Embedded Systems
Research Group

Computer Architectures

Developing a Pipeline processor

Tiago Gomes
(ESRG)
R&D Centre ALGORITMI
Department of Industrial Electronics
School of Engineering,
University of Minho, Guimarães - PORTUGAL

Features

- 5-Stage Pipeline
- 3 addresses Load-Store Architecture
- Data size:
  - 8 bits
- Instructions size:
  - 16 bits

Features

- Register File
  - 8 registers x 8 bits
- Instruction Memory
  - 256 x 16 bits
- Data Memory
  - 256 x 8 bits
- 1 x Input and 1 x Output port: 8 bits

Summary

- Microprocessor ISA definition
- Micro-Architecture
- Hazards
- Results

- Instructions:
  - Arithmetic and logical
    - ADD, SUB, AND, OR, XOR

      Rd = Ra op Rb
    - NOT

      Rd = ~Ra

- Data transfer instructions
  - Immediate Load

    Rd = value
  - Load

    Rd = mem[addr]
  - Store

    mem[addr] = value
  - Input / Output

    Rd = Input

    Output = Ra

- Control instructions
  - JMP

    PC = Ra
  - BRZ

    If (Rb ==0) PC = Ra
  - BRNZ

    If (Rb !=0) PC = Ra

- Miscellaneous
  - NOP

Summary

- Microprocessor ISA definition
- Micro-Architecture
- Hazards
- Results

- First step: instruction encoding
  - Our ISA specified 13 instructions
    - How many bits for the opcode?
    - How many bits to address the Register File?
      - 8 Registers
    - How can an ALU instruction be encoded?

      ADD Rd, Ra, Rb

- # ADD instruction

				opcode	Rd	Ra	Rb				
add	Rd,	Ra,	Rb	1 1 0 0	b b b	b b b	b b b	x	x	x	Rd = Ra + Rb

- # Other ALU instructions

				opcode	Rd	Ra	Rb				
add	Rd,	Ra,	Rb	1 1 0 0	b b b	b b b	b b b	x	x	x	Rd = Ra + Rb
sub	Rd,	Ra,	Rb	0 0 0 1	b b b	b b b	b b b	x	x	x	Rd = Ra - Rb
and	Rd,	Ra,	Rb	0 0 1 0	b b b	b b b	b b b	x	x	x	Rd = Ra & Rb
or	Rd,	Ra,	Rb	0 0 1 1	b b b	b b b	b b b	x	x	x	Rd = Ra | Rb
xor	Rd,	Ra,	Rb	0 1 0 0	b b b	b b b	b b b	x	x	x	Rd = Ra ^ Rb
not	Rd,	Ra		0 1 0 1	b b b	b b b	x x x	x x x			Rd = ~Ra

- Load and Store

loadi Rd, imm

opcode	Rd		immediate
0 1 1 0	b b b	x	b b b b b b b b

Rd = imm

load Rd, @Rb

opcode	Rd		Rb		i
0 1 1 1	b b b	x x x	b b b	x x	0

Rd = mem[Rb]

store Ra, @Rb

opcode		Ra	Rb		i
1 0 0 0	x x x	b b b	b b b	x x	0

mem[Rb] = Ra

load Rd, imm

opcode	Rd	immediate	i
0 1 1 1	b b b	b b b b b b b b	1

Rd = mem[imm]

store Ra, imm

opcode	Imm [7:5]	Ra	imm[4:0]	i
1 0 0 0	b b b	b b b	b b b b b	1

mem[imm] = Ra

- Other instructions

		opcode	Rd	Ra		i	
input	Rd	1 1 0 1	b b b	x x x	x x x x x	0	Rd = input
output	Ra	1 1 0 1	x x x	b b b	x x x x x	1	output = Ra

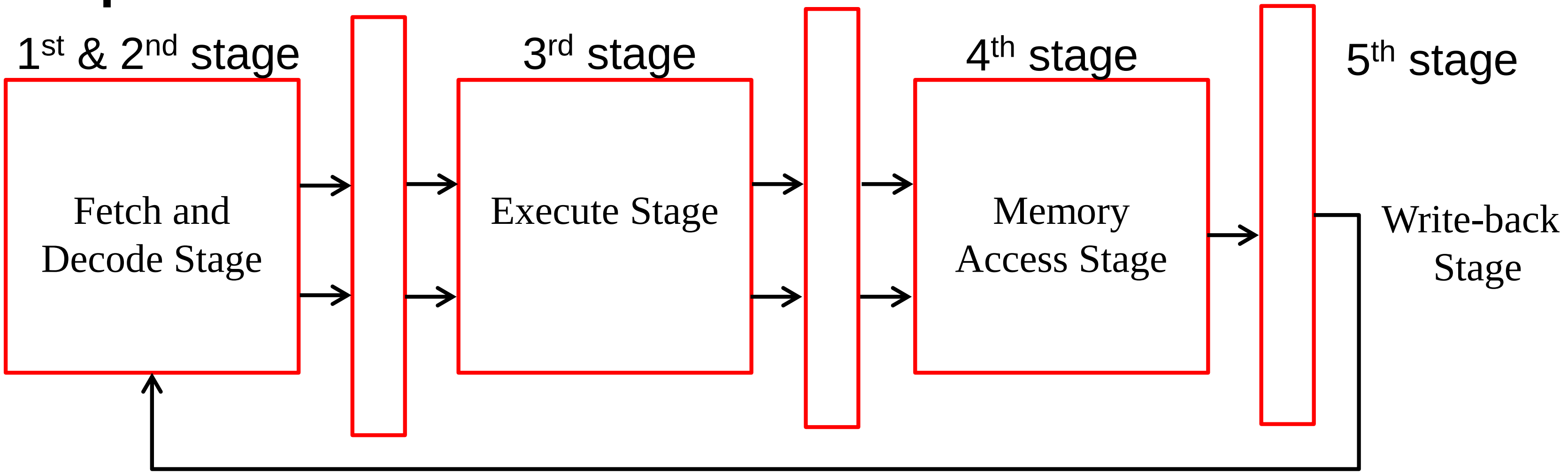
		opcode	Rd	Ra		
jmp	@Ra	1 0 0 1	x x x	b b b	x x x x x x	PC = Ra

		opcode	Rd	Ra	Rb		
brz	Rb, @Ra	1 0 1 0	x x x	b b b	b b b	x x x	if(Rb==0) PC = Ra
brnz	Rb, @Ra	1 0 1 1	x x x	b b b	b b b	x x x	if(Rb≠0) PC = Ra

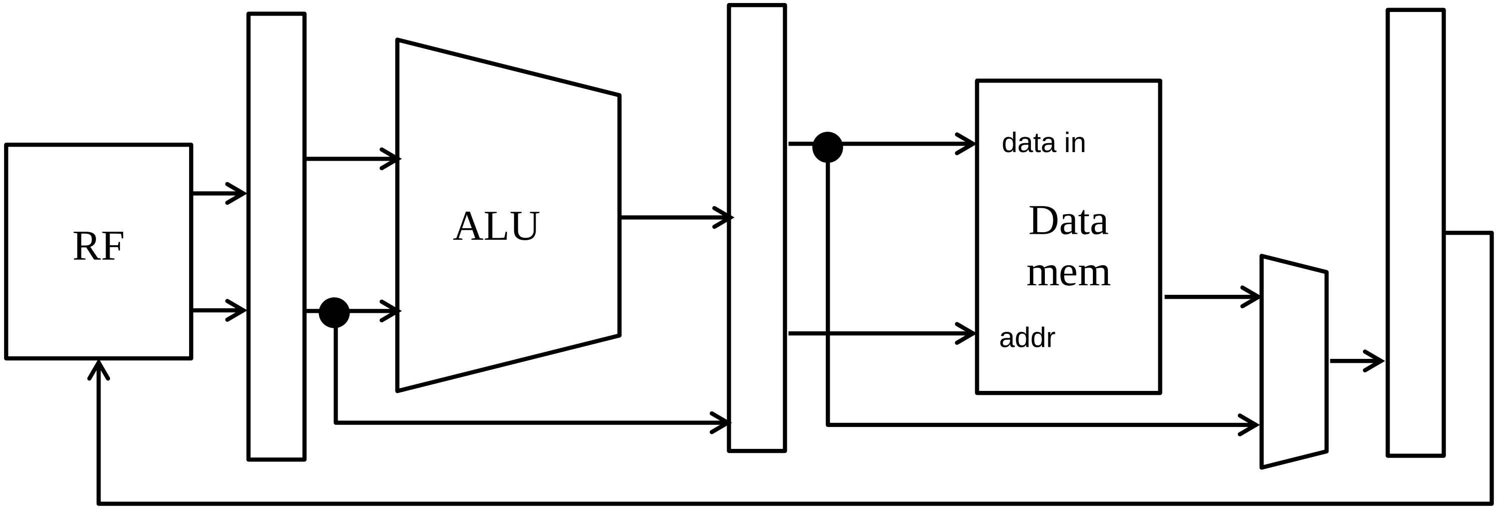
	opcode	
nop	0 0 0 0	x x x x x x x x x x x x

- Is it possible to divide the instructions in several tasks/stages?
  - Fetch & Decode Stage
    - Read instruction from memory and decode it
  - Execute Stage
    - Perform the calculations (ALU)
  - Memory Access Stage
    - Read or Write to Data Memory
  - Write-Back
    - Write back the result to the Register File

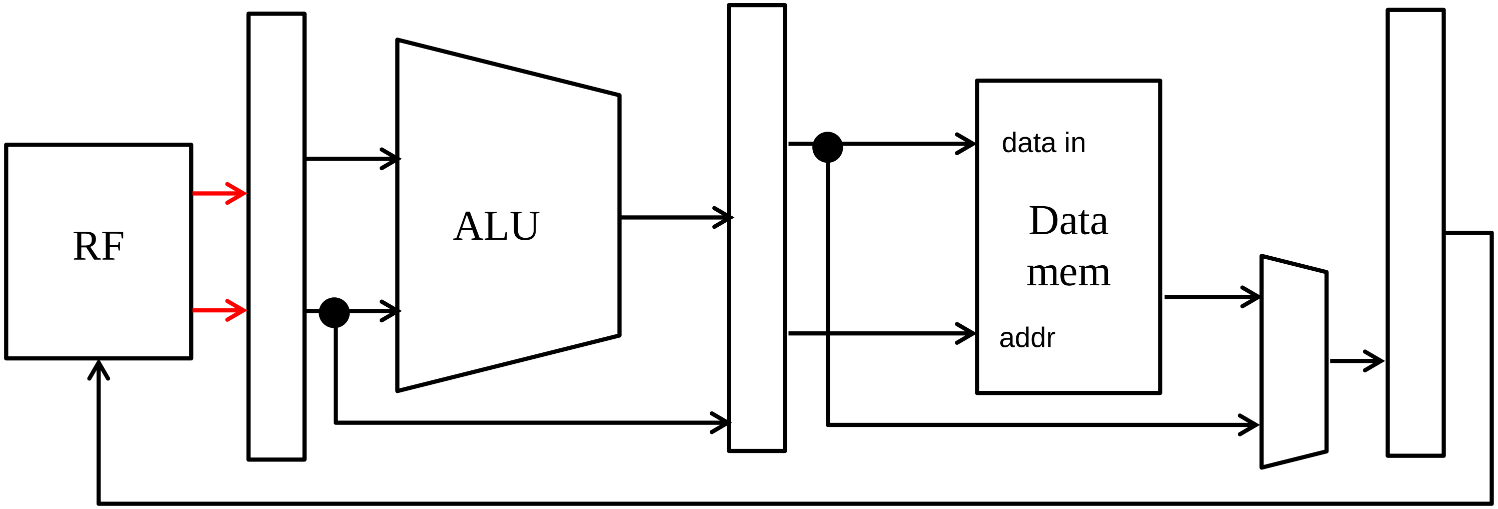
- Pipeline


1st & 2nd stage
3rd stage
4th stage
5th stage
Fetch and Decode Stage
Execute Stage
Memory Access Stage
Write-back Stage

- ALU instruction by stages



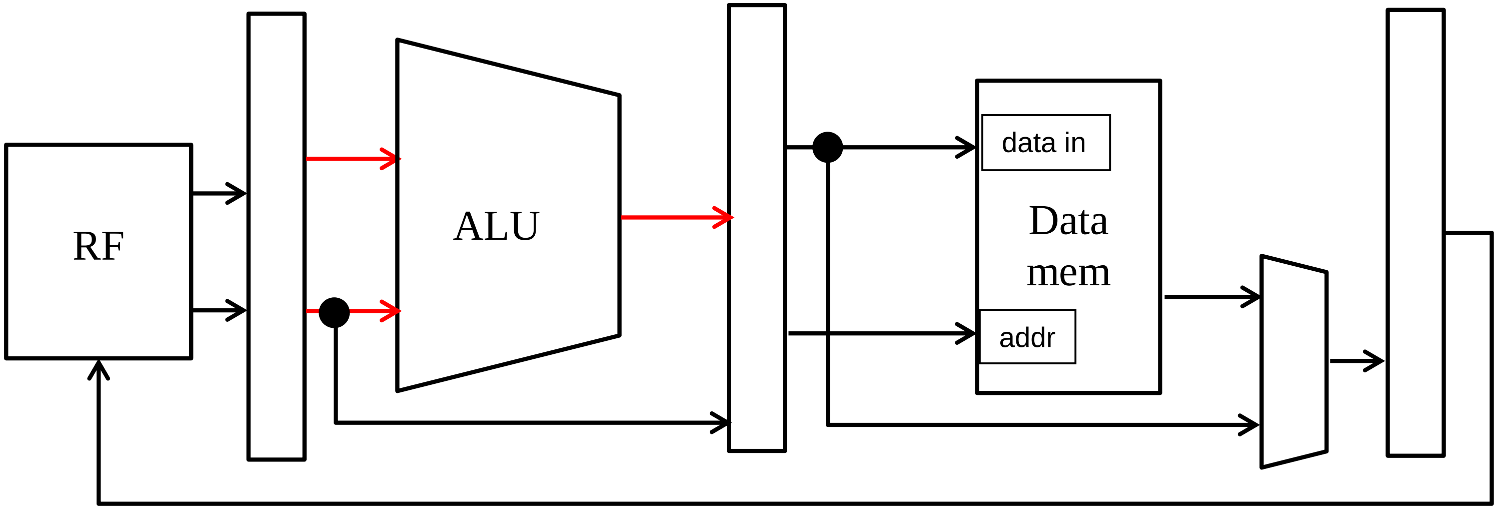
- Add Rd, Ra, Rb
  - 1st & 2nd stages:
    - Fetch instruction and Read Operands



