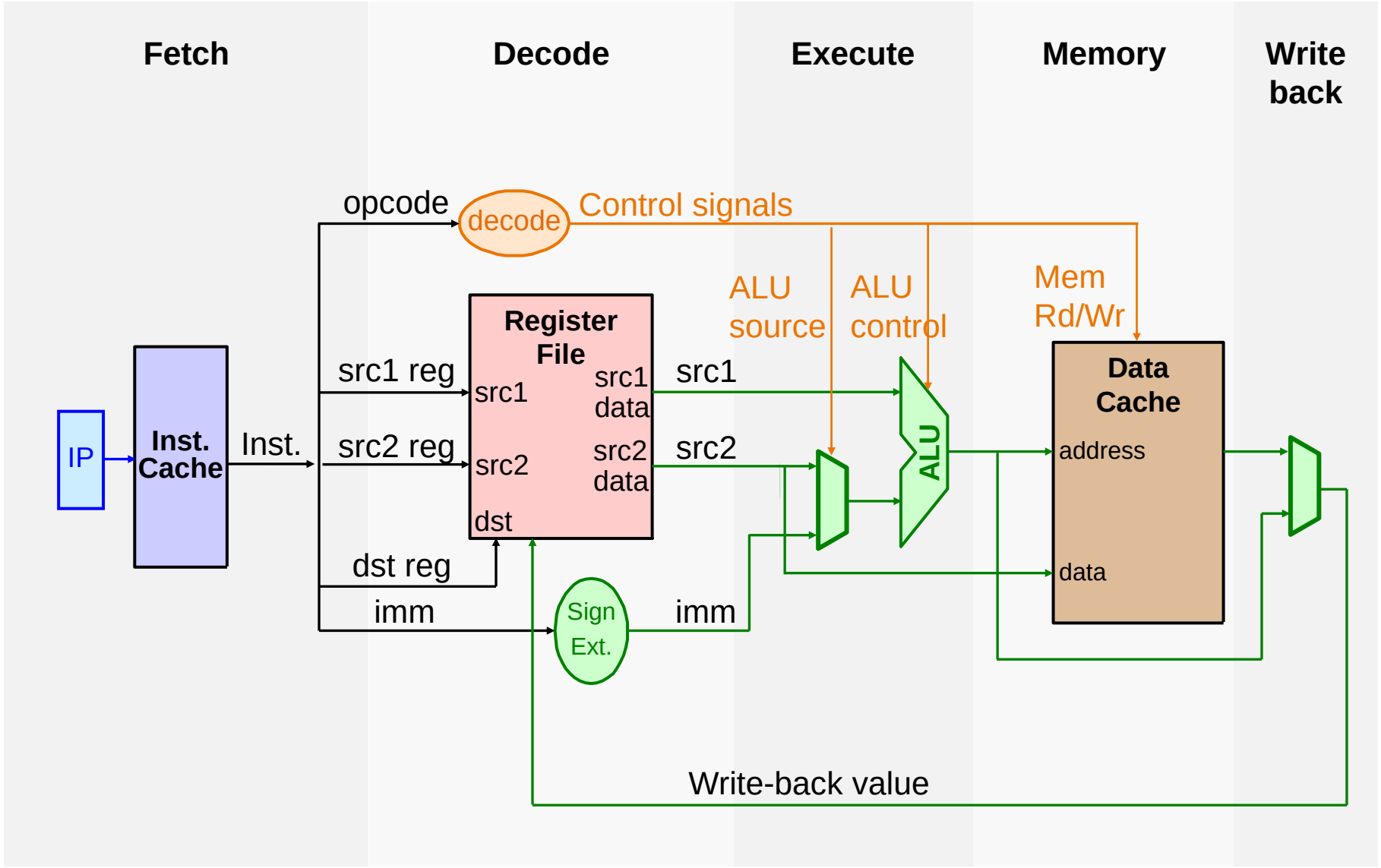
Computer Structure

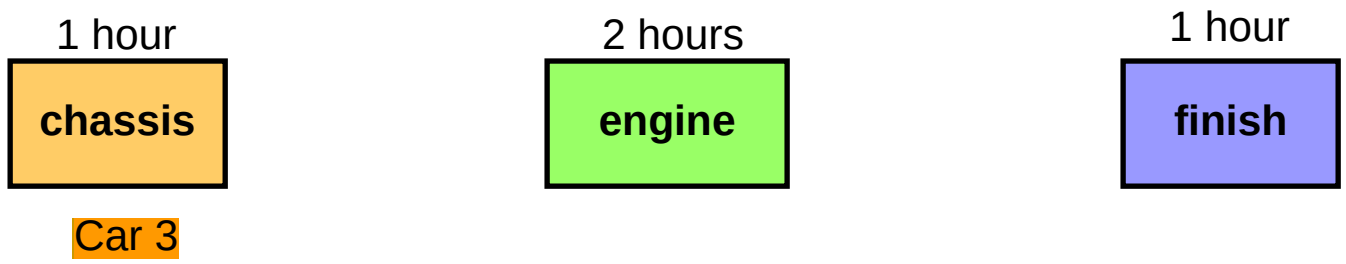
Pipeline

Lecturers:
Lihu Rappoport
Adi Yoaz

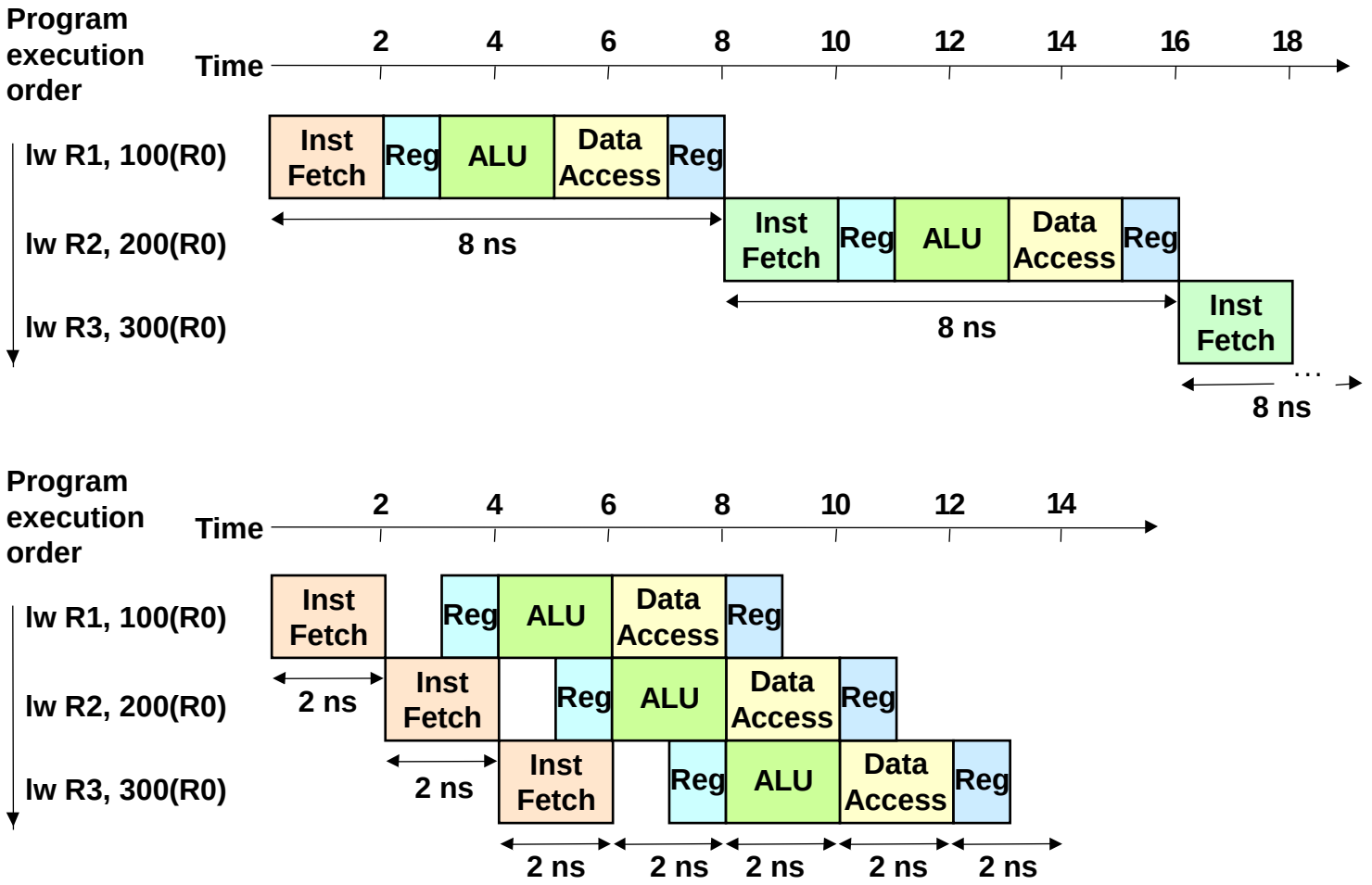
A Basic Processor



Pipelined Car Assembly



Pipelining Instructions

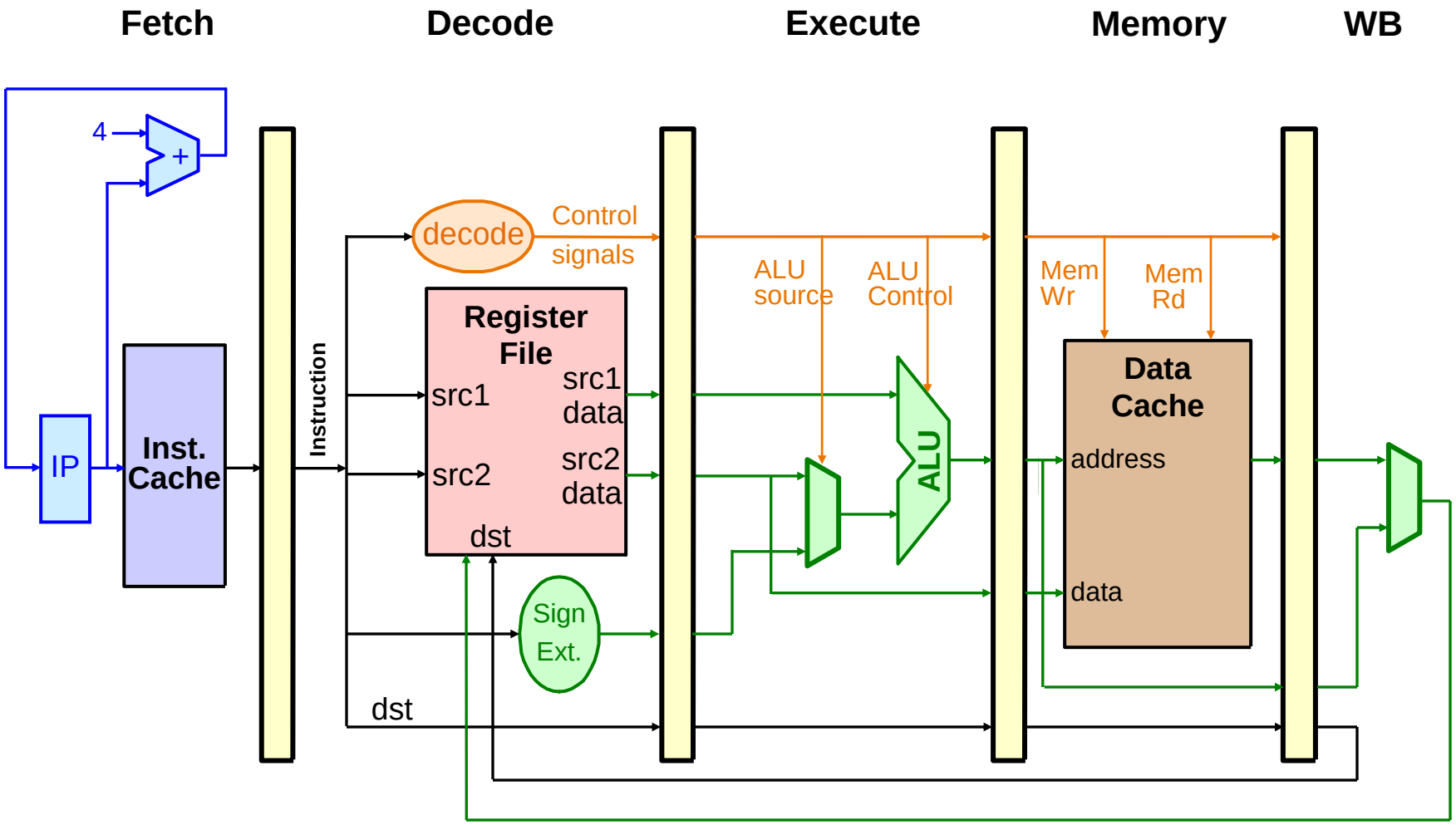


Ideal speedup is number of stages in the pipeline. Do we achieve this?

Pipelining

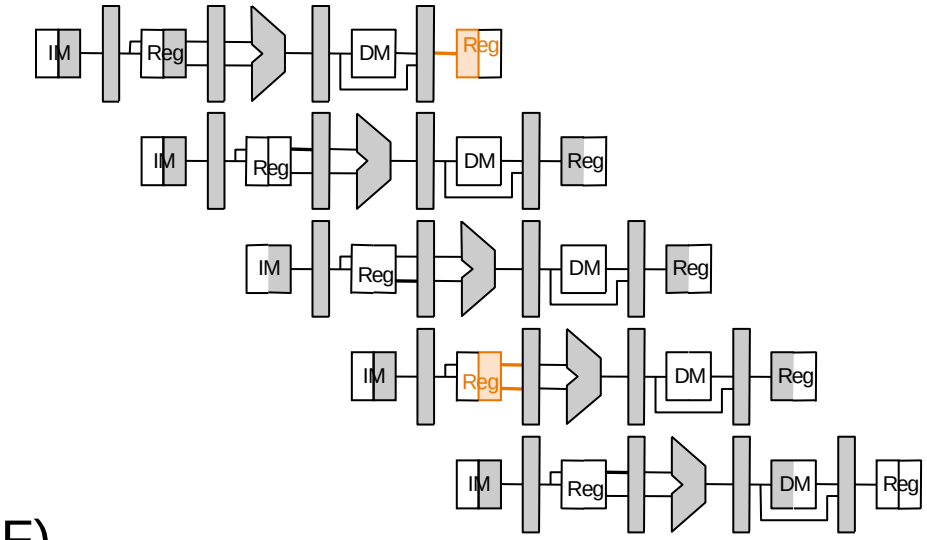
- Pipelining does not reduce the latency of single task, it increases the throughput of entire workload
- Potential speedup = Number of pipe stages
  - Pipeline rate is limited by the slowest pipeline stage
    - ⇒ Partition the pipe to many pipe stages
    - ⇒ Make the longest pipe stage to be as short as possible
    - ⇒ Balance the work in the pipe stages
- Pipeline adds overhead (e.g., latches)
  - Time to “fill” pipeline and time to “drain” it reduces speedup
  - Stall for dependencies
    - ⇒ Too many pipe-stages start to loose performance
- IPC of an ideal pipelined machine is 1
  - Every clock one instruction finishes

Pipelined CPU



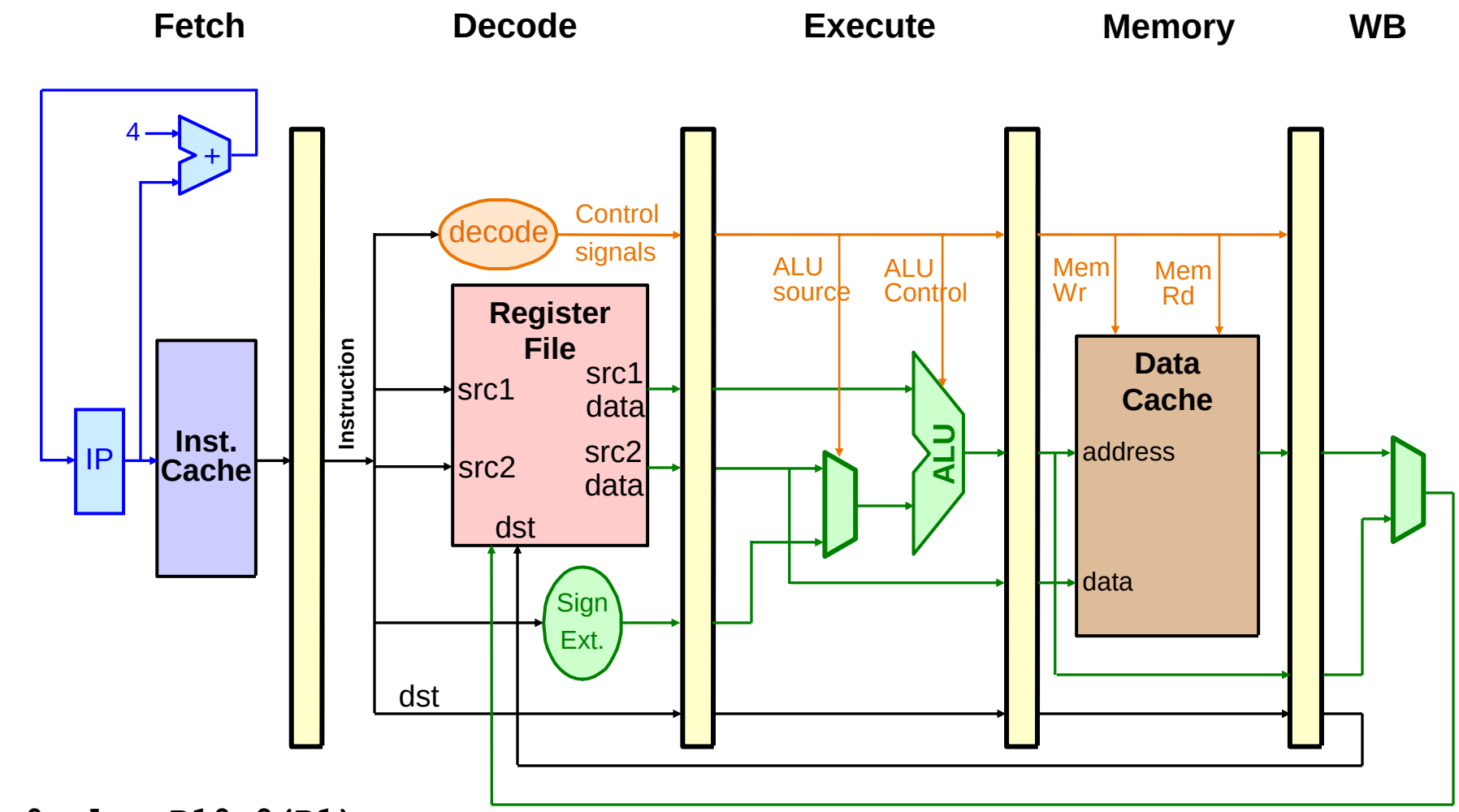
Structural Hazard

- **Different instructions using the same resource at the same time**
- **Register File**:
  - Accessed in 2 stages:
    - Read during stage 2 (ID)
    - Write during stage 5 (WB)
  - Solution: 2 read ports, 1 write port



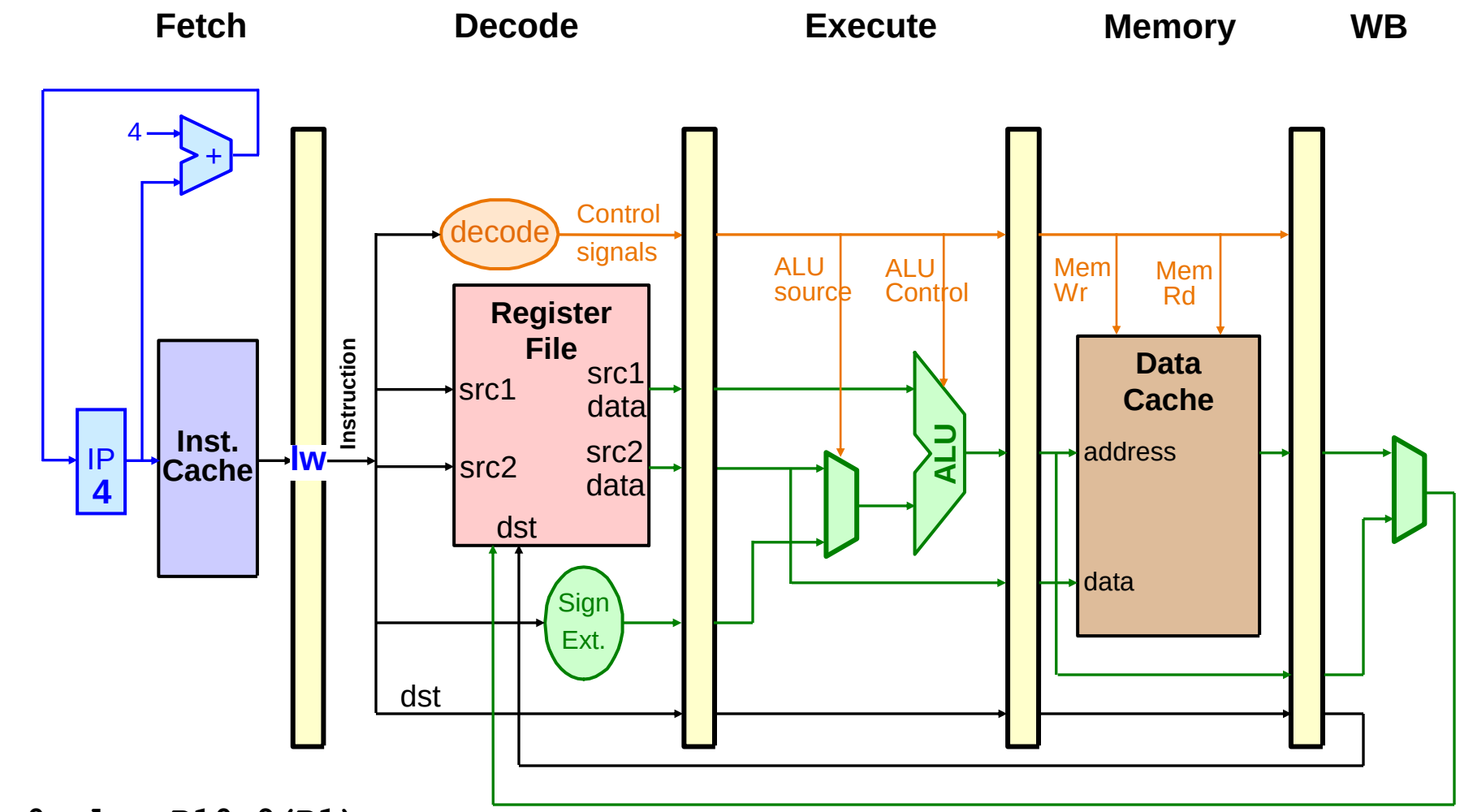
- **Memory**
  - Accessed in 2 stages:
    - Instruction Fetch during stage 1 (IF)
    - Data read/write during stage 4 (MEM)
  - Solution: separate instruction cache and data cache
- **Each functional unit can only be used once per instruction**
- **Each functional unit must be used at the same stage for all instructions**

