Embedded Systems Research Group

Computer Architectures

Developing a Pipeline processor

22/04/2013

Tiago Gomes

(ESRG)

Centro ALGORITMI

Department of Industrial Electronics

School of Engineering,

University of Minho, Guimarães - PORTUGAL

Features

Embedded Systems Research Group

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

Features

Embedded Systems Research Group

- Register File
  - 8 registers x 8-bits
- Instruction Memory
  - 256 x 16-bits
- Data Memory
  - 256 x 8-bits
- IO ports
  - 8-bit Input port
  - 8-bit Output port

Summary

Embedded Systems Research Group

- Instruction Set Architecture
- Micro-Architecture
- Hazards
- Results

Architecture

Embedded Systems Research Group

- Register File:
  - RF = {R0, R1, R2, R3, R4, R5, R6, R7}
- 6 Arithmetic/Logic instructions
- 4 Data transfer instructions
- 3 Control instructions

Architecture

Embedded Systems Research Group

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

      Rd = Ra op Rb (eg., R3 = R2 + R1)
    - NOT

      Rd = ~Ra (eg., R5 = ~R7)

Architecture

Embedded Systems Research Group

- Data transfer instructions
  - Immediate Load

    Rd = value (eg., R3 = 0x55)
  - Load

    Rd = mem[addr] (eg., R2 = mem[3])
  - Store

    mem[addr] = value (eg., mem[5] = R0)
  - Input / Output

    Rd = Input (eg., R5 = input)

    Output = Ra (eg., Output = R2)

Architecture

Embedded Systems Research Group

- Control instructions
  - JMP

    PC = Ra (eg., PC = R1)
  - BRZ

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

    If (Rb !=0) PC = Ra
- Miscellaneous
  - NOP (PC = PC + 1)

Summary

Embedded Systems Research Group

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

- First step: instruction encoding
  - Our ISA specifies 13 instructions
    - How many bits for the opcode?
    - How many bits to address the Register File?

    ...

    - How can an ALU instruction be encoded?

      ADD Rd, Ra, Rb

Micro-Architecture

Embedded Systems Research Group

- 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

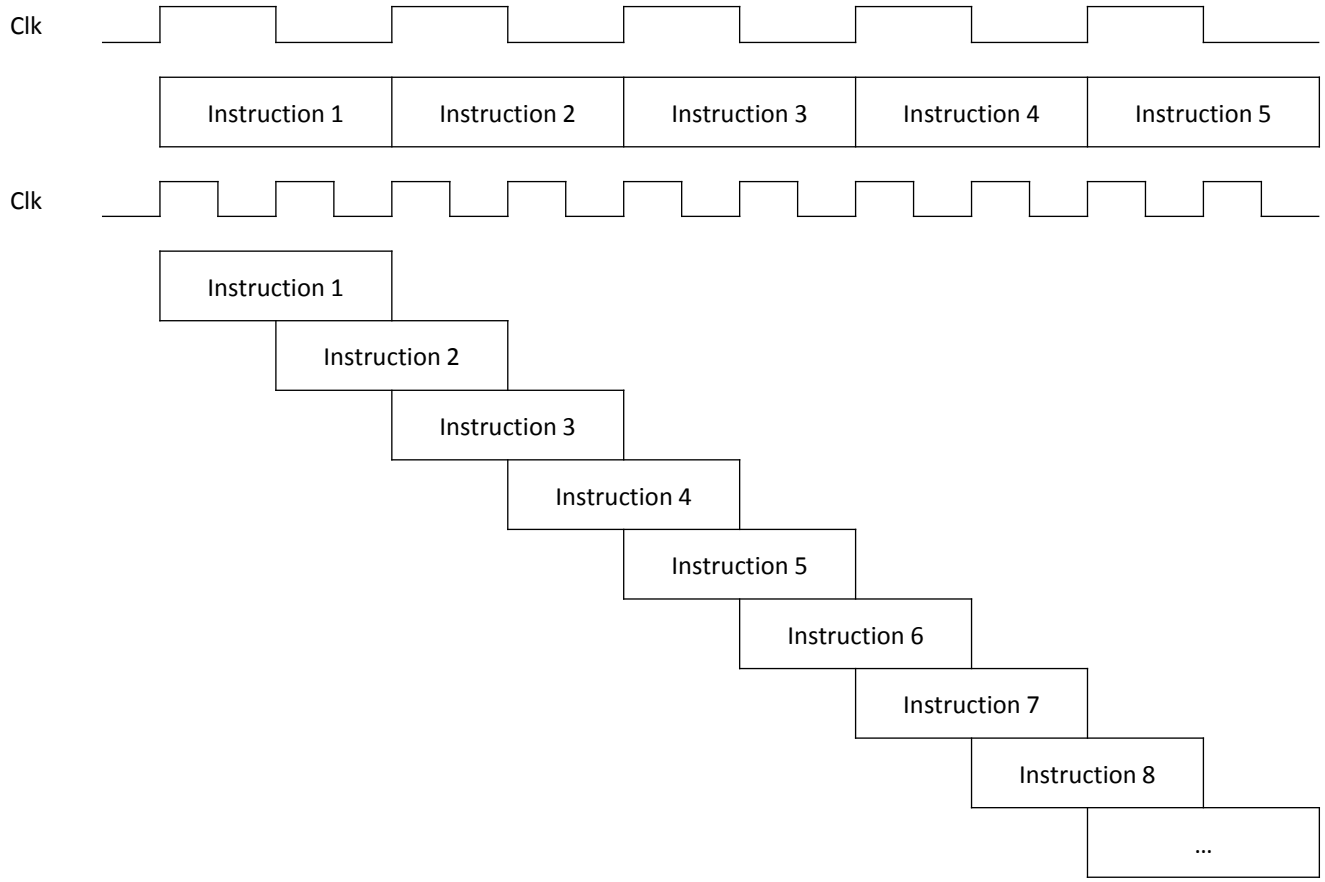
	opcode	
nop	0 0 0 0	x x x x x x x x x x x x

