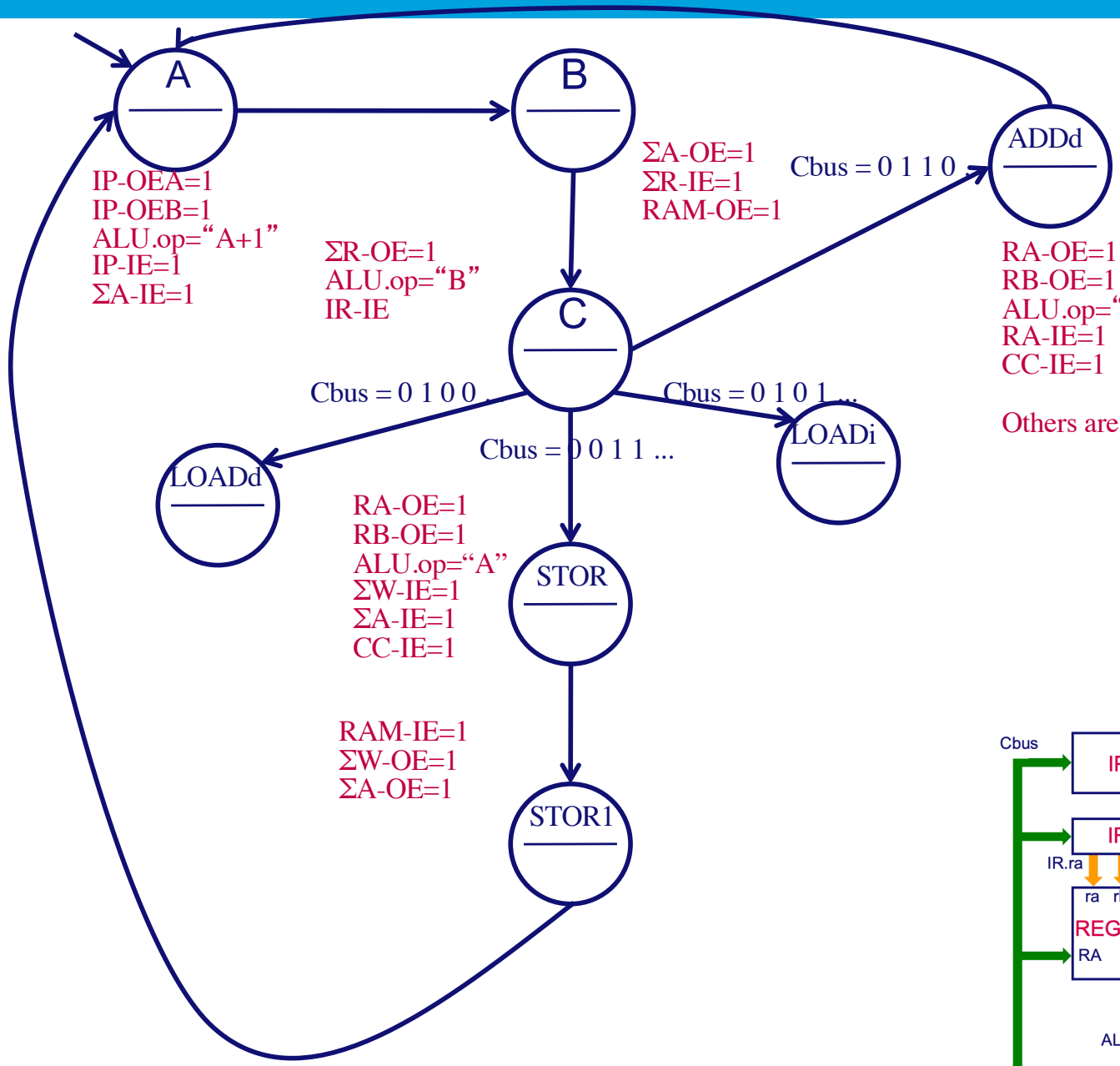
After the last step the Conductor returns to state 0; this is left *implicit*.

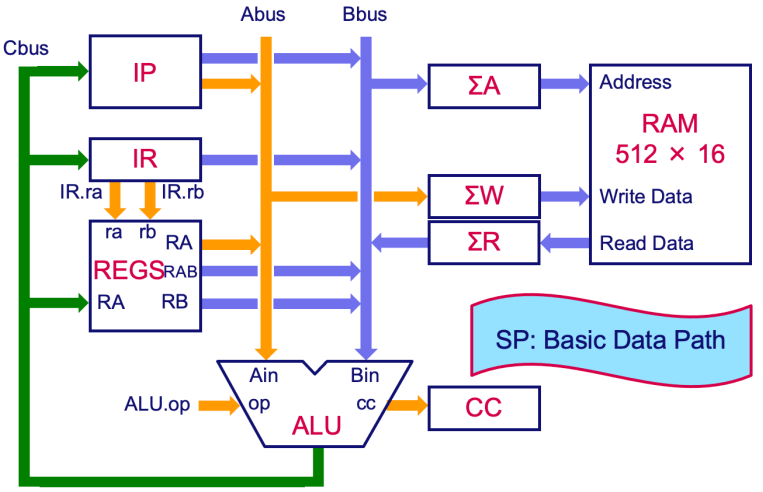
Execution of this instruction requires 4 clock cycles, 3 of which for IF!

RA is the register in REGS identified by IR.ra;

RB is the register in REGS identified by IR.rb;

Conductor: execute ADD, direct

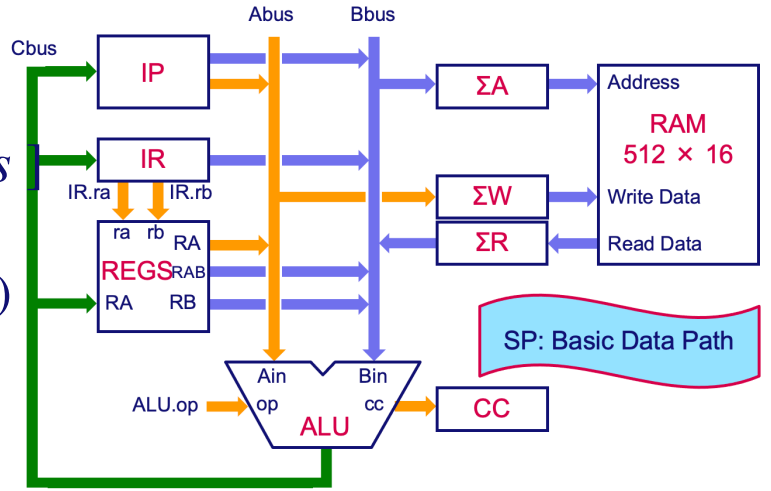




Example: Instruction execution (XOR, indirect)

Instruction: XOR RA [*address*]

Required action: RA ← RA ⊕ RAM [*address*]