Pipeline Example: cycle 1



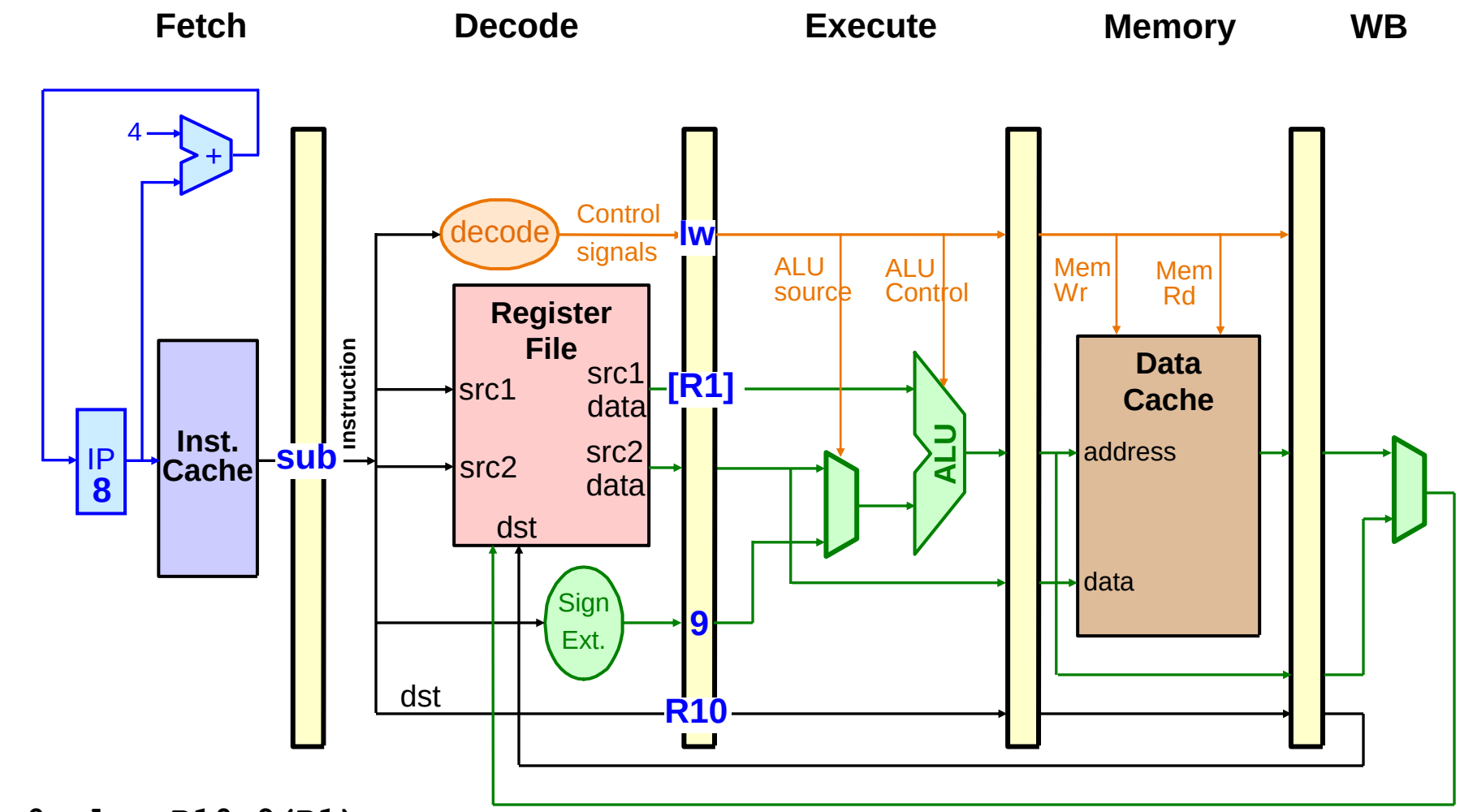

```
 0   lw   R10,9(R1)
 4   sub  R11,R2,R3
 8   and  R12,R4,R5
12   or   R13,R6,R7
```

Pipeline Example: cycle 2



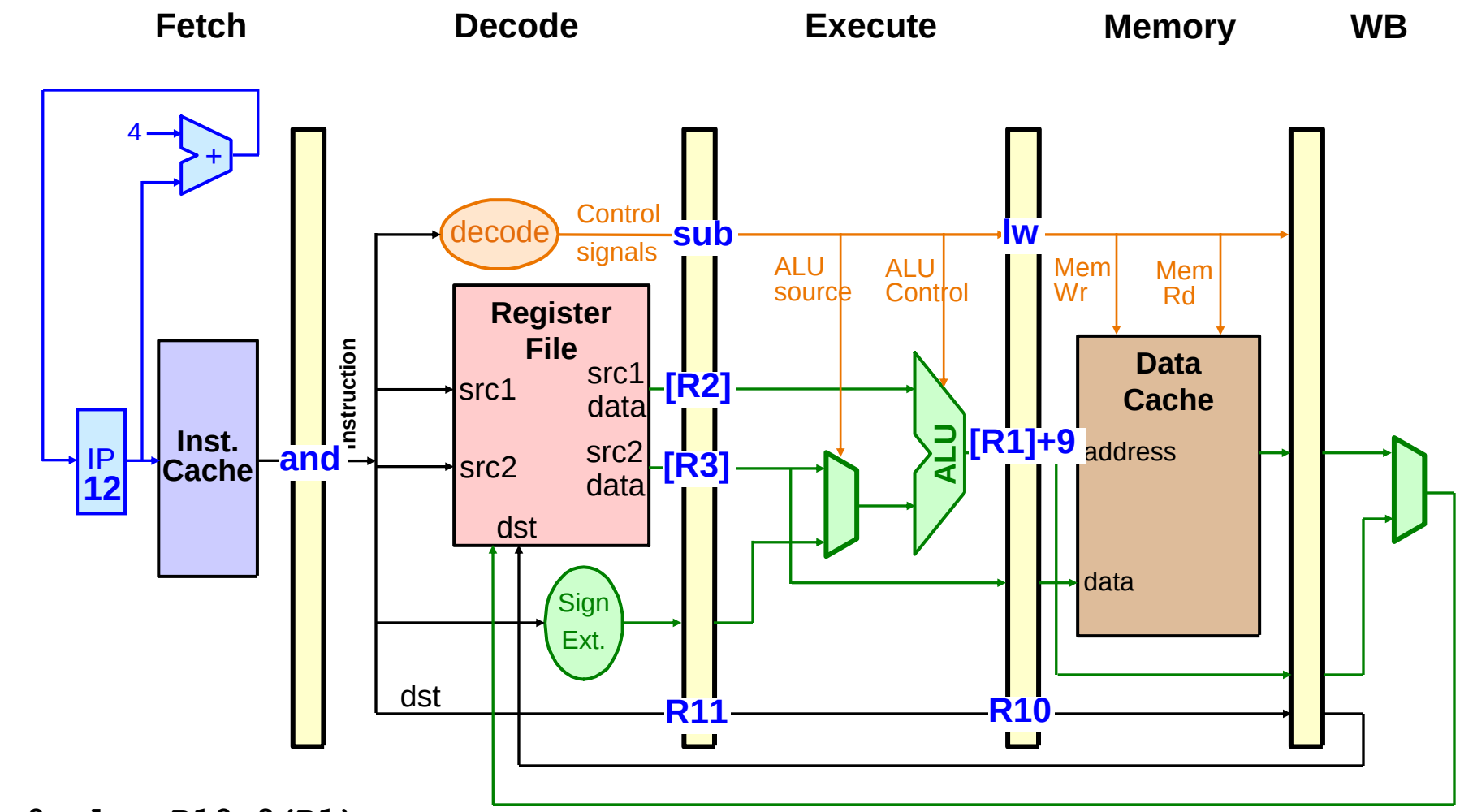

```
 0   lw   R10,9(R1)
 4   sub  R11,R2,R3
 8   and  R12,R4,R5
12   or   R13,R6,R7
```

Pipeline Example: cycle 3



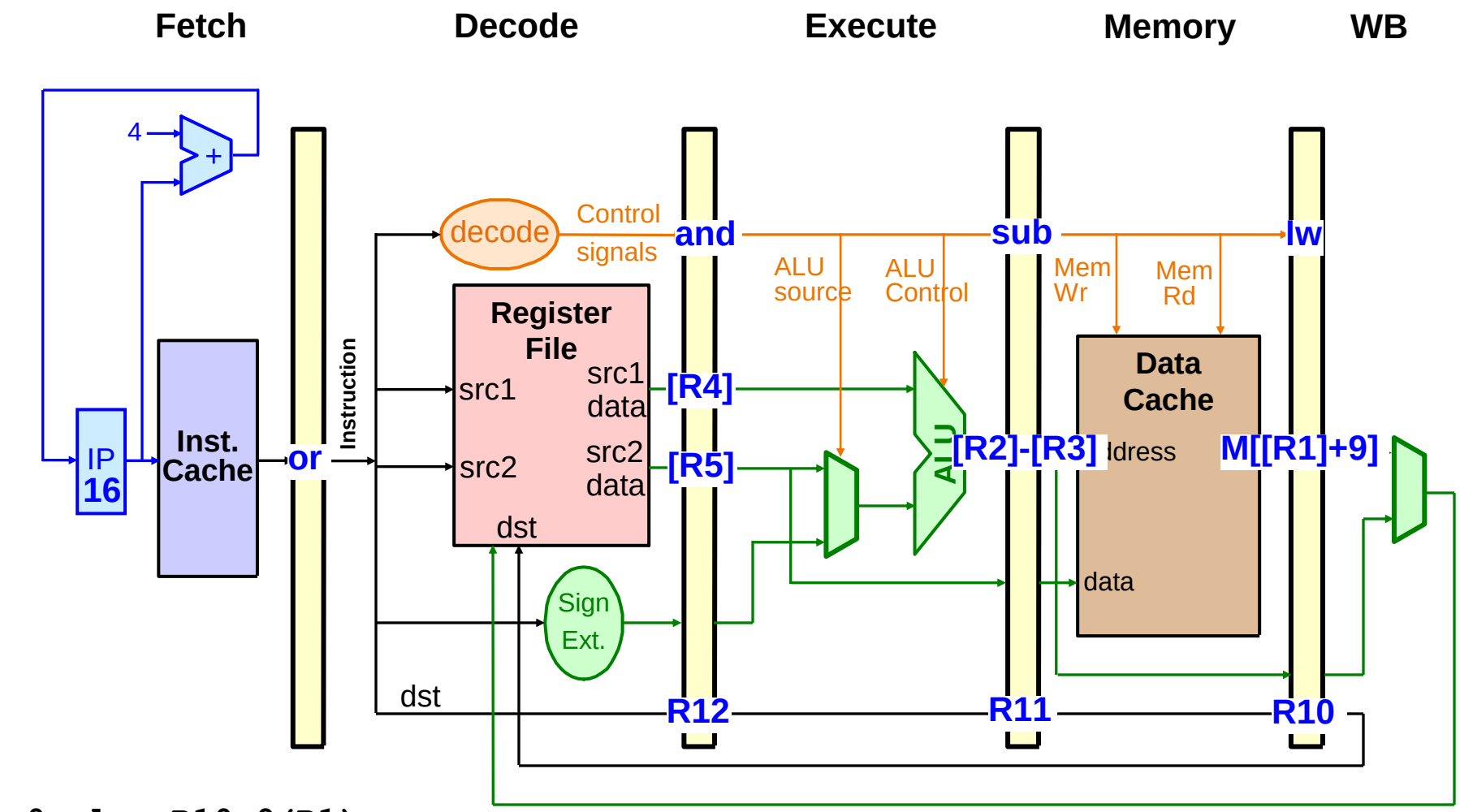

```
 0   lw   R10,9(R1)
 4   sub  R11,R2,R3
 8   and  R12,R4,R5
12   or   R13,R6,R7
```

Pipeline Example: cycle 4




```
 0   lw   R10,9(R1)
 4   sub  R11,R2,R3
 8   and  R12,R4,R5
12   or   R13,R6,R7
```

Pipeline Example: cycle 5

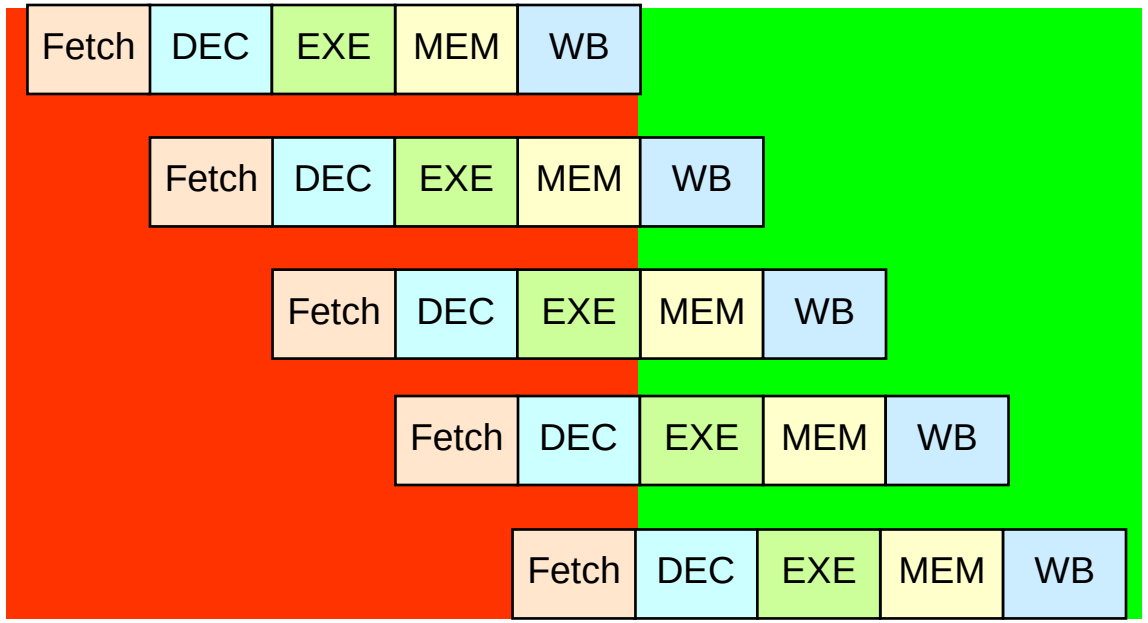



```
 0   lw   R10,9(R1)
 4   sub  R11,R2,R3
 8   and  R12,R4,R5
12   or   R13,R6,R7
```

RAW Dependency

Program execution order

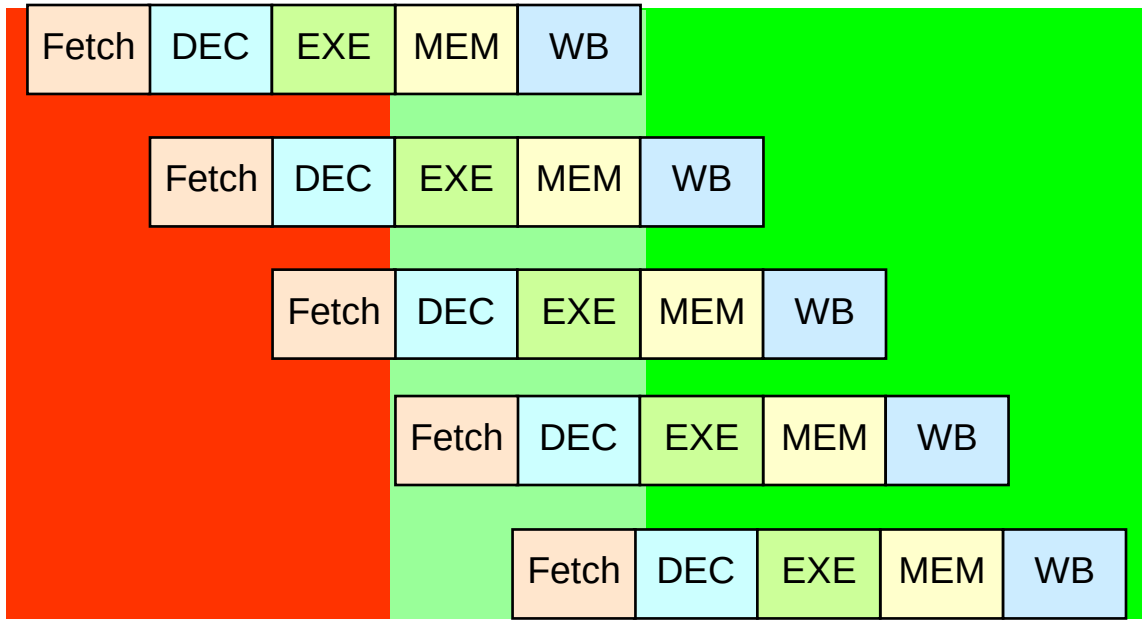
```
sub R2, R1, R3
and R12,R2, R5
or  R13,R6, R2
add R14,R2, R2
sw  R15,100(R2)
```



Using Bypass to Solve RAW Dependency

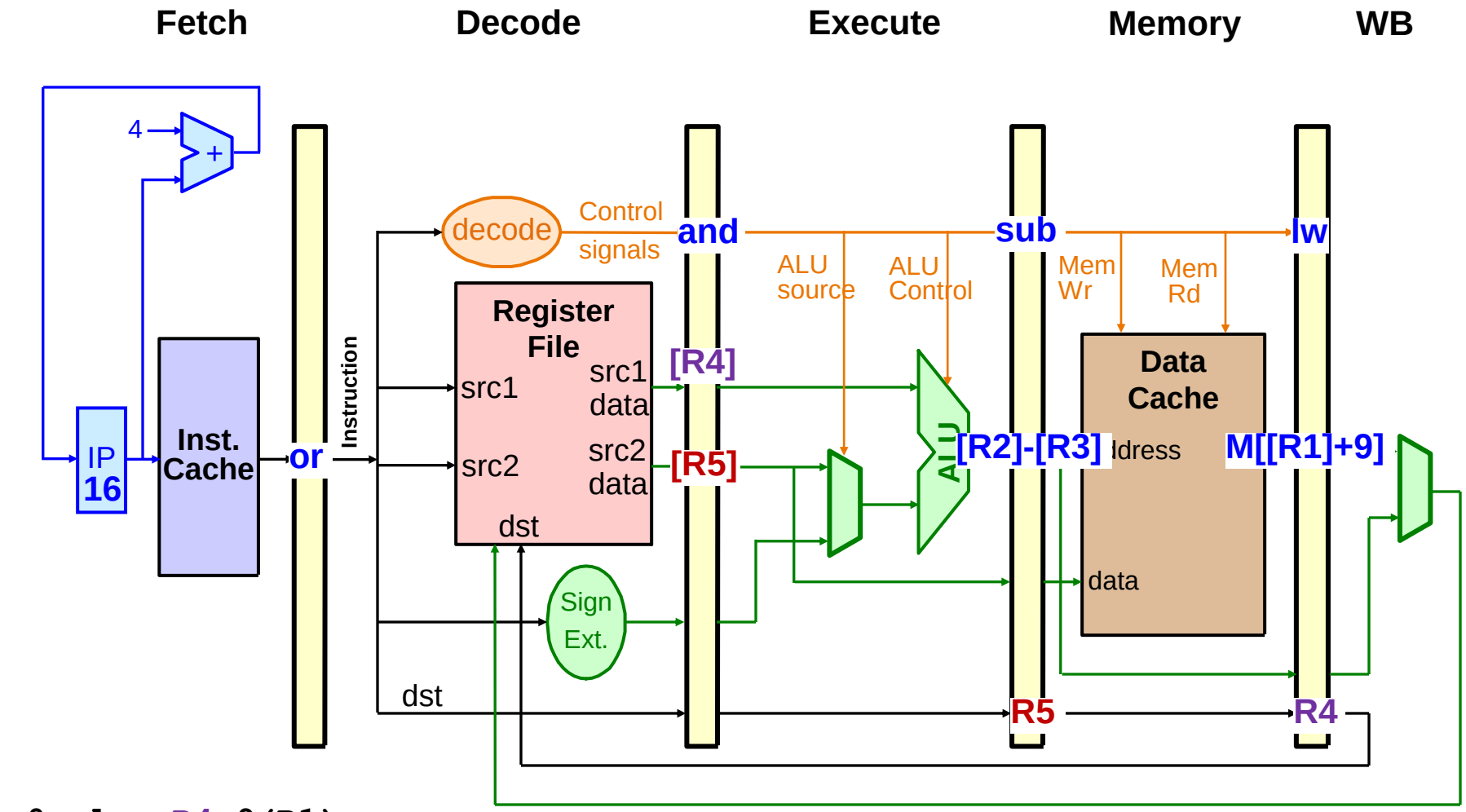
Program execution order

```
sub R2, R1, R3
and R12,R2, R5
or  R13,R6, R2
add R14,R2, R2
sw  R15,100(R2)
```



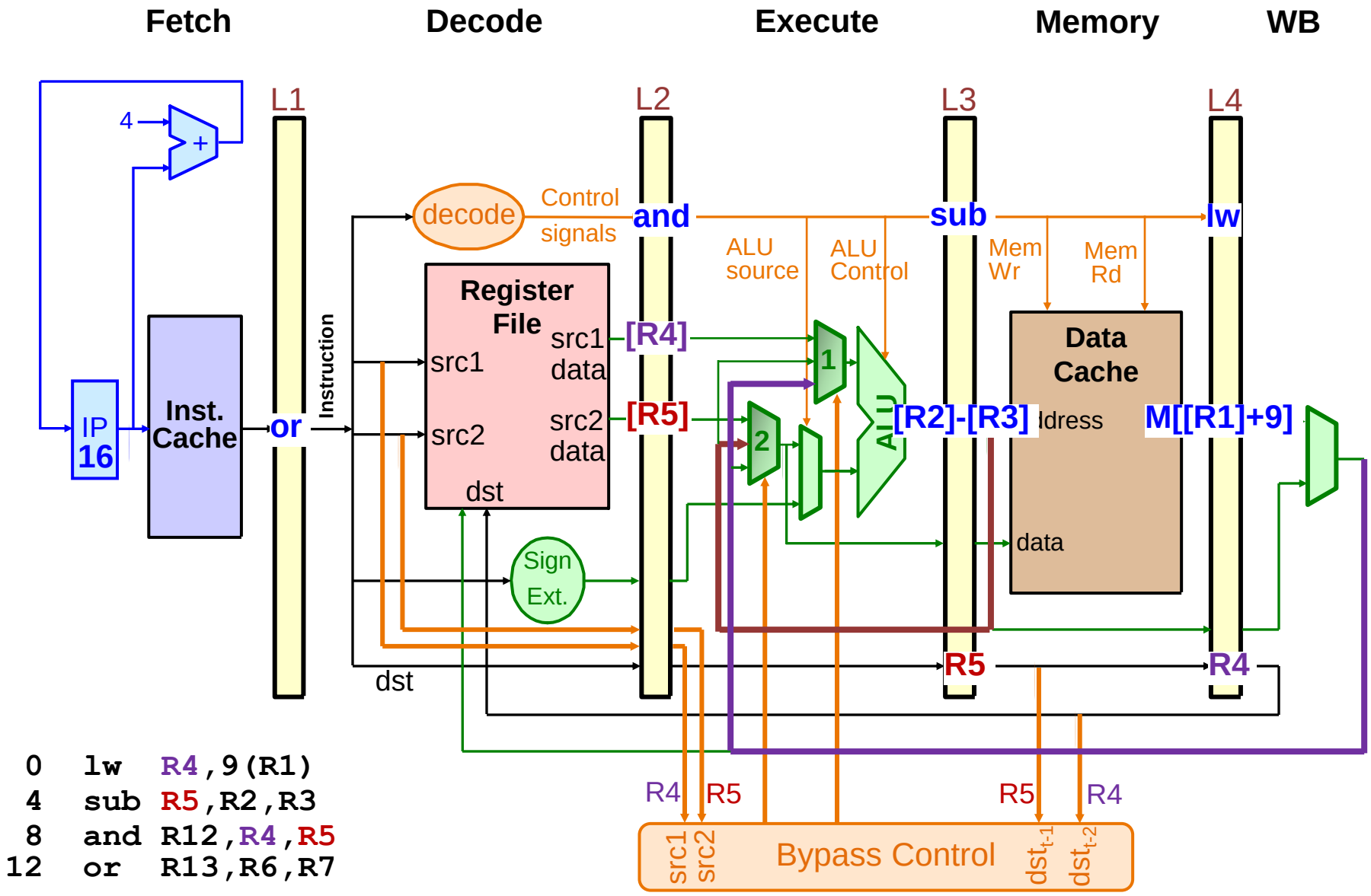
Bypass result directly from EXE output to EXE input