Pipeline

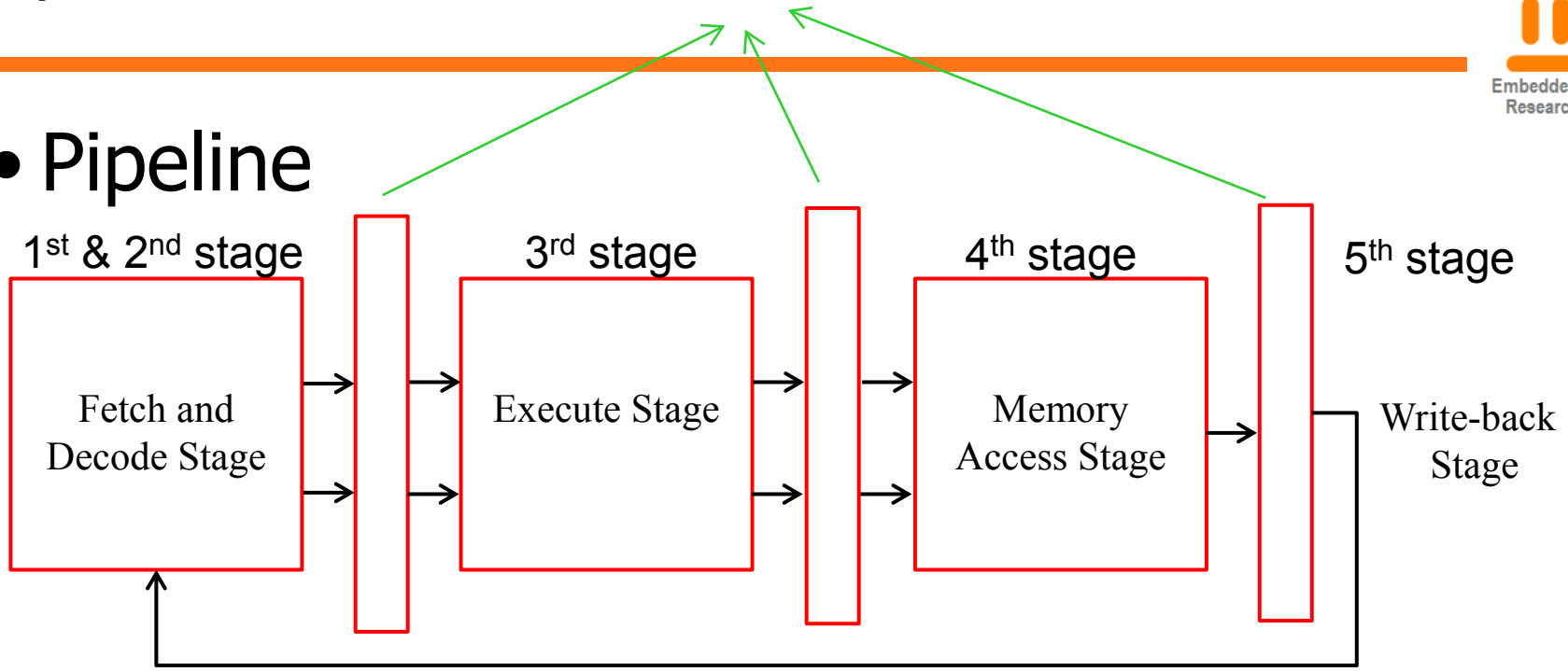
Embedded Systems Research Group

- Is it possible to divide instructions in several tasks/stages?
  - Fetch & Decode Stages
    - 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


Clk
Instruction 1
Instruction 2
Instruction 3
Instruction 4
Instruction 5
Clk
Instruction 1
Instruction 2
Instruction 3
Instruction 4
Instruction 5
Instruction 6
Instruction 7
Instruction 8
...