•0 ΣA, IP ← IP(Bbus), IP(Abus)+1 (ALU: "A+1")

•1 ΣR ← RAM [ΣA]

•2 IR ← ΣR (ALU: "B")

•3 ΣA ← IR.val …… ΣA ← *address*

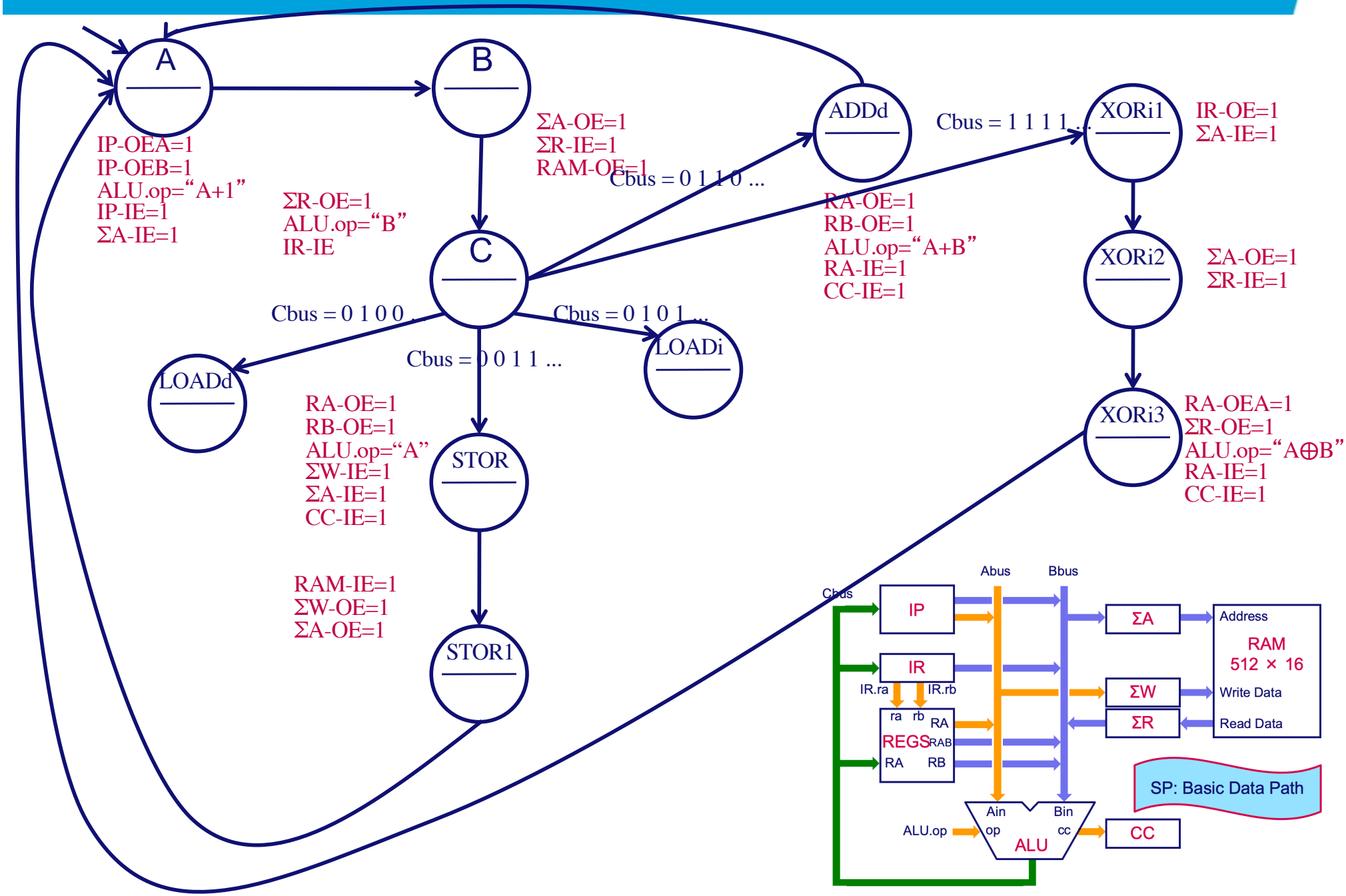
•4 ΣR ← RAM [ΣA] …… fetch operand RAM [*address*]

•5 RA, CC ← RA ⊕ ΣR (ALU: "A⊕B"), ALU.cc

Again, after the last step the Conductor returns to state 0: 6 cycles used.

Two additional cycles –compared to "ADD RA RB"– for the *operand fetch*.

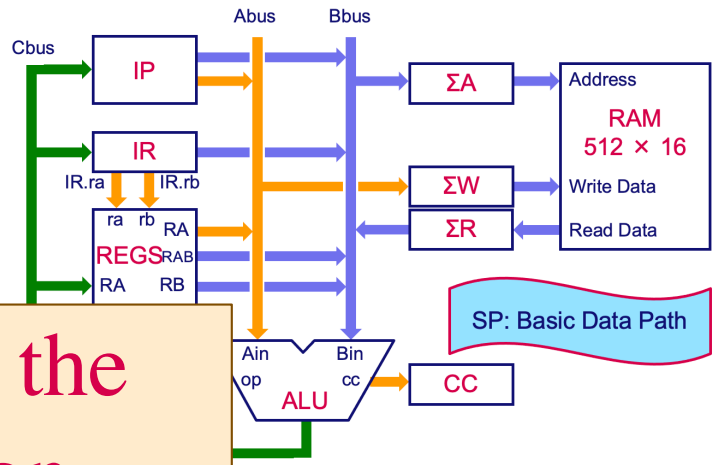
Conductor: execute XOR, indirect



Example: instruction execution (BEQ)

Instruction: BEQ *disp*, case branch taken (Z=1)

Required action: IP ← IP + *disp*



•0 ΣA, IP ← IP(Bbus), IP(Abus)+1 (ALU: "A+1")

•1 ΣR ← RAM [ΣA]

•2 IR ← ΣR (ALU: "B")

•3 IP ← IP(Abus) + IR.comp (ALU:

Interpret the instruction and the Z flag.

After the last step the Conductor returns to state 0: 4 cycles used.

IR.comp is the sign extended value. This is needed for negative jumps,

if the memory is larger than 512 bytes large.