RAW Dependency



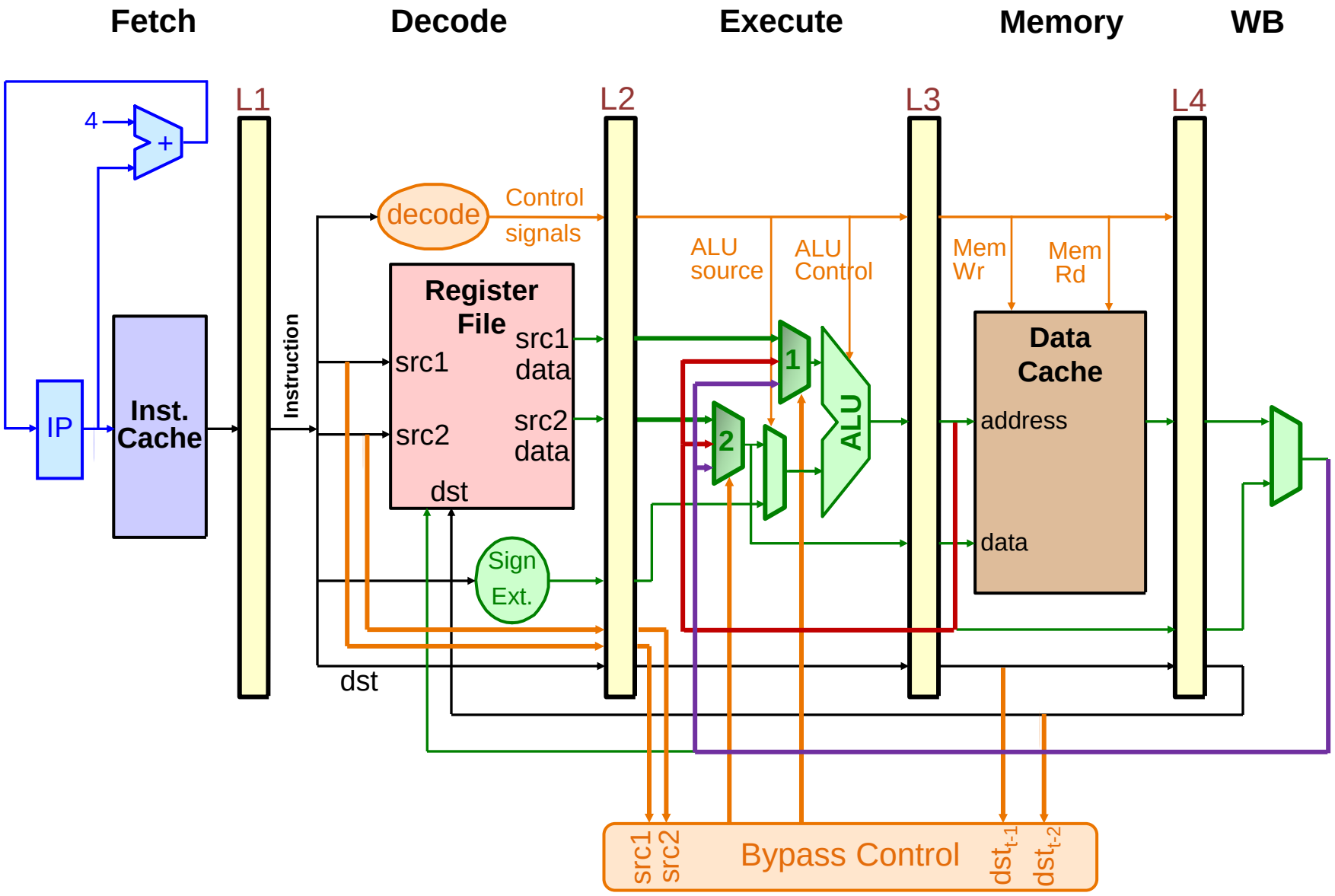

```
 0   lw   R4,9(R1)
 4   sub  R5,R2,R3
 8   and  R12,R4,R5
12   or   R13,R6,R7
```

Bypass Hardware




```
 0   lw   R4,9(R1)
 4   sub  R5,R2,R3
 8   and  R12,R4,R5
12   or   R13,R6,R7
```

Bypass Hardware

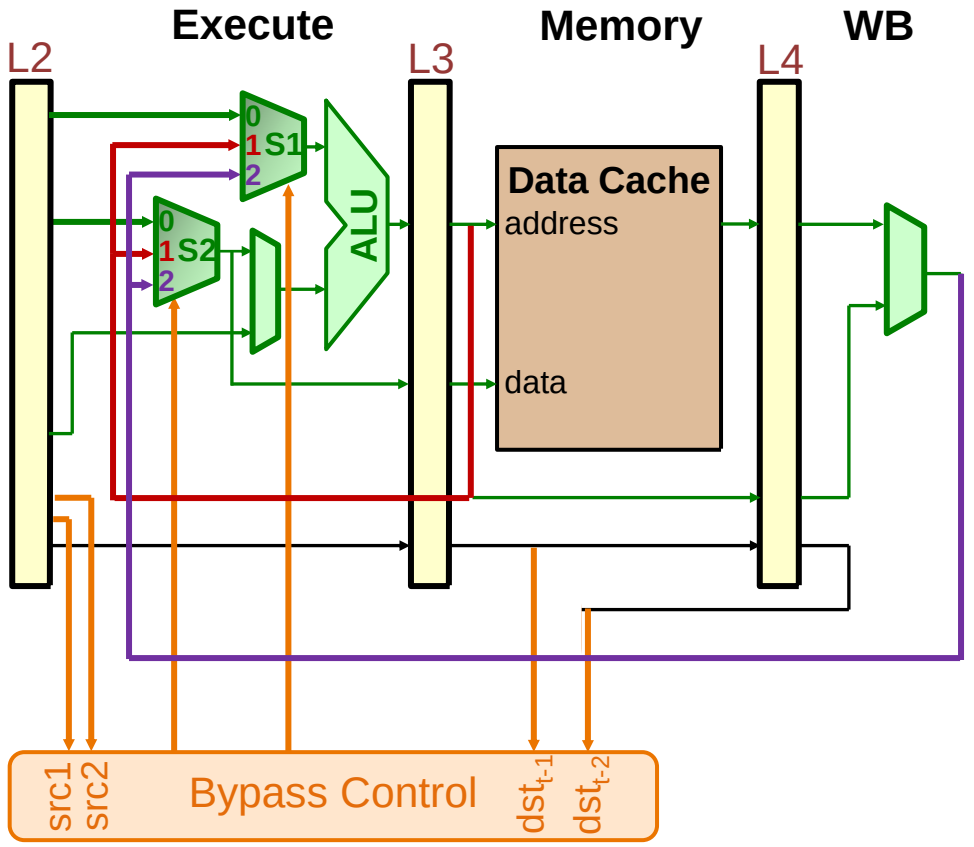


Bypass Control

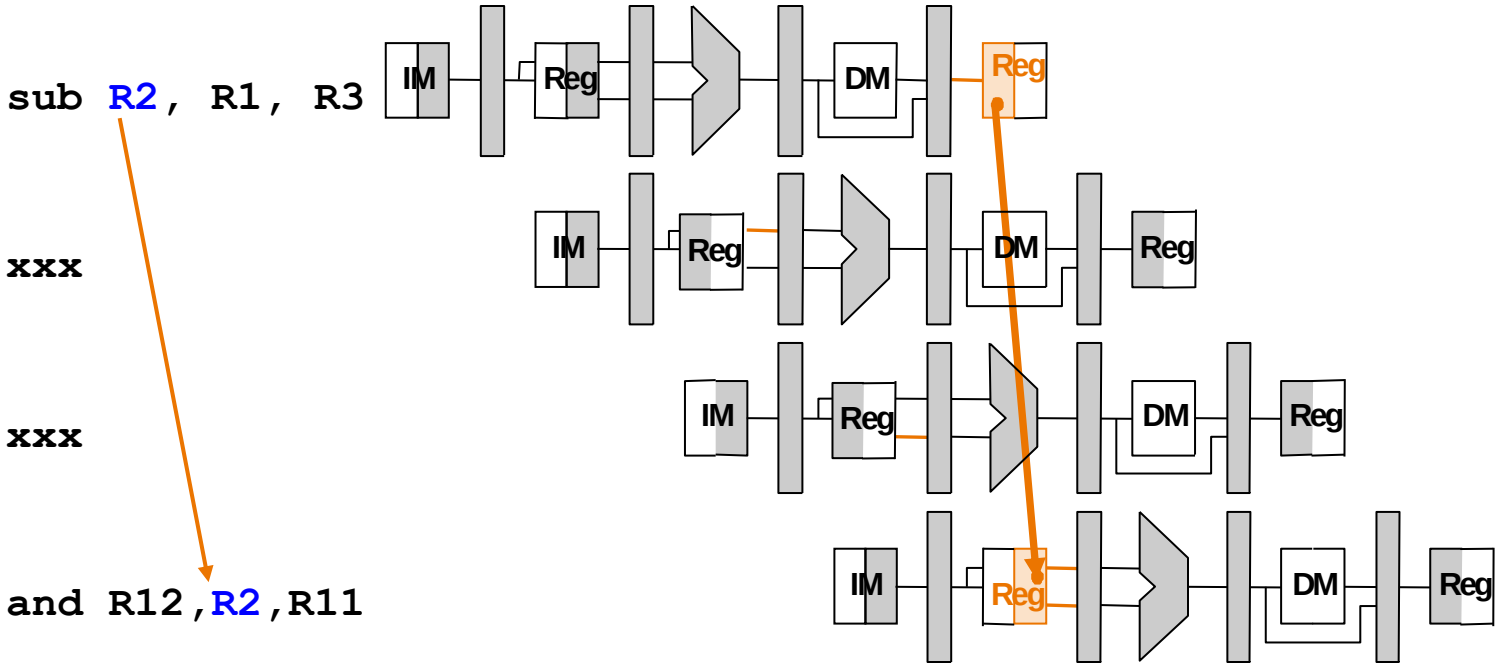
- L3_reg_wr = L3.RegWrite and (L3.opcode != lw)

- **Forwarding from EXE (L3)**
  - if (L3_reg_wr and (L3.dst == L2.src1))
    ALUSelA = 1
  - if (L3_reg_wr and (L3.dst == L2.src2))
    ALUSelB = 1

- **Forwarding from MEM (L4)**
  - if (L4.RegWrite and
    ((not L3_reg_wr) or (L3.dst ≠ L2.src1)) and
    (L4.dst = L2.src1)) ALUSelA = 2
  - if (L4.RegWrite and
    ((not L3_reg_wr) or (L3.dst ≠ L2.src2)) and
    (L4.dst = L2.src2)) ALUSelB = 2



Register File Split

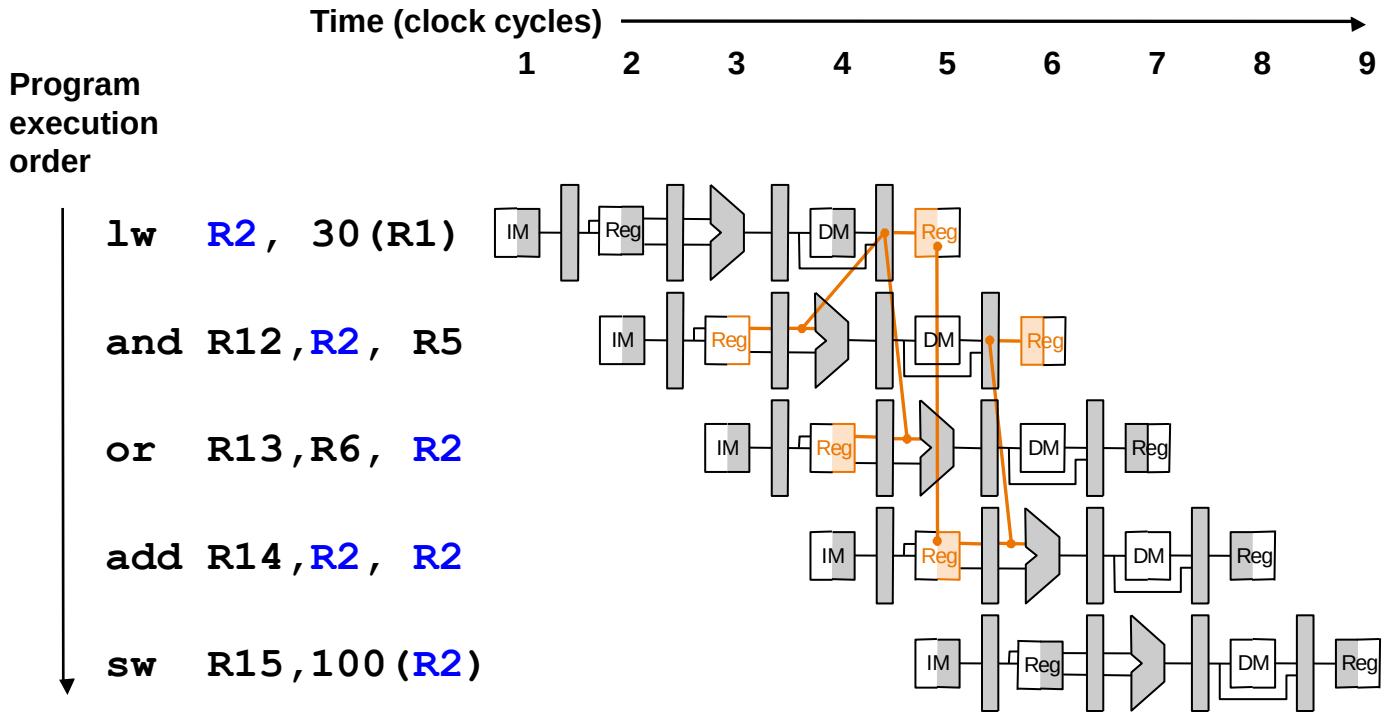


- Register file is written during first half of the cycle
- Register file is read during second half of the cycle

⇒ Register file is written before it is read ⇒ returns the correct data

Can't Always Bypass

- **Load word can still cause a hazard**
  - an instruction tries to read a register following a load instruction that writes to the same register

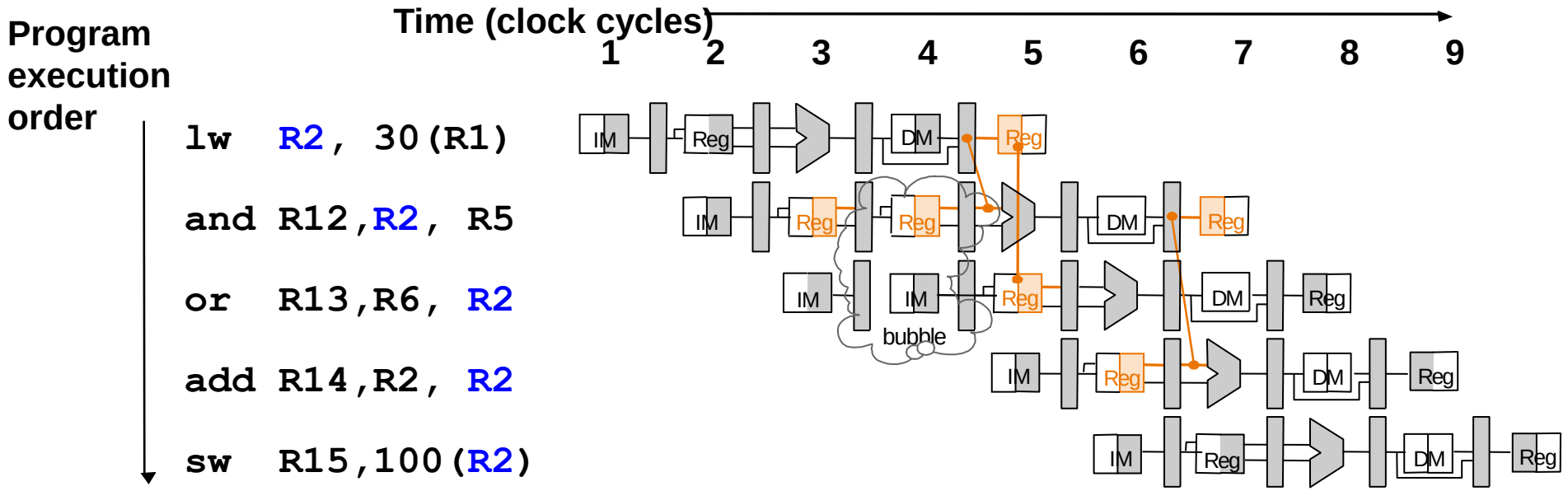


⇒ A hazard detection unit is needed to “stall” the load instruction

Stall If Cannot Bypass

if (L2.RegWrite and (L2.opcode == lw) and ((L2.dst == L1.src1) or (L2.dst == L1.src2)) then stall

- De-assert the enable to the L1 latch, and to the IP
  - The dependent instruction (**`and`**) stays another cycle in L1
- Issue a NOP into the L2 latch (instead of the stalled inst.)
- Allow the stalling instruction (**`lw`**) to move on



Software Scheduling to Avoid Load Hazards

Example: code for (assume all variables are in memory):

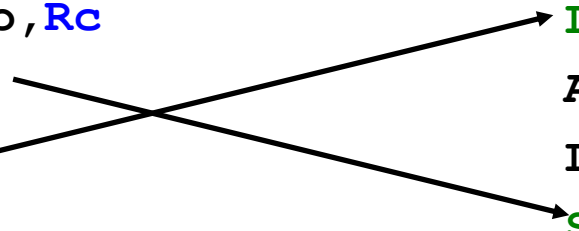
```
a = b + c;
d = e - f;
```

Slow code

```
        LW      Rb,b
        LW      Rc,c
Stall   ADD     Ra,Rb,Rc
        SW      a,Ra
        LW      Re,e
        LW      Rf,f
Stall   SUB     Rd,Re,Rf
        SW      d,Rd
```

Fast code

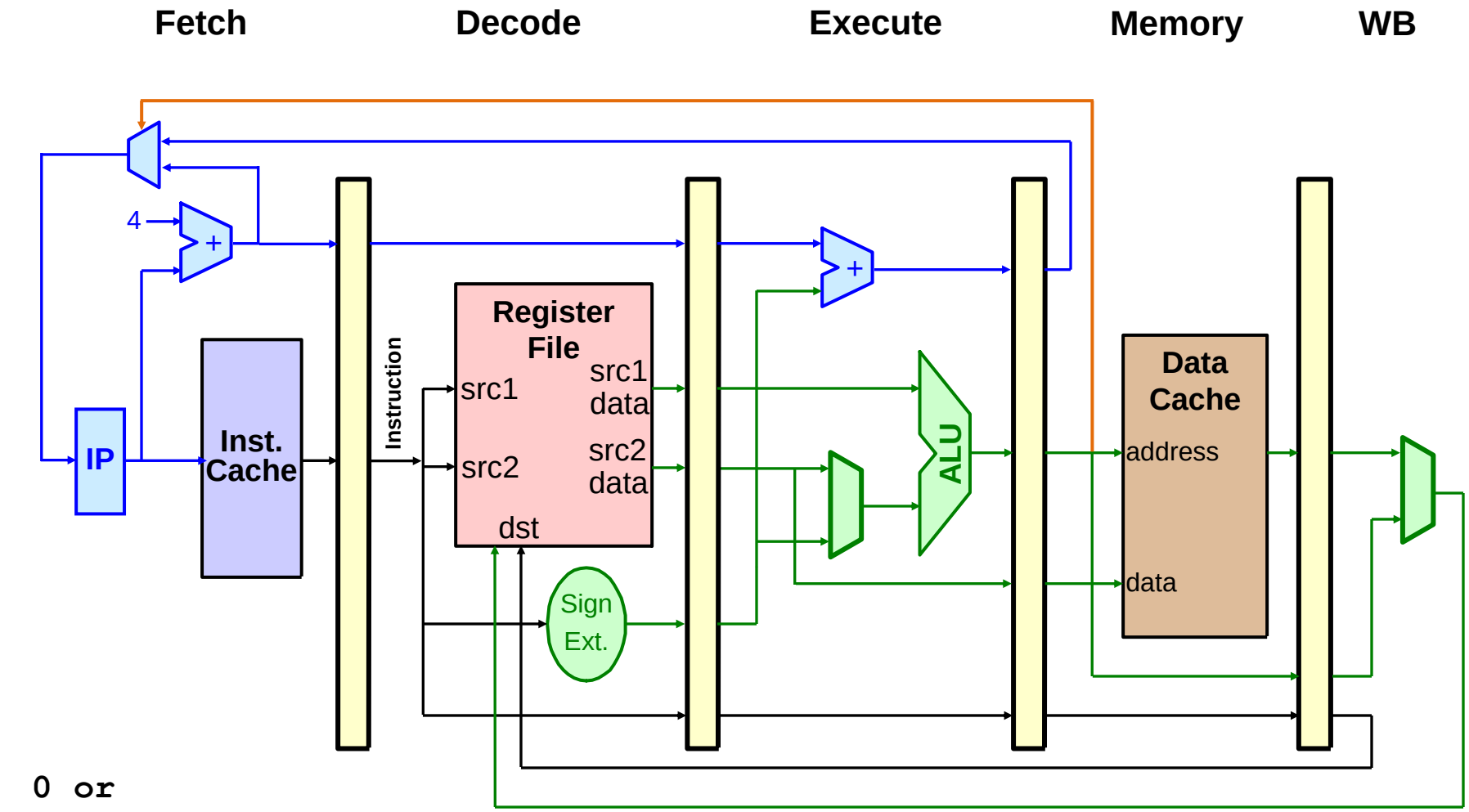
```
LW      Rb,b
LW      Rc,c
LW      Re,e
ADD     Ra,Rb,Rc
LW      Rf,f
SW      a,Ra
SUB     Rd,Re,Rf
SW      d,Rd
```