add Rd, Ra, Rb

opcode	Rd	Ra	Rb			
1 1 0 0	b b b	b b b	b b b	x	x	x

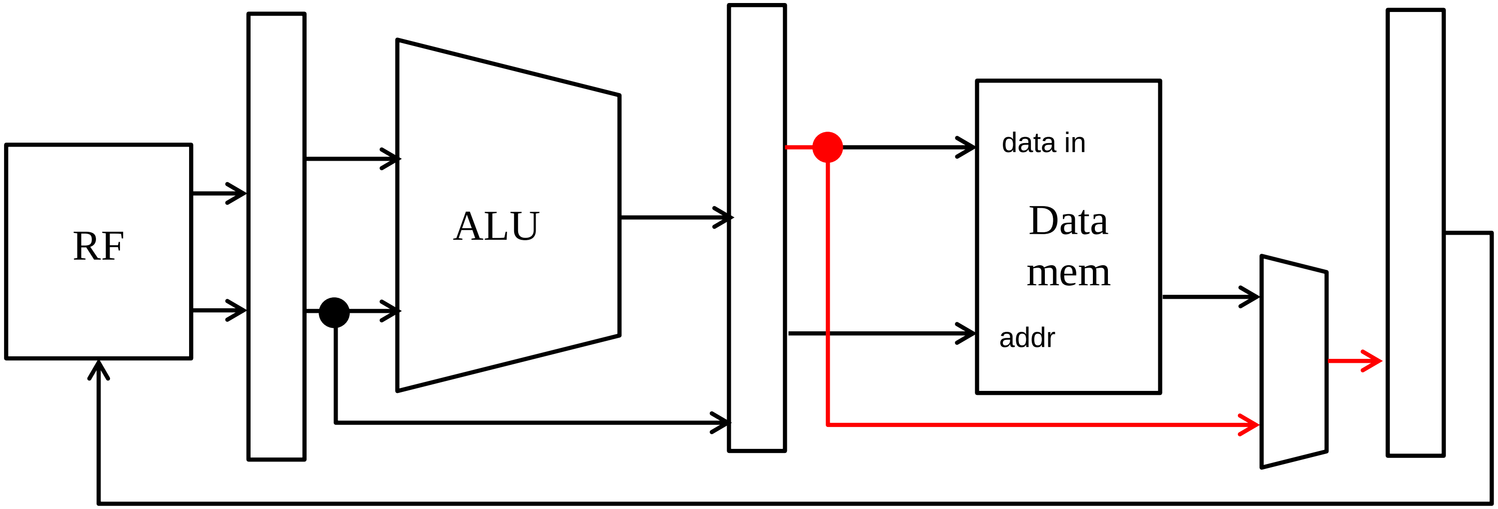
- Add Rd, Ra, Rb
  - 3rd stage:
    - Calculation



add Rd, Ra, Rb

opcode	Rd	Ra	Rb			
1 1 0 0	b b b	b b b	b b b	x	x	x

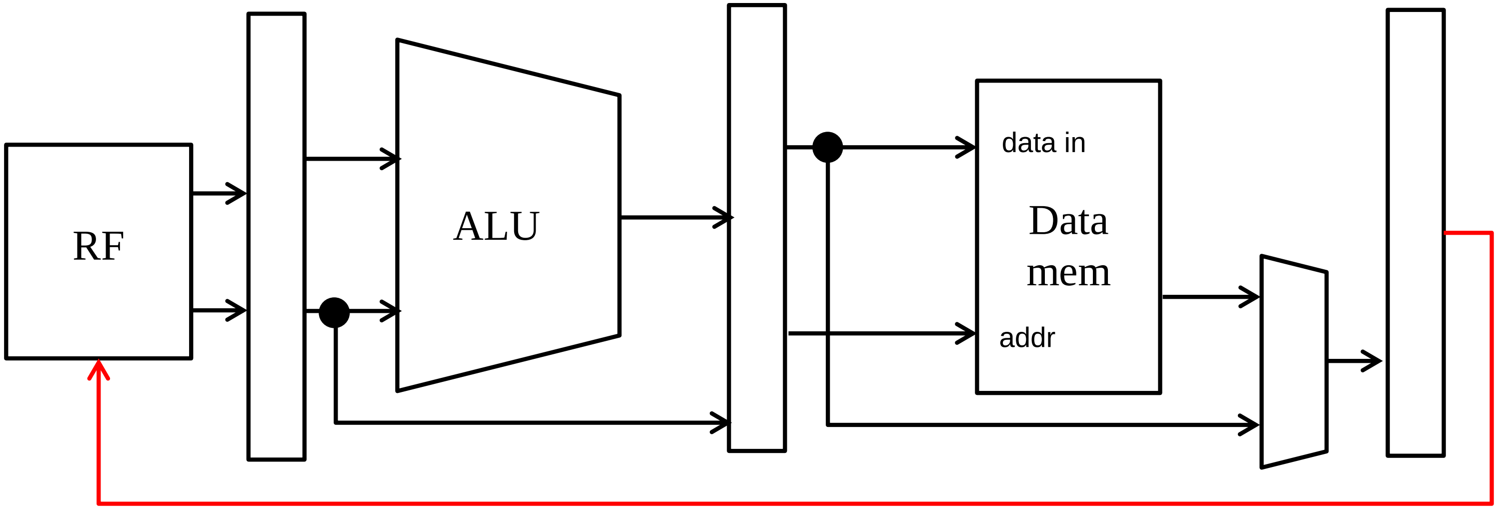
- Add Rd, Ra, Rb
  - 4th stage:
    - Do nothing



add Rd, Ra, Rb

opcode	Rd	Ra	Rb			
1 1 0 0	b b b	b b b	b b b	x	x	x

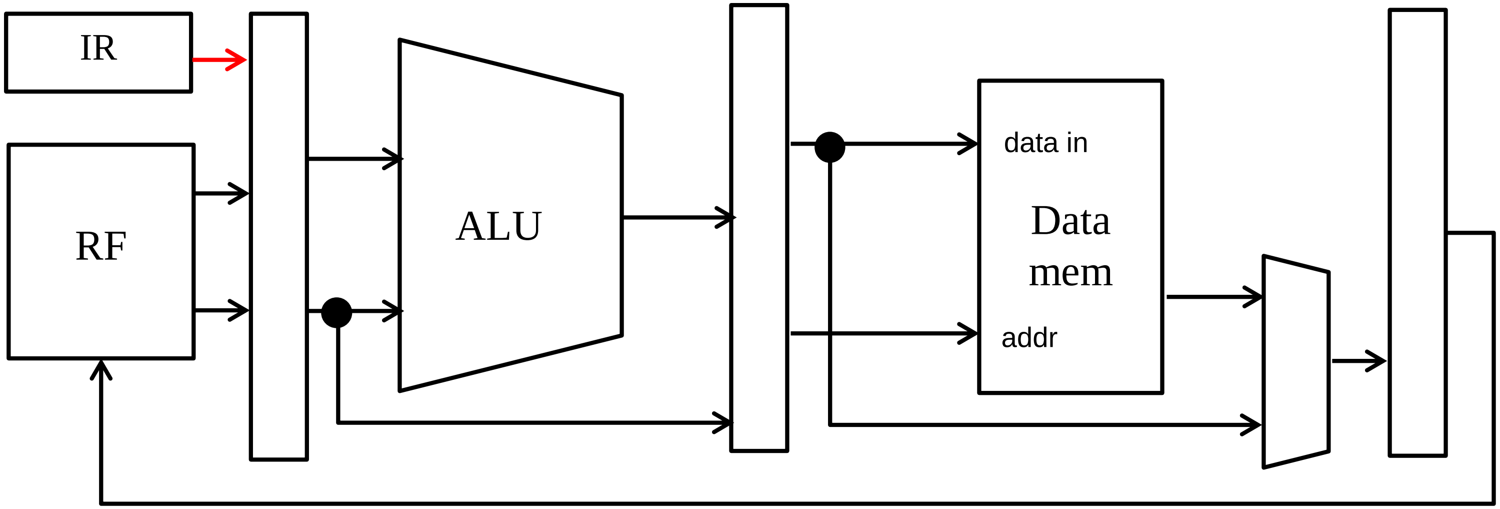
- Add Rd, Ra, Rb
  - 5th stage:
    - Write-back the result



add Rd, Ra, Rb

opcode	Rd	Ra	Rb			
1 1 0 0	b b b	b b b	b b b	x	x	x

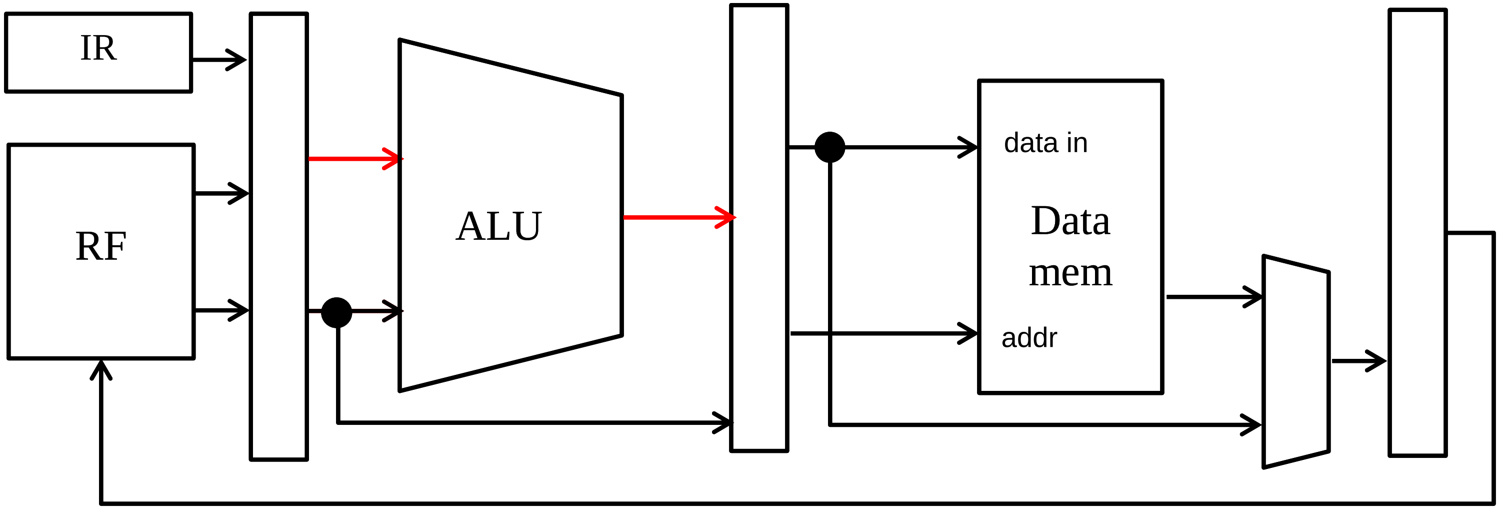
- Loadi Rd, value
  - 1st & 2nd stages:
    - Fetch instruction and Read Operands



loadi Rd, imm

opcode				Rd				immediate							
0	1	1	0	b	b	b	x	b	b	b	b	b	b	b	b

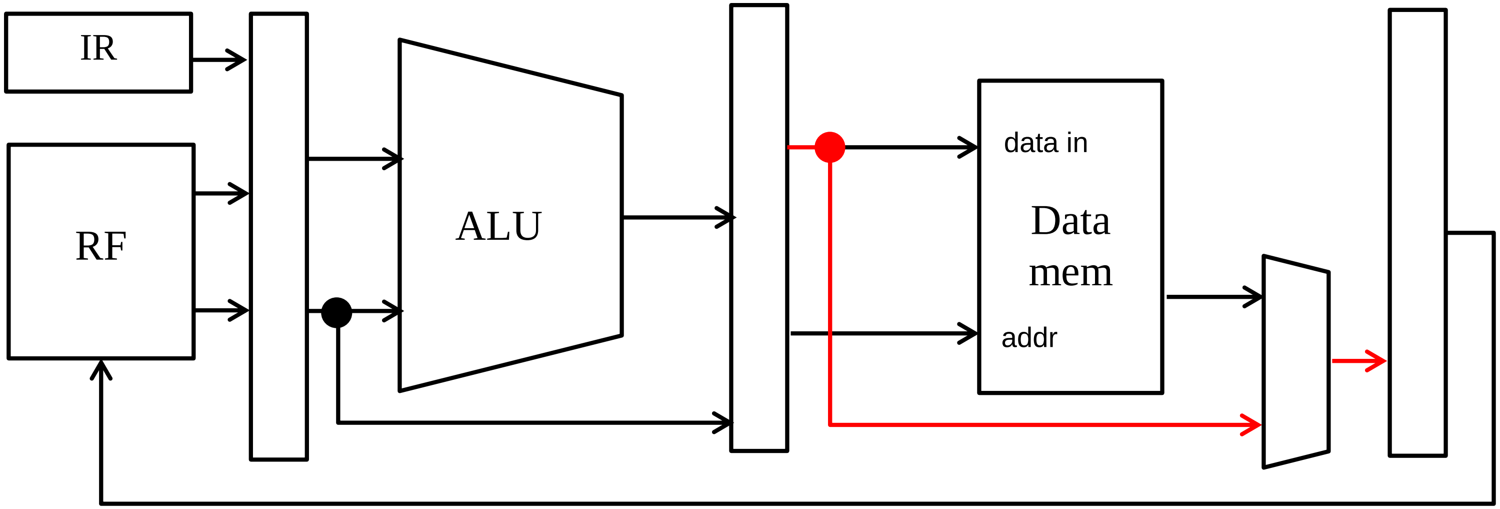
- Loadi Rd, value
  - 3rd stage:



loadi Rd, imm

opcode	Rd		immediate
0 1 1 0	b b b	x	b b b b b b b b

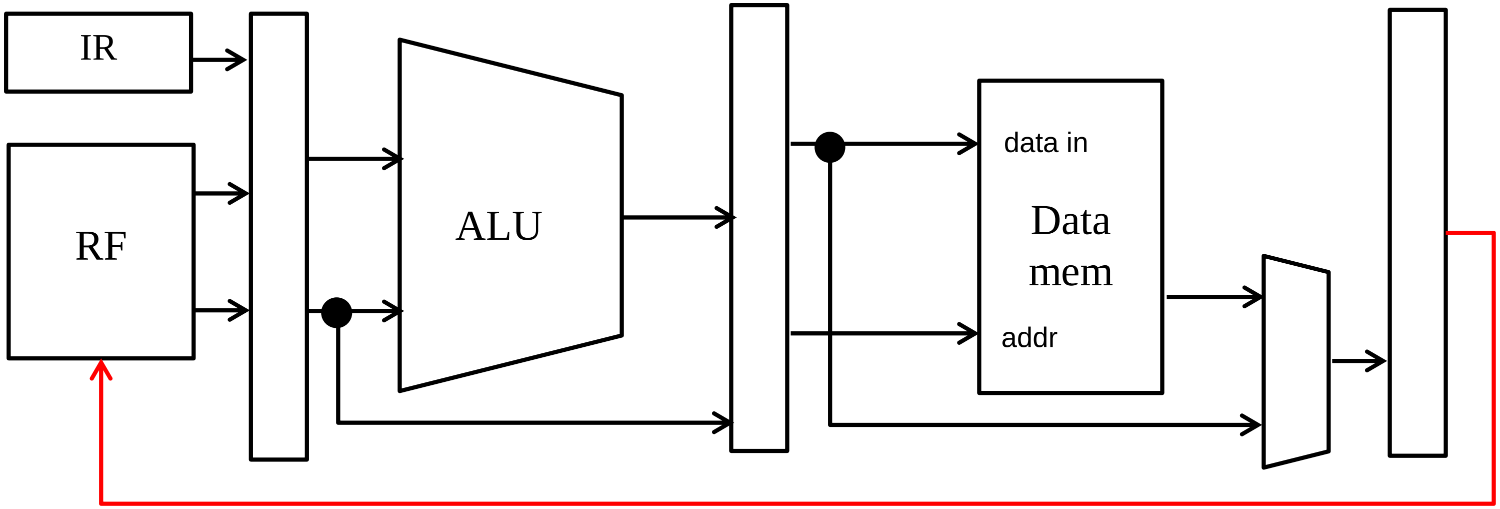
- Loadi Rd, value
    - 4th stage:



loadi Rd, imm

opcode	Rd		immediate
0 1 1 0	b b b	x	b b b b b b b b

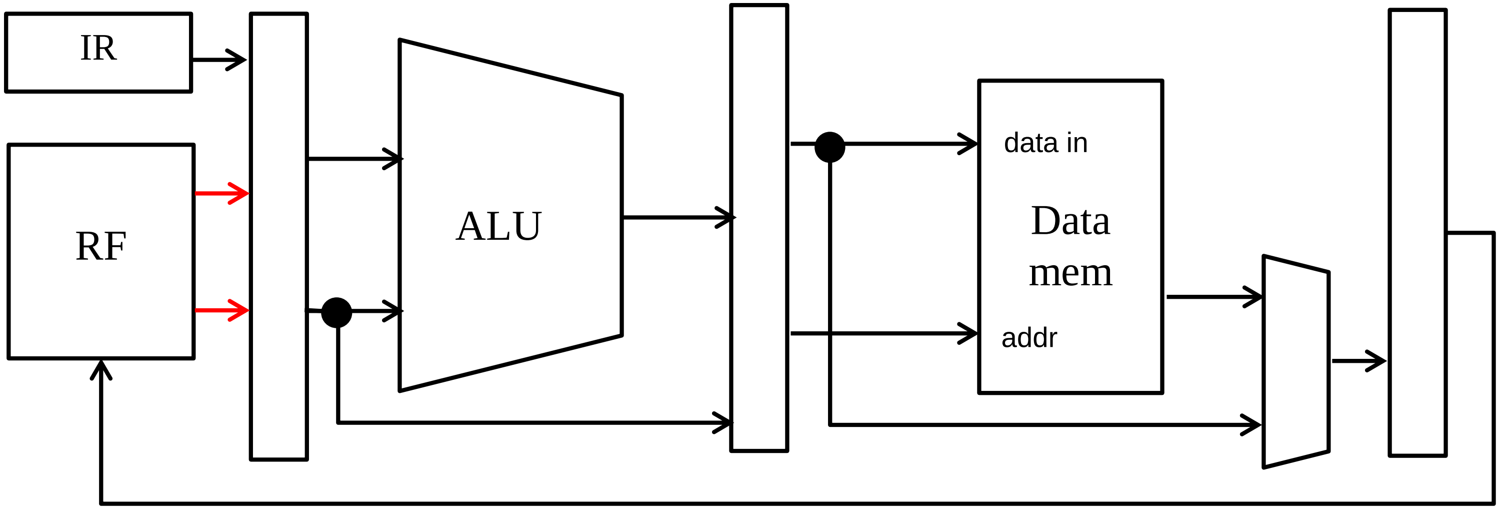
- Loadi Rd, value
  - 5th stage:
    - Write-back the result



loadi Rd, imm

opcode	Rd		immediate
0 1 1 0	b b b	x	b b b b b b b b

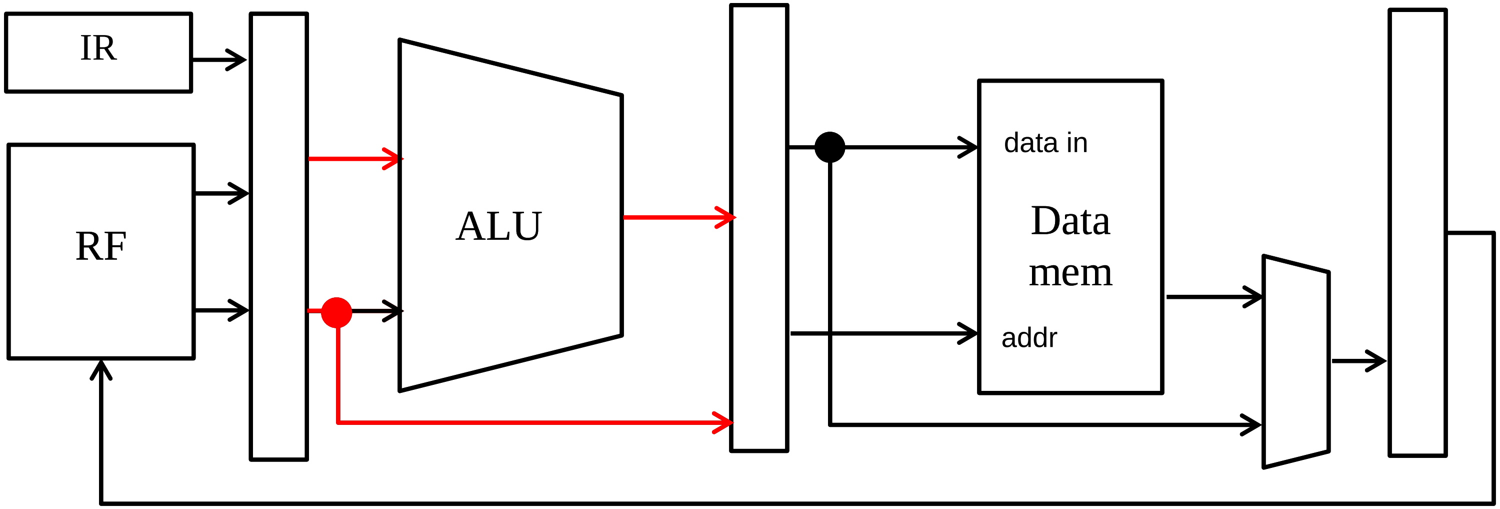
- Store Ra, @Rb (mem[Rb] = Ra)
  - 1st & 2nd stages:
    - Fetch instruction and Read Operands



store Ra, @Rb

opcode								Ra			Rb					i
1	0	0	0	x	x	x	b	b	b	b	b	b	x	x	0	

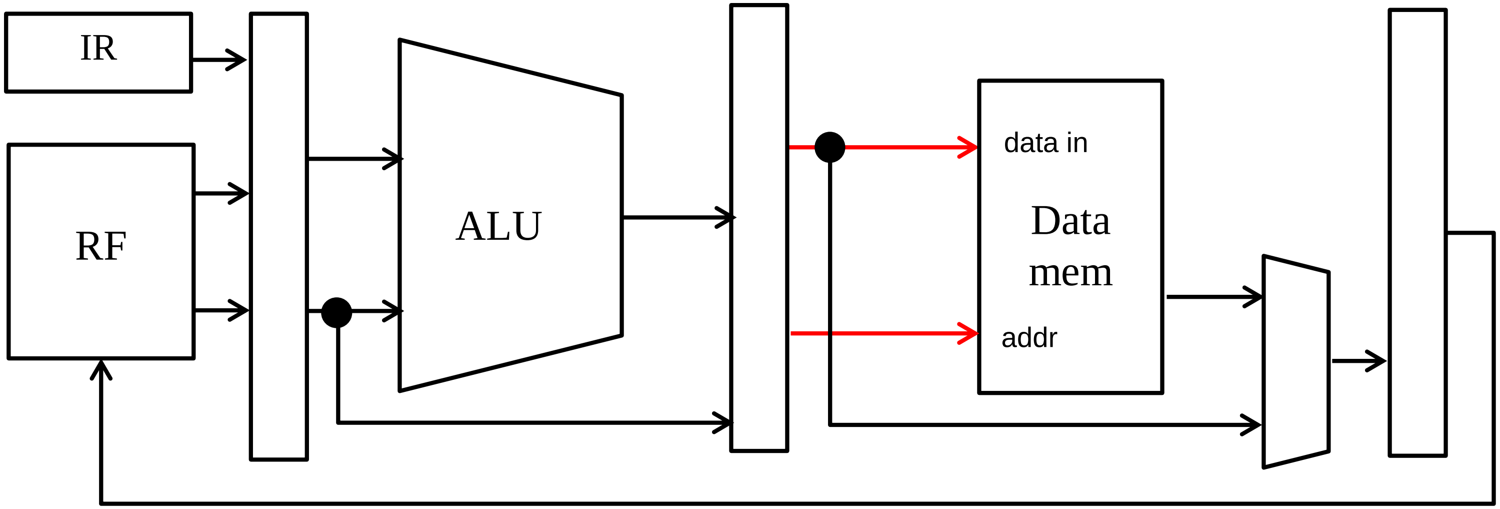
- Store Ra, @Rb (mem[Rb] = Ra)
  - 3rd stage:



store Ra, @Rb

opcode		Ra	Rb		i
1 0 0 0	x x x	b b b	b b b	x x	0

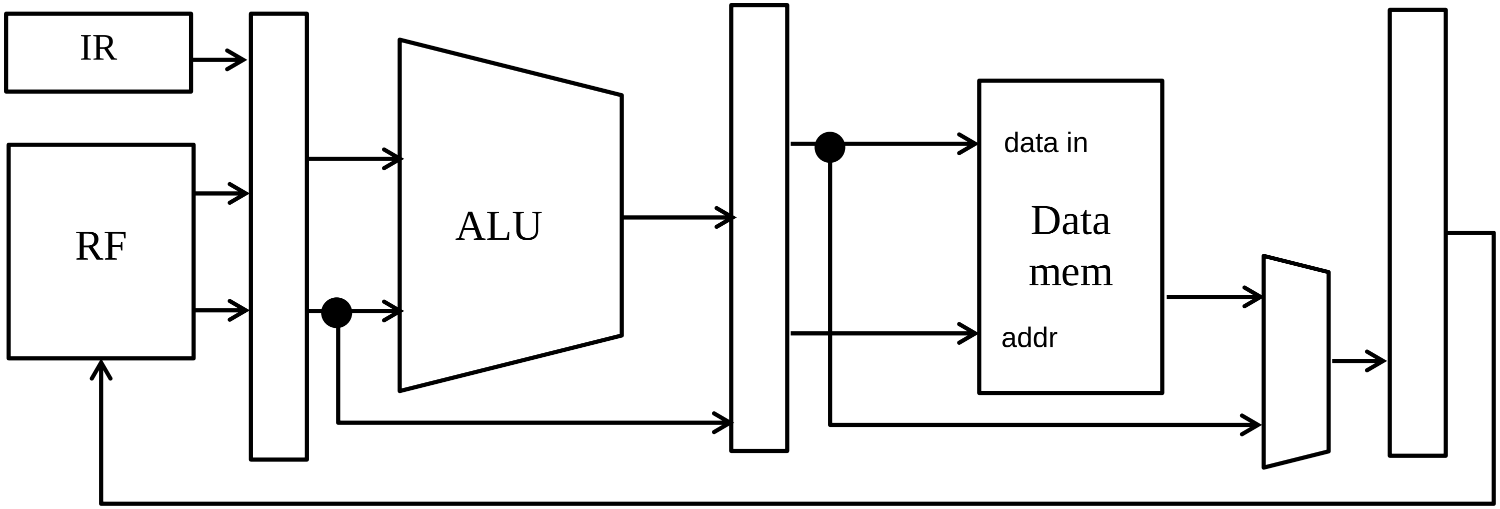
- Store Ra, @Rb (mem[Rb] = Ra)
    - 4th stage:



store Ra, @Rb

opcode															i
1	0	0	0	x	x	x	b	b	b	b	b	b	x	x	0

- Store Ra, @Rb (mem[Rb] = Ra)
  - 5th stage:
    - Do nothing

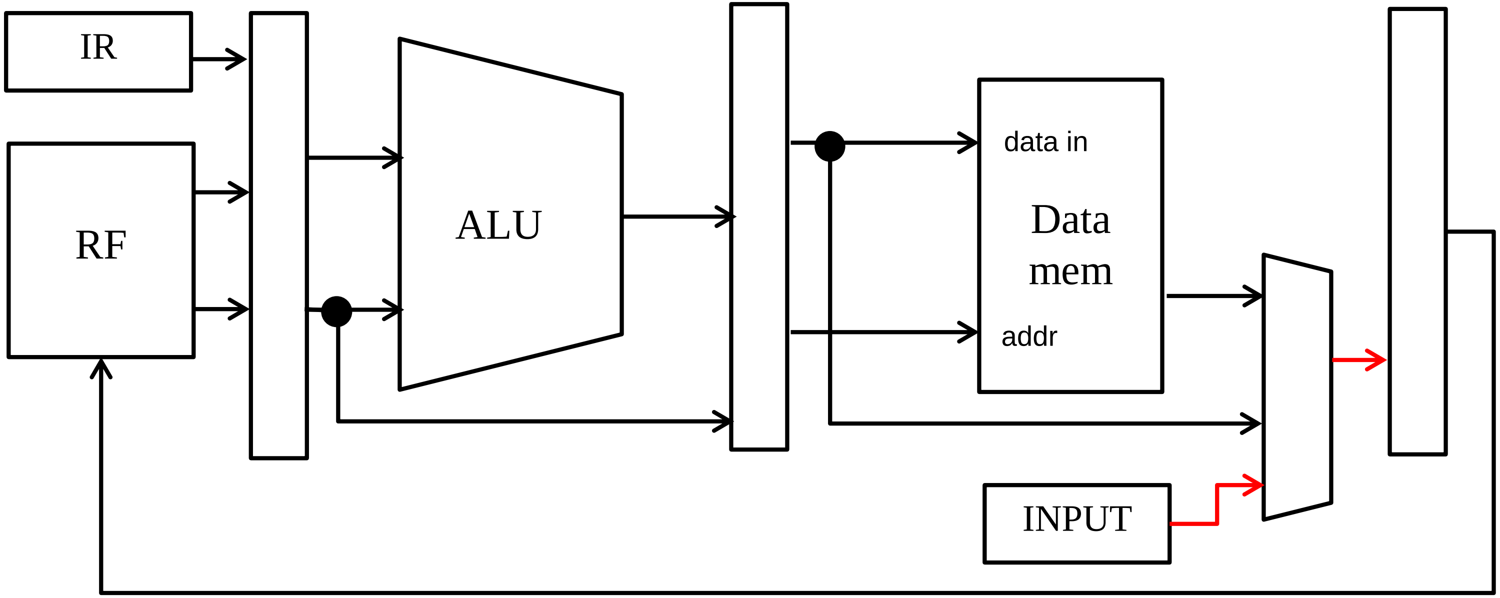


store Ra, @Rb

opcode										Ra			Rb					i
1	0	0	0	x	x	x	b	b	b	b	b	b	x	x	0

Embedded Systems Research Group

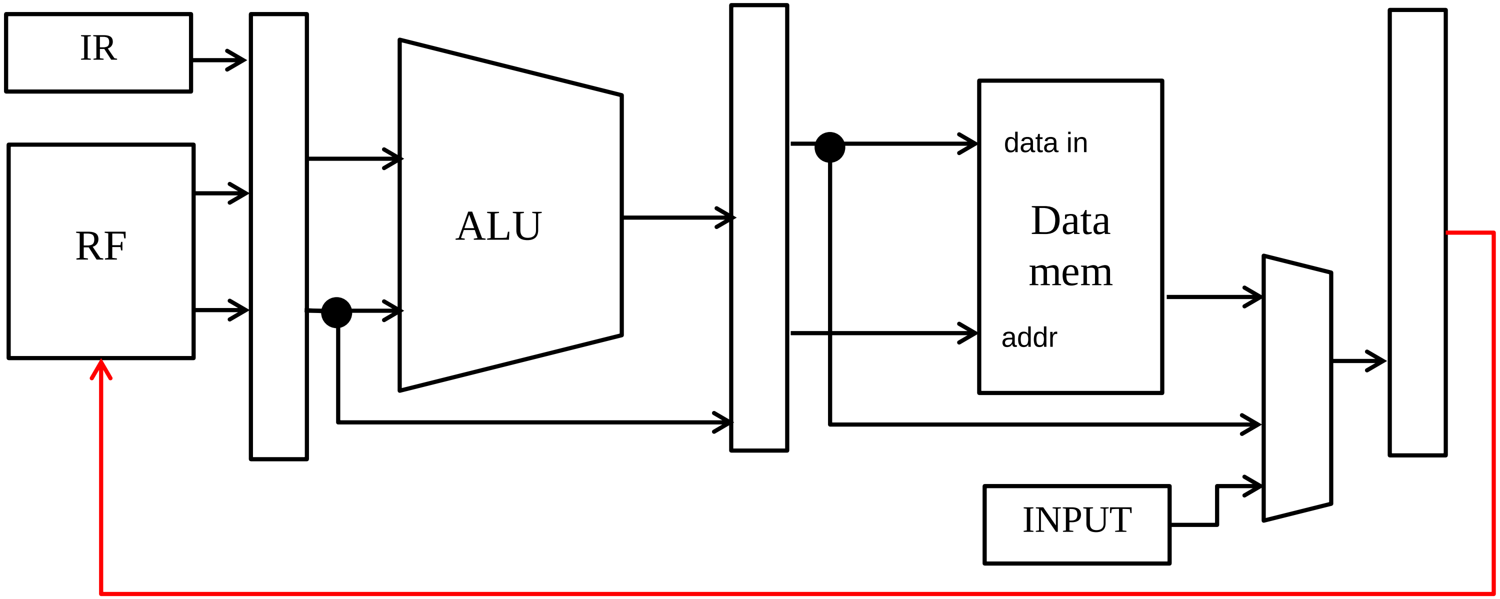
- Input Rd (Rd = Input)
  - 4th stage:
    - Read Input pins