Instruction format: conditional jumps

Instruction format: 0 0 0 1 c c c d d d d d d d d d

16 bits

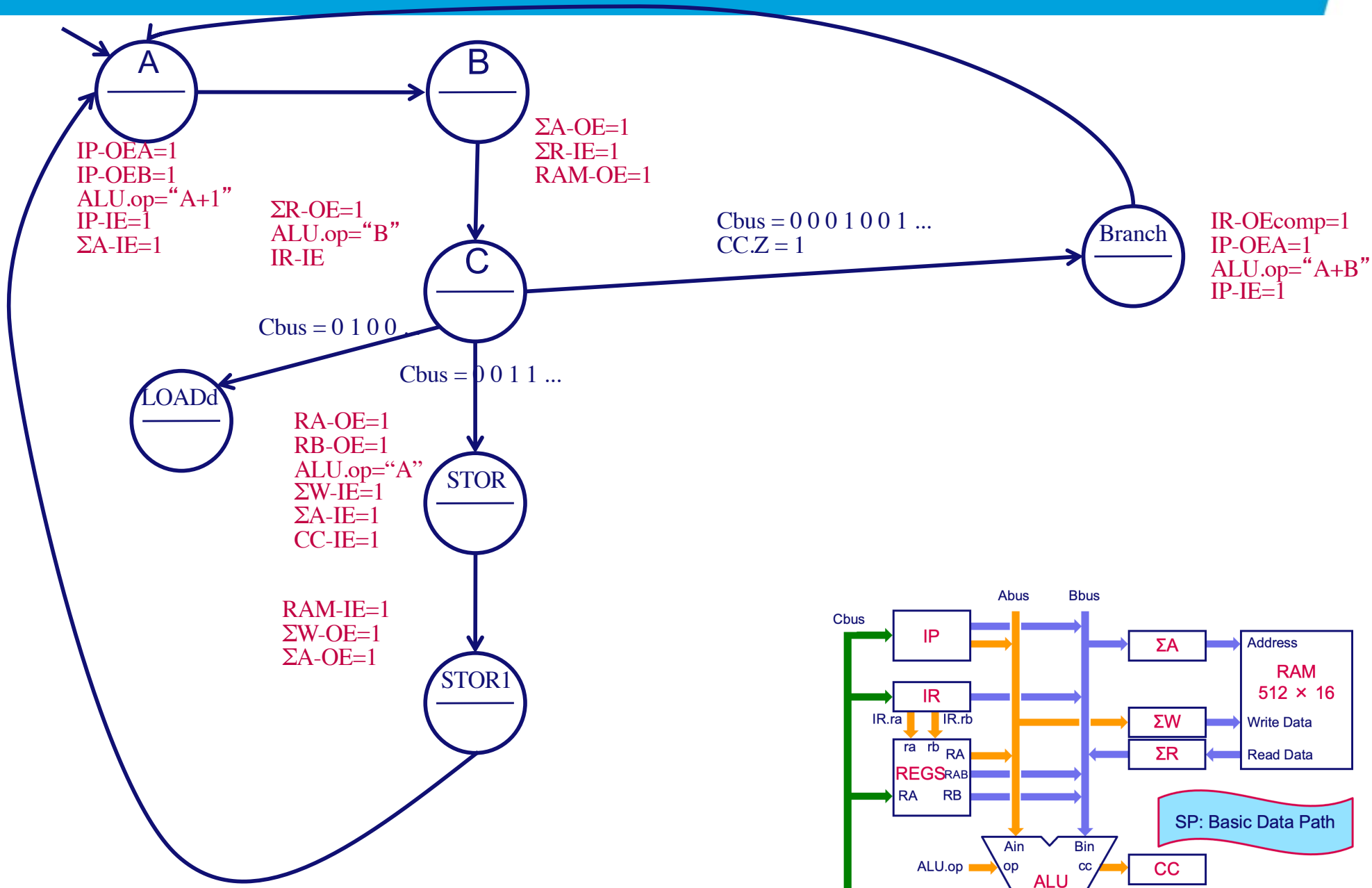
BNE	0 0 0	Z = 0
BEQ	0 0 1	Z = 1
BPL	0 1 0	N = 0
BMI	0 1 1	N = 1
BVC	1 0 0	V = 0
BVS	1 0 1	V = 1
BCC	1 1 0	C = 0
BCS	1 1 1	C = 1

BEQ -37
0 0 0 1 0 0 1 1 1 1 0 1 1 0 1 1

BVC 171
0 0 0 1 1 0 0 0 1 0 1 0 1 1 0 0

Code relocation requires relative jumps.

Conductor: execute jump to new location



Example: instruction execution (BEQ)

Instruction: BEQ *disp*, case branch *not* taken (Z=0)

Required action: "do nothing, just continue"

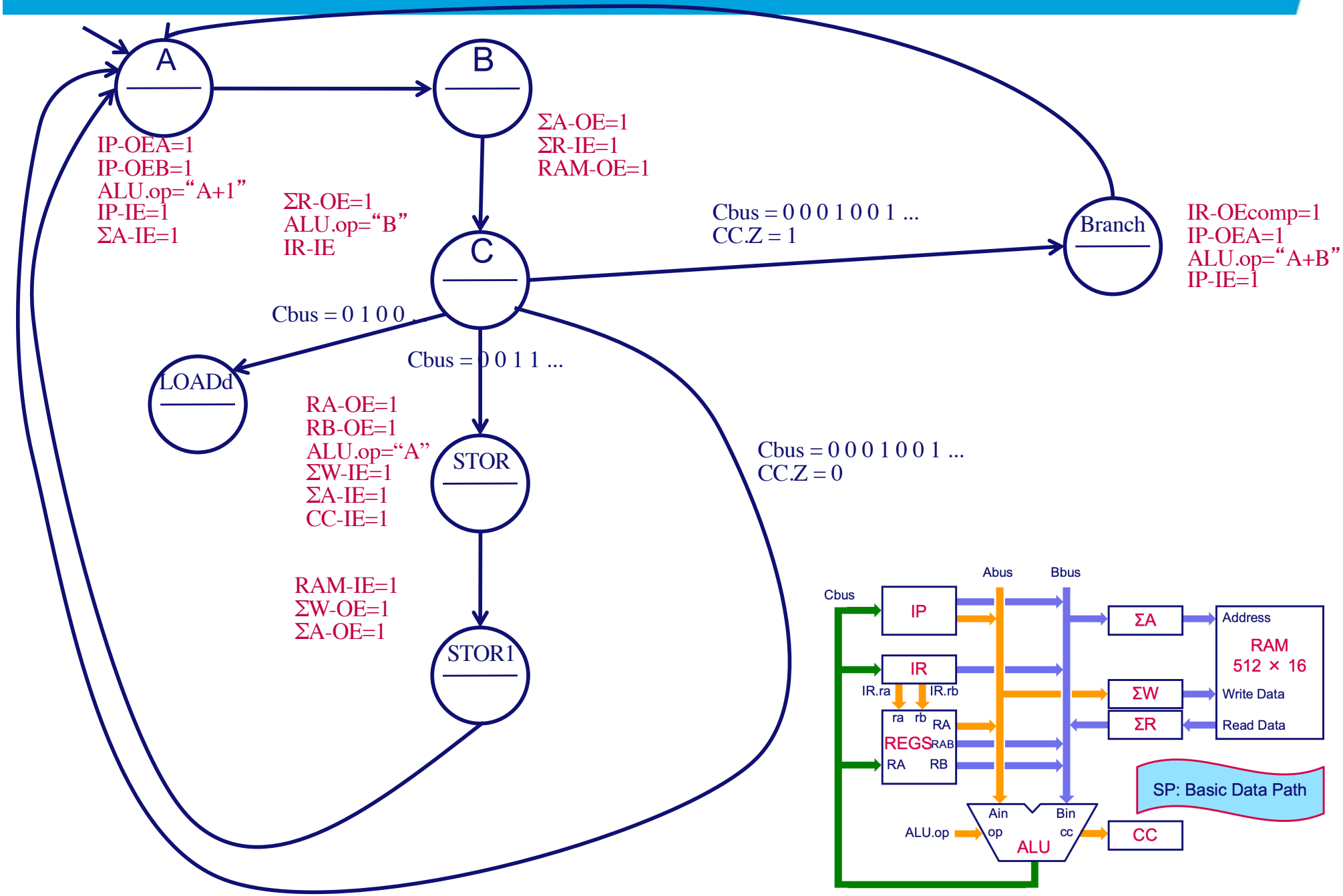
- 0 ΣA, IP ← IP(Bbus), IP(Abus)+1 (ALU: "A+1")
- 1 ΣR ← RAM [ΣA]
- 2 IR ← ΣR (ALU: "B")