Instruction order can be changed as long as the correctness is kept

Control Hazards

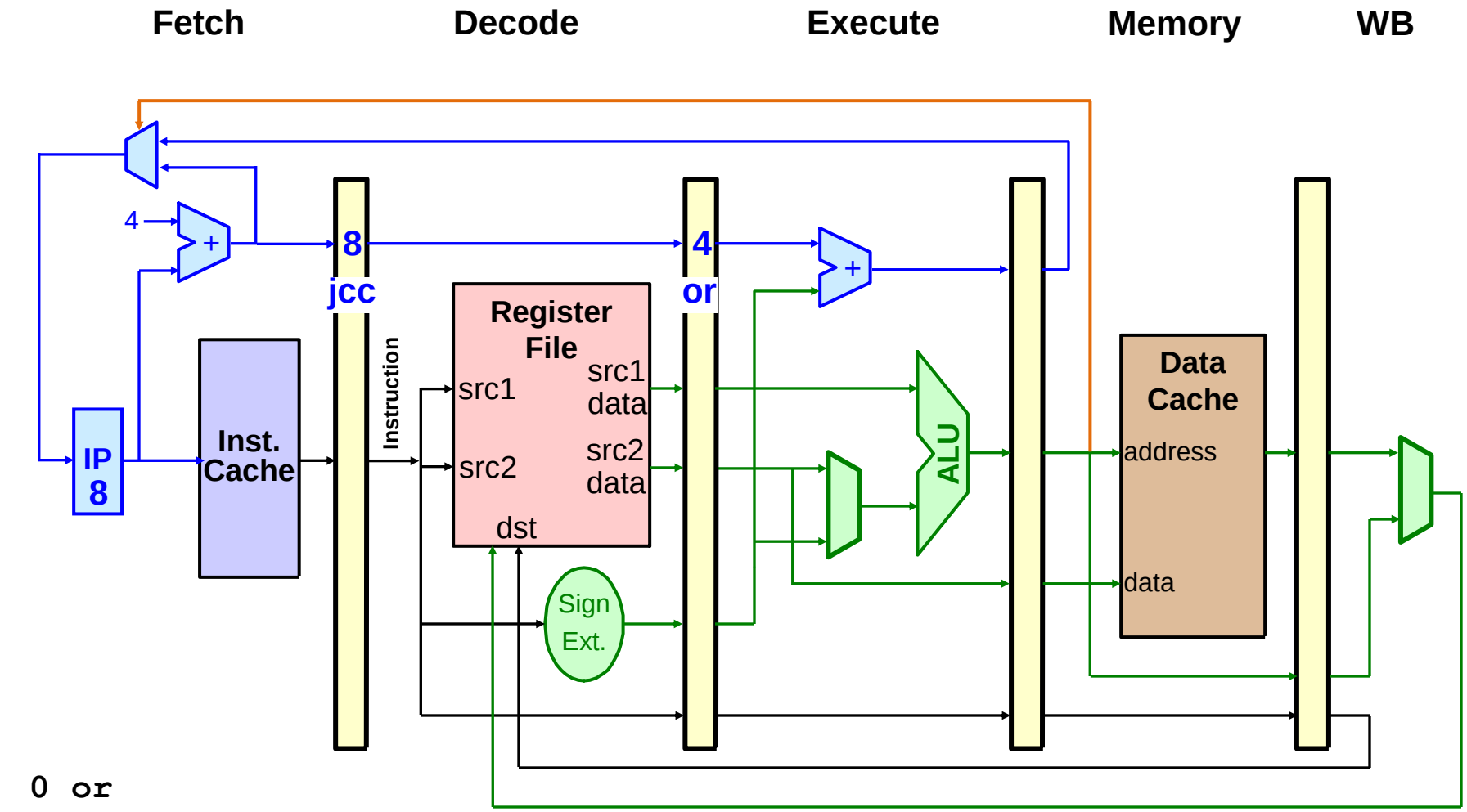
Control Hazard on Branches




```
 0 or
 4 jcc 50H
 8 and
12 sw
16 sub
```

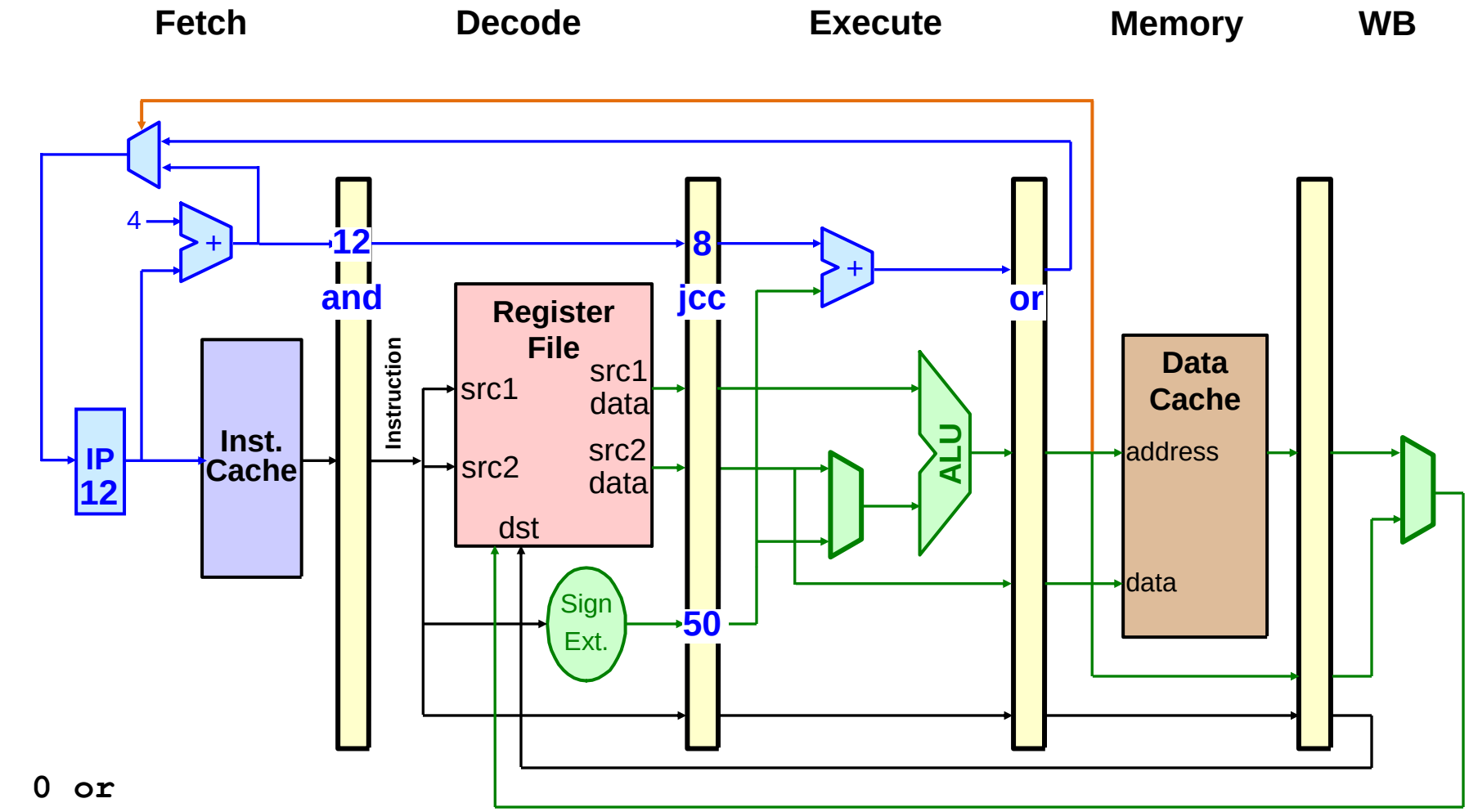
jcc target; if cond then IP ← target

Control Hazard on Branches




```
 0 or
 4 jcc 50H
 8 and
12 sw
16 sub
```

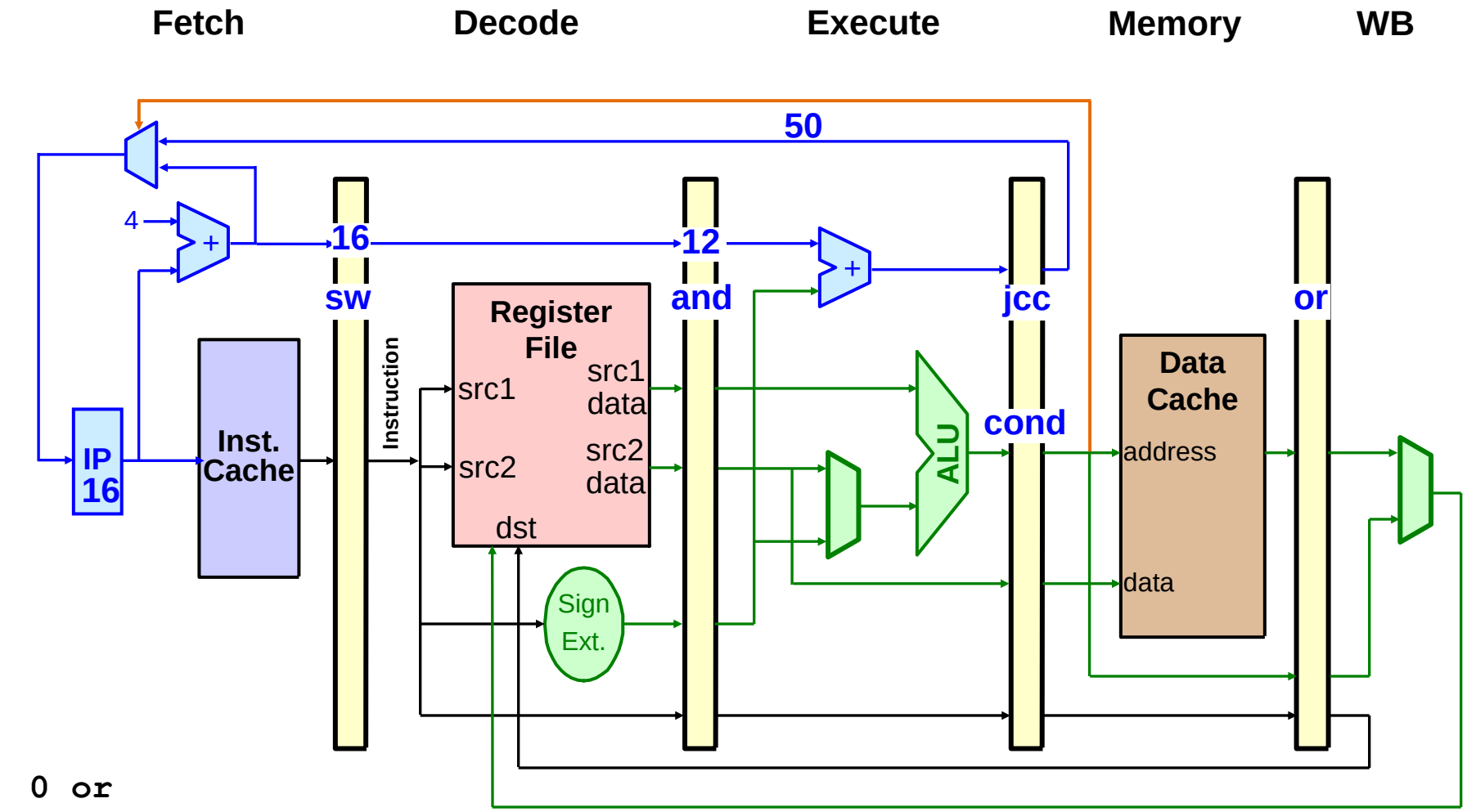
Control Hazard on Branches




```
 0 or
 4 jcc 50H
 8 and
12 sw
16 sub
```

Instructions following the branch get into the pipe

Control Hazard on Branches

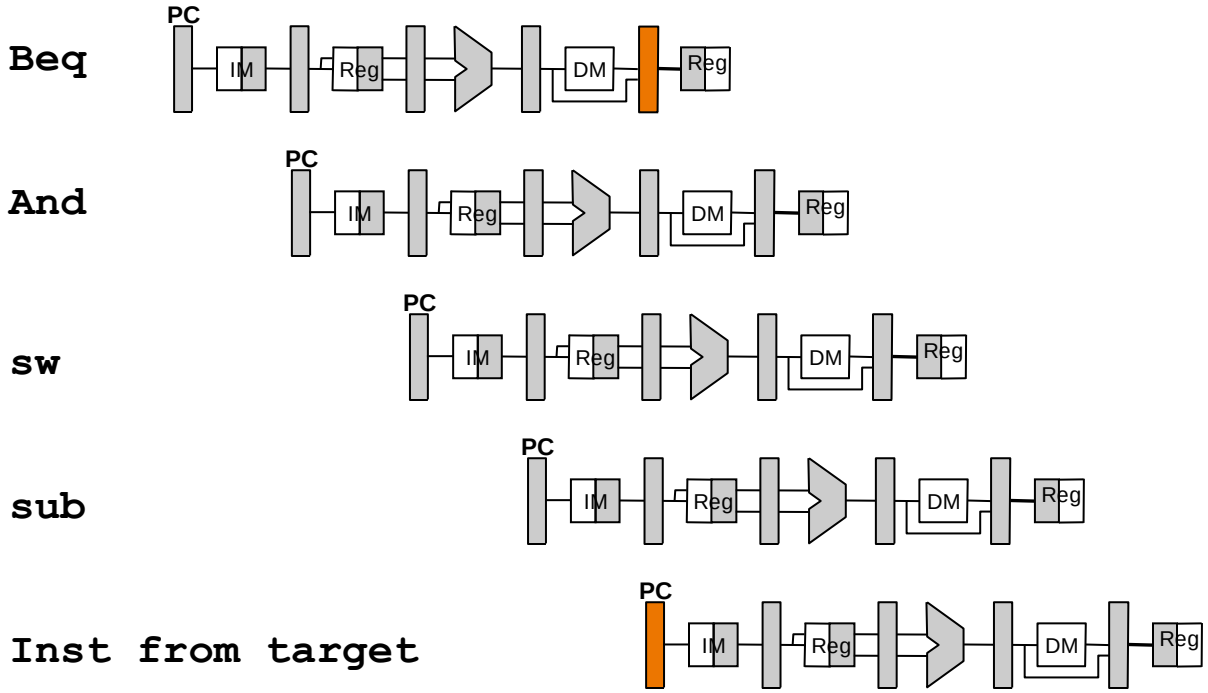



```
 0 or
 4 jcc 50H
 8 and
12 sw
16 sub
```

Instructions following the branch get into the pipe

Control Hazard on Branches

The 3 instructions following the branch get into the pipe even if the branch is taken



Control Hazard: Stall

- **Stall pipe when branch is encountered until resolved**

- **Stall impact: assumptions**
  - CPI = 1
  - 20% of instructions are branches
  - Stall 3 cycles on every branch

    $\Rightarrow$ $CPI_{new} = 1 + 0.2 \times 3 = 1.6$

($CPI_{new} = CPI_{Ideal}$ + avg. stall cycles / instr.)

We loose 60% of the performance

Control Hazard: Predict Not Taken

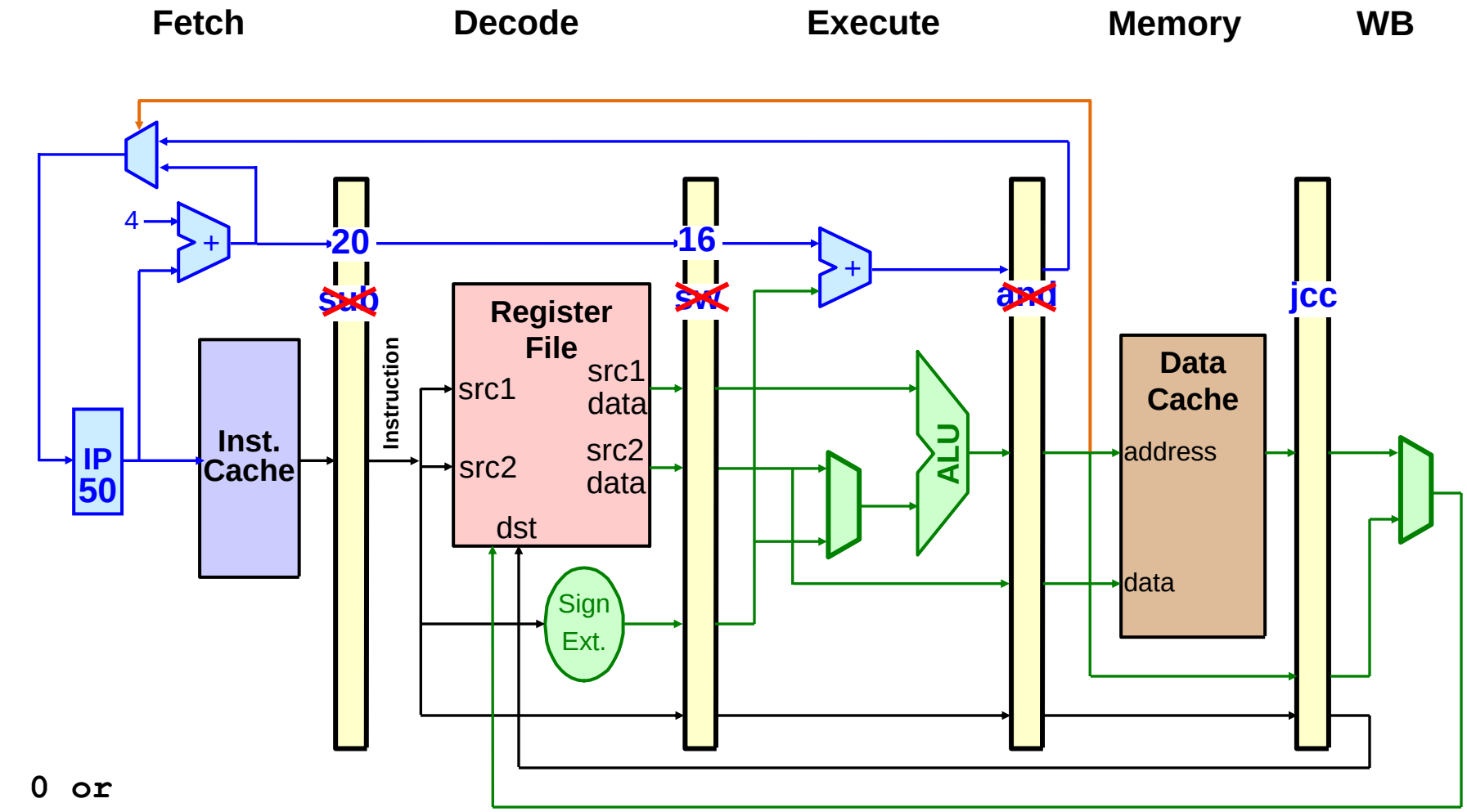
- **Execute instructions from the fall-through (not-taken) path**
  - As if there is no branch
  - If the branch is not-taken (~50%), no penalty is paid

- **If branch actually taken**
  - Flush the fall-through path instructions before they change the machine state (memory / registers)
  - Fetch the instructions from the correct (taken) path

- **Assuming 20% of instructions are branches and ~50% branches not taken on average**

  CPI new = 1 + (0.2 × 0.5) × 3 = 1.3

Control Hazard on Branches

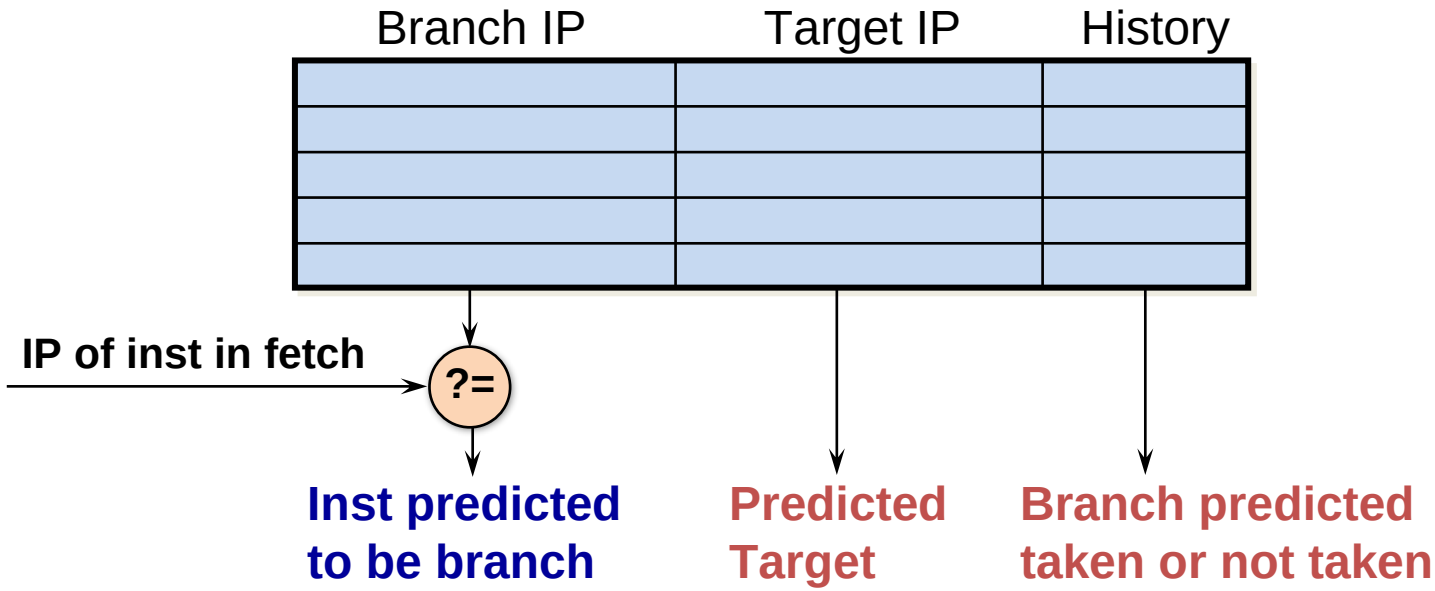



```
 0 or
 4 jcc 50H
 8 and
12 sw
16 sub
```

If the jcc is resolved as taken at EXE ⇒ flush the pipeline:
Reset the pipe latches, and set IP ← target

Dynamic Branch Prediction

- **Add a Branch Target Buffer (BTB) that predicts (at fetch)**
  - Instruction is a branch
  - Branch taken / not-taken
  - Taken branch target



- **BTB allocated at execute – after all branch info is known**
- **BTB is looked up at instruction fetch**

BTB

- **Allocation – at Decode / EXE**
  - Allocate instructions identified as branches (after decode)
    - Both conditional and unconditional branches are allocated
  - Not taken branches need not be allocated
    - BTB miss implicitly predicts not-taken
- **Prediction – at Fetch**
  - BTB lookup is done parallel to IC lookup
  - BTB provides
    - Indication that the instruction is a branch (BTB hits)
    - Branch predicted target
    - Branch predicted direction
    - Branch predicted type (e.g., conditional, unconditional)
- **Update (when branch outcome is known – at EXE)**
  - Branch target
  - Branch history (taken / not-taken)

BTB (cont.)