input Rd

opcode	Rd	Ra		i
1 1 0 1	b b b	x x x	x x x x x	0

- Input Rd (Rd = Input)
  - 5th stage:
    - Write-back result



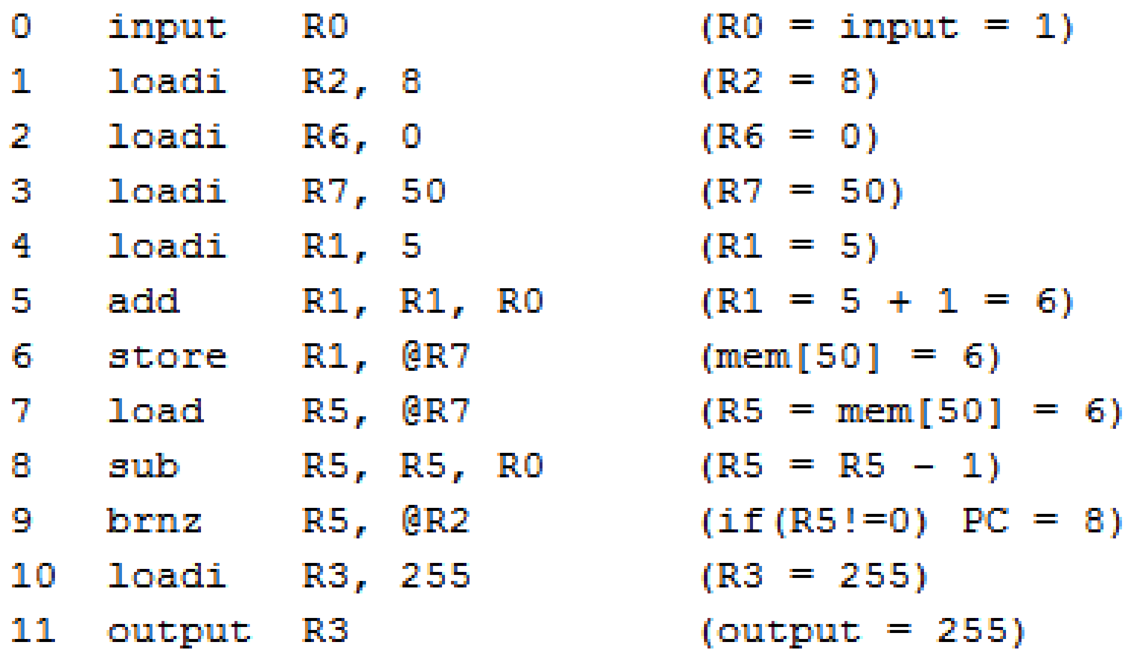
input Rd

opcode				Rd			Ra								i
1	1	0	1	b	b	b	x	x	x	x	x	x	x	x	0

Hazards

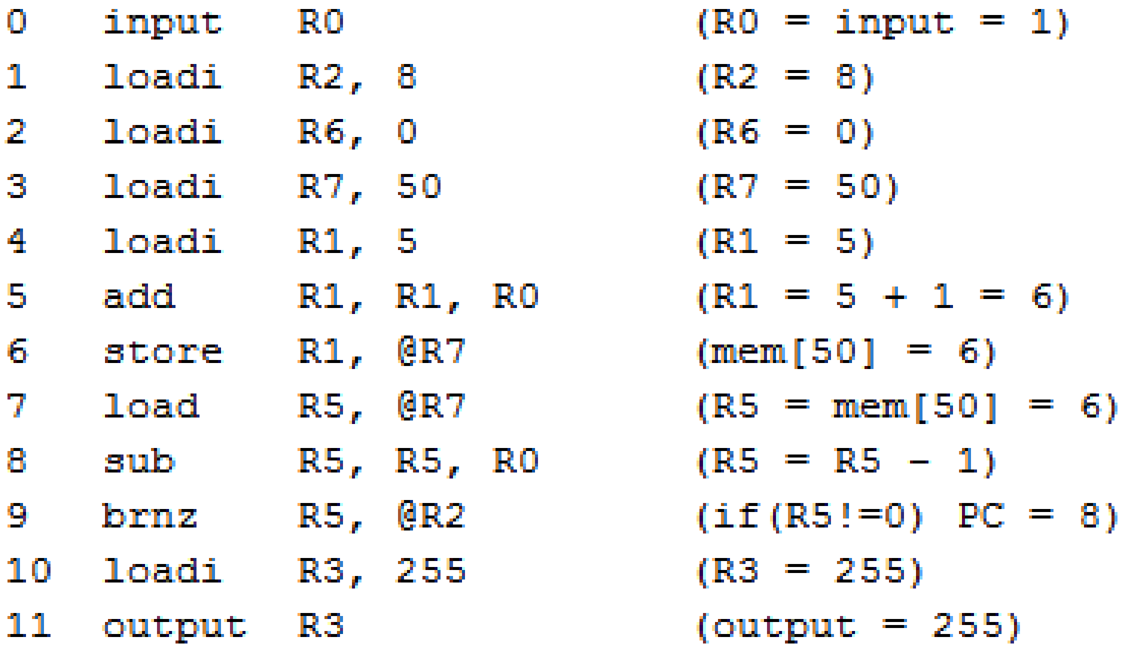
- Data Hazards
- Control Hazards
- Structural Hazards

- Let’s run this application



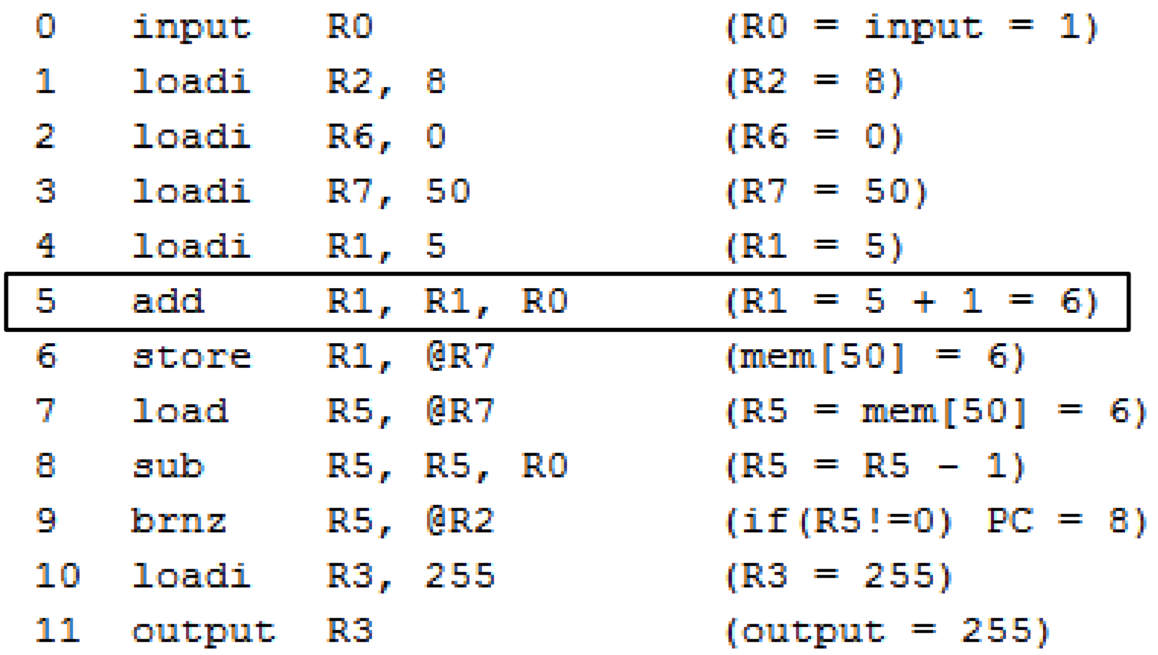
```
0    input    R0                (R0 = input = 1)
1    loadi    R2, 8             (R2 = 8)
2    loadi    R6, 0             (R6 = 0)
3    loadi    R7, 50            (R7 = 50)
4    loadi    R1, 5             (R1 = 5)
5    add      R1, R1, R0        (R1 = 5 + 1 = 6)
6    store    R1, @R7           (mem[50] = 6)
7    load     R5, @R7           (R5 = mem[50] = 6)
8    sub      R5, R5, R0        (R5 = R5 - 1)
9    brnz     R5, @R2           (if(R5!=0) PC = 8)
10   loadi    R3, 255           (R3 = 255)
11   output   R3                (output = 255)
```

- Hazards?



```
0    input    R0               (R0 = input = 1)
1    loadi    R2, 8            (R2 = 8)
2    loadi    R6, 0            (R6 = 0)
3    loadi    R7, 50           (R7 = 50)
4    loadi    R1, 5            (R1 = 5)
5    add      R1, R1, R0       (R1 = 5 + 1 = 6)
6    store    R1, @R7          (mem[50] = 6)
7    load     R5, @R7          (R5 = mem[50] = 6)
8    sub      R5, R5, R0       (R5 = R5 - 1)
9    brnz     R5, @R2          (if(R5!=0) PC = 8)
10   loadi    R3, 255          (R3 = 255)
11   output   R3               (output = 255)
```

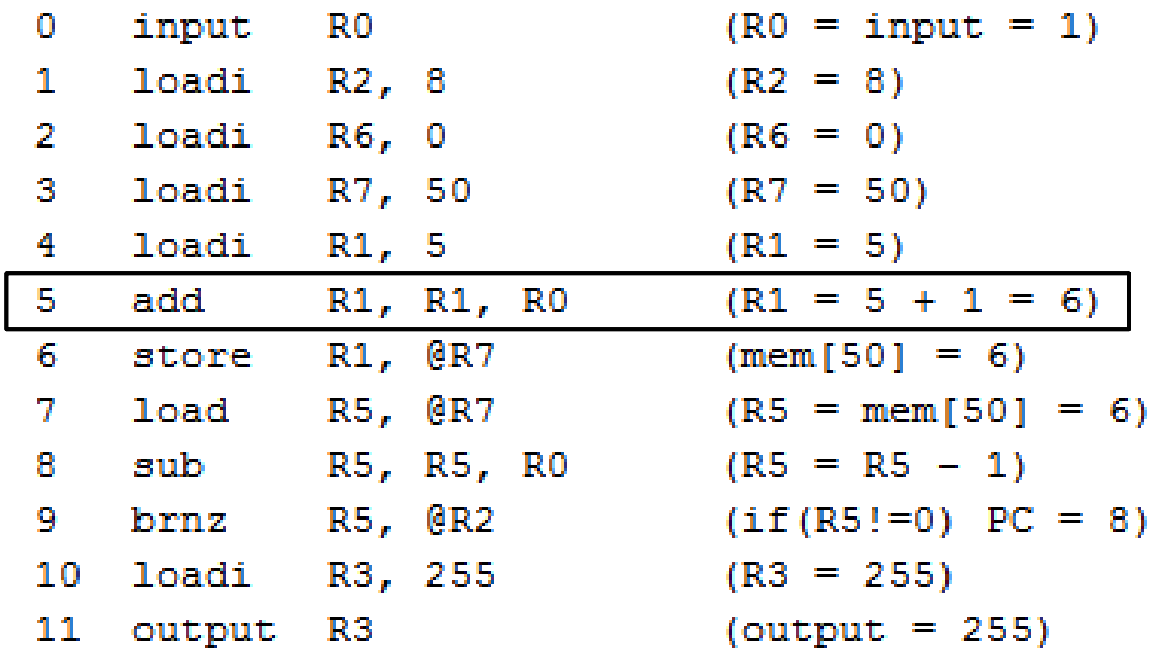
- Hazards?



```
0    input    R0                (R0 = input = 1)
1    loadi    R2, 8             (R2 = 8)
2    loadi    R6, 0             (R6 = 0)
3    loadi    R7, 50            (R7 = 50)
4    loadi    R1, 5             (R1 = 5)
5    add      R1, R1, R0        (R1 = 5 + 1 = 6)
6    store    R1, @R7           (mem[50] = 6)
7    load     R5, @R7           (R5 = mem[50] = 6)
8    sub      R5, R5, R0        (R5 = R5 - 1)
9    brnz     R5, @R2           (if(R5!=0) PC = 8)
10   loadi    R3, 255           (R3 = 255)
11   output   R3                (output = 255)
```

Data Hazard

- How to solve this hazard?