Interpret the instruction and the Z flag.

After step 2 the Conductor returns to state 0: 3 cycles used.

Hence, in this case "do nothing" really means doing nothing!

Conductor: skip jump to new location

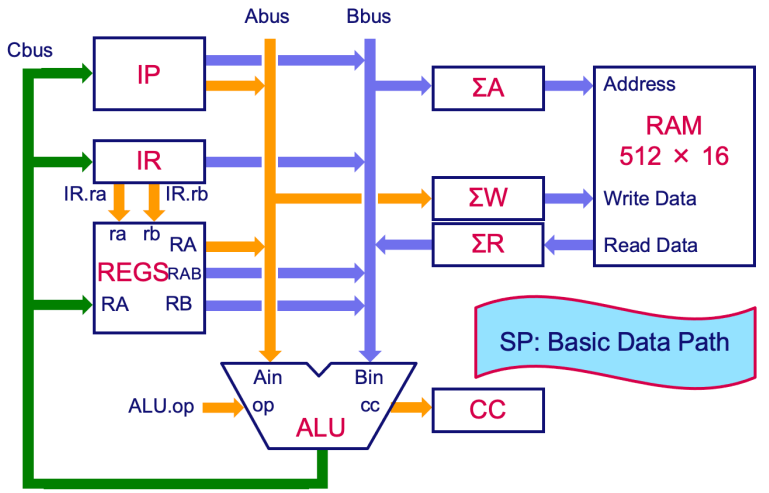


Example: Instruction execution (BRS)

Instruction: BRS *reg displacement*

Required action: *reg*, RAM[*reg*-1], IP ← *reg*-1, IP, IP + *displacement*

- •0 ΣA, IP ← IP(Bbus), IP(Abus)+1 (ALU: "A+1")
- •1 ΣR ← RAM [ΣA]
- •2 IR ← ΣR (ALU: "B")
- •3 RA ← RA (Abus)-1
- •4 ΣA, ΣW ← RAB (Bbus), IP (Abus)
- •5 IP, RAM [ΣA] ← IP (Abus)+IR.compl(Bbus), ΣW



Micro programming

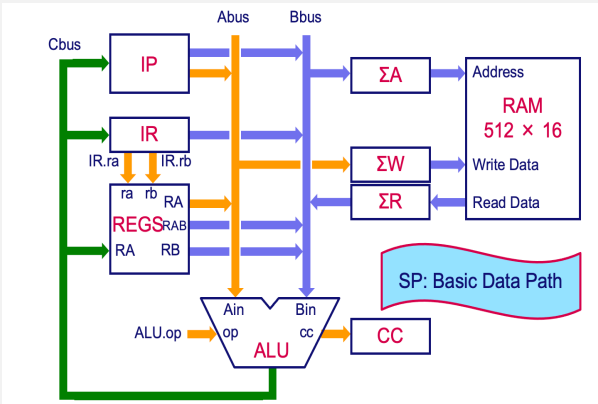
Program in memory

LOAD R1 1
ADD R1 [R2]
BEQ else_case
....

Alternative for encoding a state machine.

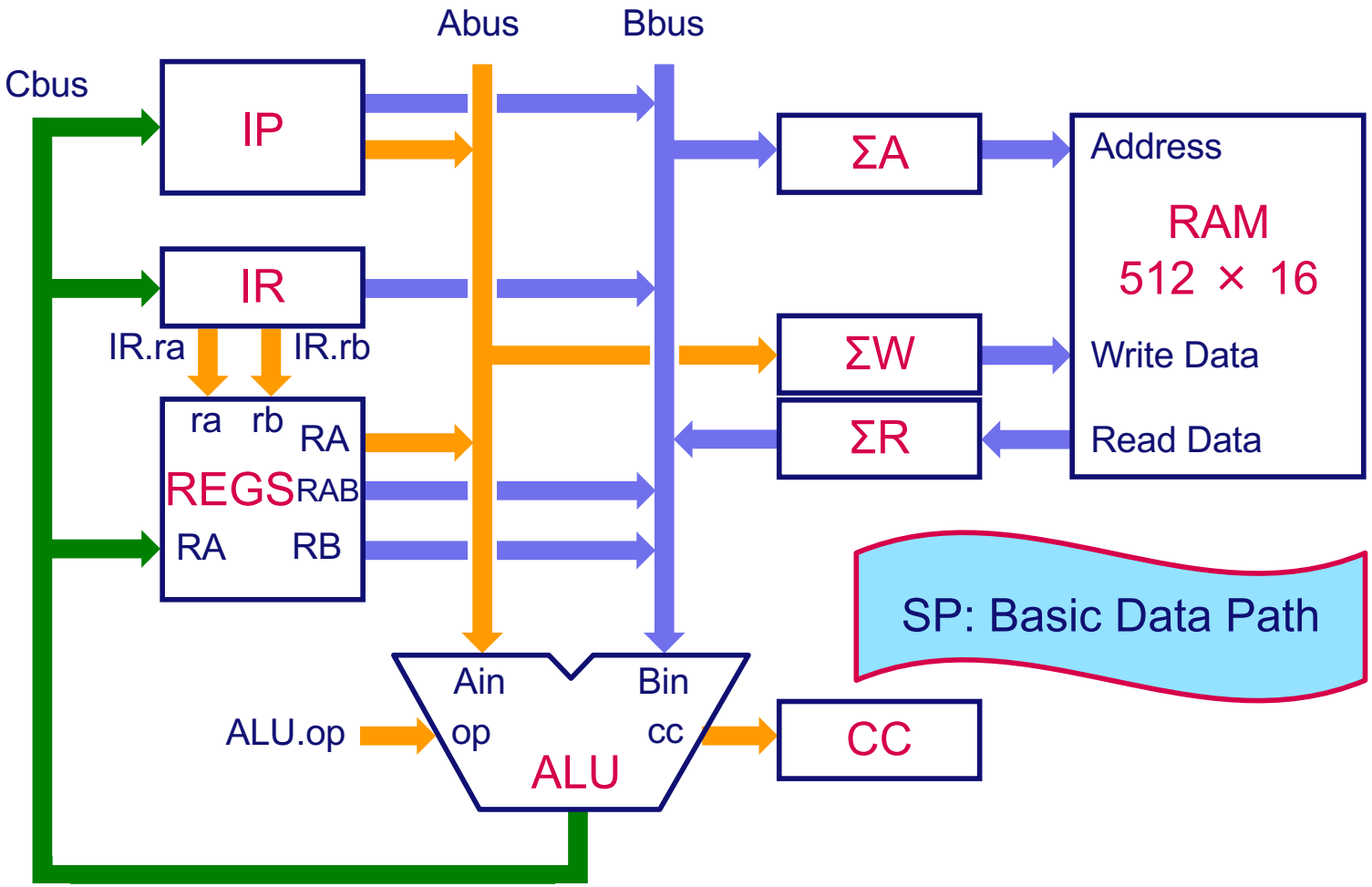
Microprogram:

- •3 RA ← RA (Abus)-1
- •4 ΣA, ΣW ← RAB (Bbus), IP (Abu
- •5 IP, RAM [ΣA] ← IP (Abus)+IR.



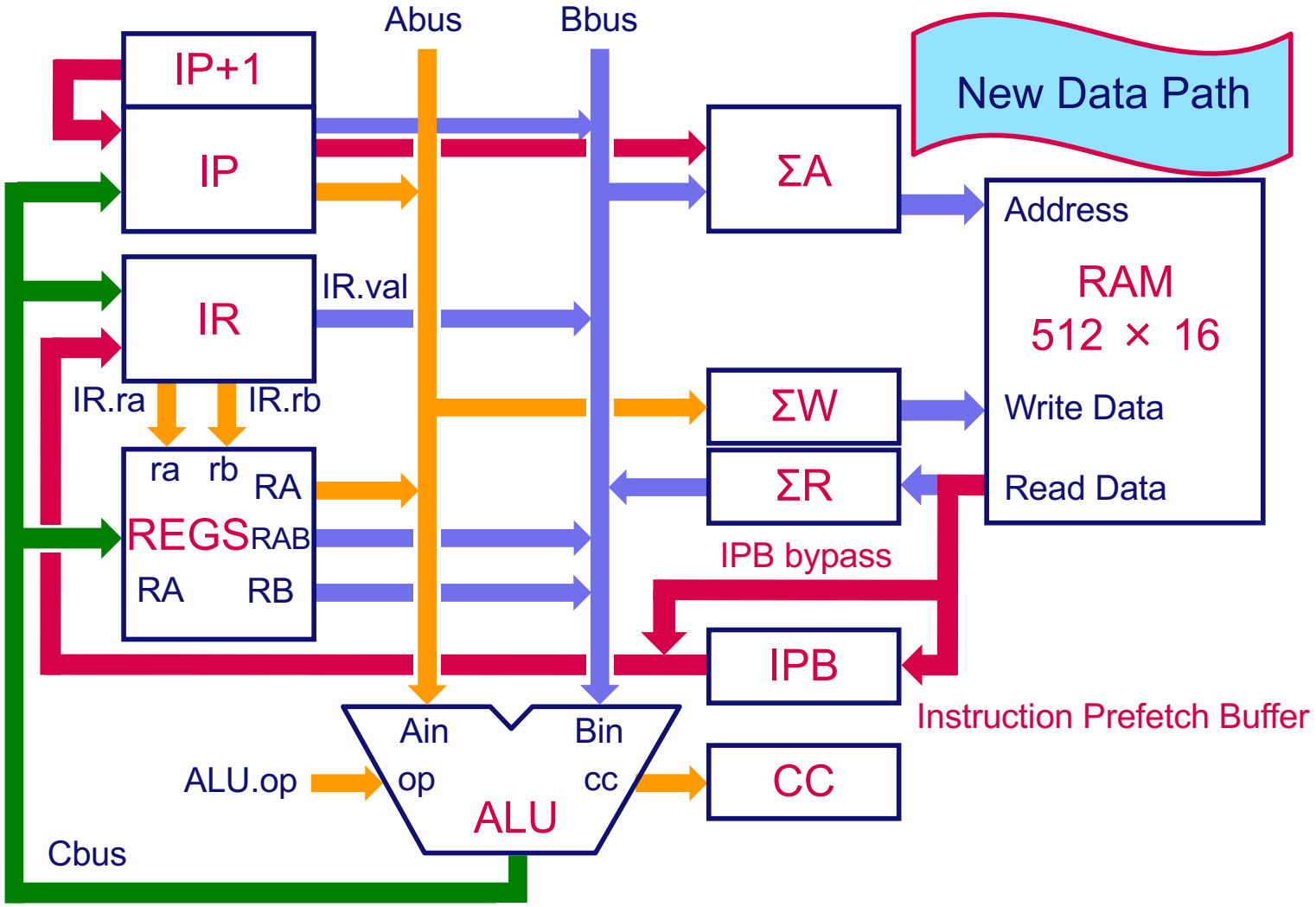
Questions?

Simple processor: data path



- Abus: IP or RA
- Bbus: IP, IR, ΣR, RB or RAB
- REGS: 2 Reads and 1 Write, simultaneously, ra[1..0] and rb[1..0] select registers

Simple processor: speeding up the machine



- IPB: allows prefetching without overwriting IR too early

Example: faster instruction execution (1)

Instruction: ADD RA RB

Required action: RA ← RA + RB

- •0 ΣA , IP ← IP , IP+1 ……using the IP-incrementer
- •1 IR , ΣA , IP ← RAM [ΣA] , IP, IP+1 ....……IR via the IPB-bypass
- •2 RA , CC , IR , ΣA , IP ←