- **Wrong prediction**
  - Predict not-taken, actual taken
  - Predict taken, actual not-taken, or actual taken but wrong target

- **In case of wrong prediction – flush the pipeline**
  - Reset latches (same as making all instructions to be NOPs)
  - Select the PC source to be from the correct path
    - Need get the fall-through with the branch
  - Start fetching instruction from correct path

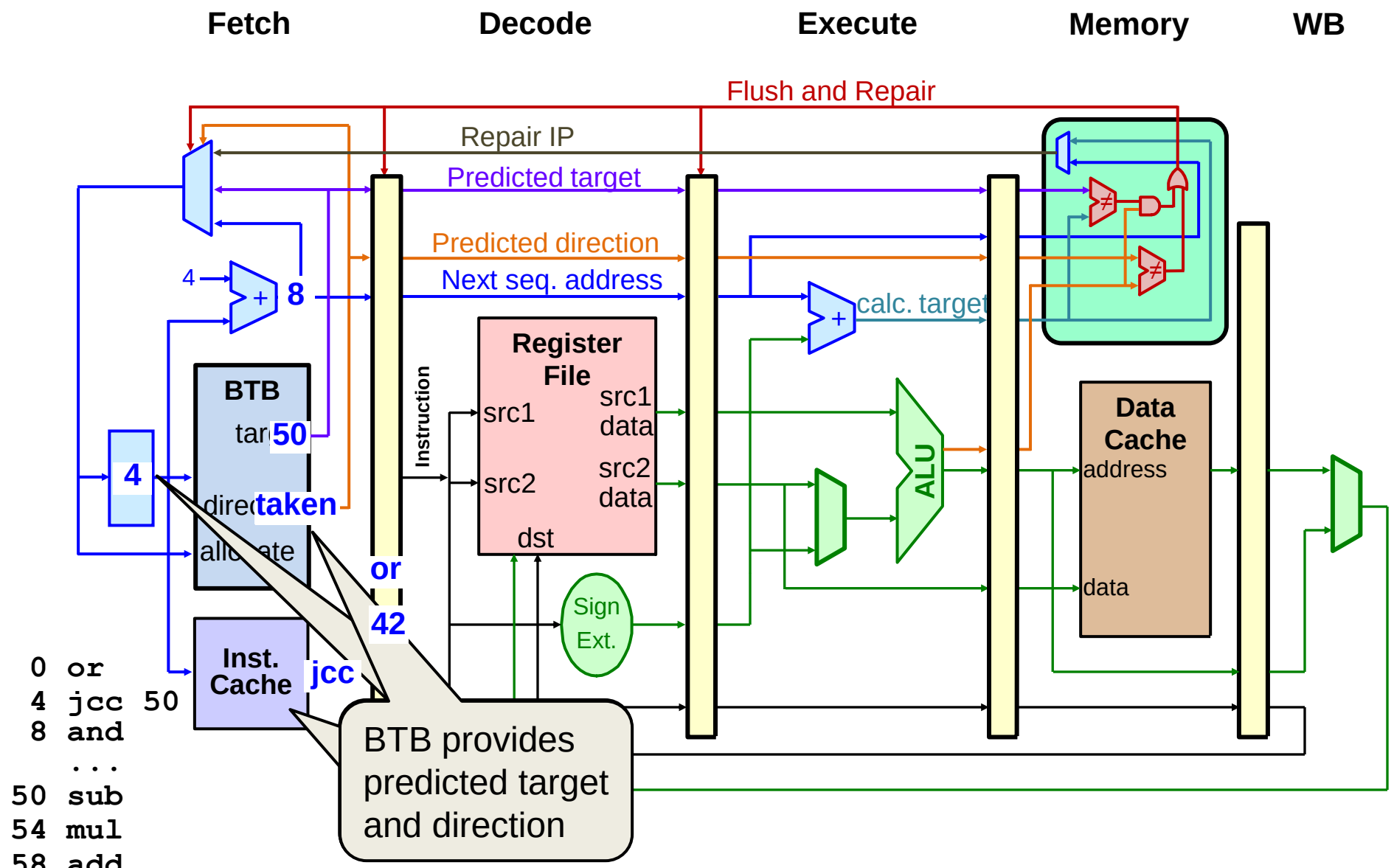
- **Assuming P% correct prediction rate**