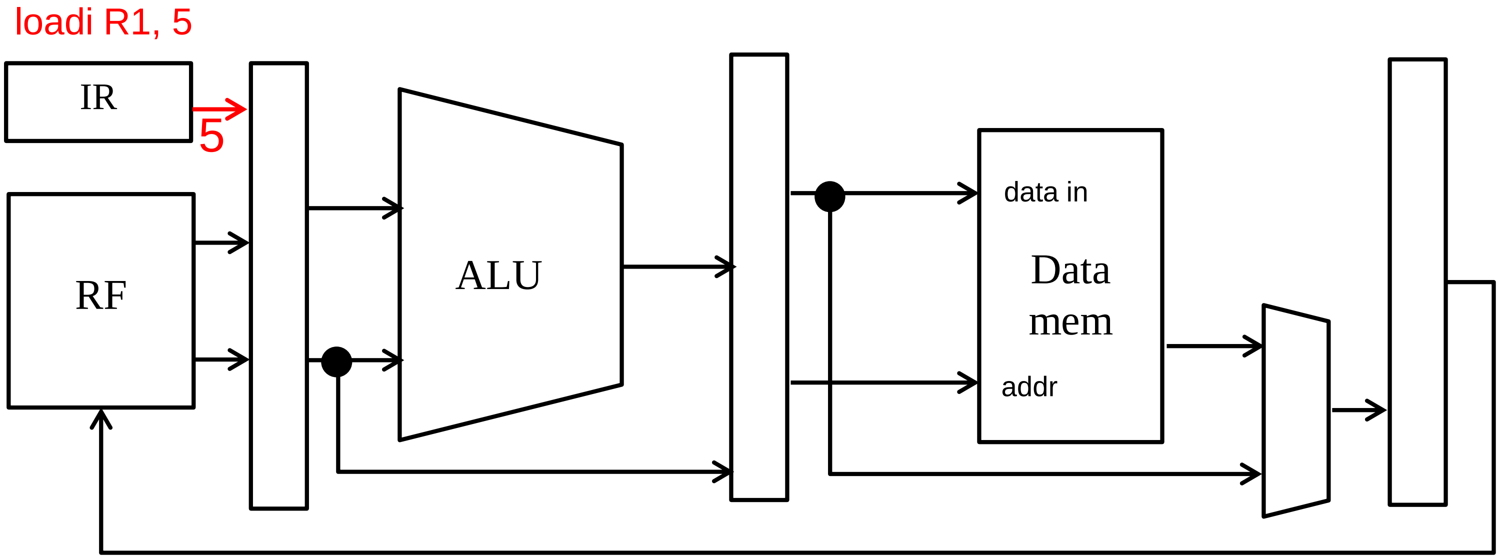
```
0    input    R0                    (R0 = input = 1)
1    loadi    R2, 8                 (R2 = 8)
2    loadi    R6, 0                 (R6 = 0)
3    loadi    R7, 50                (R7 = 50)
4    loadi    R1, 5                 (R1 = 5)
5    add      R1, R1, R0            (R1 = 5 + 1 = 6)
6    store    R1, @R7               (mem[50] = 6)
7    load     R5, @R7               (R5 = mem[50] = 6)
8    sub      R5, R5, R0            (R5 = R5 - 1)
9    brnz     R5, @R2               (if(R5!=0) PC = 8)
10   loadi    R3, 255               (R3 = 255)
11   output   R3                    (output = 255)
```

Data Hazard

- First solution: Using NOPs

```
4    loadi    R1, 5              (R1 = 5)
5    add      R1, R1, R0         (R1 = 5 + 1 = 6)
```

Data Hazard



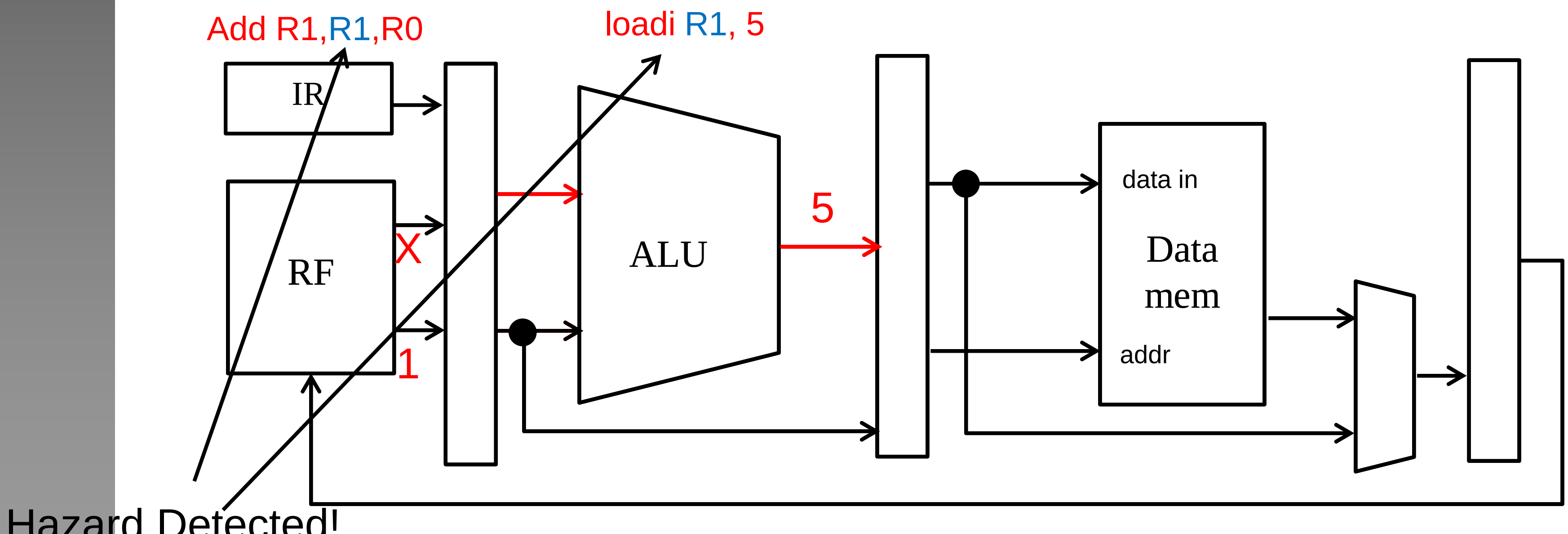
loadi Rd, imm

opcode				Rd				immediate							
0	1	1	0	b	b	b	x	b	b	b	b	b	b	b	b

- # First solution: Using NOPs

```
4    loadi    R1, 5             (R1 = 5)
5    add      R1, R1, R0        (R1 = 5 + 1 = 6)
```

Data Hazard



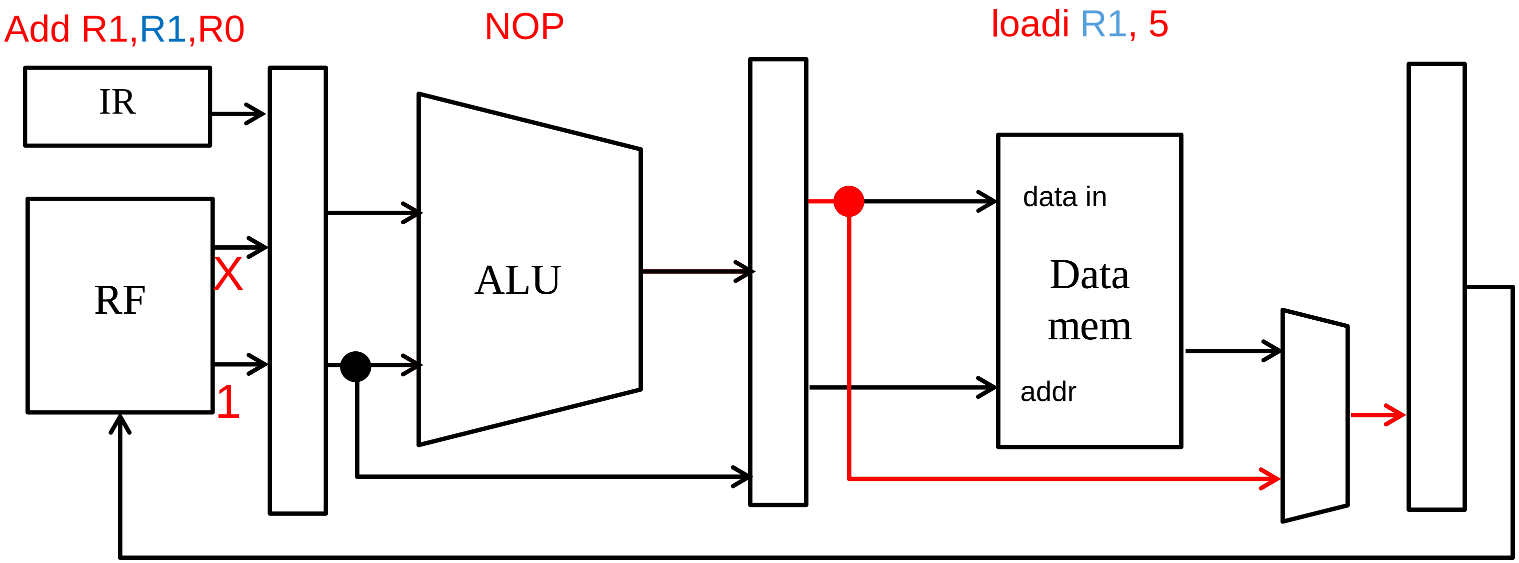
loadi Rd, imm

opcode	Rd		immediate
0 1 1 0	b b b	x	b b b b b b b b

- First solution: Using NOPs

```
4    loadi    R1, 5            (R1 = 5)
5    add      R1, R1, R0       (R1 = 5 + 1 = 6)
```

Data Hazard



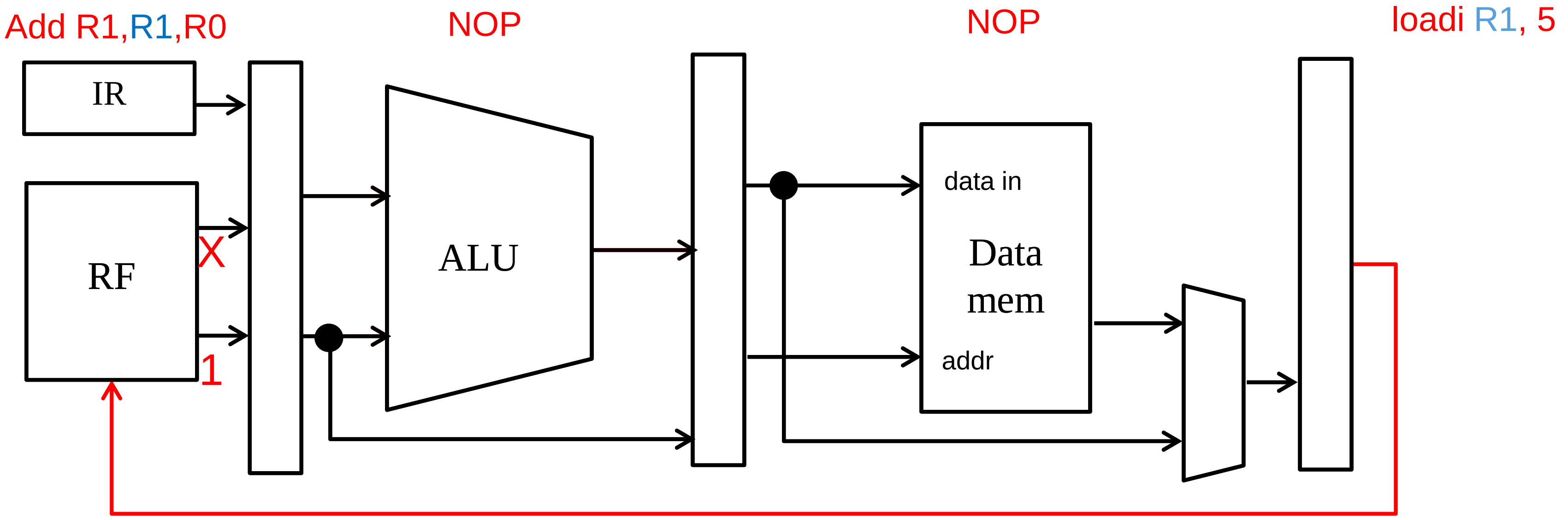
loadi Rd, imm

opcode				Rd				immediate							
0	1	1	0	b	b	b	x	b	b	b	b	b	b	b	b

- First solution: Using NOPs

```
4    loadi    R1, 5             (R1 = 5)
5    add      R1, R1, R0        (R1 = 5 + 1 = 6)
```

Data Hazard



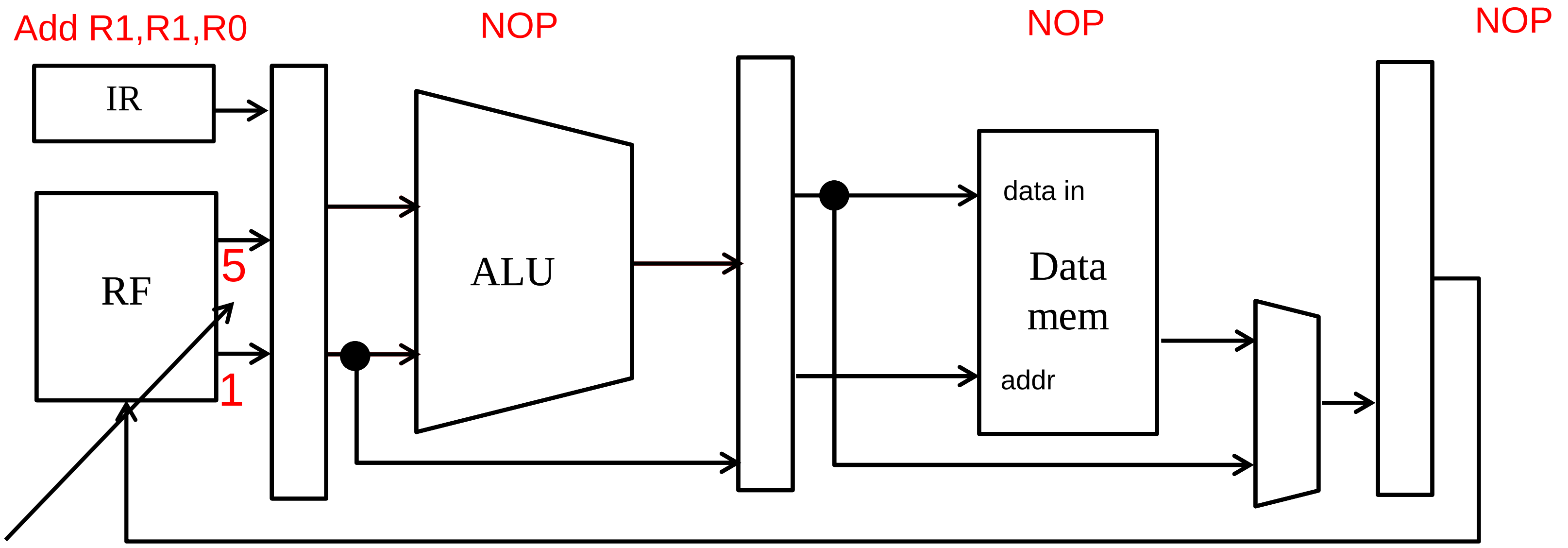
loadi Rd, imm

opcode	Rd		immediate
0 1 1 0	b b b	x	b b b b b b b b

- First solution: Using NOPs

```
4    loadi    R1, 5              (R1 = 5)
5    add      R1, R1, R0         (R1 = 5 + 1 = 6)
```