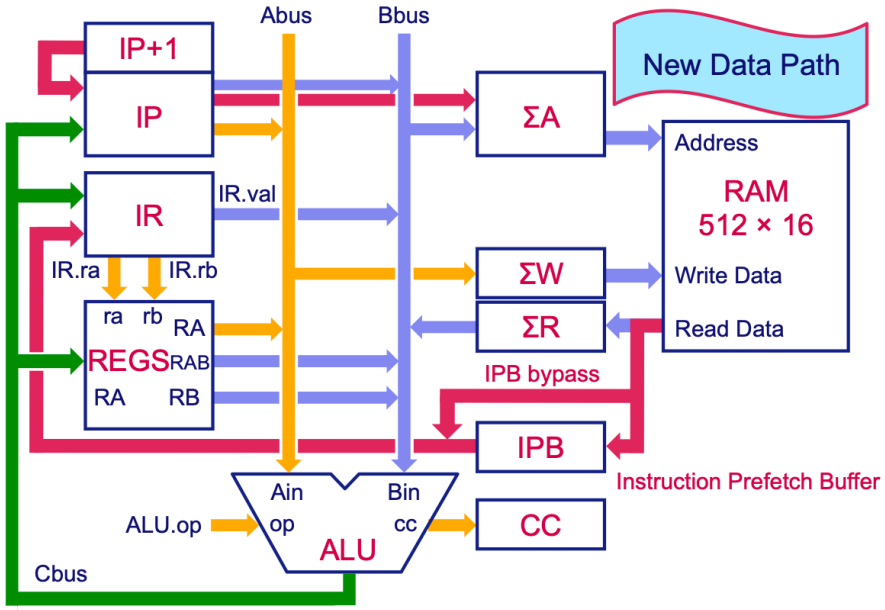
  RA + RB (ALU: "A+B") , ALU.cc , RAM [ΣA] , IP , IP+1

In state 1 the value of IP is already sent to ΣA , to initiate prefetching.

In state 2 the next instruction arrives; because the current execution is completed here, it can be placed into IR directly: IPB is not needed here.

In addition, due to the prefetch: IP points 2 instructions *ahead!*

After state 2 the Conductor goes to a new instruction, with a prefetched instruction. Thus, instructions of this type need only 1 cycle, instead of 4 !



Example: faster instruction execution (2)

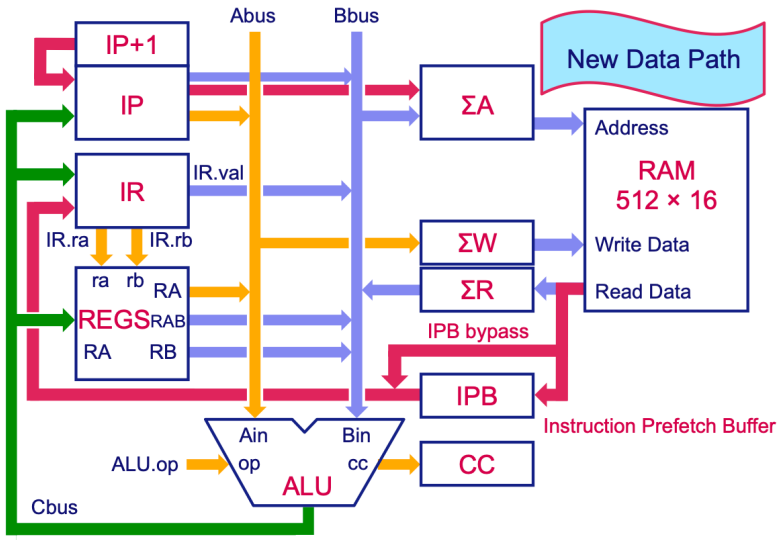
Instruction: XOR RA [*address*]

Required action: RA ← RA ⊕ RAM [*address*]

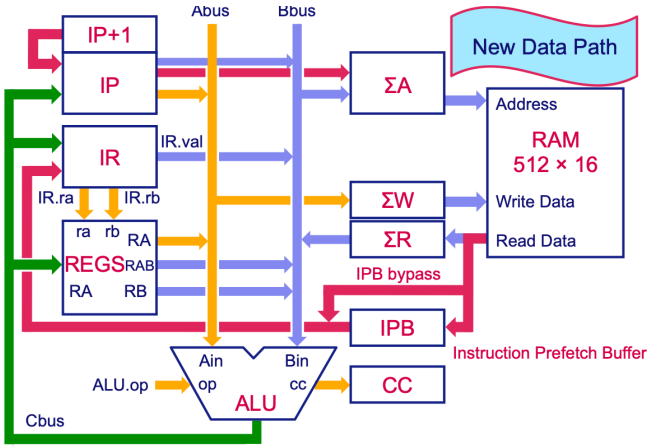
- •0 ΣA , IP ← IP , IP+1
- •1 IR , ΣA , IP ← RAM [ΣA] , IP, IP+1 …. fetch instruction, start prefetch
- •2 ΣA , IPB ← IR.val , RAM [ΣA] …… start operand fetch, save pref. instr. in IPB
- •3 ΣR ← RAM [ΣA] …… fetch operand
- •4 RA , CC , IR , ΣA , IP ← …… execute, prefetch from IPB

  RA ⊕ ΣR (ALU: "A⊕B") , ALU.cc , IPB, IP , IP+1

With a prefetch instruction starts in 2,
ends in 4: 3 cycles used.



Example: faster instruction execu



Instruction: STOR RA [RB]

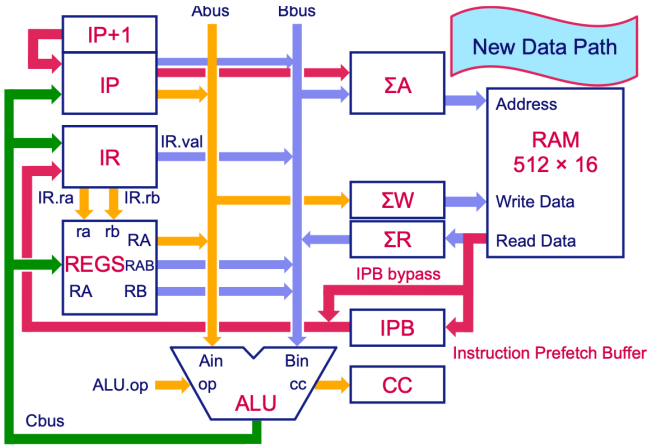
Required action: RAM [RB] ← RA

- 0 ΣA , IP ← IP , IP+1
- 1 IR , ΣA , IP ← RAM [ΣA] , IP, IP+1 ..... fetch instruction
- 2 ΣA , ΣW , CC , IPB ← RB , RA , ALU.cc (ALU: "A") , RAM [ΣA]
- 3 RAM[ΣA], IR , ΣA , IP ← ΣW, IPB, IP , IP+1 ..... fetch from IPB

Tricky: In step 3 the Memory is busy with " RAM [RB] ← RA "; hence, the instruction prefetch is initiated in 2; IPB is used as a buffer to delay the next instruction fetch.

With a prefetch the processor starts at 2: it only needs two instructions.

Example: faster instruction execu



Instruction: BEQ *disp*, case branch taken (Z=1)

Required action: IP ← IP + *disp*

- •0 ΣA , IP ← IP, IP+1
- •1 IR , ΣA , IP ← RAM [ΣA], IP, IP+1 …… initiate prefetch: will be wasted!