  $$CPI_{new} = 1 + (0.2 \times (1-P)) \times 3$$

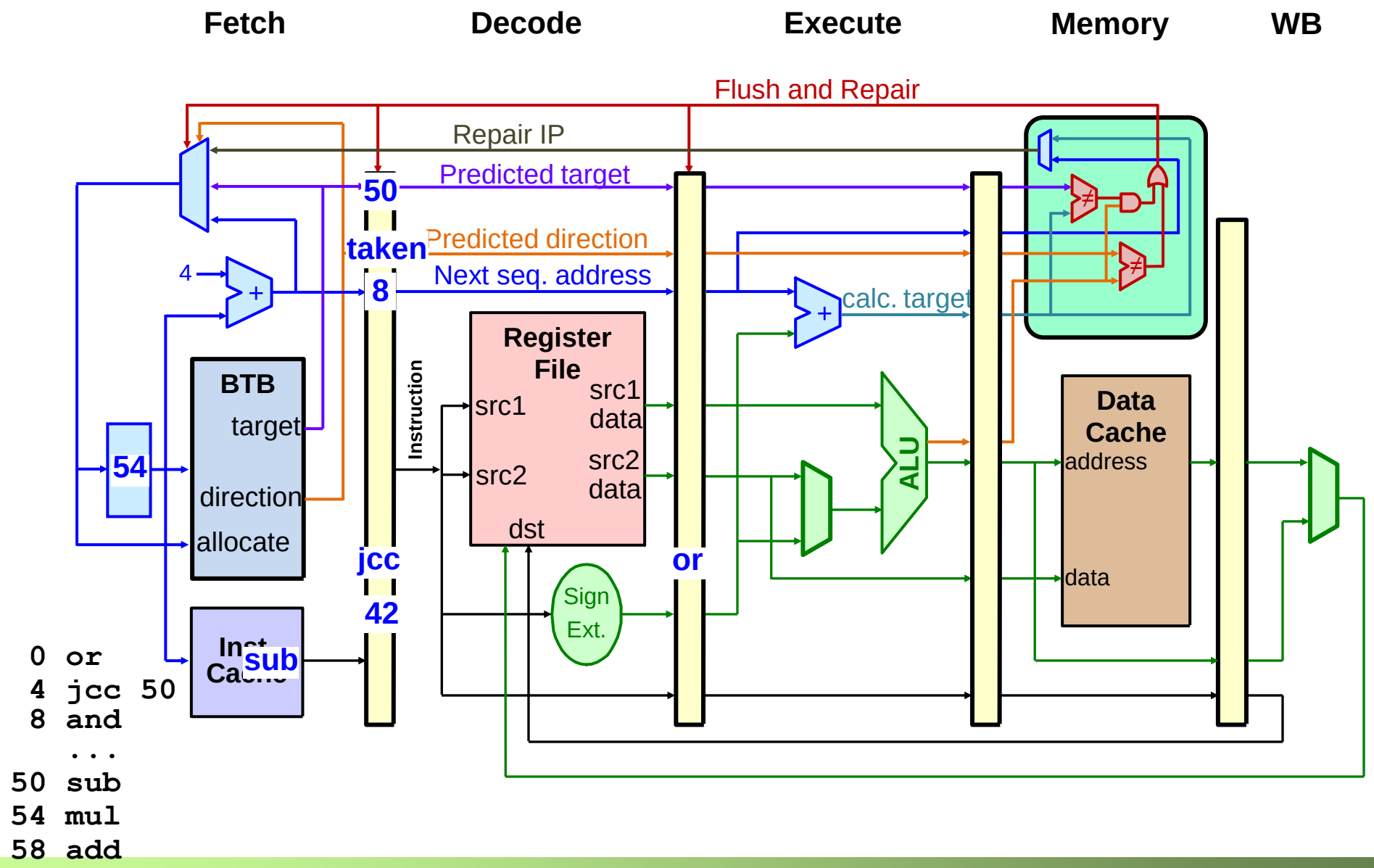
  - For example, if P=0.7

  $$CPI_{new} = 1 + (0.2 \times 0.3) \times 3 = 1.18$$

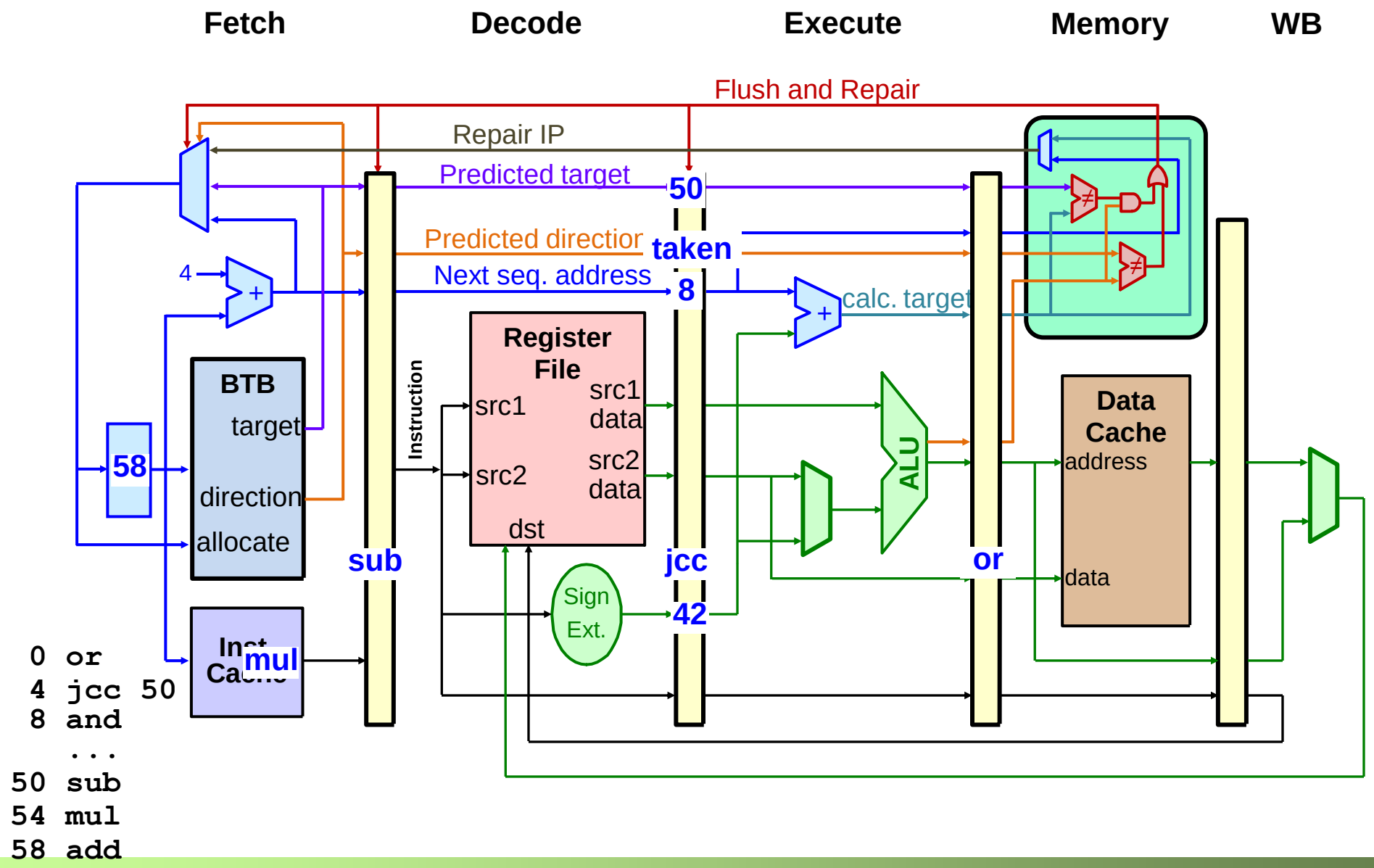
Adding a BTB to the Pipeline



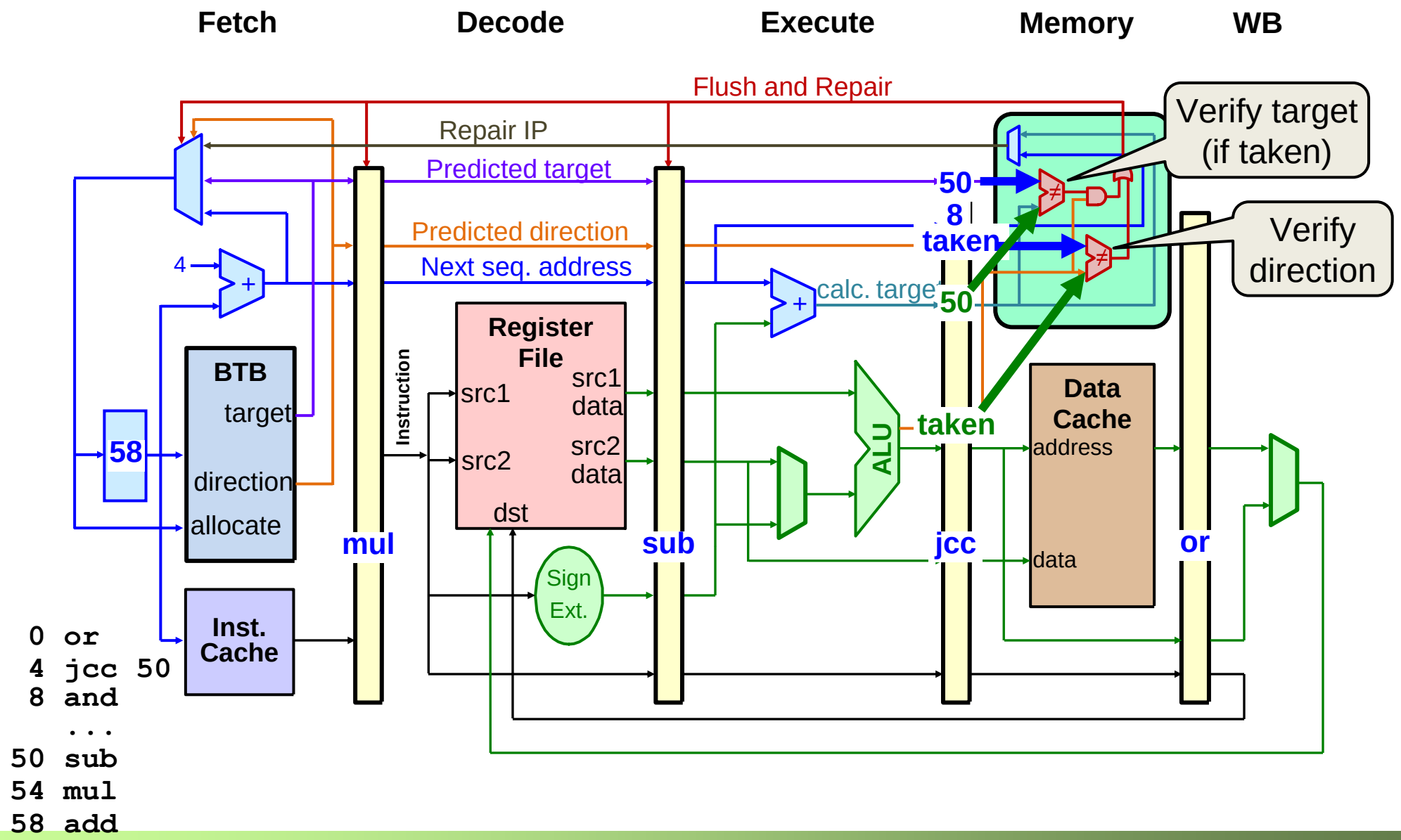
Adding a BTB to the Pipeline



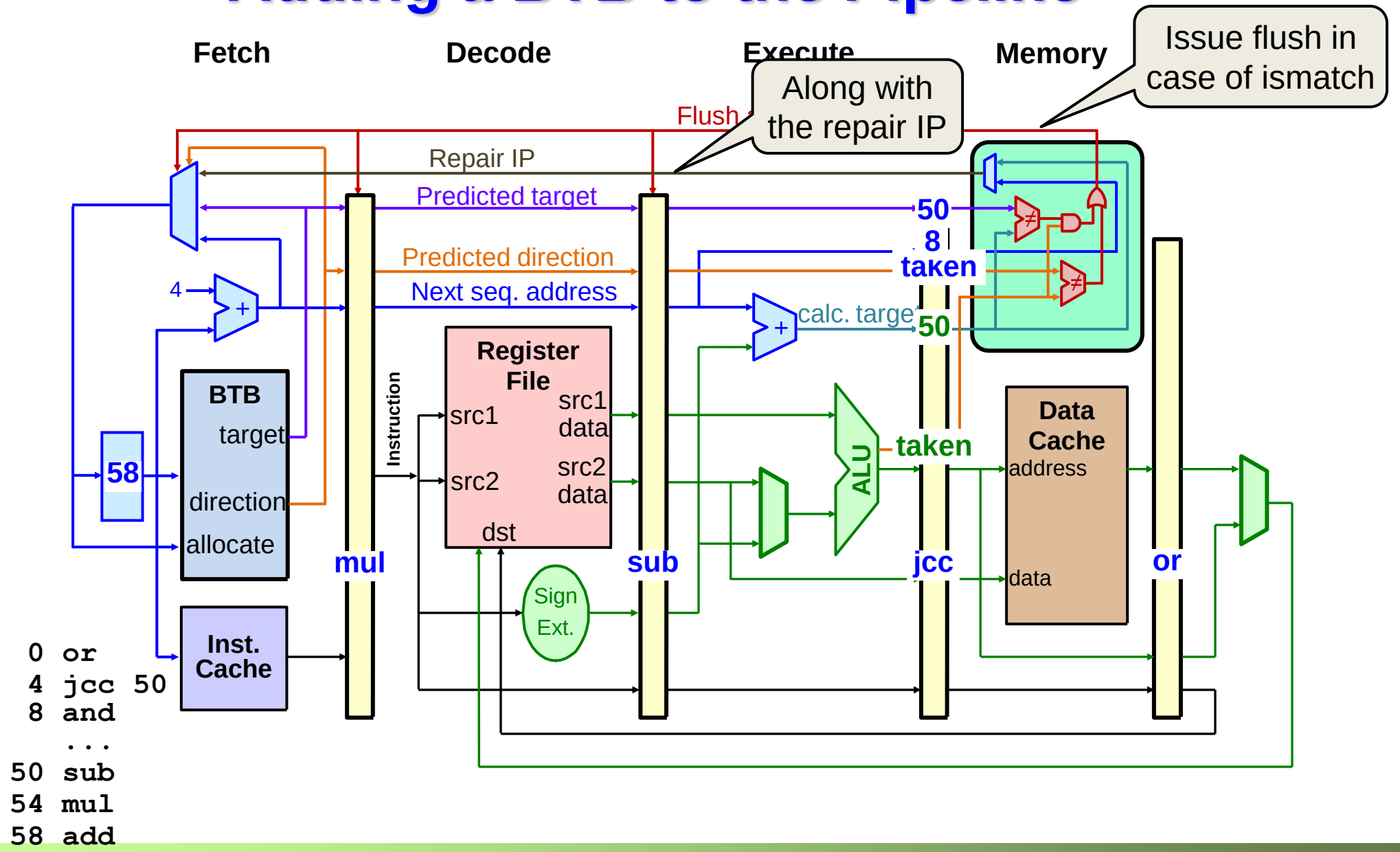
Adding a BTB to the Pipeline



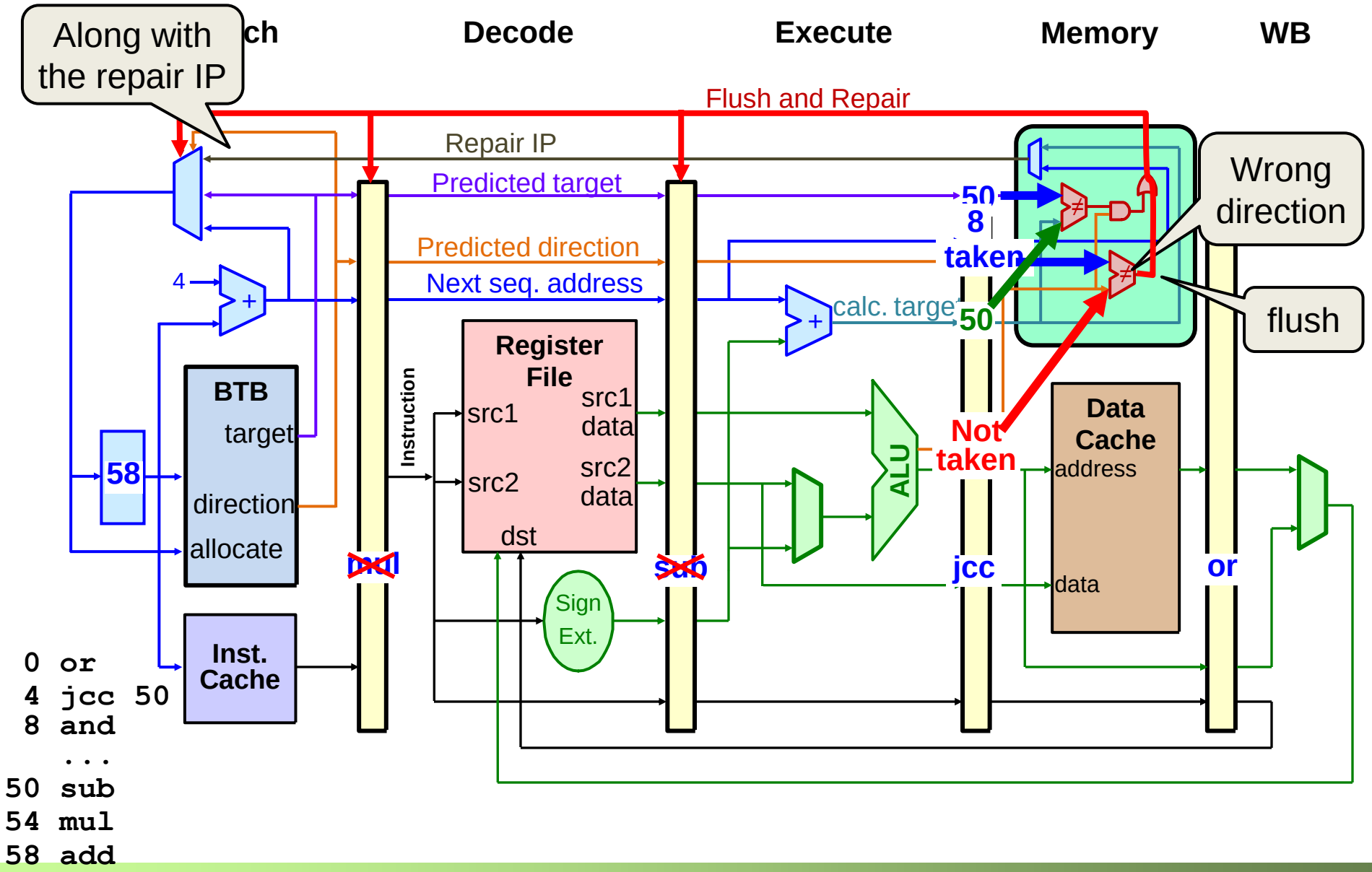
Adding a BTB to the Pipeline



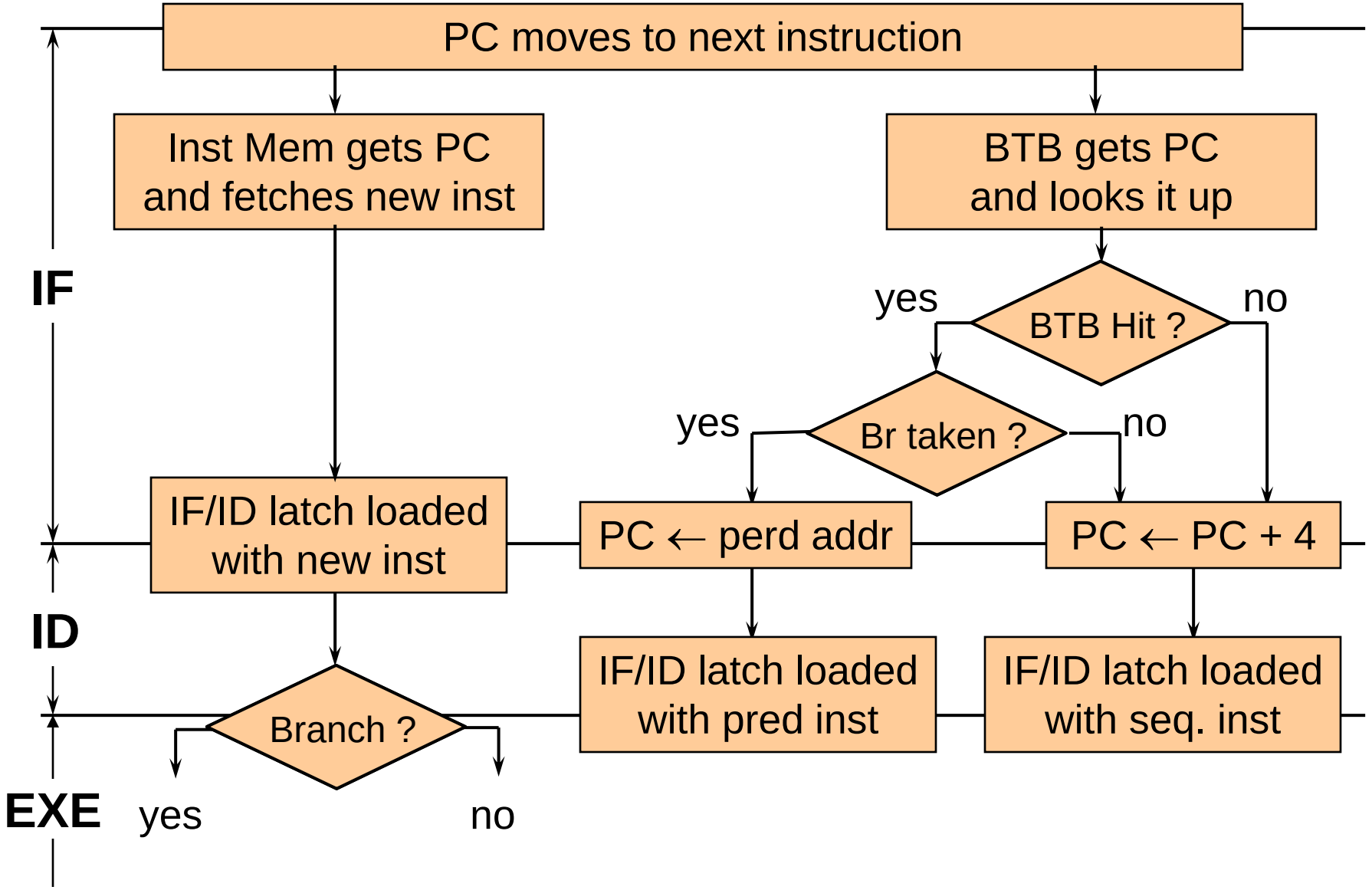
Adding a BTB to the Pipeline



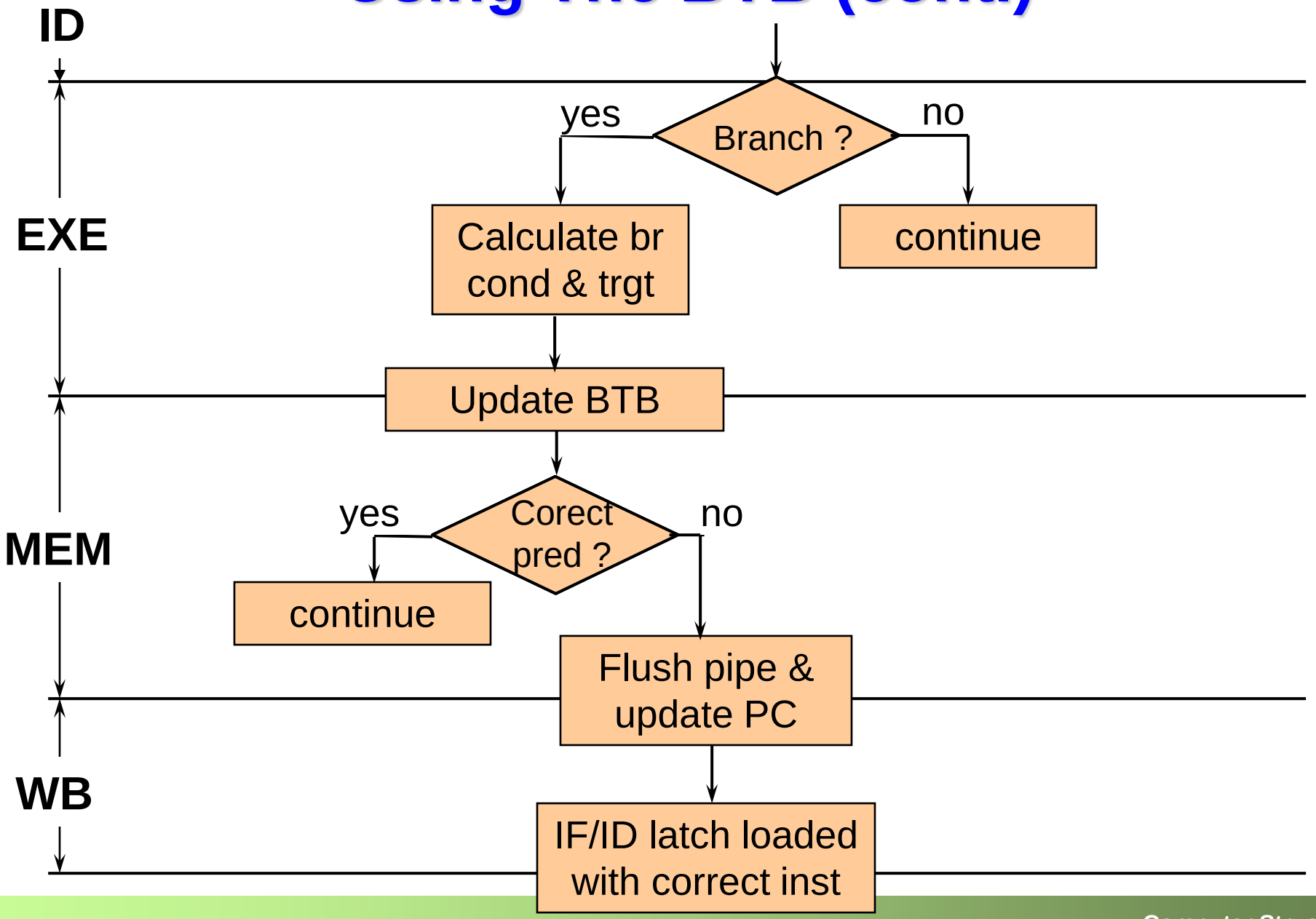
Predicted Taken Actual Not Taken



Using The BTB



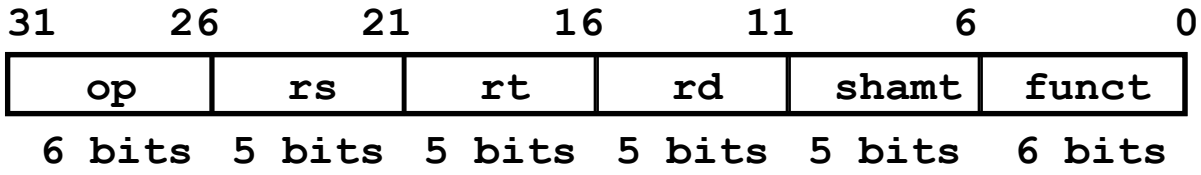
Using The BTB (cont.)



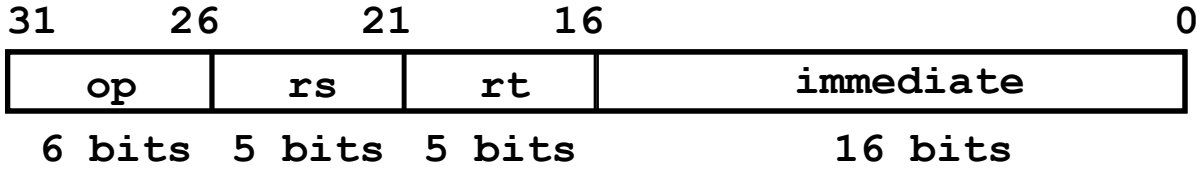
Backup

MIPS Instruction Formats

- R-type
  (register insts)



- I-type (Load, Store, Branch,
  inst's w/imm data)

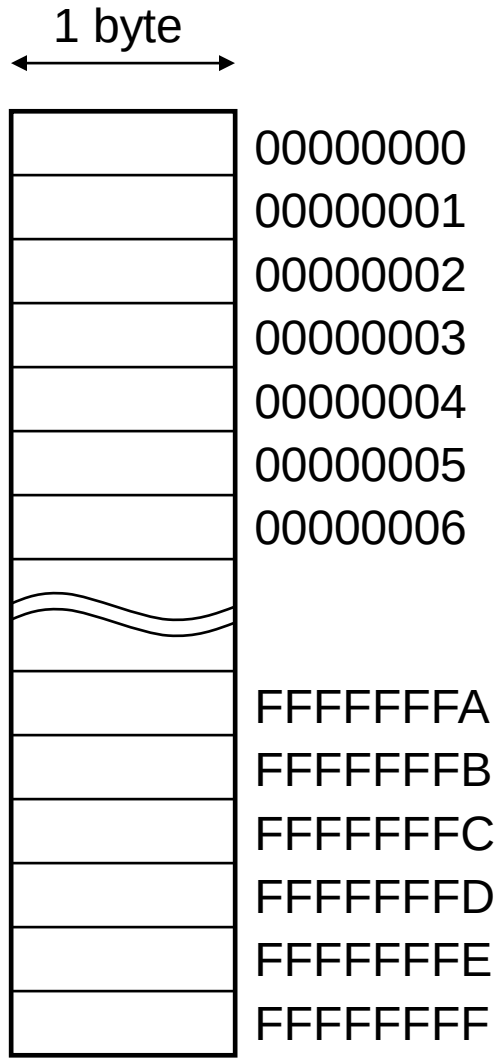


- J-type (Jump)

op: operation of the instruction
rs, rt, rd: the source and destination register specifiers
shamt: shift amount
funct: selects the variant of the operation in the "op" field
address / immediate: address offset or immediate value
target address: target address of the jump instruction

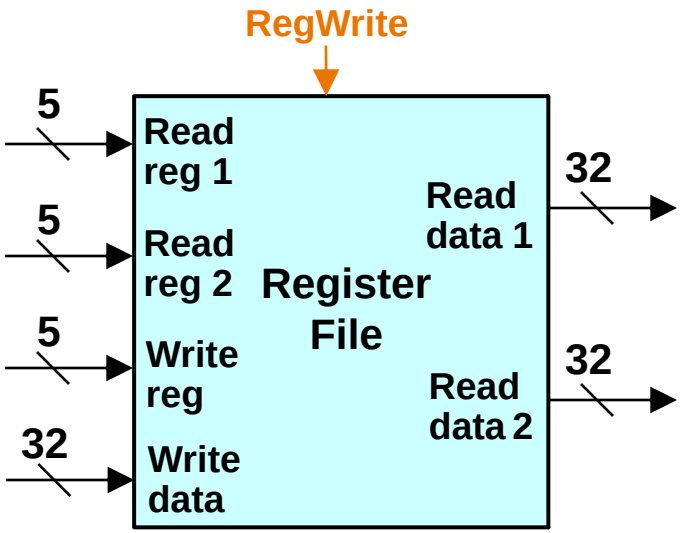
MIPS Memory Space

- **Each memory location**
  - is 8 bit = 1 byte wide
  - has an address
- **We assume 32 byte address**
  - An address space of 2^{32} bytes
- **Memory stores both instructions and data**
  - Each instruction is 32 bit wide ⇒ stored in 4 consecutive bytes in memory
  - Various data types have different width



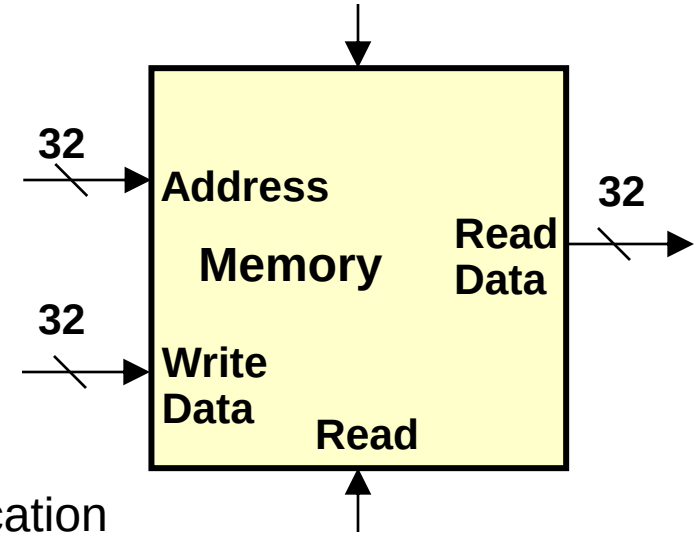
Register File

- **The Register File holds 32 registers**
- **Each register is 32 bit wide**
- **The RF supports parallel**
  - reading any two registers and
  - writing any register
- **Inputs**
  - **Read reg 1/2**: #register whose value will be output on *Read data 1/2*
  - **RegWrite**: write enable
  - **Write reg** (relevant when *RegWrite*=1)
    - #register to which the value in *Write data* is written to
  - **Write data** (relevant when *RegWrite*=1)
    - data written to *Write reg*
- **Outputs**
  - **Read data 1/2:** data read from *Read reg 1/2*



Memory Components

- **Inputs**
  - **Address**: address of the memory location we wish to access
  - **Read**: read data from location
  - **Write**: write data into location
  - **Write data** (relevant when Write=1) data to be written into specified location
- **Outputs**
  - **Read data** (relevant when Read=1) data read from specified location



Cache

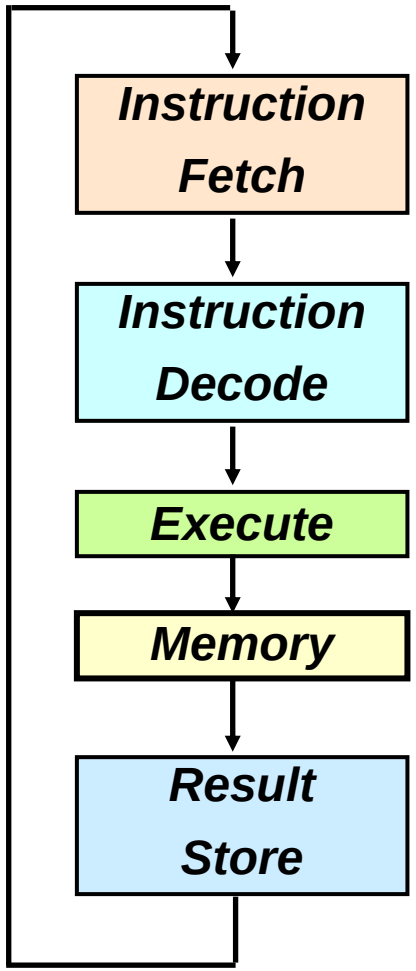
- **Memory components are slow relative to the CPU**
  - A **cache** is a fast memory which contains only small part of the memory
  - **Instruction cache** stores parts of the memory space which hold code
  - **Data Cache** stores parts of the memory space which hold data

The Program Counter (PC)

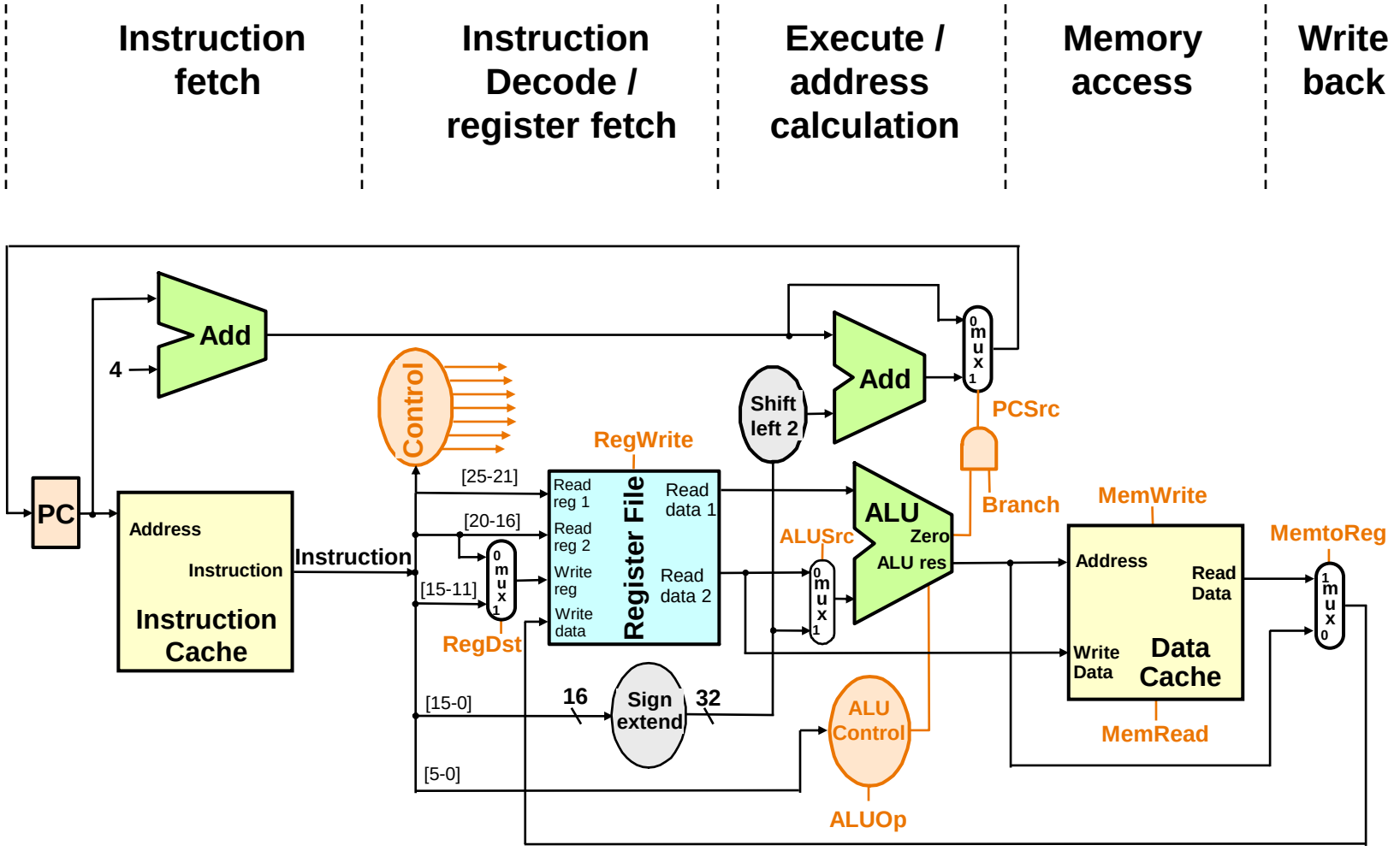
- **Holds the address (in memory) of the next instruction to be executed**
- **After each instruction, advanced to point to the next instruction**
  - If the current instruction is not a taken branch,
    the next instruction resides right after the current instruction
    PC ← PC + 4
  - If the current instruction is a taken branch,
    the next instruction resides at the branch target
    PC ← target (absolute jump)
    PC ← PC + 4 + offset×4 (relative jump)

Instruction Execution Stages

- **Fetch**
  - Fetch instruction pointed by PC from I-Cache
- **Decode**
  - Decode instruction (generate control signals)
  - Fetch operands from register file
- **Execute**
  - For a memory access: calculate effective address
  - For an ALU operation: execute operation in ALU
  - For a branch: calculate condition and target
- **Memory Access**
  - For load: read data from memory
  - For store: write data into memory
- **Write Back**
  - Write result back to register file
  - update program counter



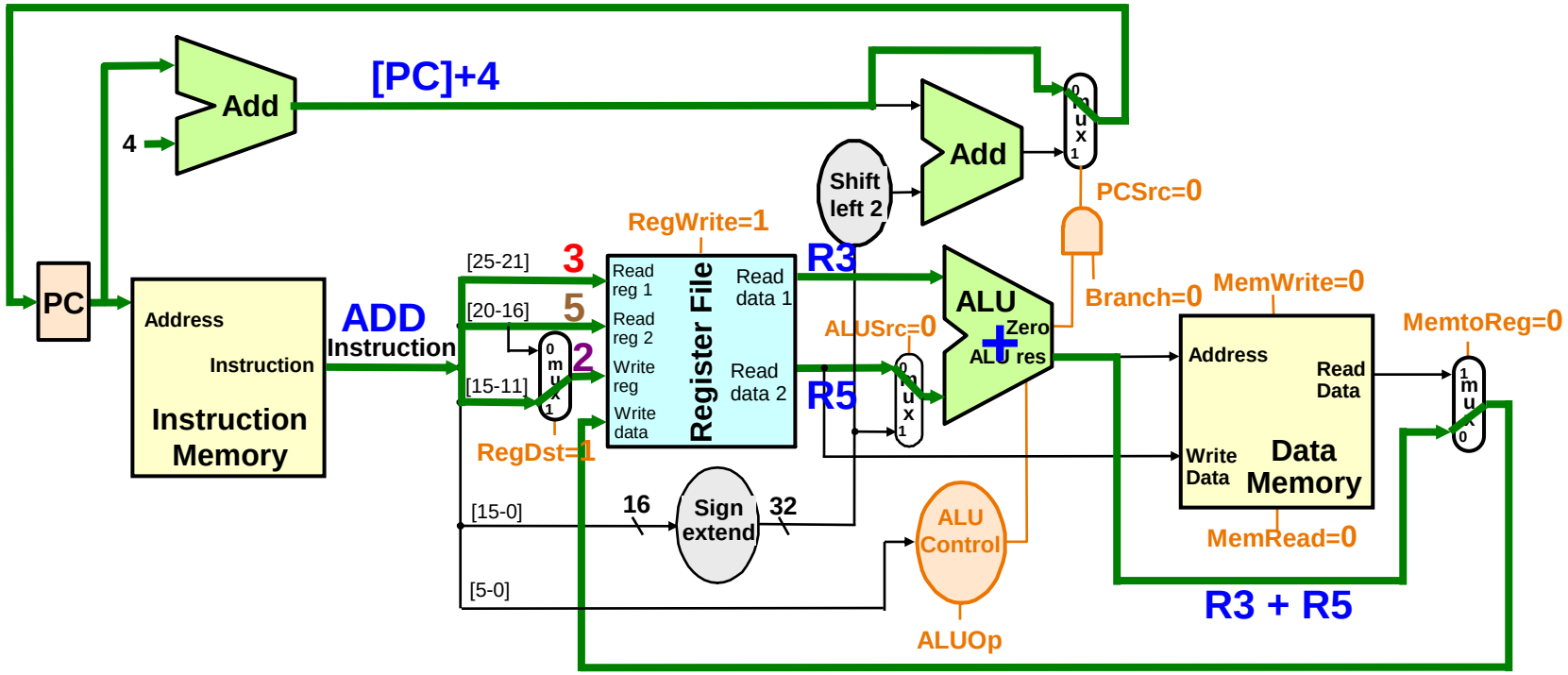
The MIPS CPU


Instruction fetch
Instruction Decode / register fetch
Execute / address calculation
Memory access
Write back
Add
4
PC
Address
Instruction
Instruction Cache
Instruction
Control
[25-21]
[20-16]
[15-11]
RegDst
RegWrite
Read reg 1
Read reg 2
Write reg
Write data
Register File
Read data 1
Read data 2
[15-0]
16
Sign extend
32
[5-0]
Shift left 2
Add
PCSrc
Branch
ALUSrc
ALU
Zero
ALU res
ALU Control
ALUOp
MemWrite
MemRead
Address
Read Data
Write Data
Data Cache
MemtoReg