Data Hazard



Ready to execute

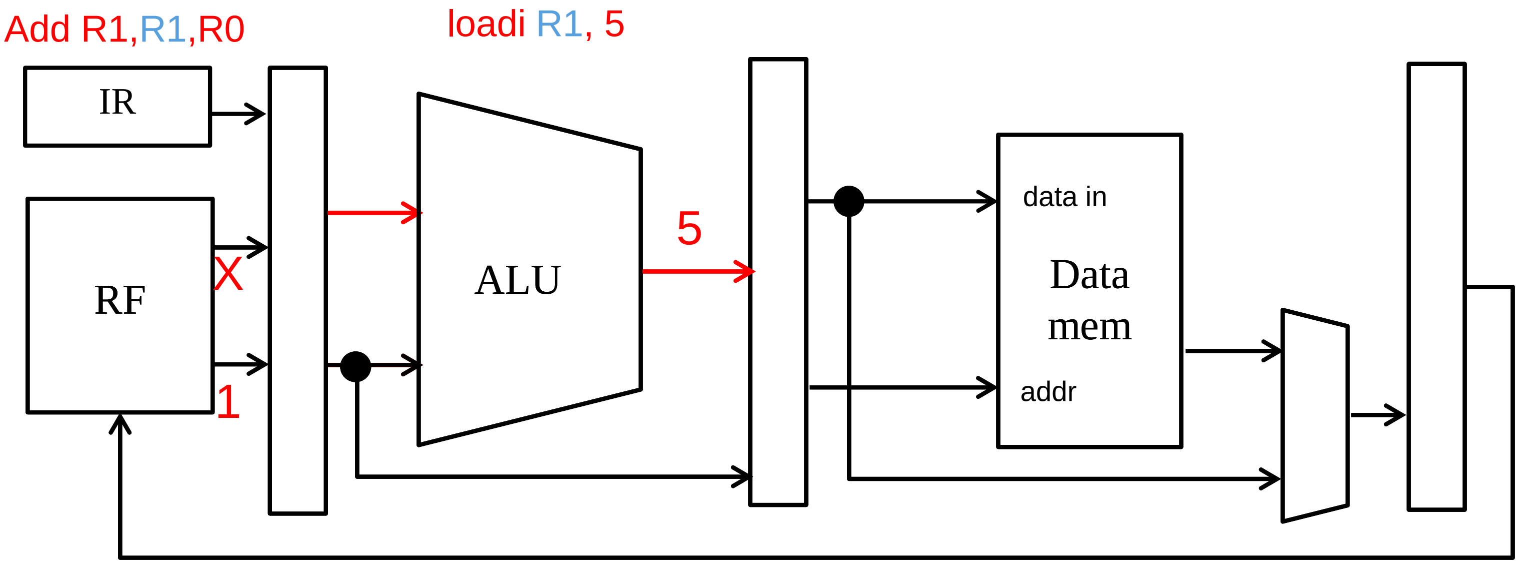
loadi Rd, imm

opcode	Rd		immediate
0 1 1 0	b b b	x	b b b b b b b b

- Second solution: Data Forward

```
4    loadi    R1, 5               (R1 = 5)
5    add      R1, R1, R0          (R1 = 5 + 1 = 6)
```

Data Hazard



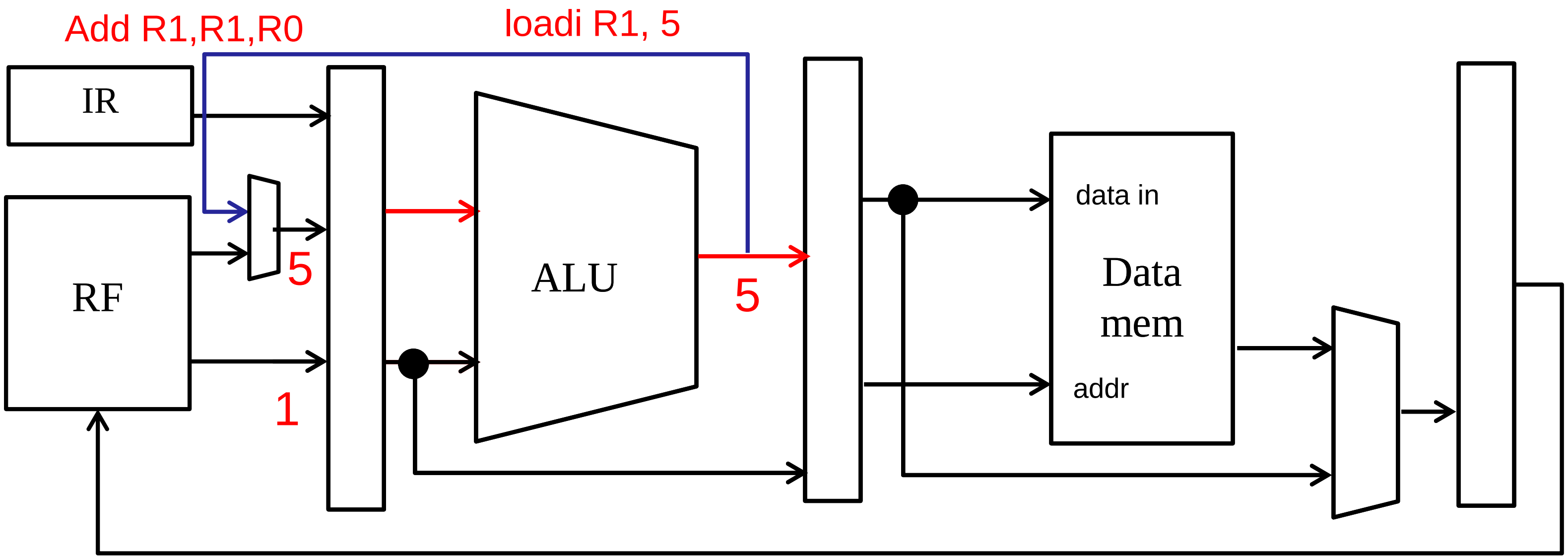
loadi Rd, imm

opcode	Rd		immediate
0 1 1 0	b b b	x	b b b b b b b b

- Second solution: Data Forward

```
4    loadi    R1, 5               (R1 = 5)
5    add      R1, R1, R0          (R1 = 5 + 1 = 6)
```

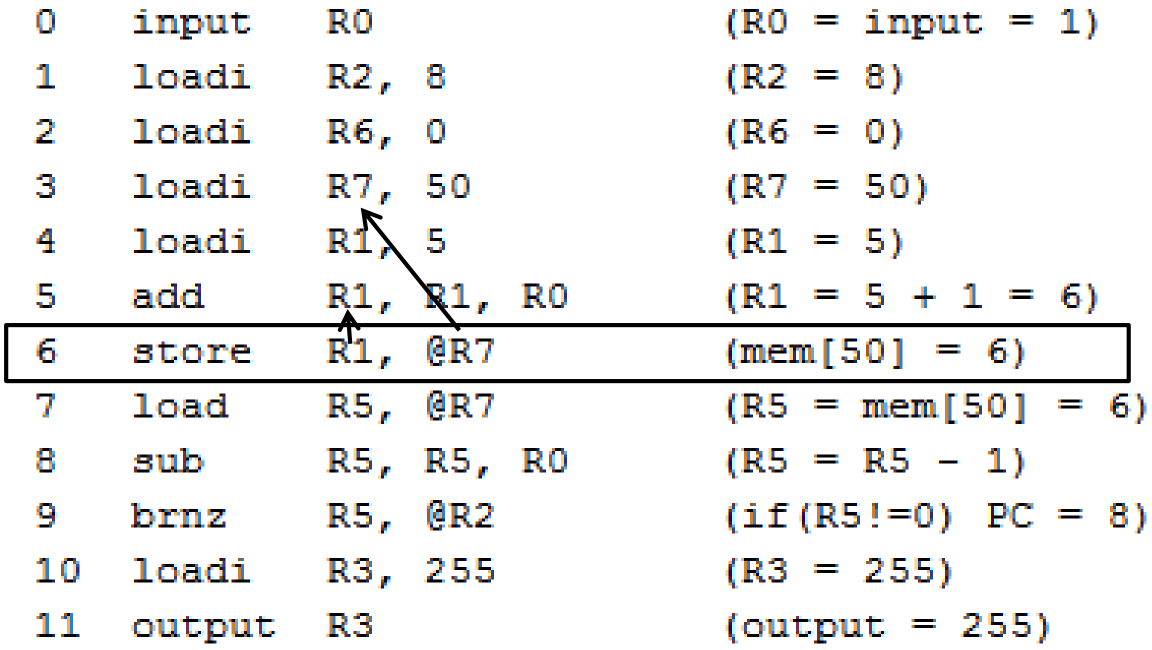
Data Hazard



loadi   Rd,   imm

opcode	Rd		immediate
0 1 1 0	b b b	x	b b b b b b b b

- How to solve this hazard?

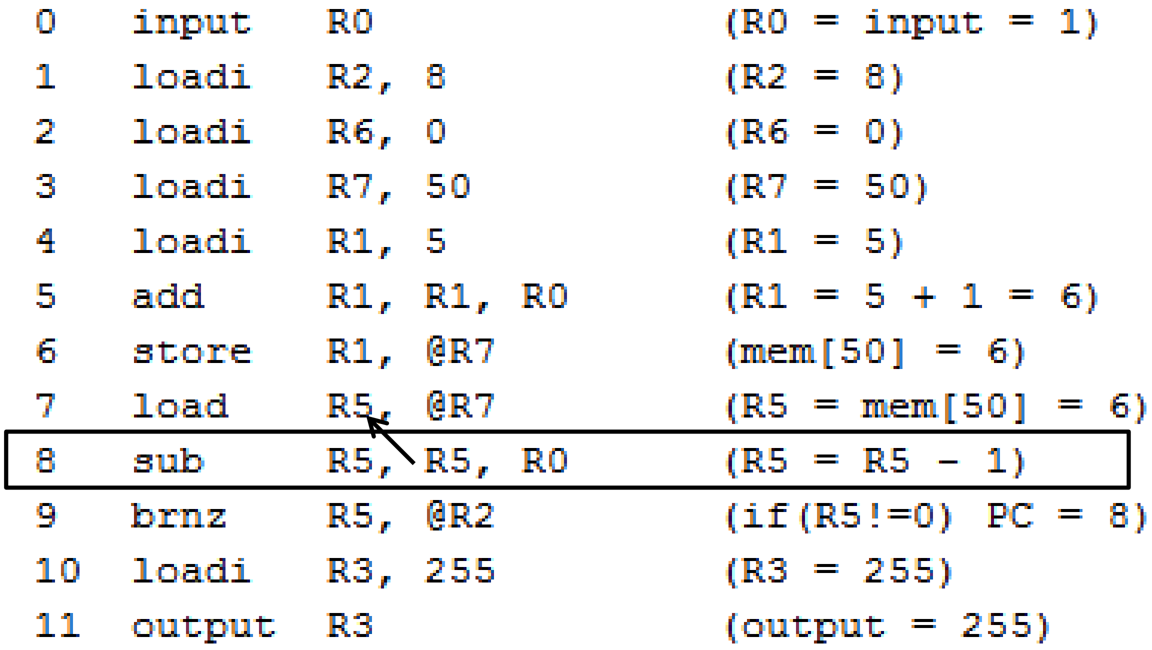


```
0    input    R0                 (R0 = input = 1)
1    loadi    R2, 8              (R2 = 8)
2    loadi    R6, 0              (R6 = 0)
3    loadi    R7, 50             (R7 = 50)
4    loadi    R1, 5              (R1 = 5)
5    add      R1, R1, R0         (R1 = 5 + 1 = 6)
6    store    R1, @R7            (mem[50] = 6)
7    load     R5, @R7            (R5 = mem[50] = 6)
8    sub      R5, R5, R0         (R5 = R5 - 1)
9    brnz     R5, @R2            (if(R5!=0) PC = 8)
10   loadi    R3, 255            (R3 = 255)
11   output   R3                 (output = 255)
```

Data Hazard

Also solved with data forwarding!

- How to solve this hazard?



```
0    input    R0                (R0 = input = 1)
1    loadi    R2, 8             (R2 = 8)
2    loadi    R6, 0             (R6 = 0)
3    loadi    R7, 50            (R7 = 50)
4    loadi    R1, 5             (R1 = 5)
5    add      R1, R1, R0        (R1 = 5 + 1 = 6)
6    store    R1, @R7           (mem[50] = 6)
7    load     R5, @R7           (R5 = mem[50] = 6)
8    sub      R5, R5, R0        (R5 = R5 - 1)
9    brnz     R5, @R2           (if(R5!=0) PC = 8)
10   loadi    R3, 255           (R3 = 255)
11   output   R3                (output = 255)
```

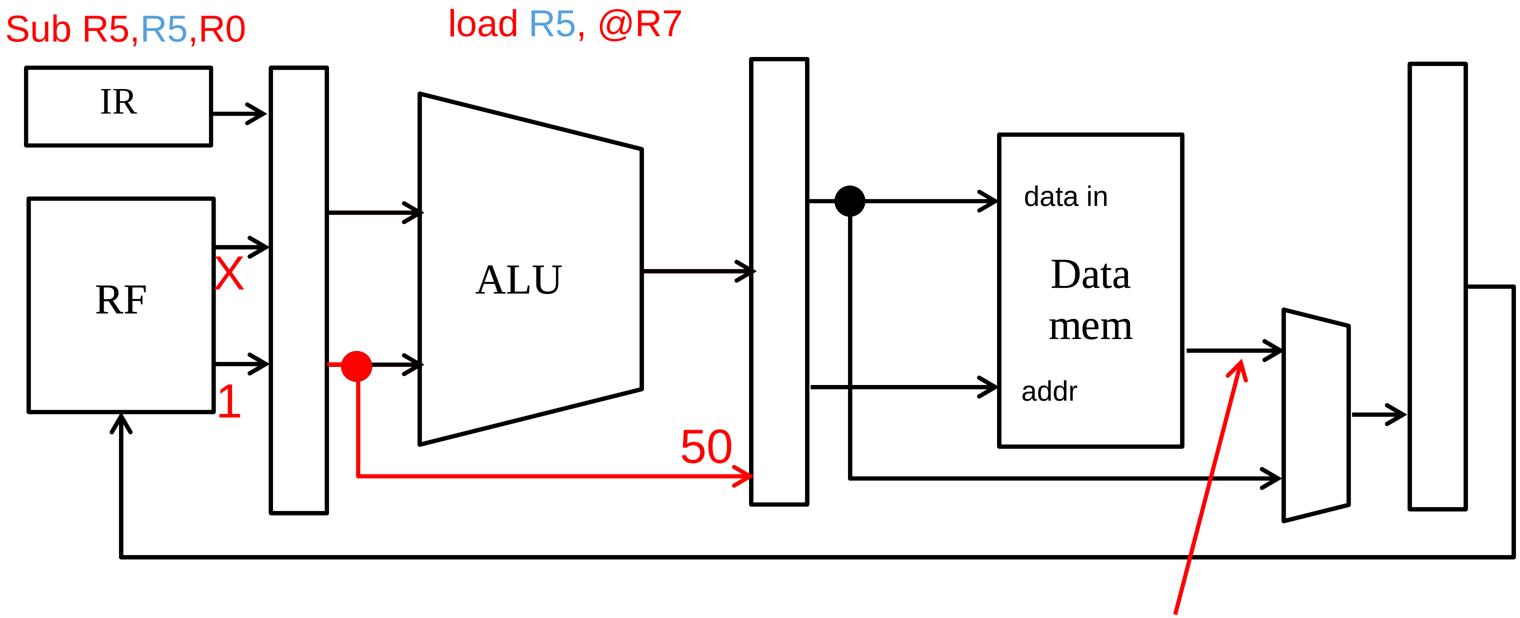
Data Hazard

It can’t be solved with data forward!!
The pipeline has to be stalled

- Second solution: Data Forward

```
7    load     R5, @R7             (R5 = mem[50] = 6)
8    sub      R5, R5, R0          (R5 = R5 - 1)
```