  { IR = "BEQ *disp*", IP's value is 1 too large , Z = 1 }
- •2 IP ← IP(Abus) – 1 (ALU: "A–1") ..…… correct IP's value
- •3 IP ← IP(Abus) + IR.comp (ALU: "A+B") …… perform the jump
- •4 ΣA , IP ← IP, IP+1 …… initiate *target* IF
- •5 IR , ΣA , IP ← RAM [ΣA], IP, IP+1 …… fetch *target* instruction

Prefetching the *next* instruction now is part of executing the *current* one.

Hence, again the Conductor goes back to state 2: 4 cycles for this case.

Warning: this is a very tricky game!

Alternative: to *prevent* IP correction an additional *buffer register* may be used,

or the ALU may be extended with an *additional operation*: "A+B–1".

Example: Faster instruction execution (4b)

Instruction: BEQ *disp* , case branch *not* taken (Z=0)

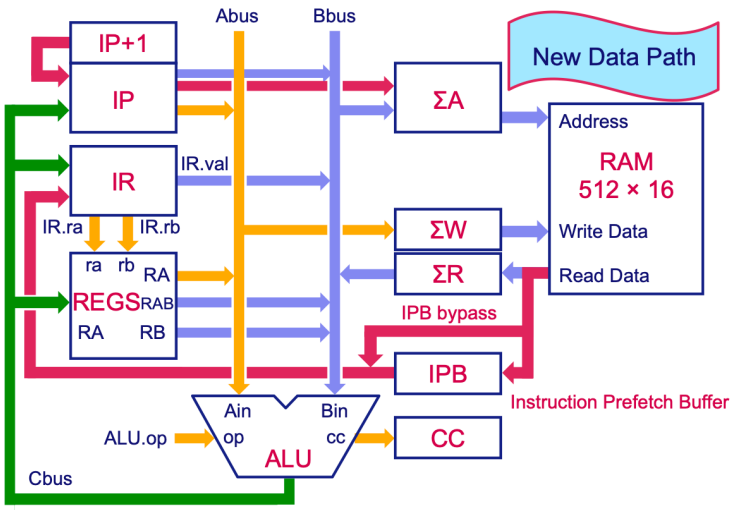
Required action: "do nothing, just continue"

- •0 ΣA , IP ← IP , IP+1
- •1 IR , ΣA , IP ← RAM [ΣA] , IP, IP+1 …… initiate prefetch: not wasted!

  { IR = "BEQ *disp*", Z = 0 }
- •2 IR , ΣA , IP ← RAM [ΣA] , IP, IP+1 …… fetch *next* instruction

Because fetching the next instruction has been initiated already in step 1, "do nothing" entails no more than completing the Instruction Fetch in step 2.

With a prefetch, the executing the instruction starts in state 2: only 1 cycle now.



Simple Processor: Summary

Instruction Execution Times, without and with prefetching

Instruction:	no prefetch:	prefetch:	memory accesses:
ADD R2 R3	4	1	1
XOR R2 [*address*]	6	3	2
STOR R0 [R3]	5	2	2
BEQ *disp*	4 / 3	4 / 1	1

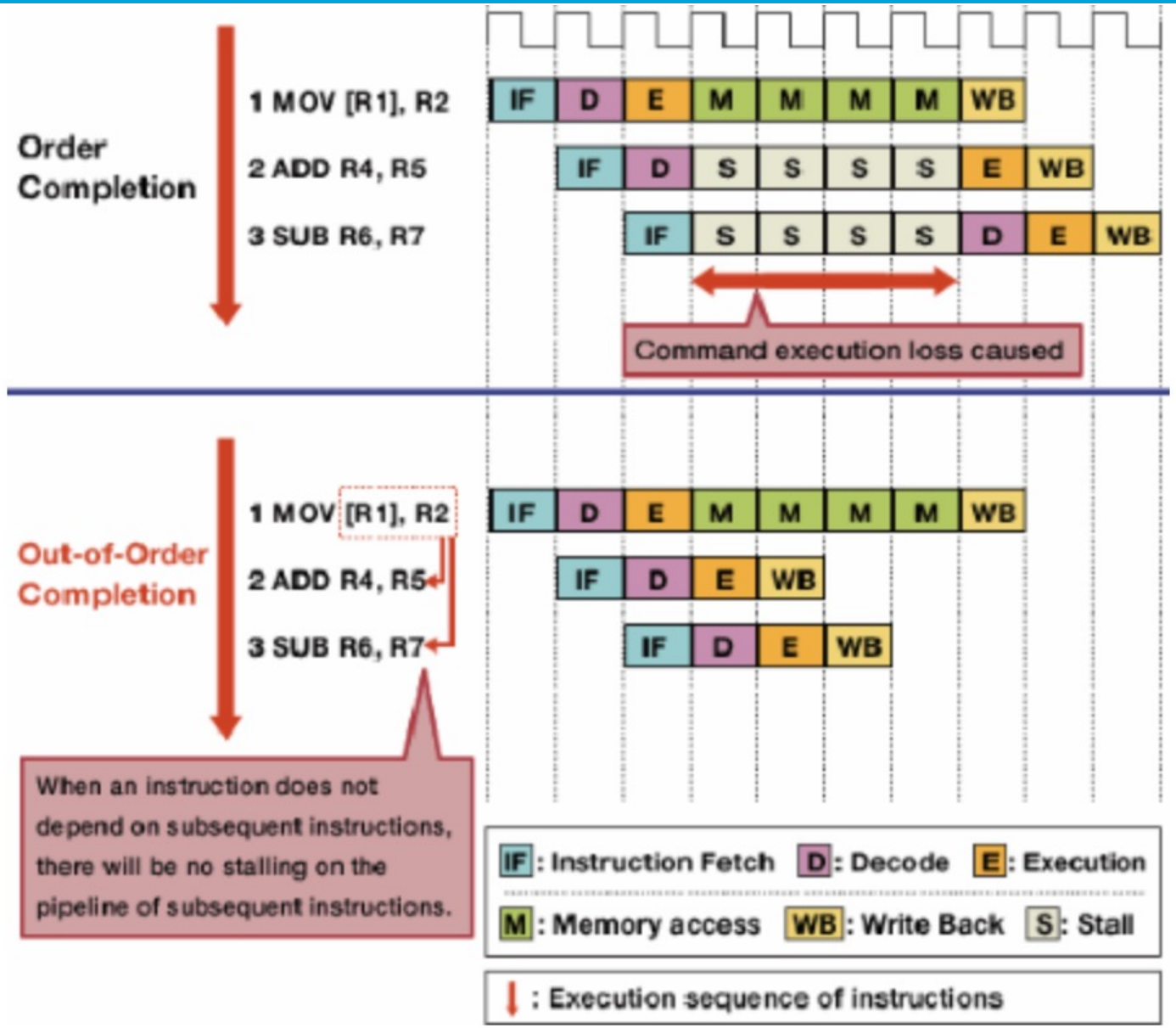
A substantial improvement!