Executing an Add Instruction

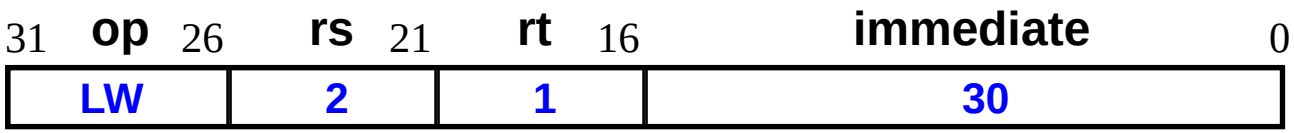
Add R2, R3, R5 ; R2 ← R3+R5

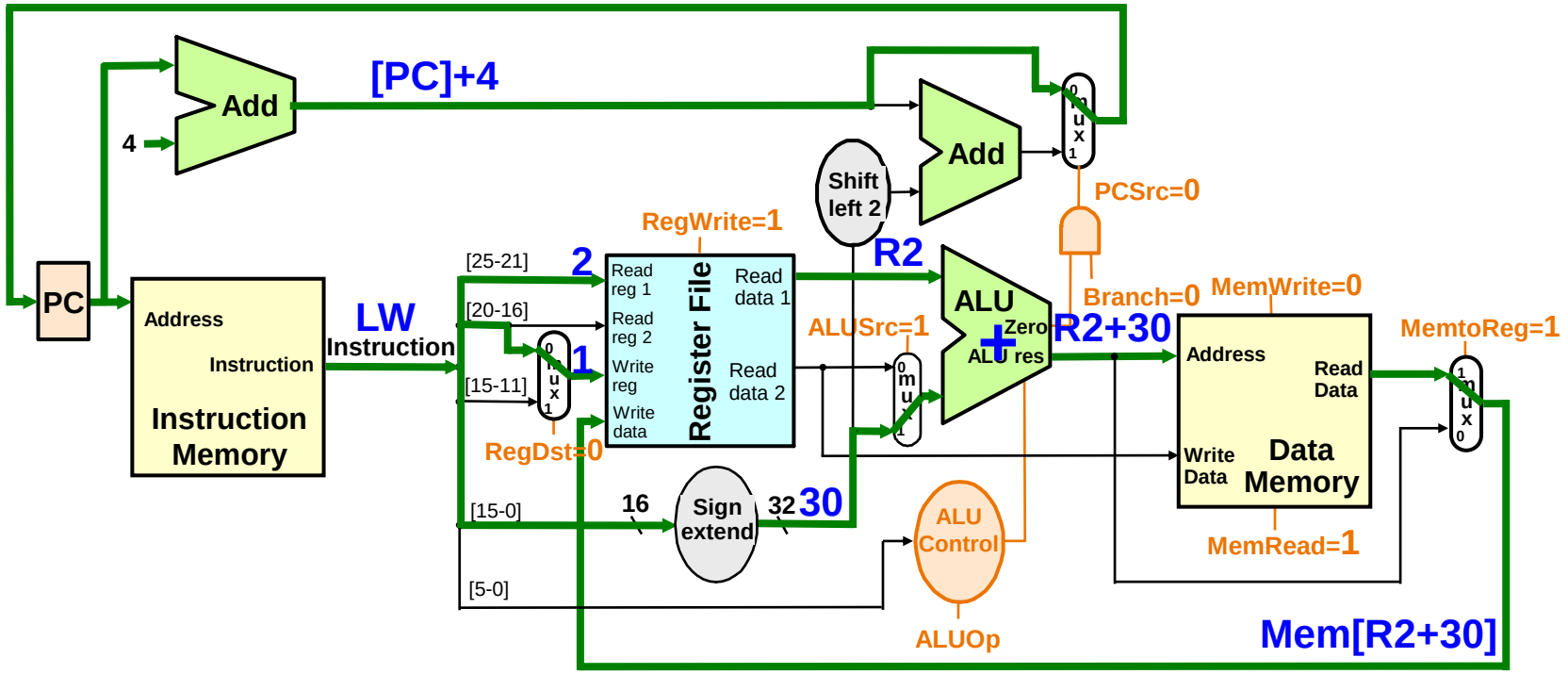
31 op 26	rs 21	rt 16	rd 11	shamt 6	funct 0
ALU	3	5	2	0	0 = Add



Executing a Load Instruction

LW R1, (30)R2 ; R1 ← Mem[R2+30]

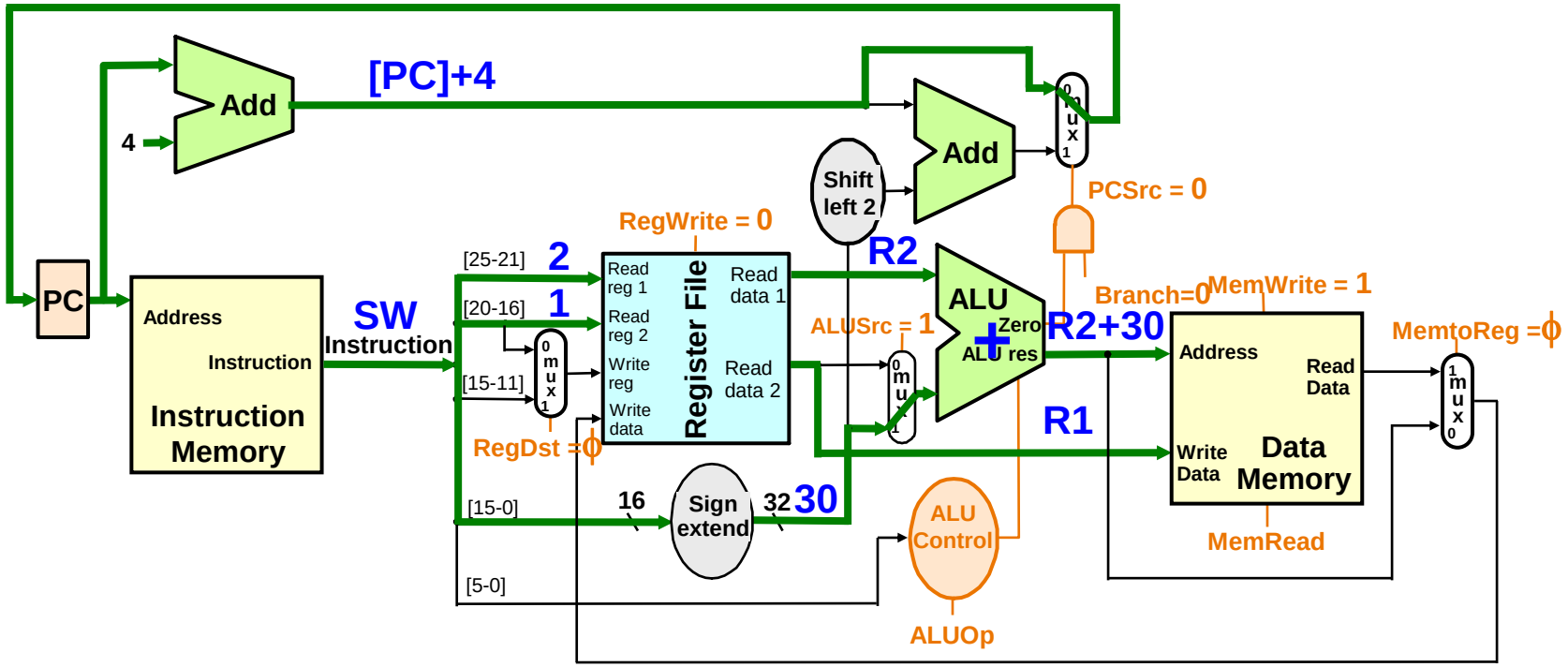




Executing a Store Instruction

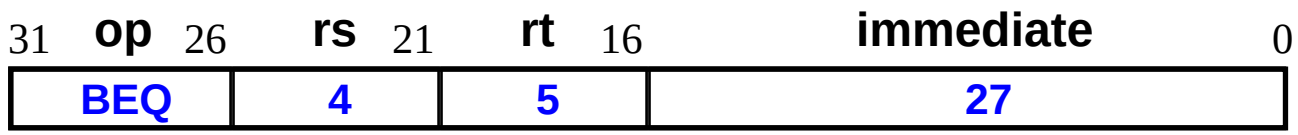
SW R1, (30)R2 ; Mem[R2+30] ← R1

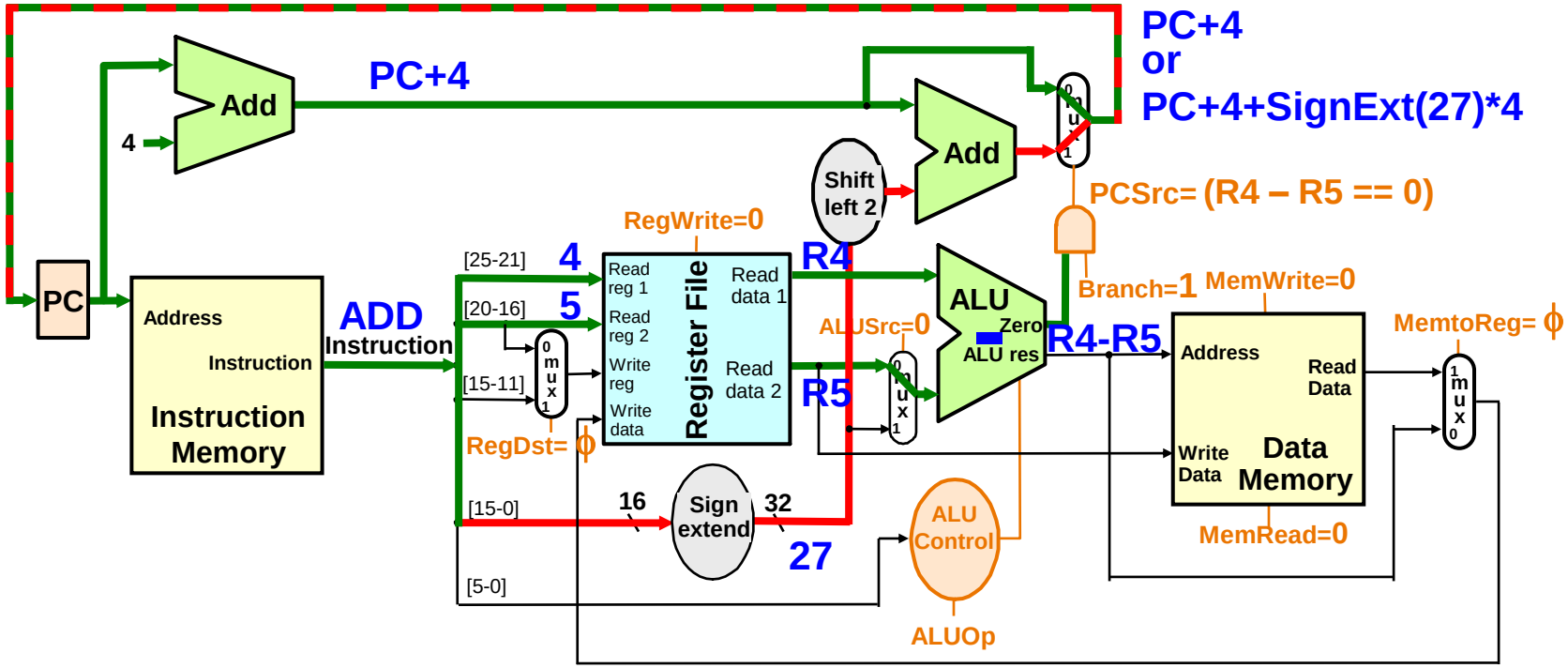
31 op 26	rs 21	rt 16	immediate 0
SW	2	1	30



Executing a BEQ Instruction

BEQ R4, R5, 27 ; if (R4-R5=0) then PC ← PC+4+SignExt(27)*4 ;
else PC ← PC+4



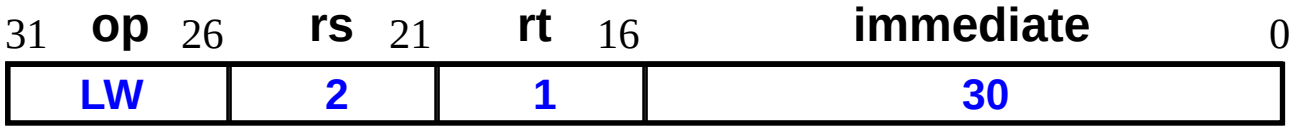


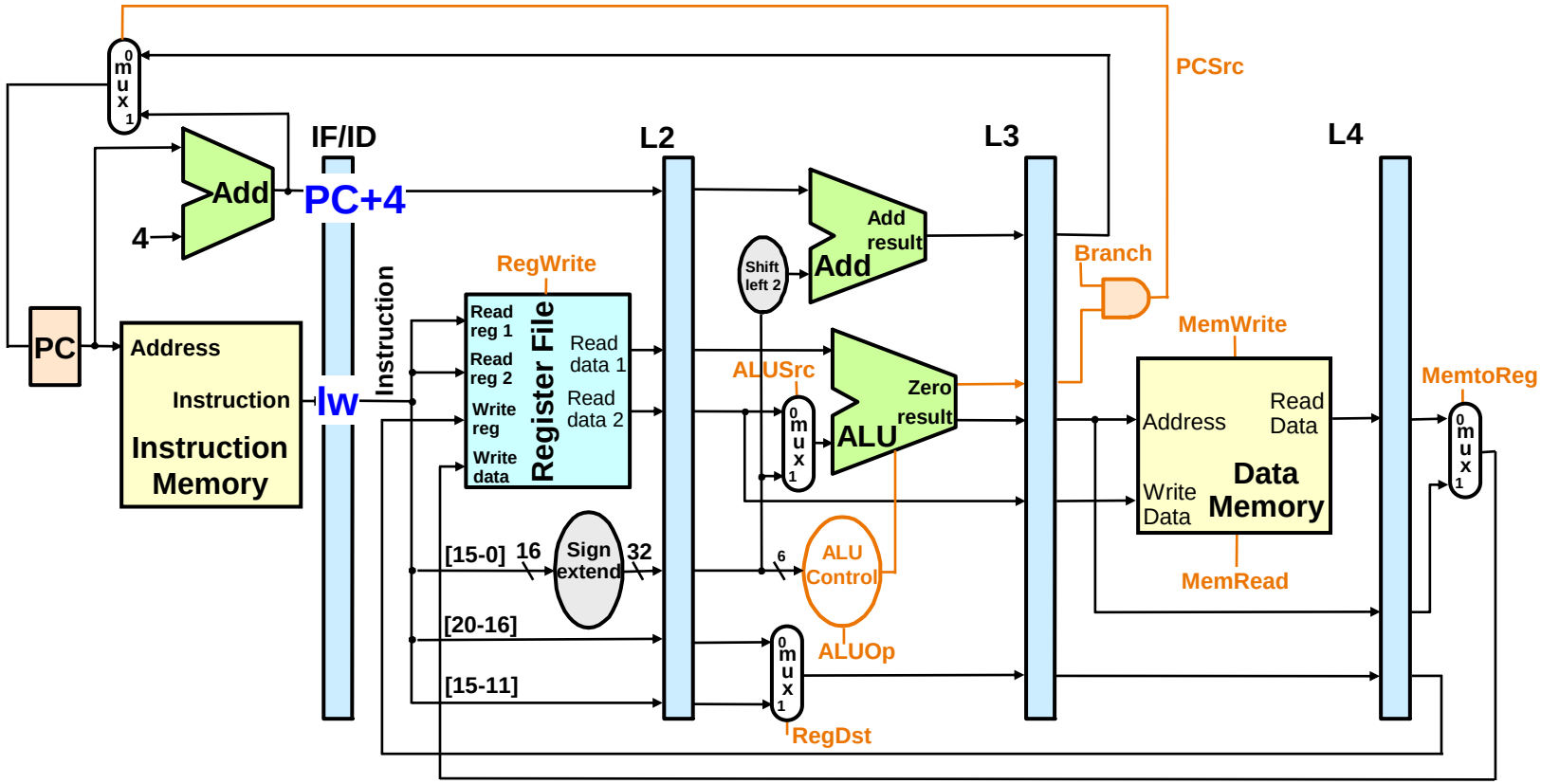
Control Signals

func	10 0000	10 0010	**Don't Care**				
op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	add	**sub**	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	1	0	0	x	x	x
ALUSrc	0	0	1	1	1	0	x
MemtoReg	0	0	0	1	x	x	x
RegWrite	1	1	1	1	0	0	0
MemWrite	0	0	0	0	1	0	0
Branch	0	0	0	0	0	1	x
Jump	0	0	0	0	0	0	1
ALUctr<2:0>	Add	Subtract	Or	Add	Add	Subtract	xxx

Pipelined CPU: Load (cycle 1 – Fetch)

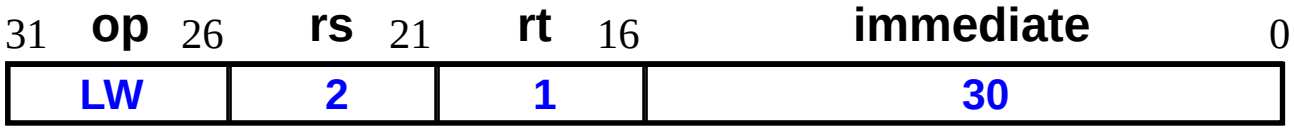
LW R1, (30)R2 ; R1 ← Mem[R2+30]

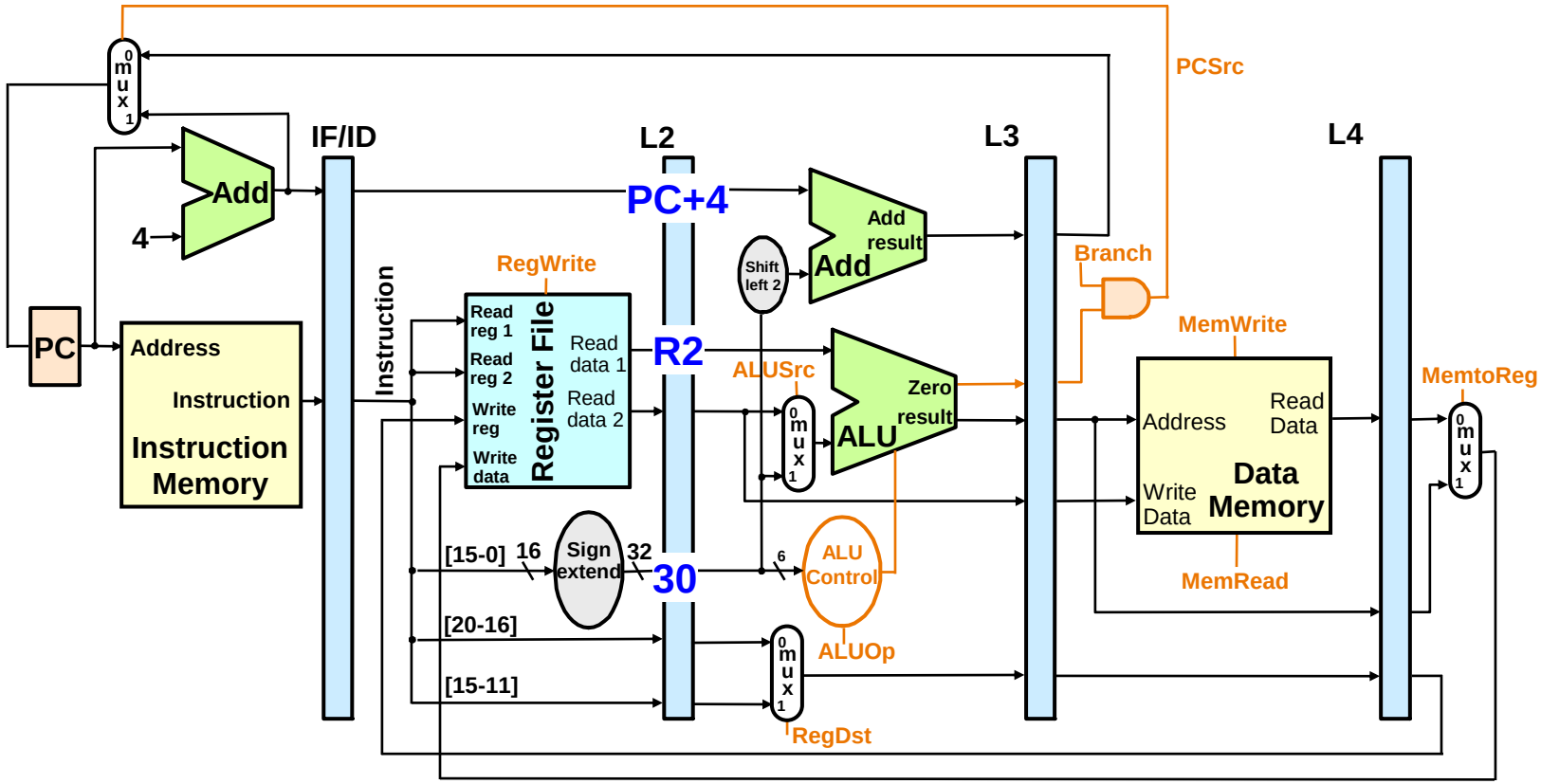




Pipelined CPU: Load (cycle 2 – Dec)

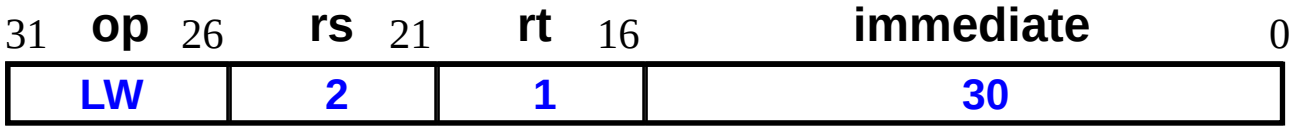
LW R1, (30)R2 ; R1 ← Mem[R2+30]

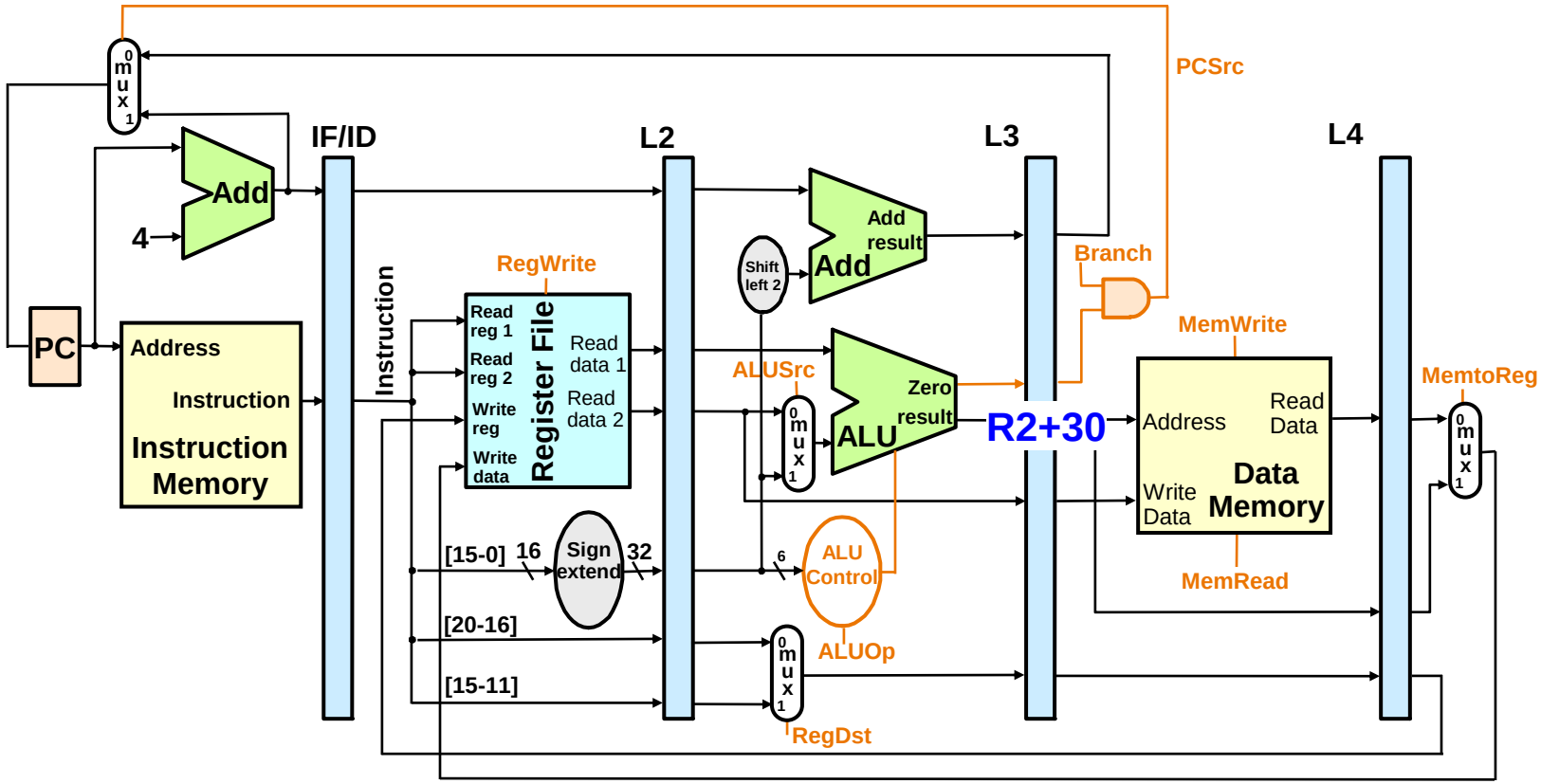




Pipelined CPU: Load (cycle 3 – Exe)