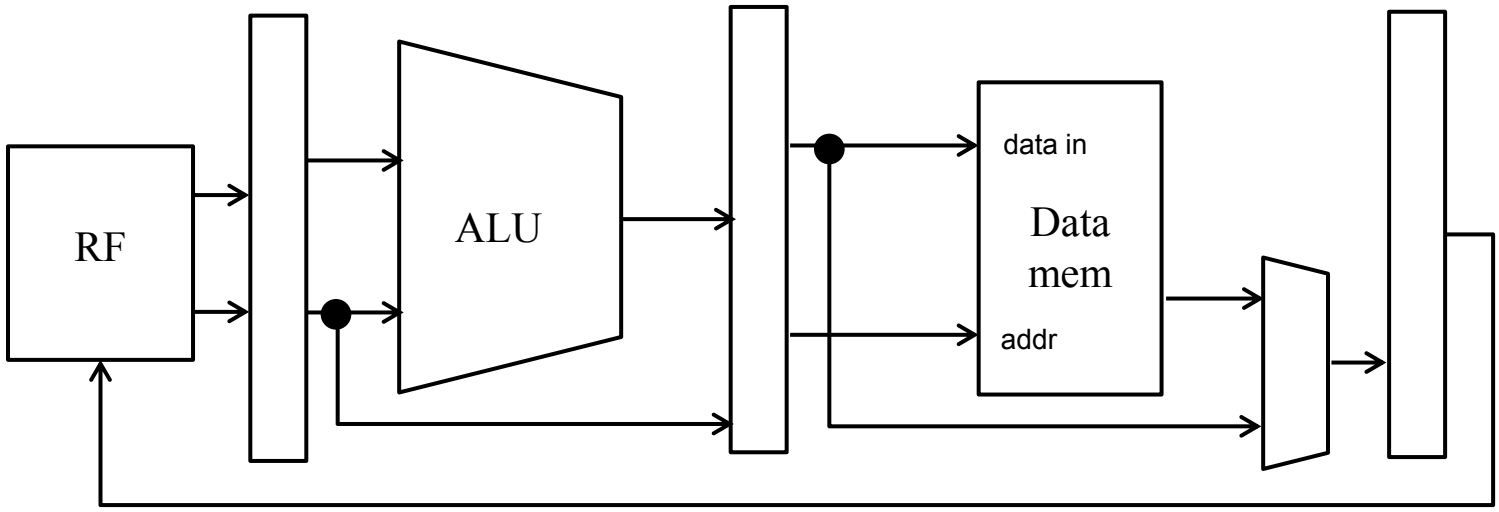
- Pipeline


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

Pipeline

Embedded Systems Research Group

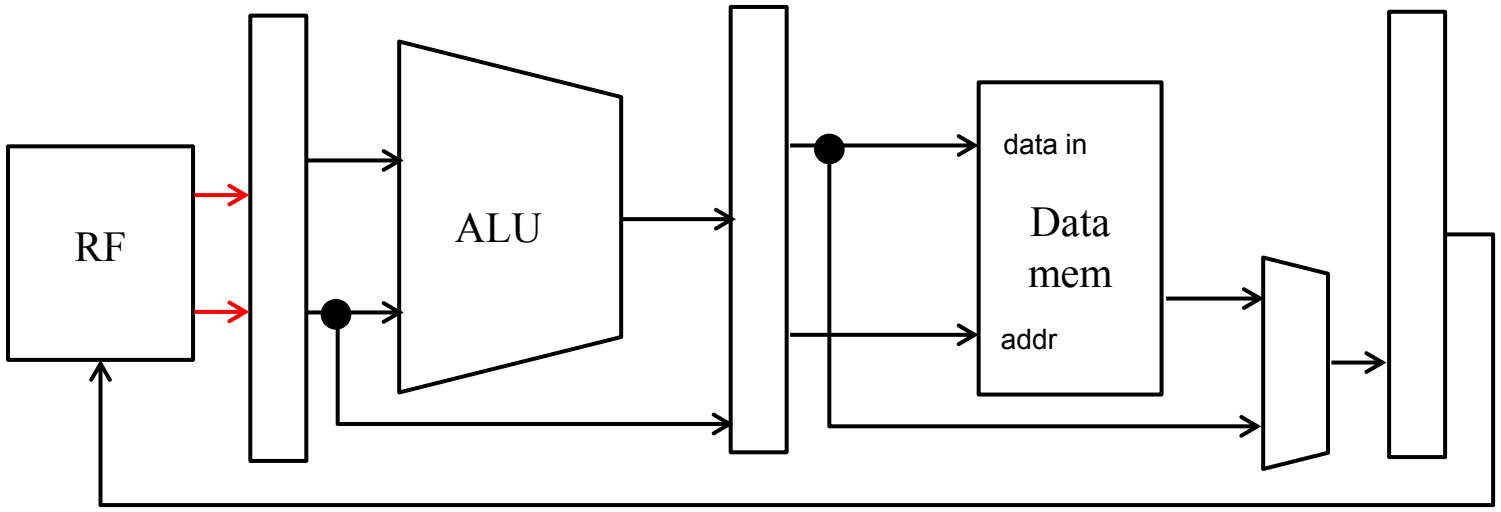
- ALU instruction by stages



Pipeline

Embedded Systems Research Group

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



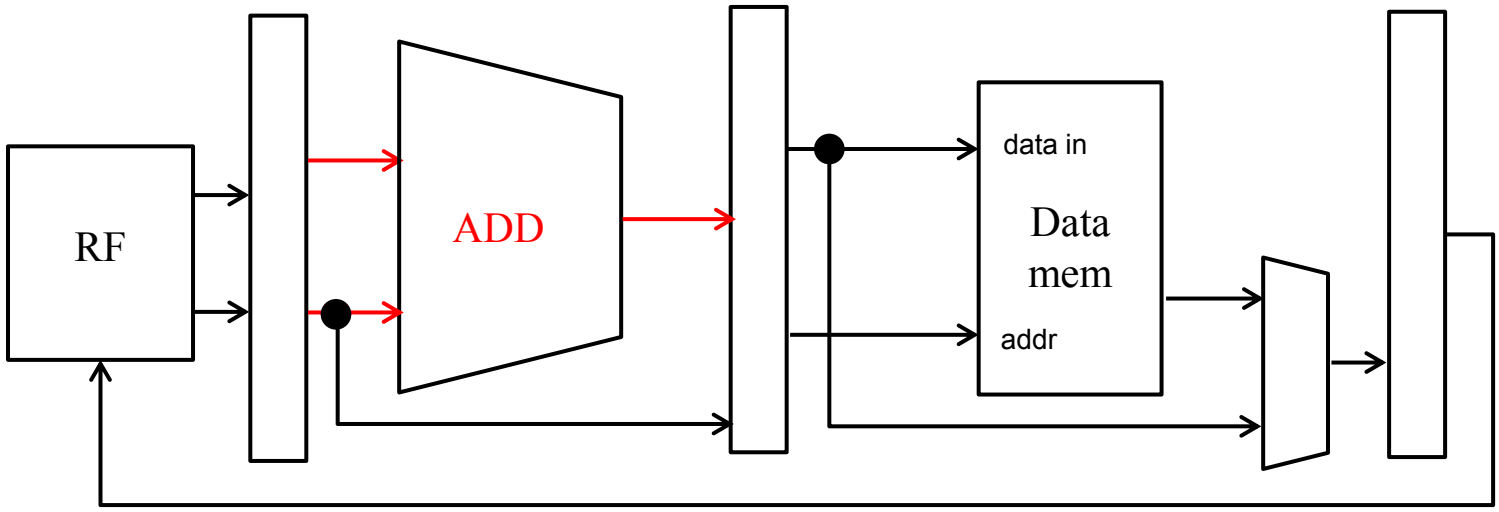
add Rd, Ra, Rb

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

Pipeline

Embedded Systems Research Group

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

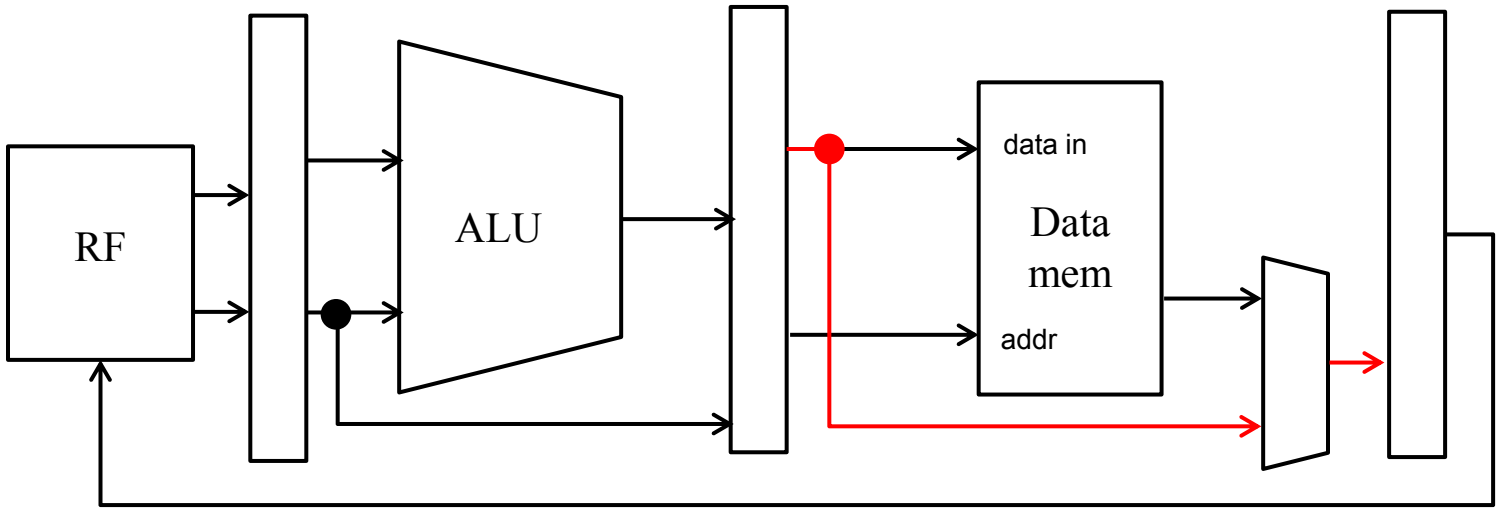


add Rd, Ra, Rb

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

Pipeline

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