Other possibilities for speeding up:

- adding more buffer registers; example: IP, to retain "old" value
- adding more bypasses; example: from ALU to ΣA .
- adding dedicated adders; example: address calculations.

The number of memory accesses needed is a *lower bound!*

Out of order execution.



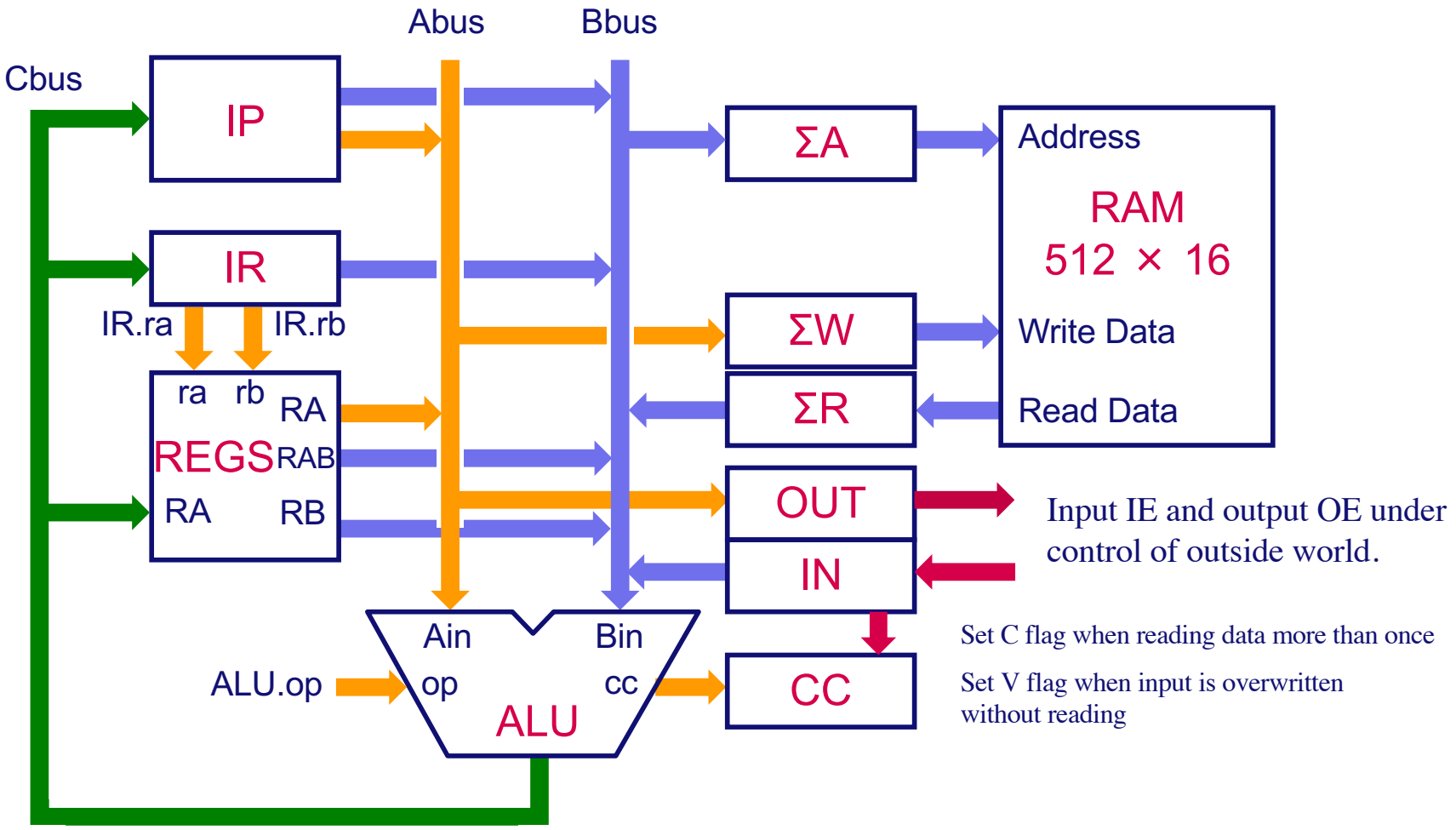
On parallel computers changing the ordering can be harmfull.

Hence, barrier instructions that explicitly prevent out of order execution.

Picture from: renesasrulz.com

Questions?

Simple processor: Input and output.



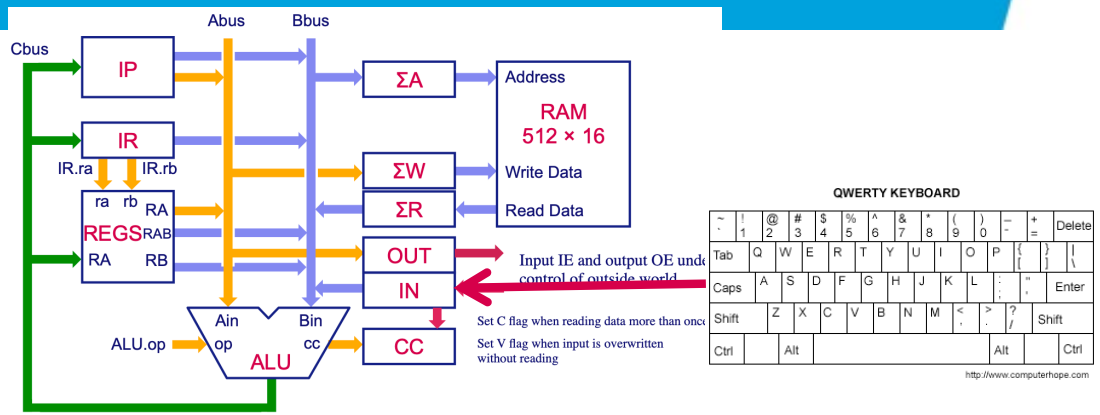
IN *register* 0 0 1 0 r r 1 0 0 0 0 0 0 0 0 0

OUT *register* 0 0 1 0 r r 0 1 0 0 0 0 0 0 0 0

Input and output: Polling.

Count keyboard inputs in register R1.

```
   LOAD R1 0;          Reset key stroke count.
start_keyboard_read:
   IN R0;              Read input buffer.
   BVS overwrite_error;
                       Data entry in buffer has
                       been missed.
   BCS start_keyboard_read;
                       If carry flag is set, there is
                       no new data, so loop back.
   ADD R1 1;           Count the key stroke.
   BRA start_keyboard_read;
overwrite_error:
                       This is the place for error handling.
```



Polling computationally expensive.

Summary

What did we learn:

- We implemented all instructions on the simple data path.
- It is possible to change the ‘simple data path’, add buses and registers if that would be useful.
- We can optimize the execution time of the instructions, for instance by prefetching instructions.