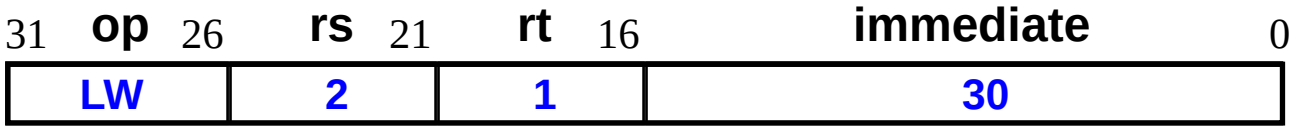
LW R1, (30)R2 ; R1 ← Mem[R2+30]

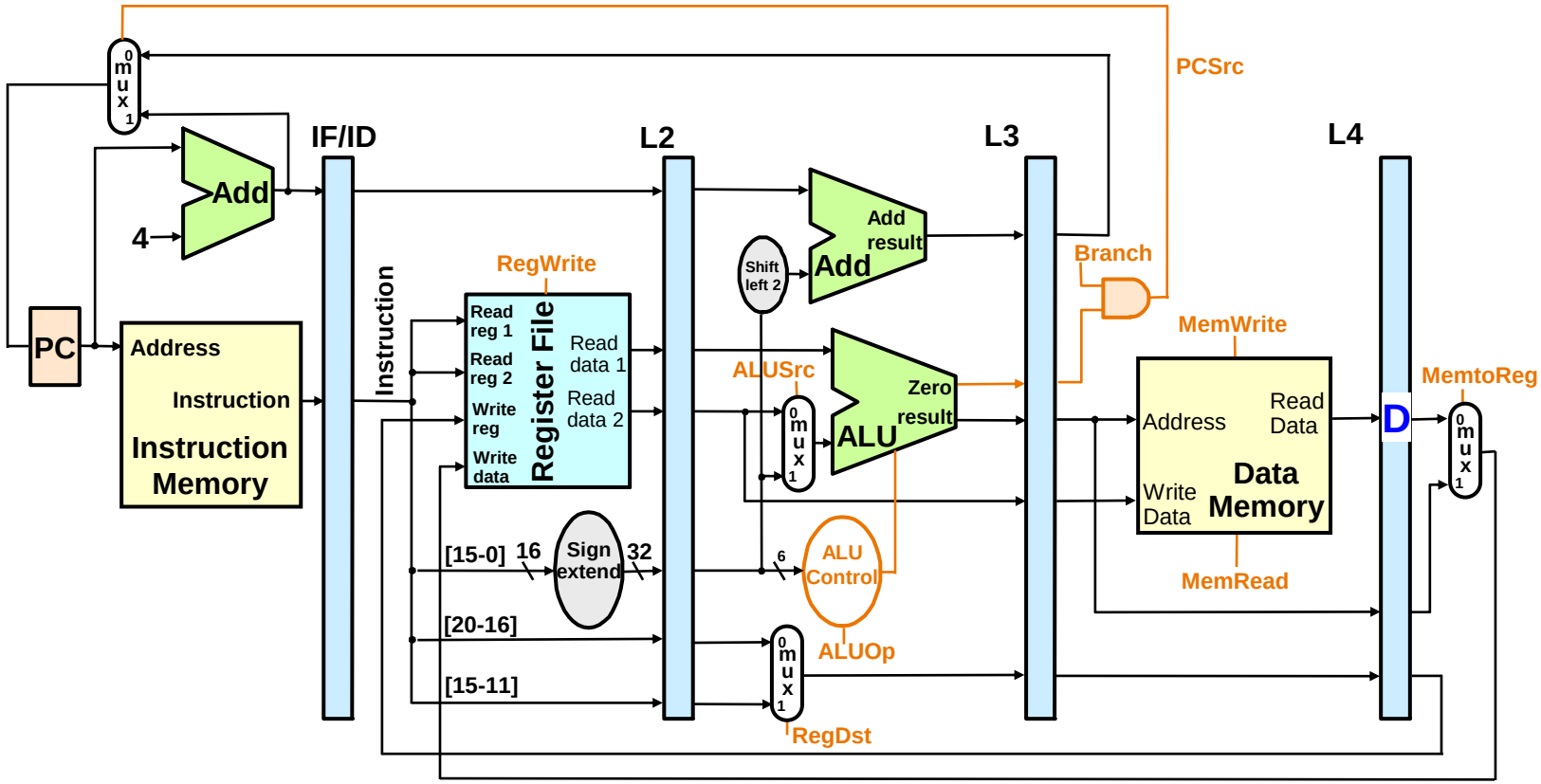




Pipelined CPU: Load (cycle 4 – Mem)

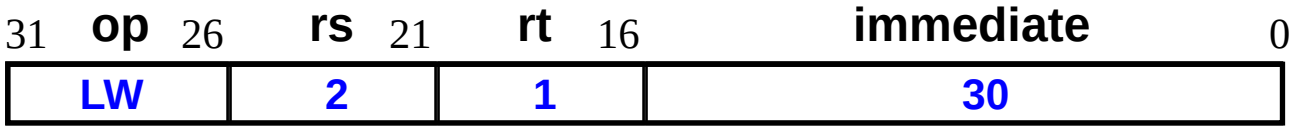
LW R1, (30)R2 ; R1 ← Mem[R2+30]

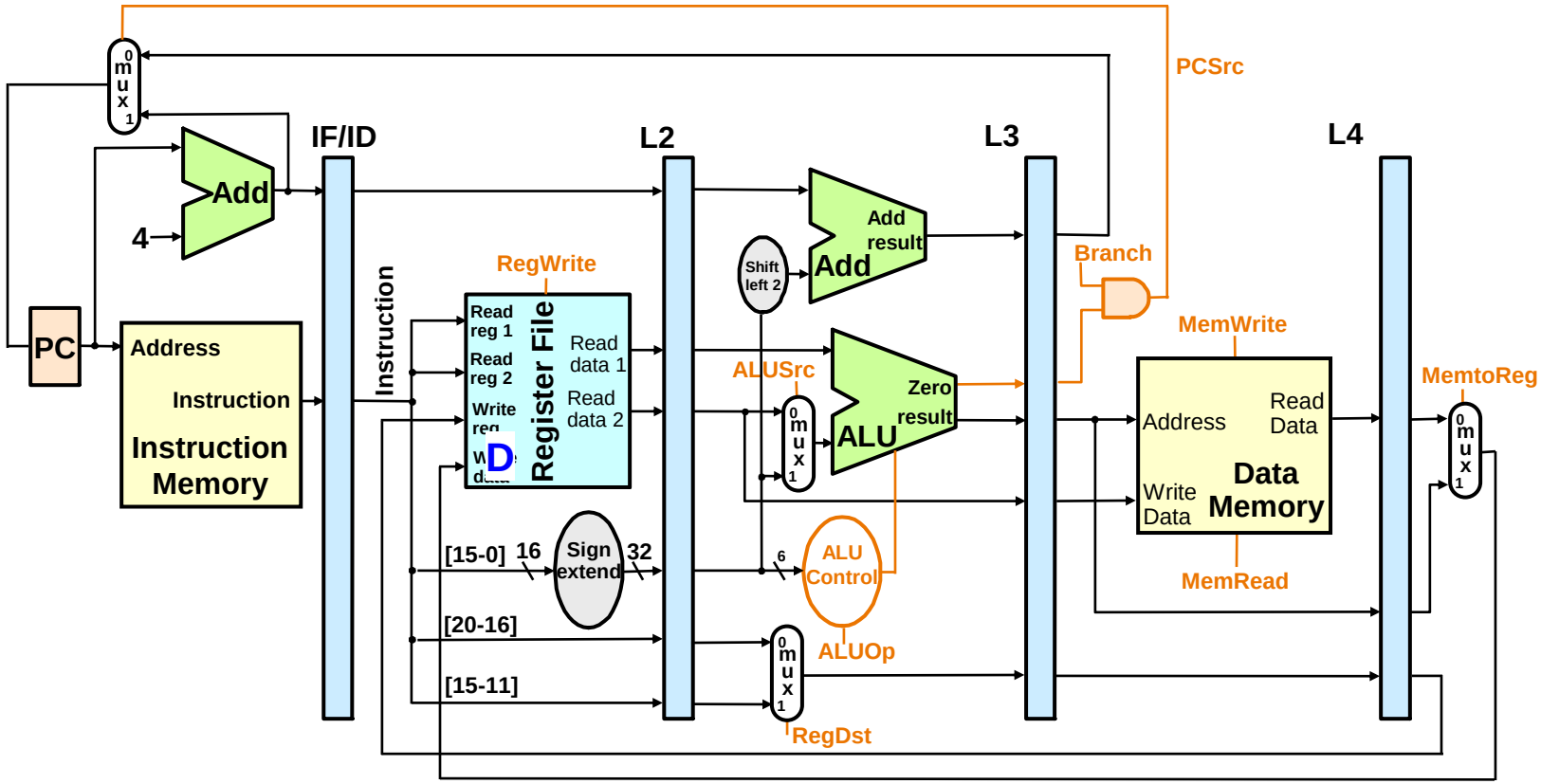




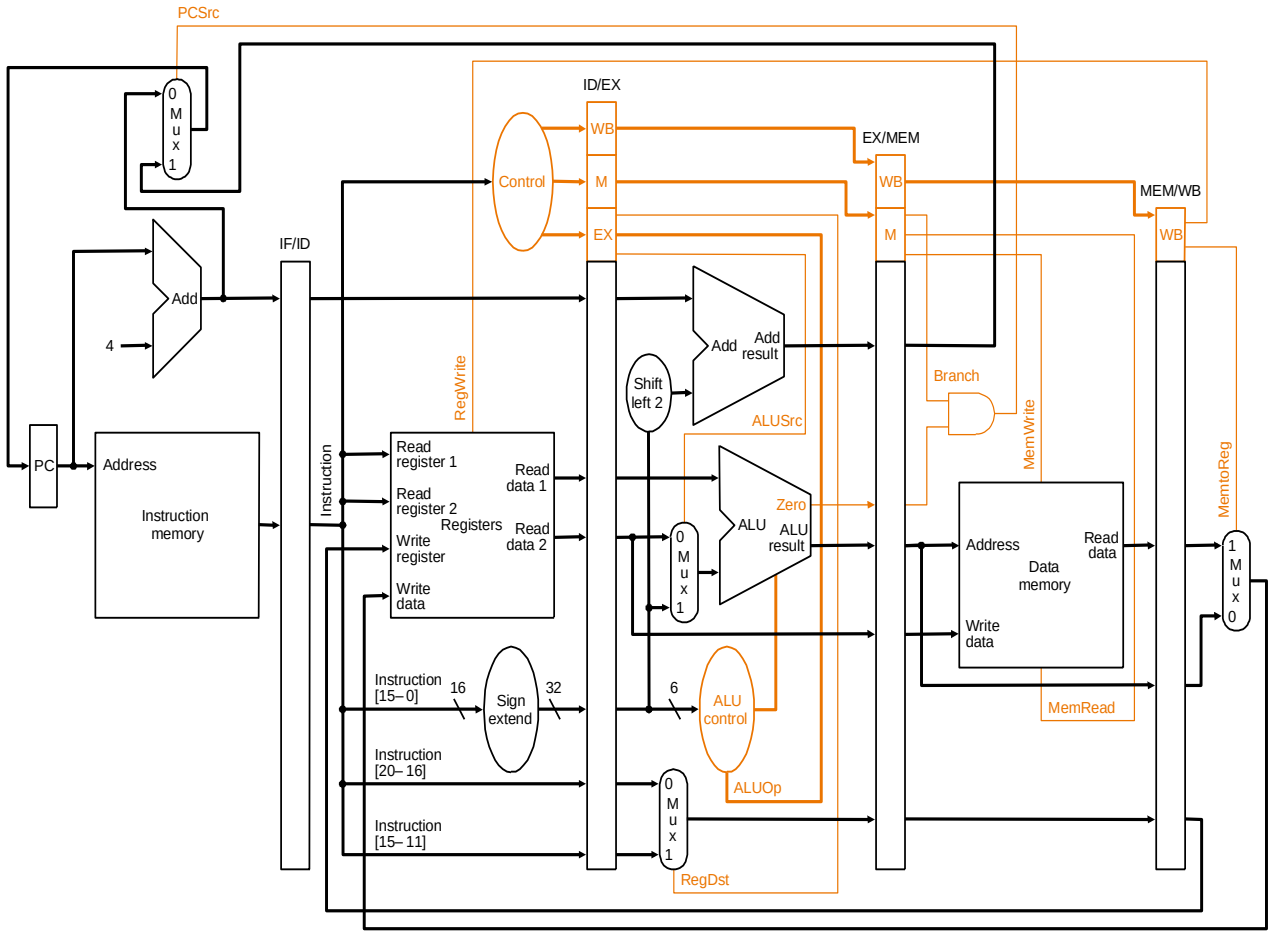
Pipelined CPU: Load (cycle 5 – WB)

LW R1, (30)R2 ; R1 ← Mem[R2+30]





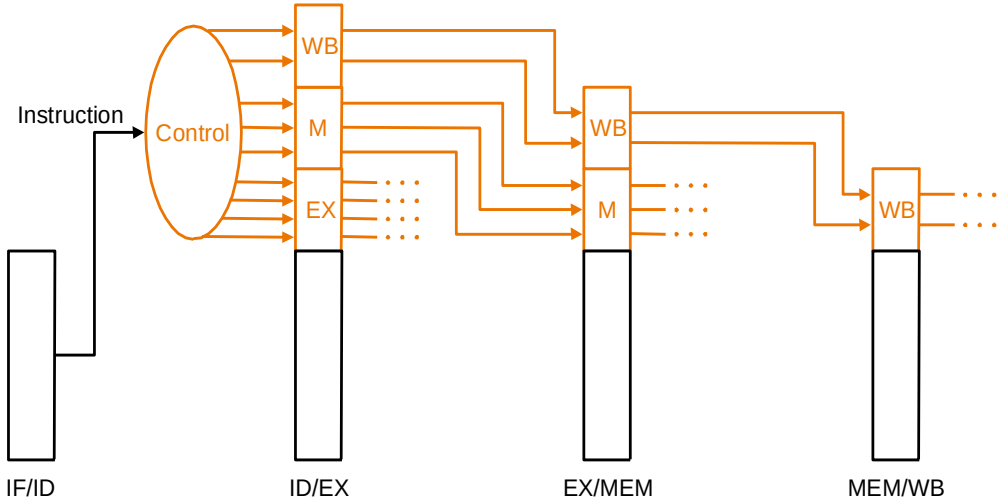
Datapath with Control



Multi-Cycle Control

- **Pass control signals along just like the data**

Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
`lw`	0	0	0	1	0	1	0	1	1
`sw`	X	0	0	1	0	0	1	0	X
`beq`	X	0	1	0	1	0	0	0	X



Five Execution Steps

- **Instruction Fetch**
  - Use PC to get instruction and put it in the Instruction Register.
  - Increment the PC by 4 and put the result back in the PC.

    IR = Memory[PC];
    PC = PC + 4;
- **Instruction Decode and Register Fetch**
  - Read registers rs and rt
  - Compute the branch address

```
A = Reg[IR[25-21]];
B = Reg[IR[20-16]];
ALUOut = PC + (sign-extend(IR[15-0]) << 2);
```

  - We aren't setting any control lines based on the instruction type
    (we are busy "decoding" it in our control logic)

Five Execution Steps (cont.)

- Execution

  ALU is performing one of three functions, based on instruction type:

  - **Memory Reference: effective address calculation.**

    ```
    ALUOut = A + sign-extend(IR[15-0]);
    ```

  - **R-type:**

    ```
    ALUOut = A op B;
    ```

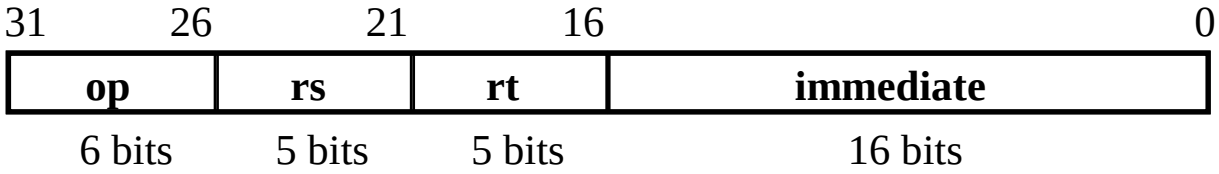
  - **Branch:**

    ```
    if (A==B) PC = ALUOut;
    ```

- Memory Access or R-type instruction completion

- Write-back step

The Store Instruction



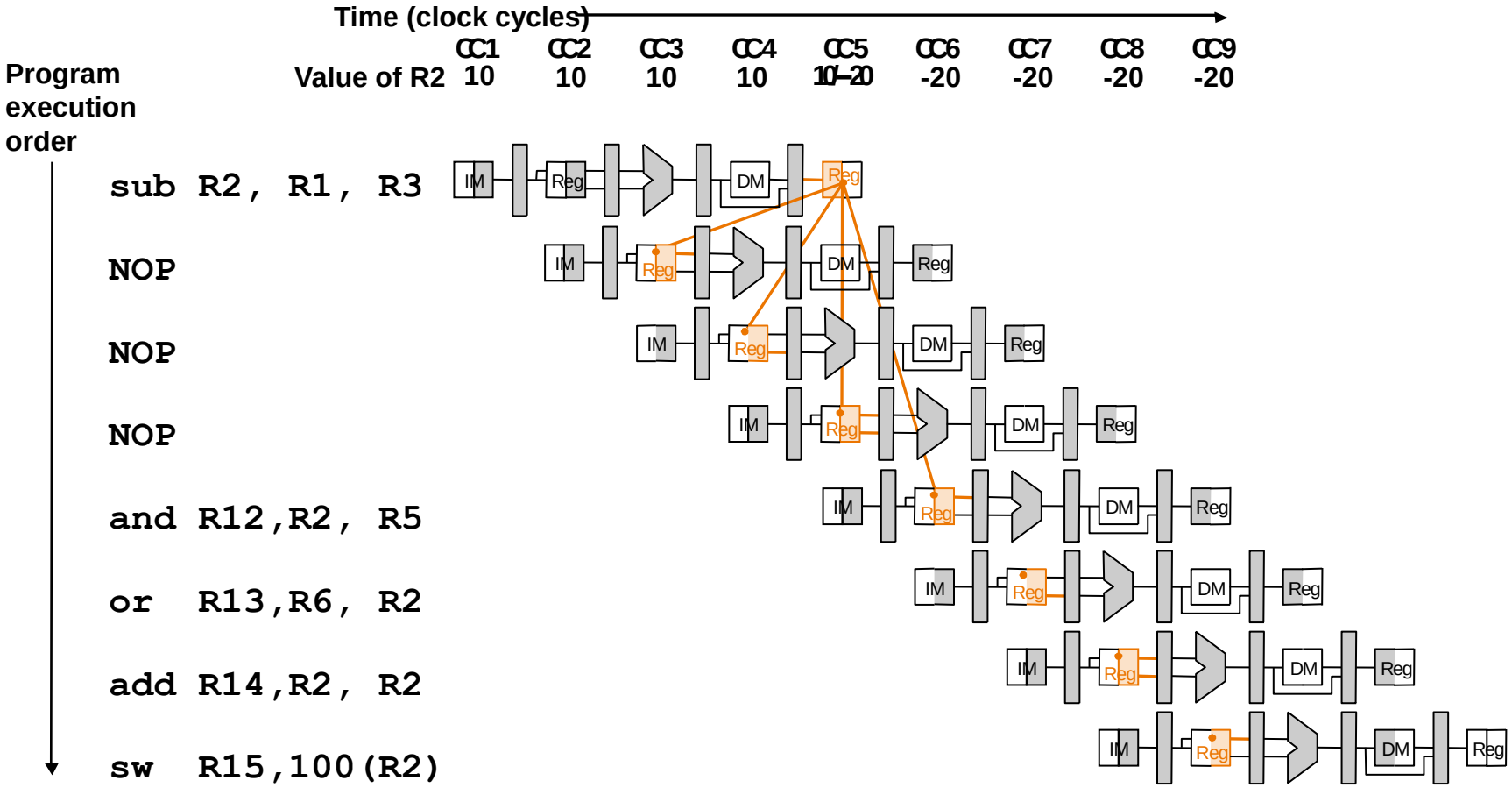
- **sw rt, rs, imm16**

  - mem[PC] Fetch the instruction from memory
  - Addr ← R[rs] + SignExt(imm16) Calculate the memory address
  - Mem[Addr] ← R[rt] Store the register into memory
  - PC ← PC + 4 Calculate the next instruction's address

Mem[Rs + SignExt[imm16]] ← Rt Example: sw rt, rs, imm16

RAW Hazard: SW Solution

- Have compiler avoid hazards by adding NOP instructions



- Problem: this really slows us down!

Delayed Branch

- **Define branch to take place AFTER *n* following instruction**
  - HW executes *n* instructions following the branch regardless of branch is taken or not
- **SW puts in the *n* slots following the branch instructions that need to be executed regardless of branch resolution**
  - Instructions that are before the branch instruction, or
  - Instructions from the converged path after the branch
- **If cannot find independent instructions, put NOP**

Original Code

```
r3 = 23
R4 = R3+R5
If (r1==r2) goto x
R1 = R4 + R5
X: R7 = R1
```

⇒

New Code

```
If (r1==r2) goto x
r3 = 23
R4 = R3 +R5
NOP
R1 = R4 + R5
X: R7 = R1
```

Delayed Branch Performance

- **Filling 1 delay slot is easy, 2 is hard, 3 is harder**
- **Assuming we can effectively fill *d*% of the delayed slots**

$$CPI_{new} = 1 + 0.2 \times (3 \times (1-d))$$

- **For example, for d=0.5, we get $CPI_{new} = 1.3$**
- **Mixing architecture with micro-arch**
  - New generations requires more delay slots
  - Cause computability issues between generations