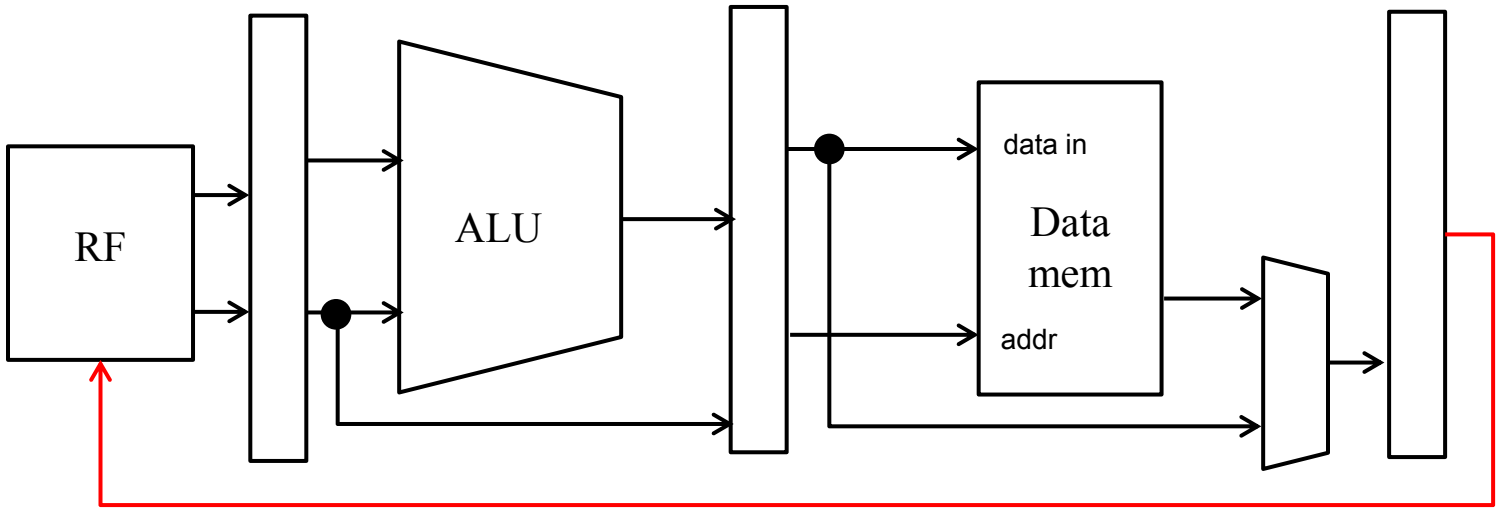


add Rd, Ra, Rb

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

Pipeline

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



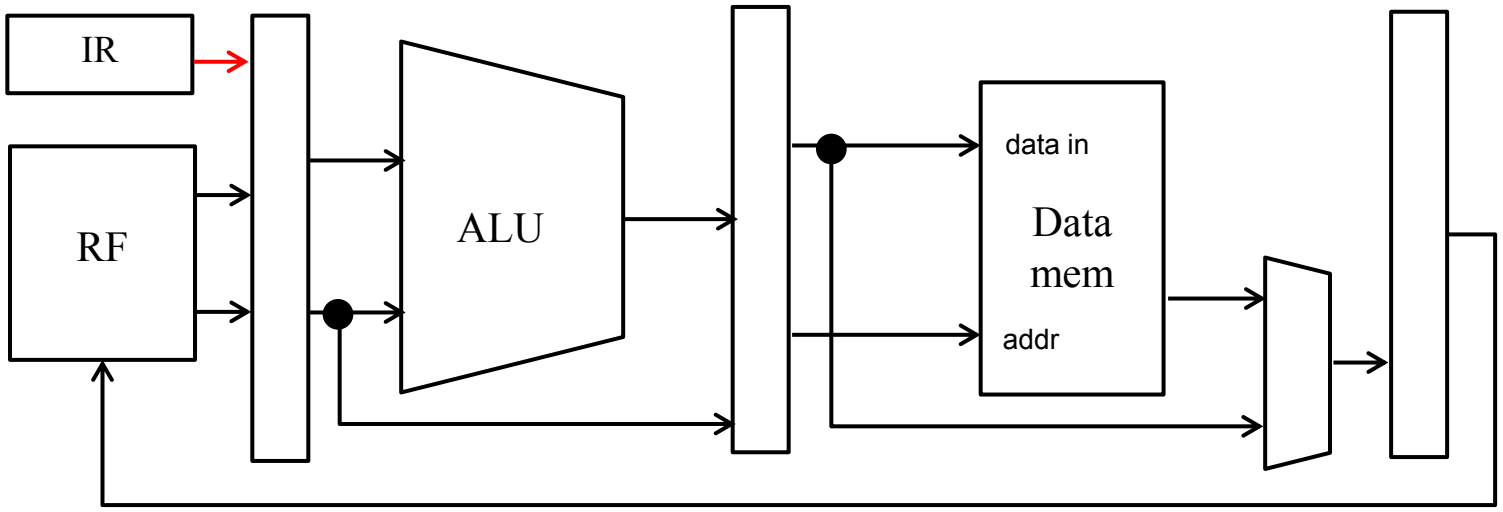
add Rd, Ra, Rb

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

Pipeline

Embedded Systems Research Group

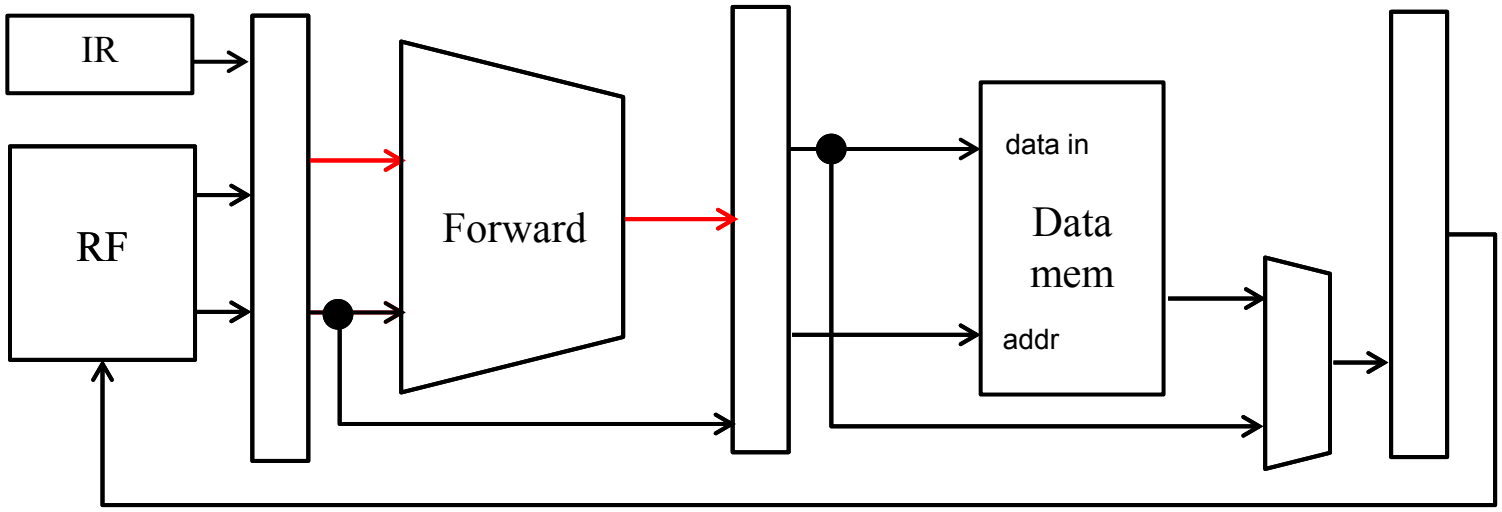
- Loadi Rd, value
  - 1st & 2nd stages:
    - Fetch instruction and Read Operands



loadi Rd, imm

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

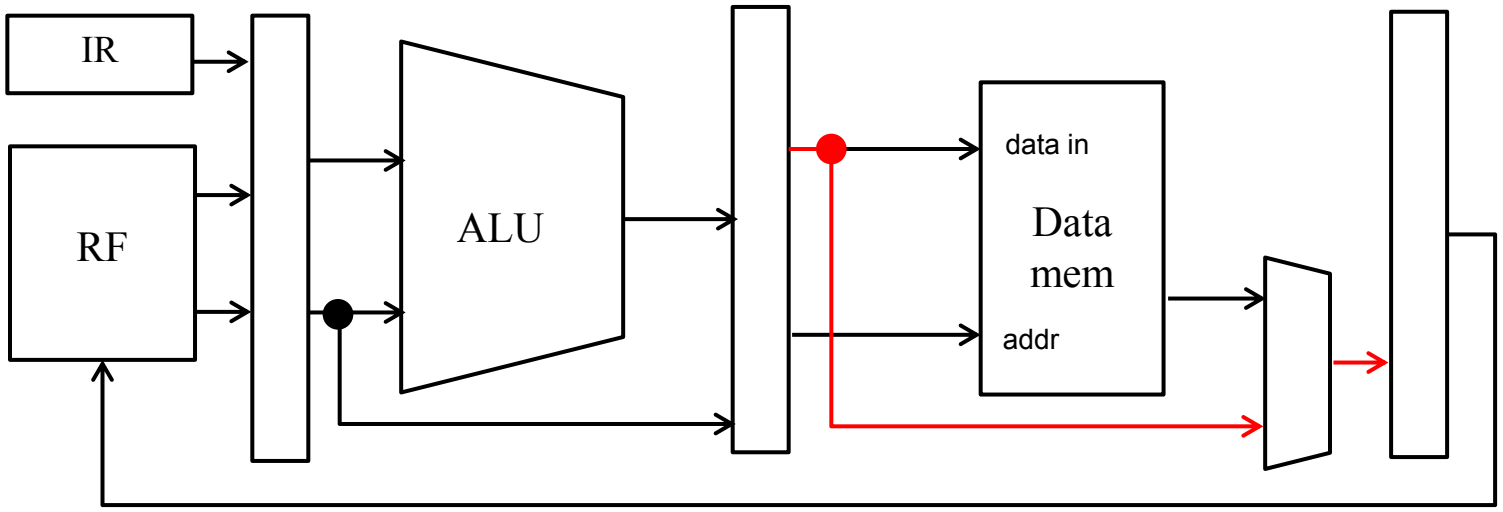
- Loadi Rd, value
  - 3rd stage:



loadi Rd, imm

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

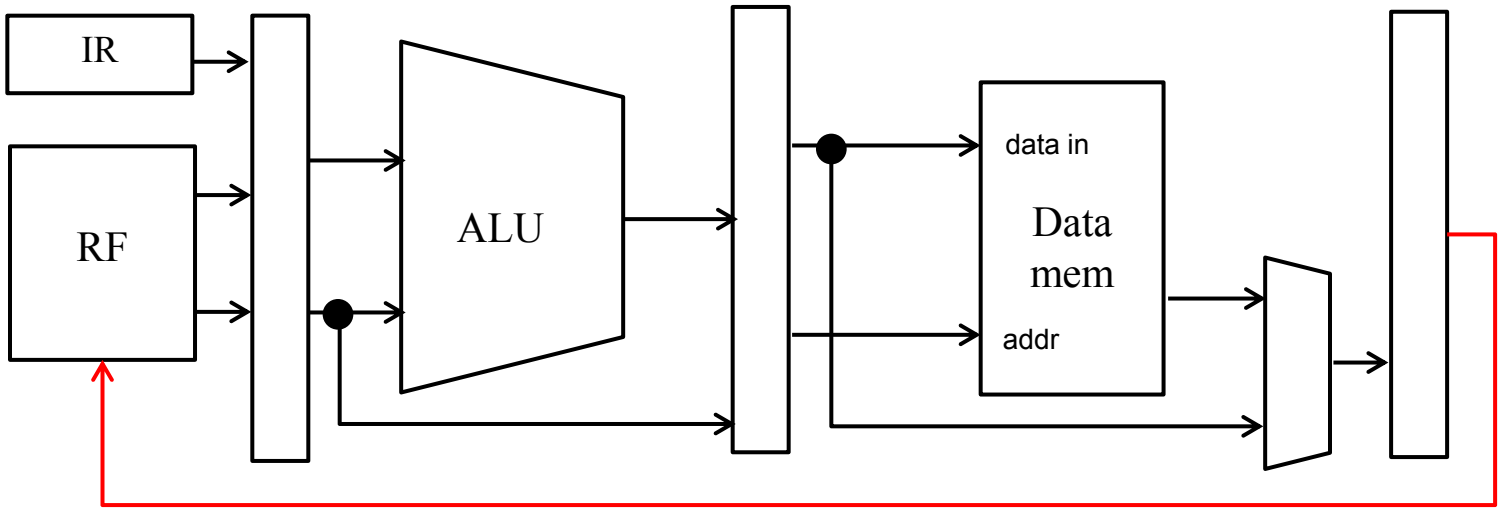
- Loadi Rd, value
  - 4th stage:



loadi Rd, imm

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

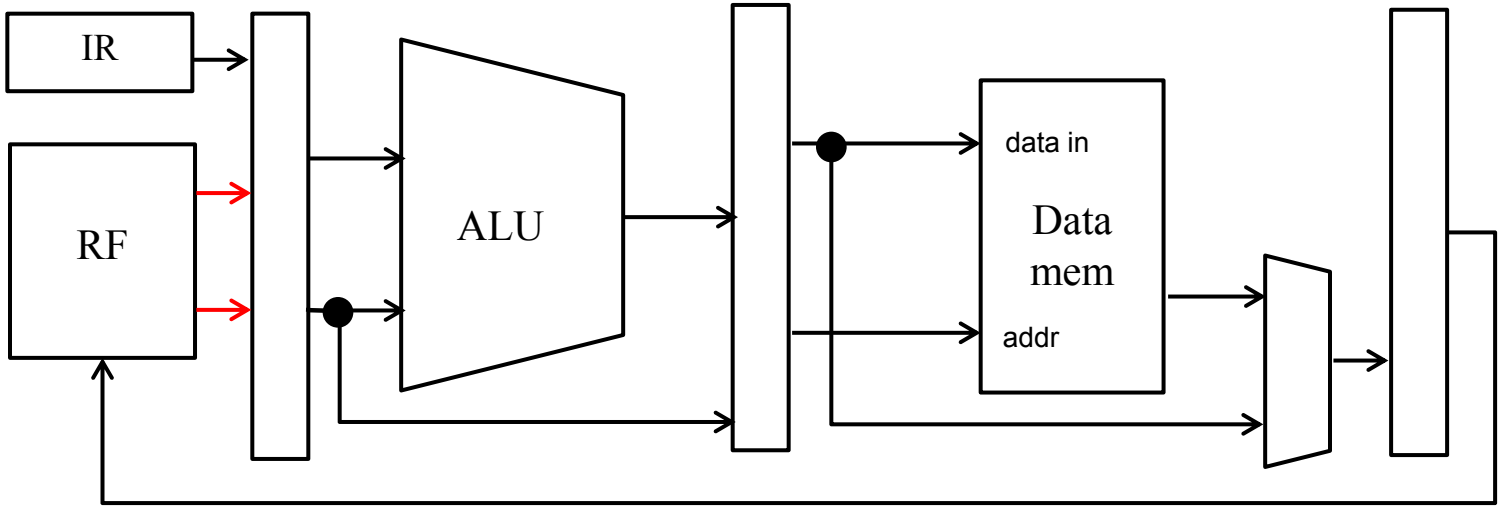
- Loadi Rd, value
  - 5th stage:
    - Write-back the result



loadi Rd, imm

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

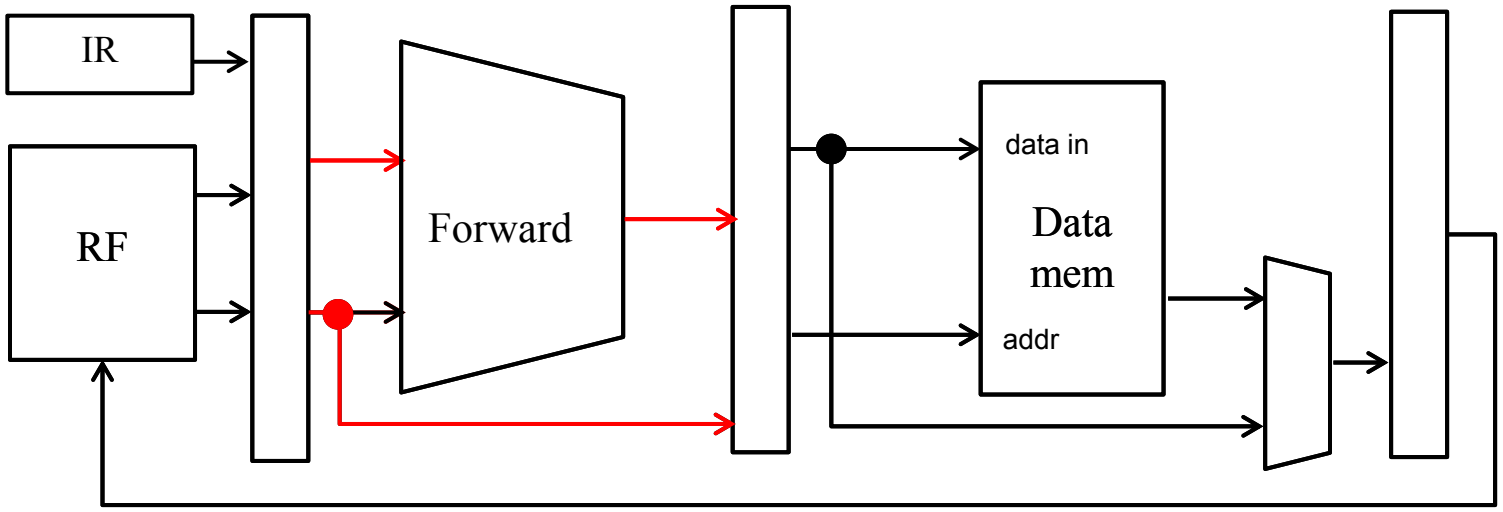
- Store Ra, @Rb (mem[Rb] = Ra)
  - 1st & 2nd stages:
    - Fetch instruction and Read Operands



store Ra, @Rb

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

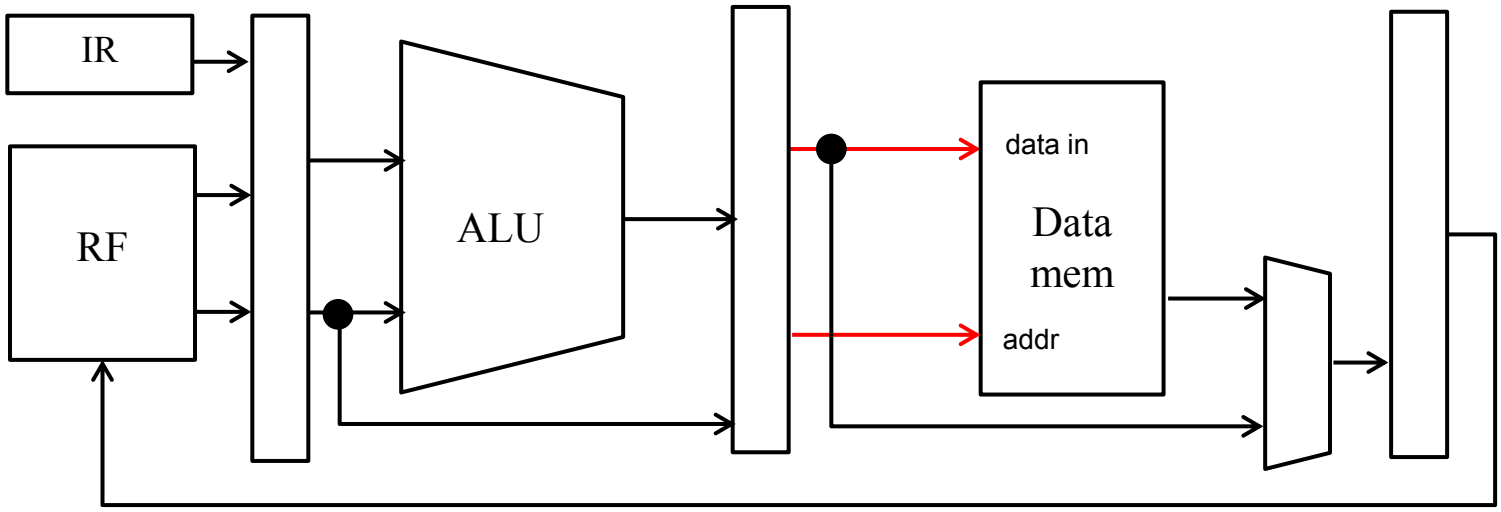
- Store Ra, @Rb (mem[Rb] = Ra)
  - 3rd stage:



store Ra, @Rb

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

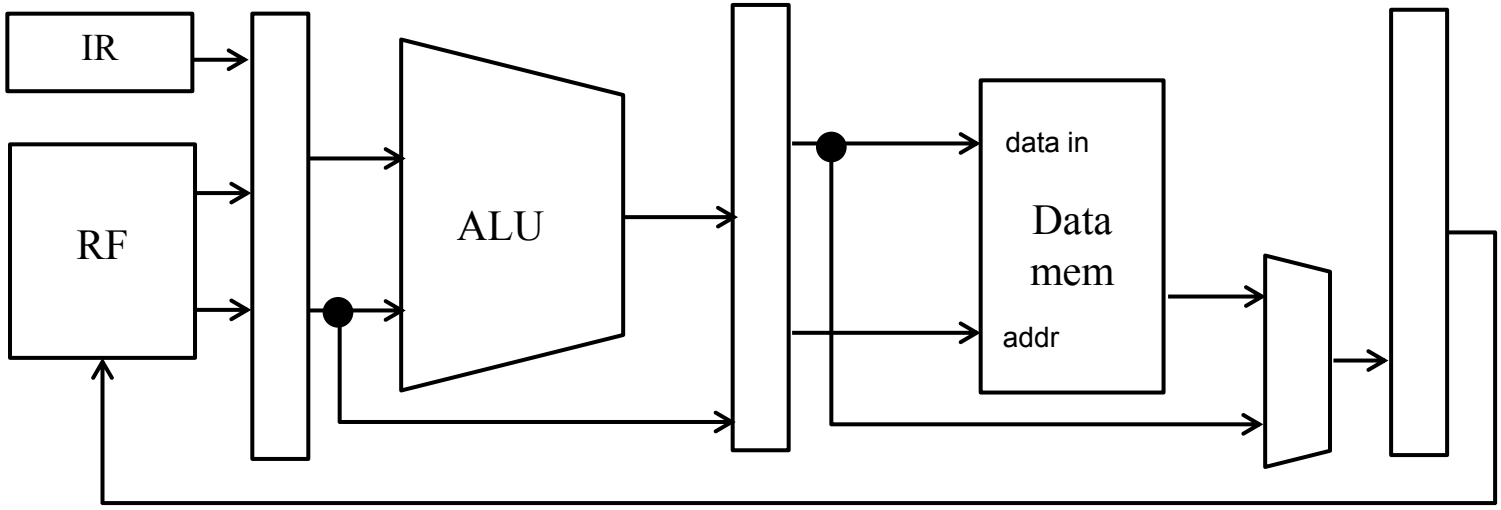
- Store Ra, @Rb (mem[Rb] = Ra)
  - 4th stage:



store Ra, @Rb

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

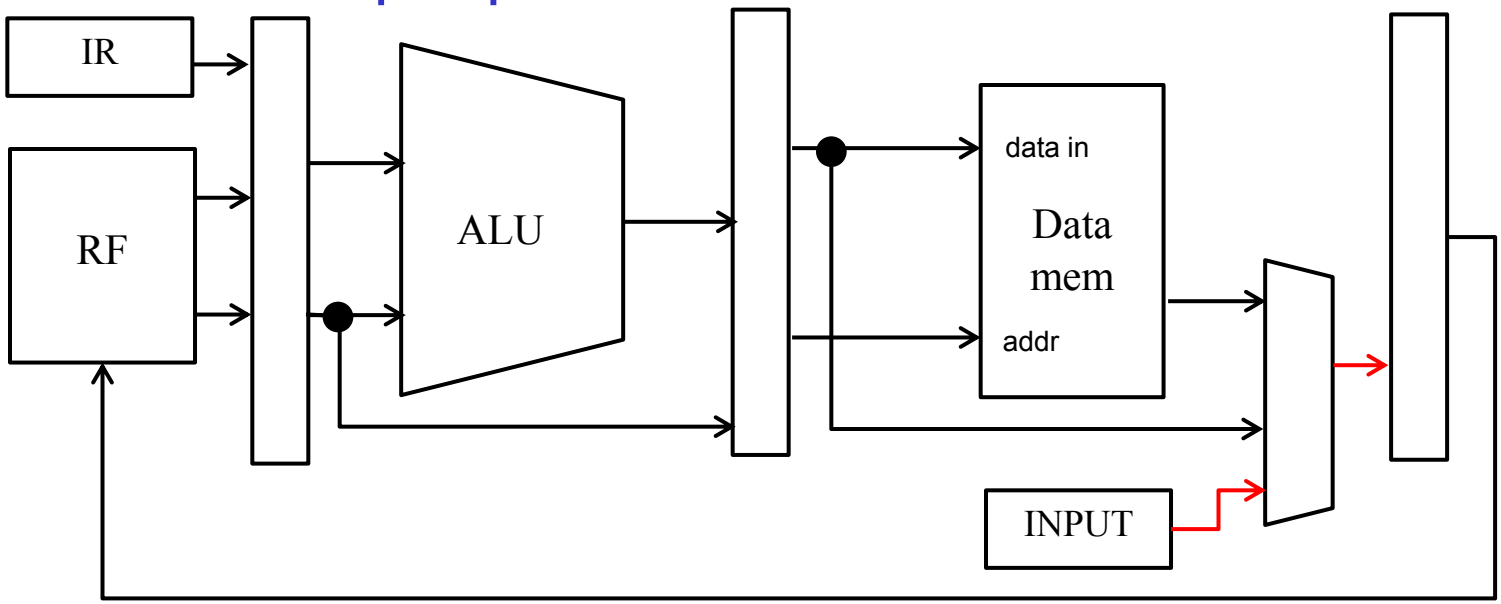
- Store Ra, @Rb (mem[Rb] = Ra)
  - 5th stage:
    - Do nothing