Data Hazard

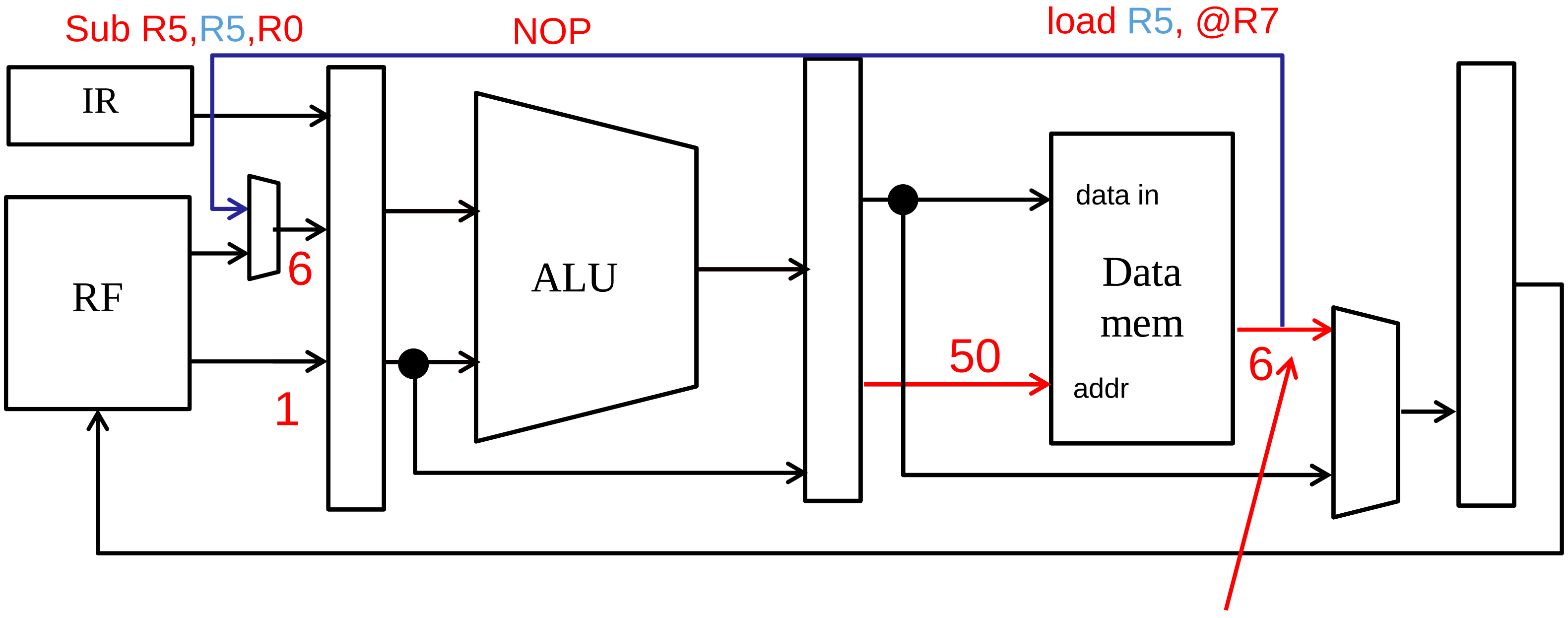


R5 is only available here!!!

- Second solution: Data Forward

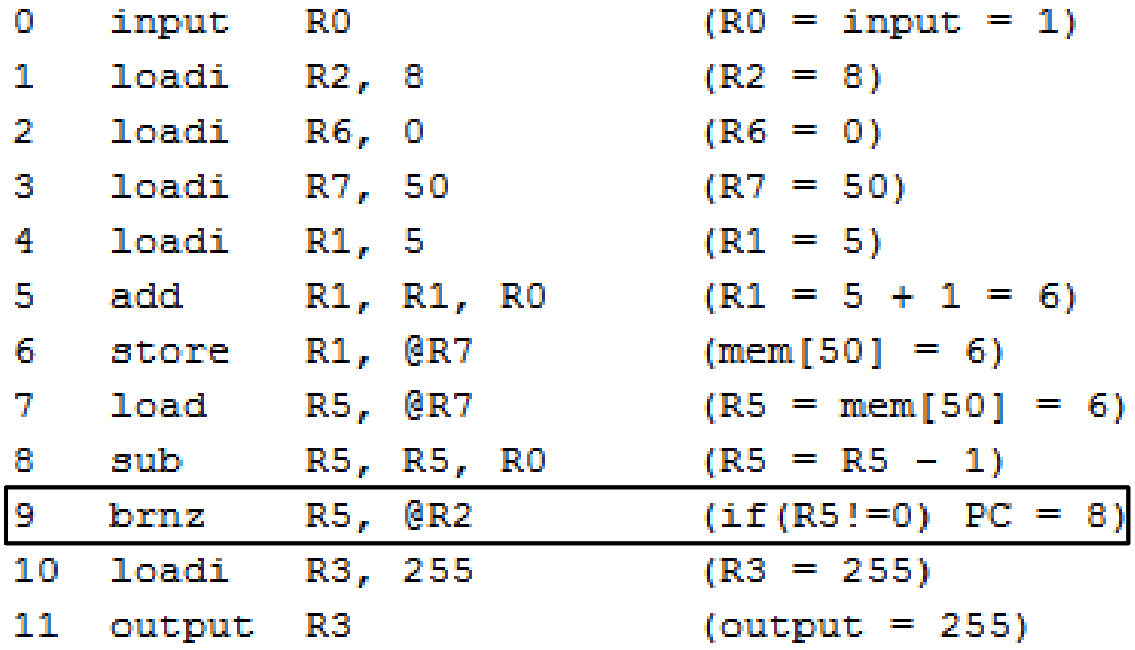
```
7    load     R5, @R7          (R5 = mem[50] = 6)
8    sub      R5, R5, R0       (R5 = R5 - 1)
```

Data Hazard



R5 is only available here!!!

- How to solve this hazard?



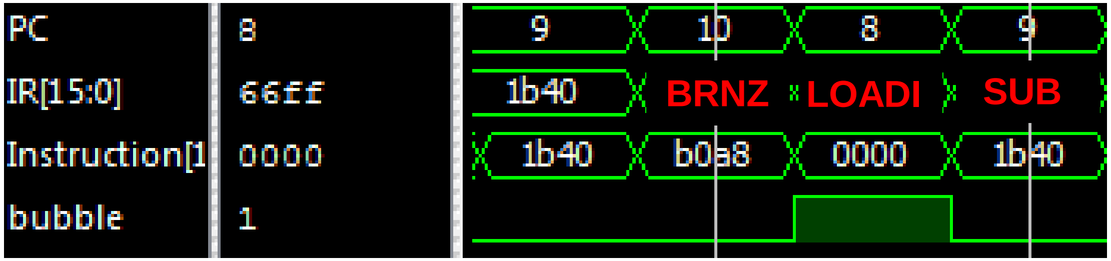
```
0    input    R0                    (R0 = input = 1)
1    loadi    R2, 8                 (R2 = 8)
2    loadi    R6, 0                 (R6 = 0)
3    loadi    R7, 50                (R7 = 50)
4    loadi    R1, 5                 (R1 = 5)
5    add      R1, R1, R0            (R1 = 5 + 1 = 6)
6    store    R1, @R7               (mem[50] = 6)
7    load     R5, @R7               (R5 = mem[50] = 6)
8    sub      R5, R5, R0            (R5 = R5 - 1)
9    brnz     R5, @R2               (if(R5!=0) PC = 8)
10   loadi    R3, 255               (R3 = 255)
11   output   R3                    (output = 255)
```

Control Hazard

- How to solve this hazard?

```
8    sub      R5, R5, R0         (R5 = R5 - 1)
9    brnz     R5, @R2            (if(R5!=0) PC = 8)
10   loadi    R3, 255            (R3 = 255)
```

Control Hazard

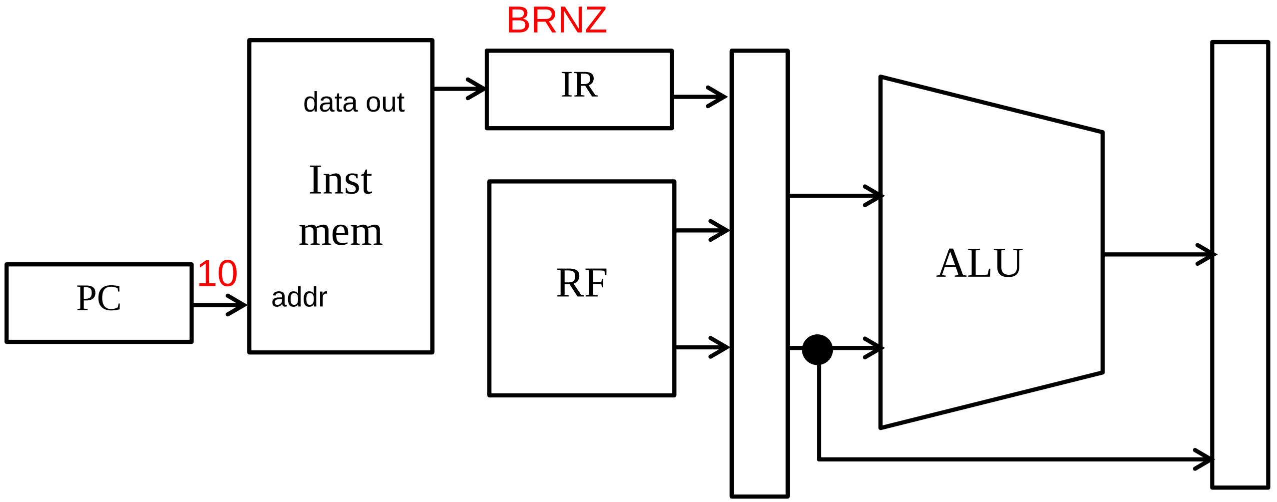


When the BRNZ instruction is fetched, the PC already points to 10, so in the next posedge Clk, the LOADI will be fetched.
However the instruction that should be executed is the SUB instruction since the branch condition is true

To solve this hazard a bubble is introduced

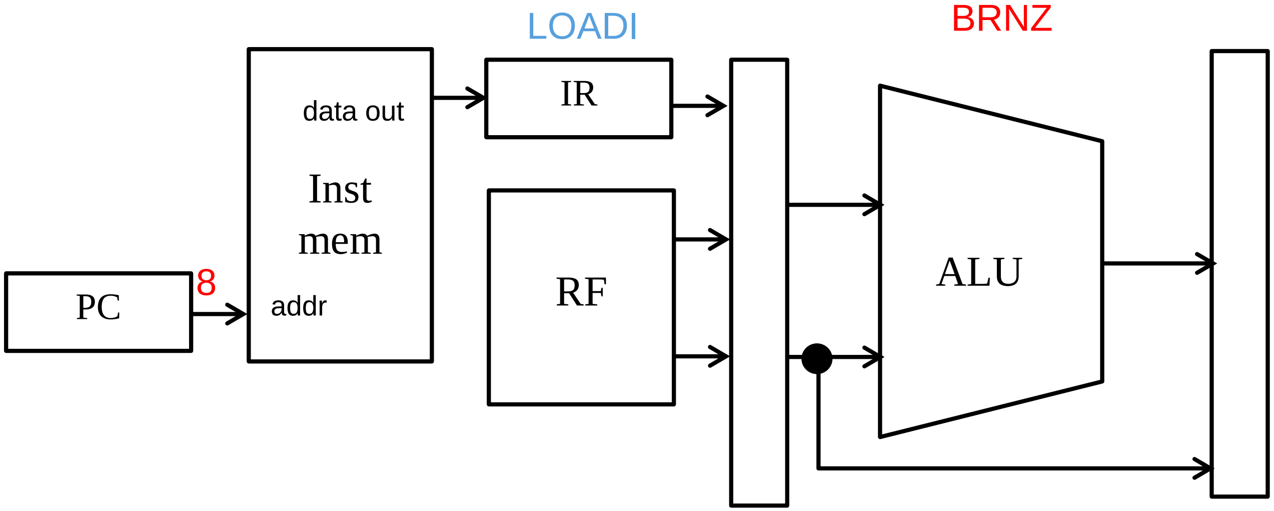
```
8    sub      R5, R5, R0         (R5 = R5 - 1)
9    brnz     R5, @R2            (if(R5!=0) PC = 8)
10   loadi    R3, 255            (R3 = 255)
```

Control Hazard



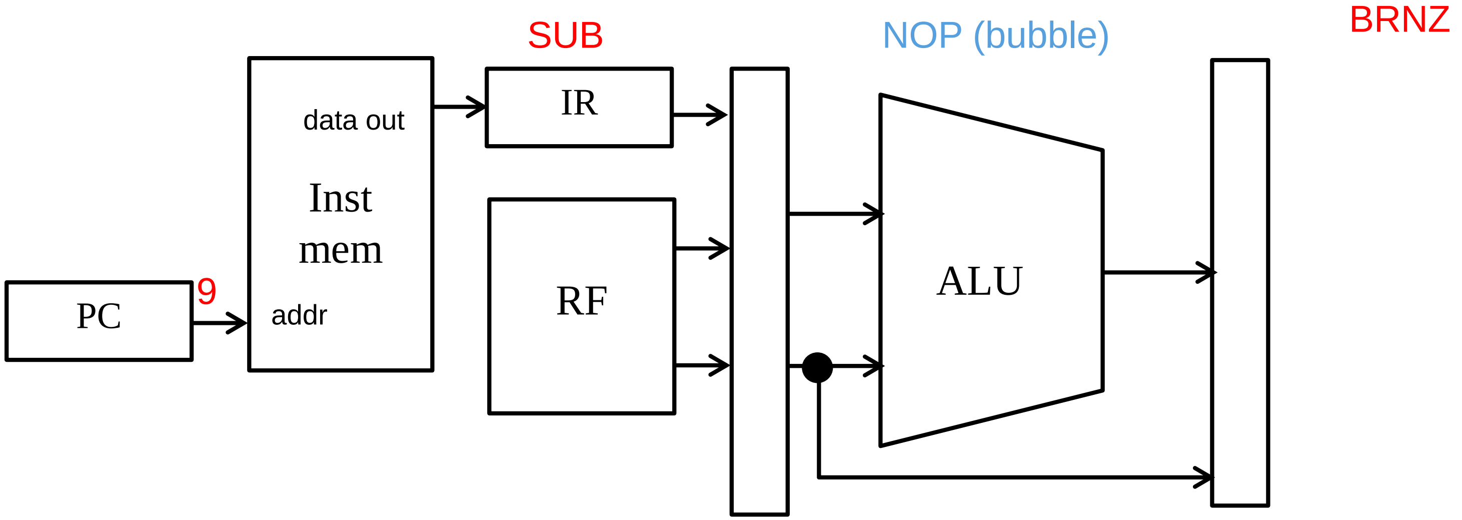
```
8    sub      R5, R5, R0         (R5 = R5 - 1)
9    brnz     R5, @R2            (if(R5!=0) PC = 8)
10   loadi    R3, 255            (R3 = 255)
```