store Ra, @Rb

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

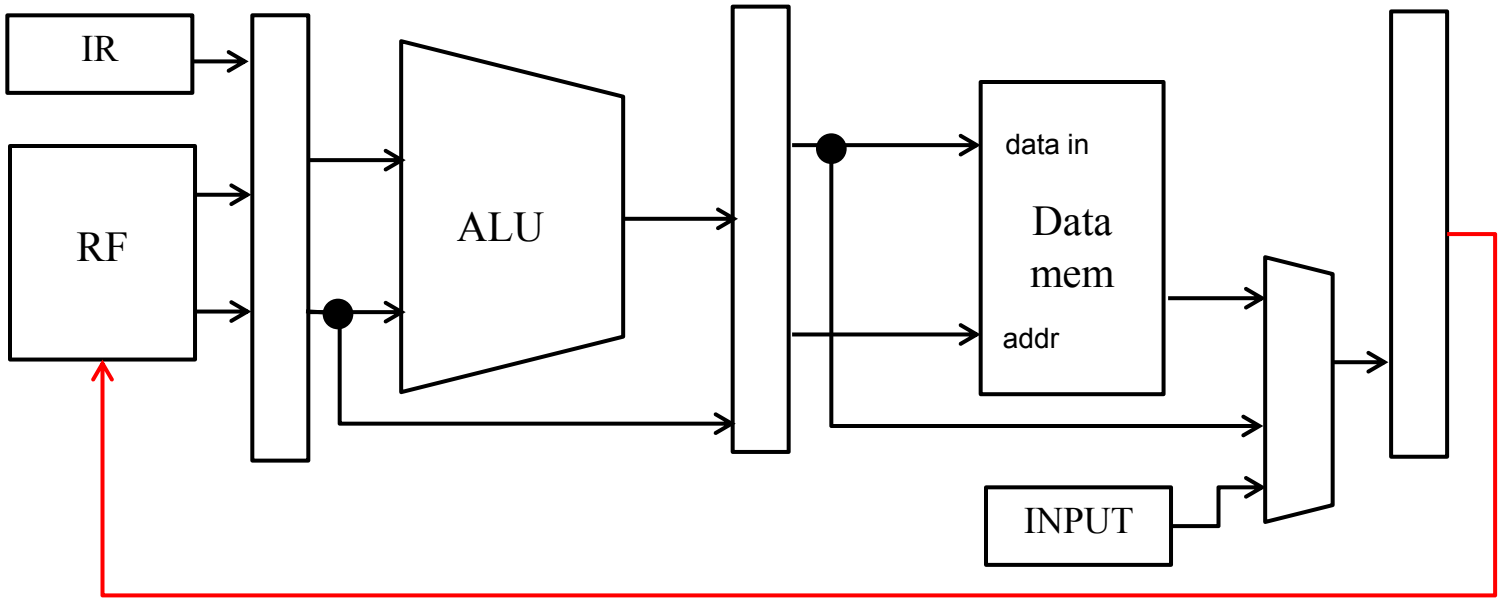
- Input Rd (Rd = Input)
  - 4th stage:
    - Read Input pins



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

Embedded Systems Research Group

- Data Hazards
- Control Hazards
- Structural Hazards

- Let's run this code

```
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?

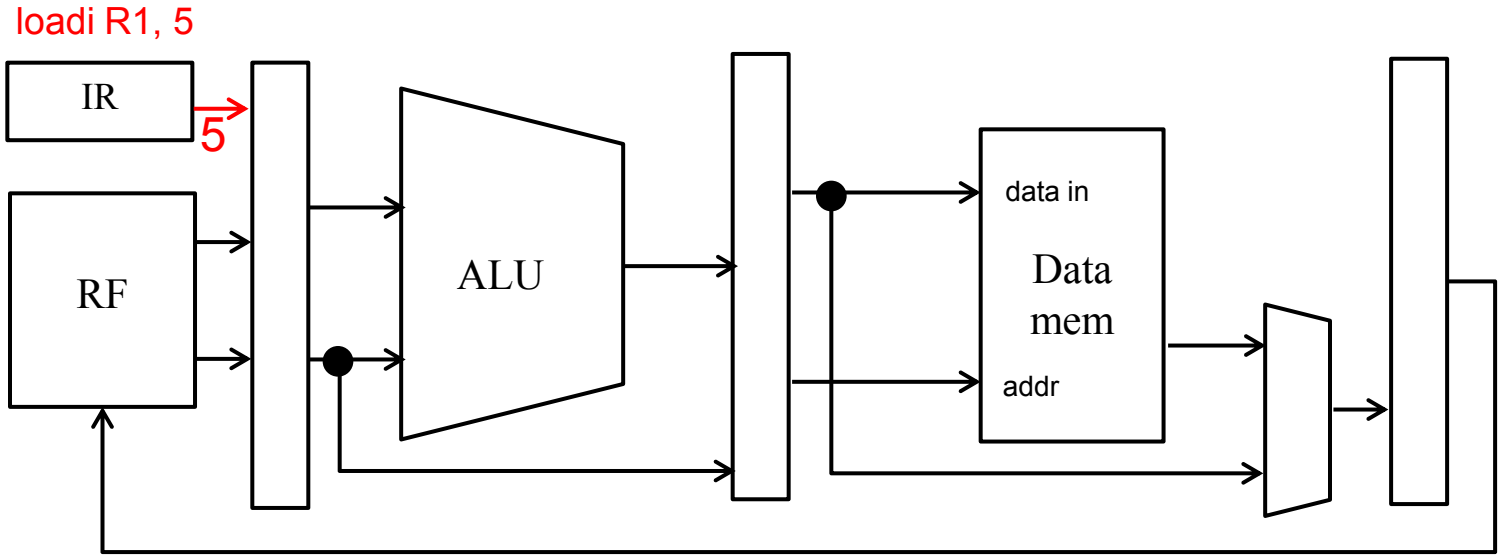
```
0    input   R0              (R0 = input = 1)
1    loadi   R2, 8           (R2 = 8)
2    loadi   R6, 0           (R6 = 0)
3    loadi   R7, 50          (R7 = 50)
4    loadi   R1, 5           (R1 = 5)
5    add     R1, R1, R0      (R1 = 5 + 1 = 6)
6    store   R1, @R7         (mem[50] = 6)
7    load    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



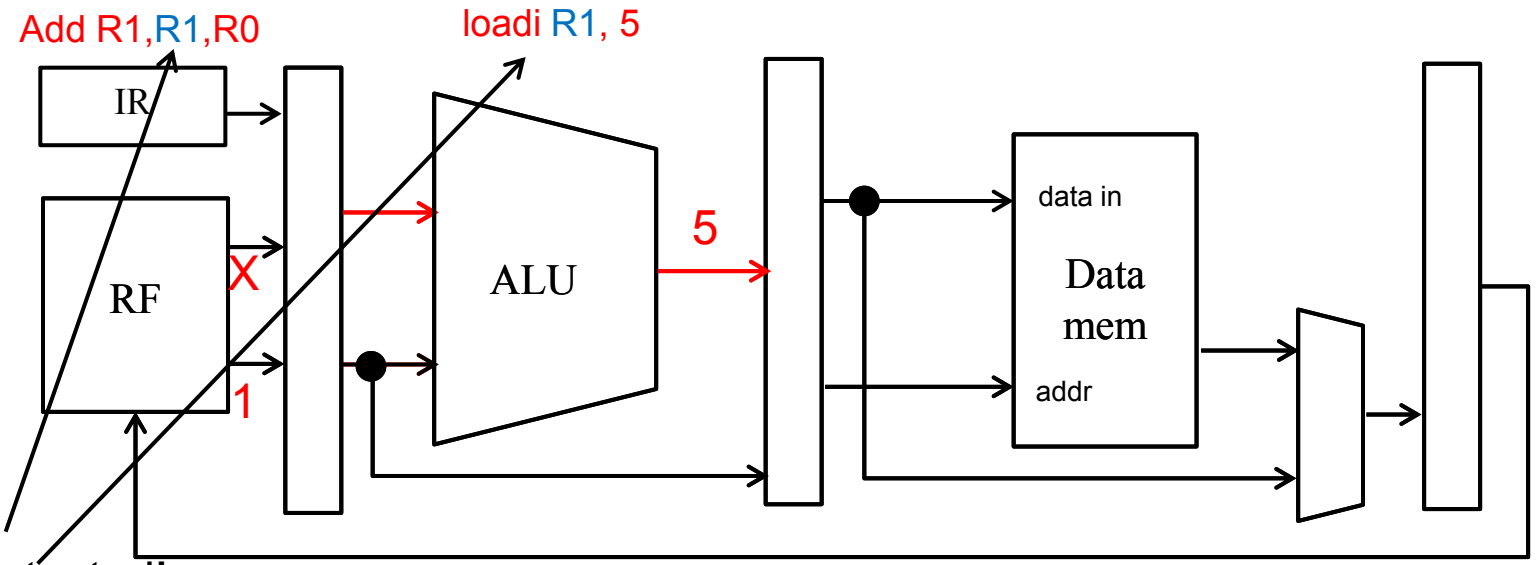
loadi Rd, imm

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



Hazard Detected!