Control Hazard

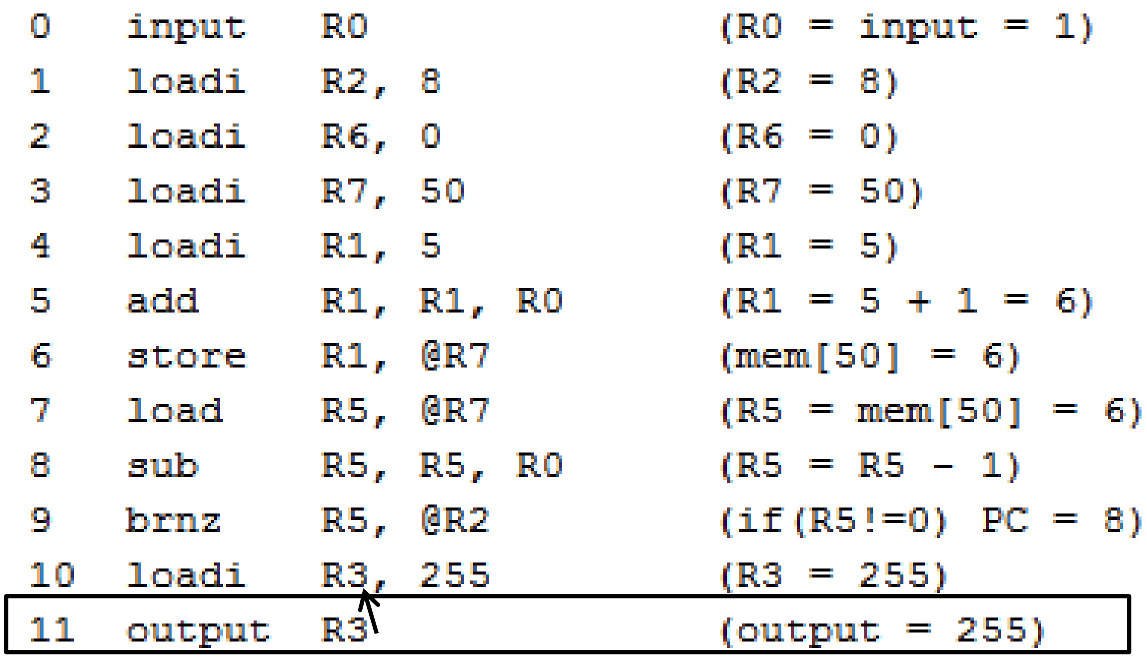


```
8    sub     R5, R5, R0       (R5 = R5 - 1)
9    brnz    R5, @R2          (if(R5!=0) PC = 8)
10   loadi   R3, 255          (R3 = 255)
```

Control Hazard



- How to solve this hazard?

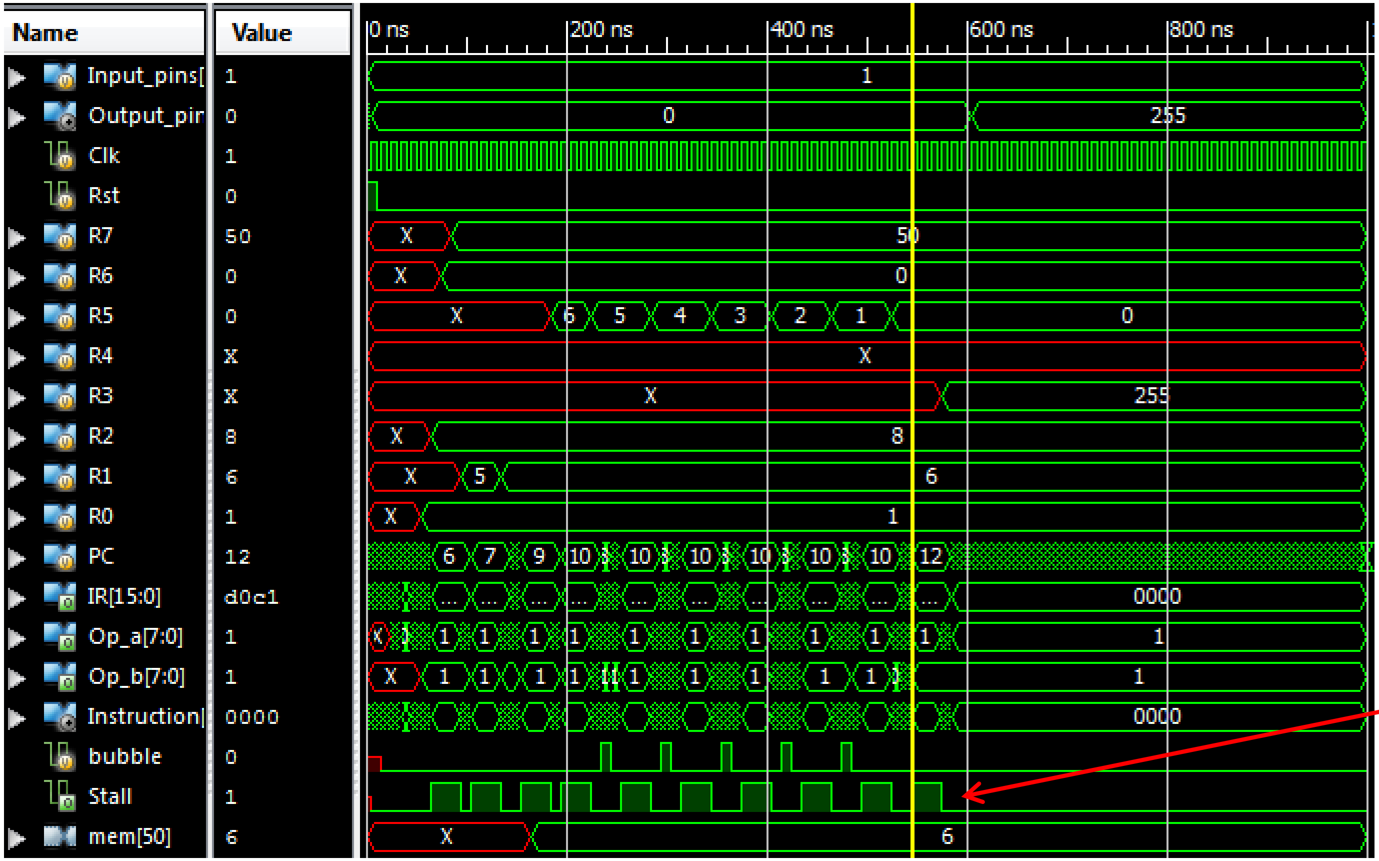


```
0    input    R0                   (R0 = input = 1)
1    loadi    R2, 8                (R2 = 8)
2    loadi    R6, 0                (R6 = 0)
3    loadi    R7, 50               (R7 = 50)
4    loadi    R1, 5                (R1 = 5)
5    add      R1, R1, R0           (R1 = 5 + 1 = 6)
6    store    R1, @R7              (mem[50] = 6)
7    load     R5, @R7              (R5 = mem[50] = 6)
8    sub      R5, R5, R0           (R5 = R5 - 1)
9    brnz     R5, @R2              (if(R5!=0) PC = 8)
10   loadi    R3, 255              (R3 = 255)
11   output   R3                   (output = 255)
```

Data Hazard

Also solved with data forwarding!

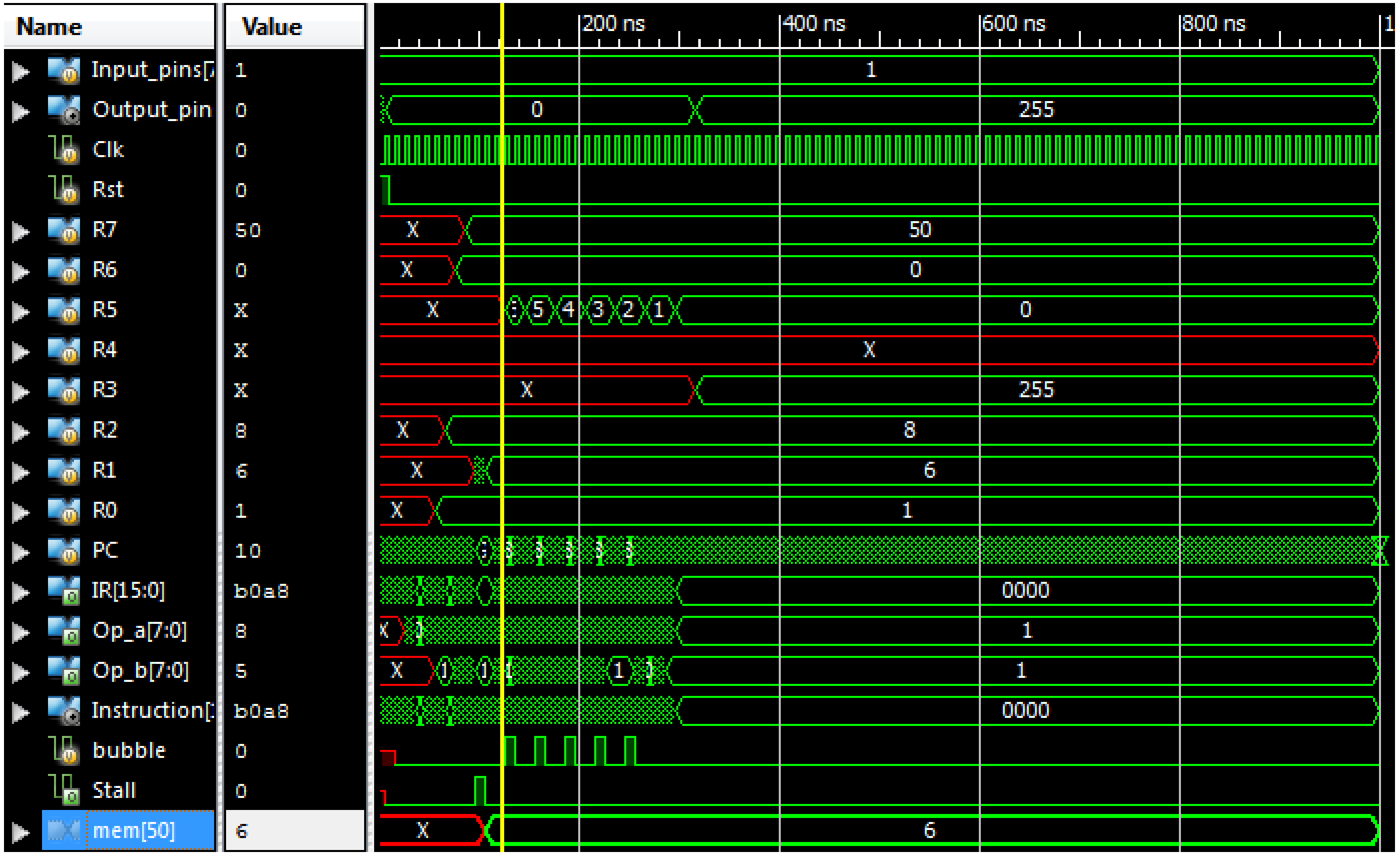
- Pipeline without data forwarding executes this pipeline in 605ns



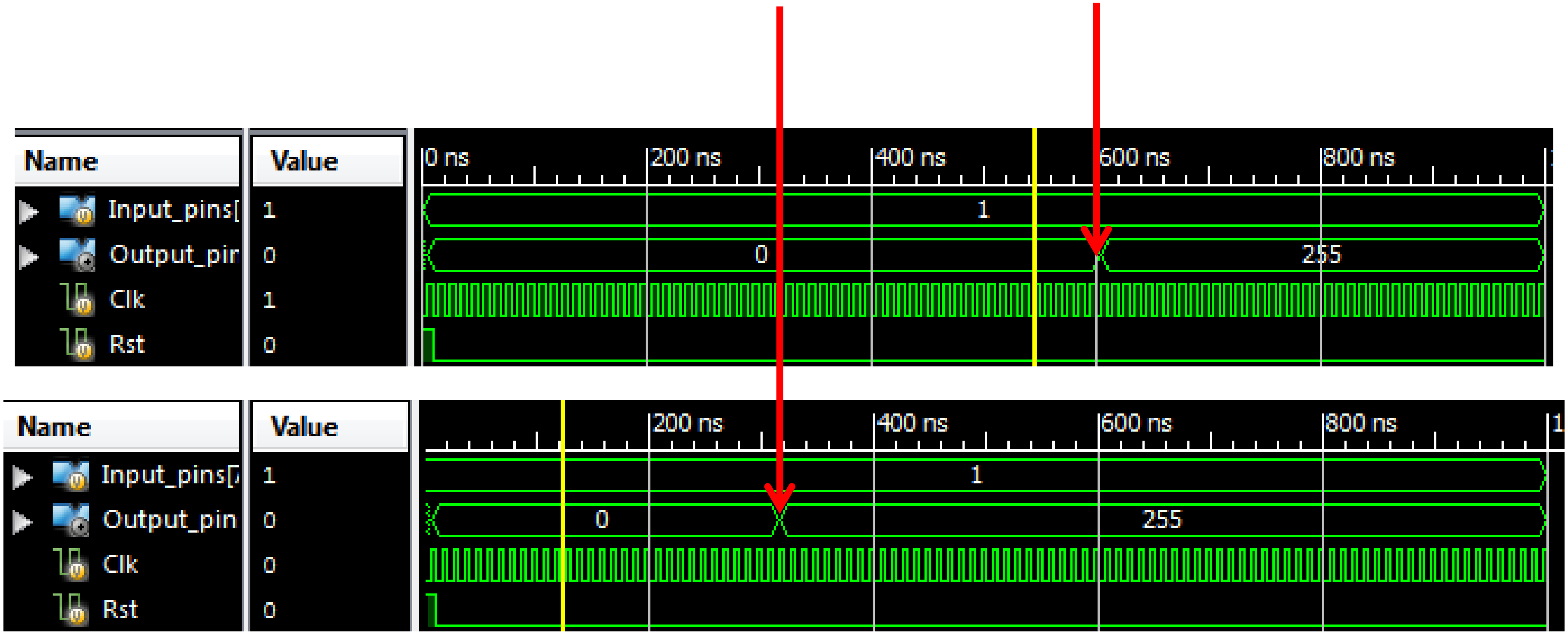
Name	Value
Input_pins[	1
Output_pir	0
Clk	1
Rst	0
R7	50
R6	0
R5	0
R4	X
R3	X
R2	8
R1	6
R0	1
PC	12
IR[15:0]	d0c1
Op_a[7:0]	1
Op_b[7:0]	1
Instruction	0000
bubble	0
Stall	1
mem[50]	6

Too many NOPs

- While the version with data forward implemented executes in 315ns (almost half of the time!)



Name	Value
Input_pins[	1
Output_pin	0
Clk	0
Rst	0
R7	50
R6	0
R5	x
R4	x
R3	x
R2	8
R1	6
R0	1
PC	10
IR[15:0]	b0a8
Op_a[7:0]	8
Op_b[7:0]	5
Instruction[	b0a8
bubble	0
Stall	0
mem[50]	6



Name	Value
Input_pins[	1
Output_pir	0
Clk	1
Rst	0

Name	Value
Input_pins[/	1
Output_pin	0
Clk	0
Rst	0

Future work

- Extend the microprocessor
    - Add more instructions
        - Multiply
        - Mov between Registers
        - Other branch conditions
        - Halt
    - Add support to Jump and Link
- Develop Assembler

- THE END