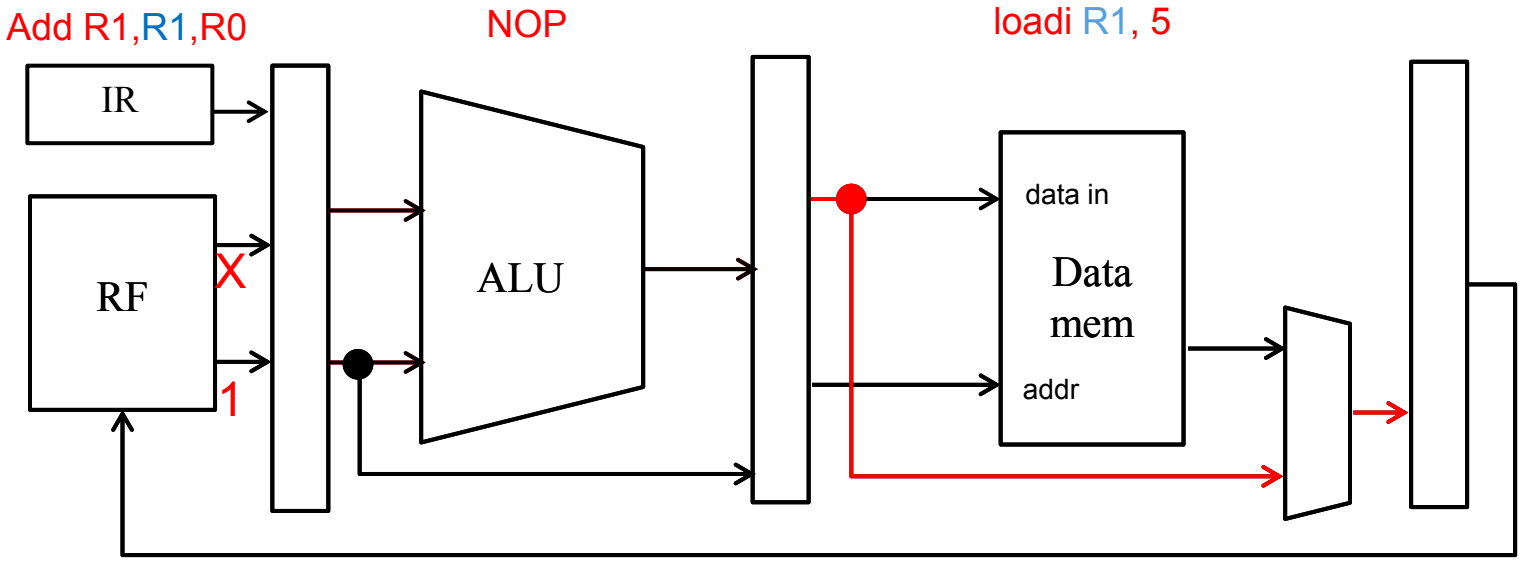
loadi Rd, imm

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



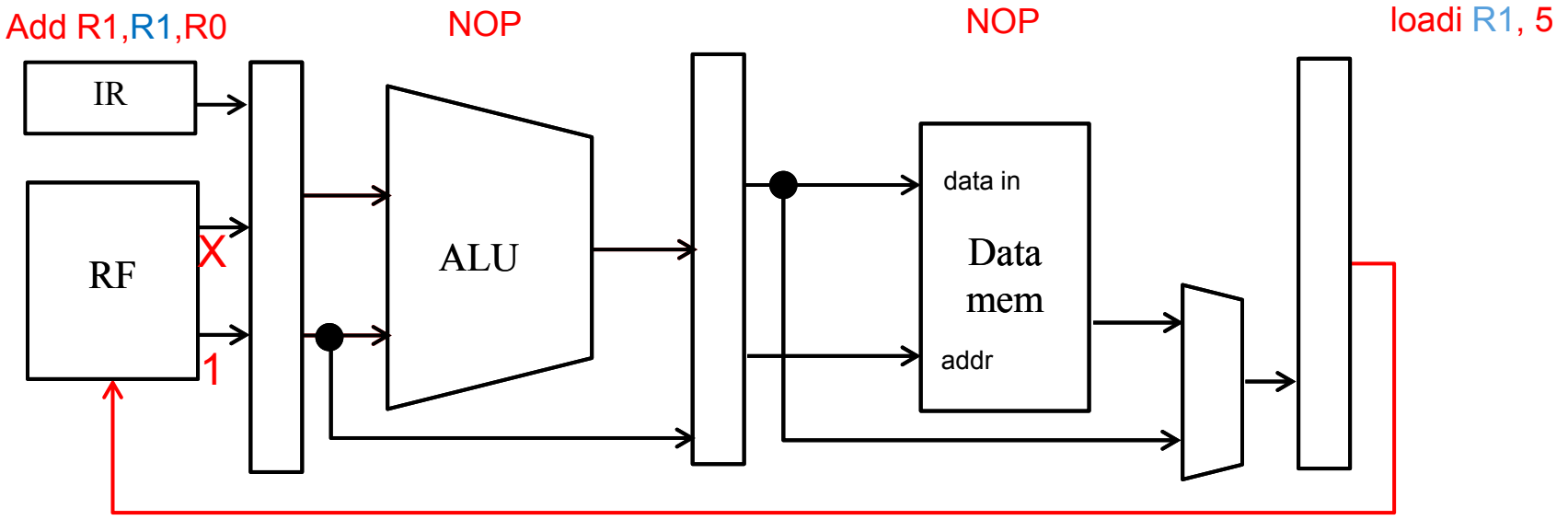
loadi Rd, imm

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



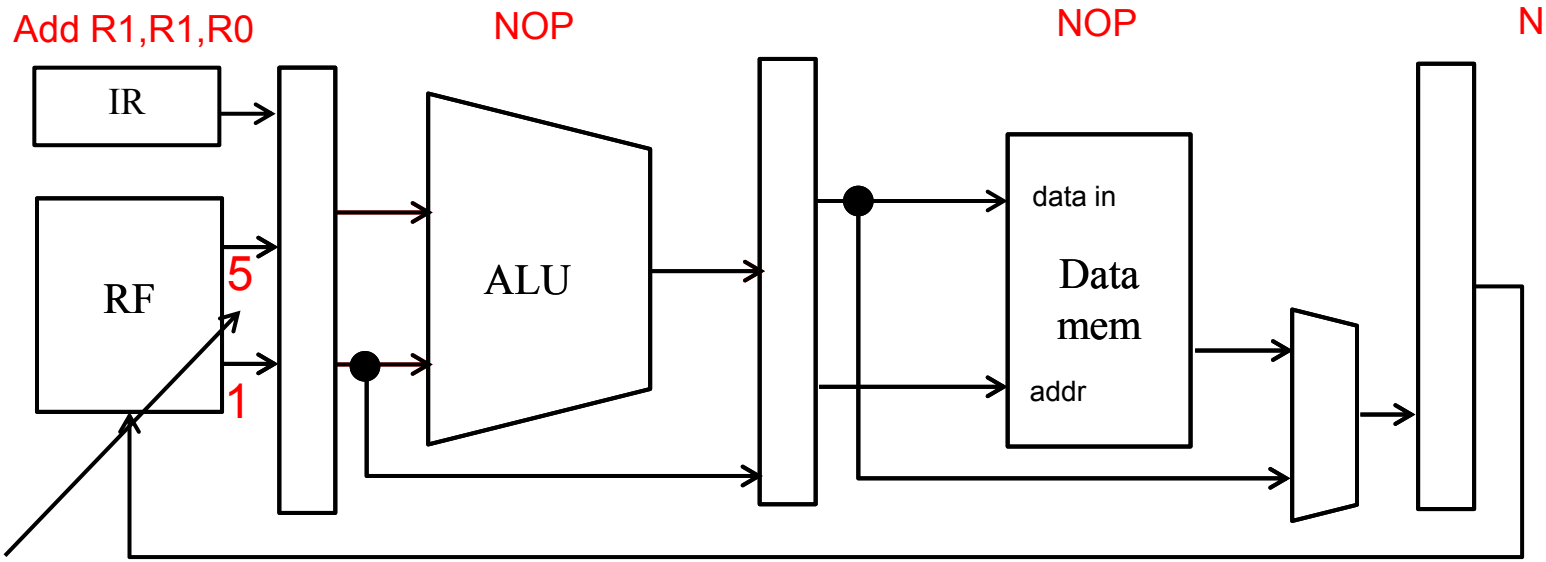
loadi Rd, imm

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



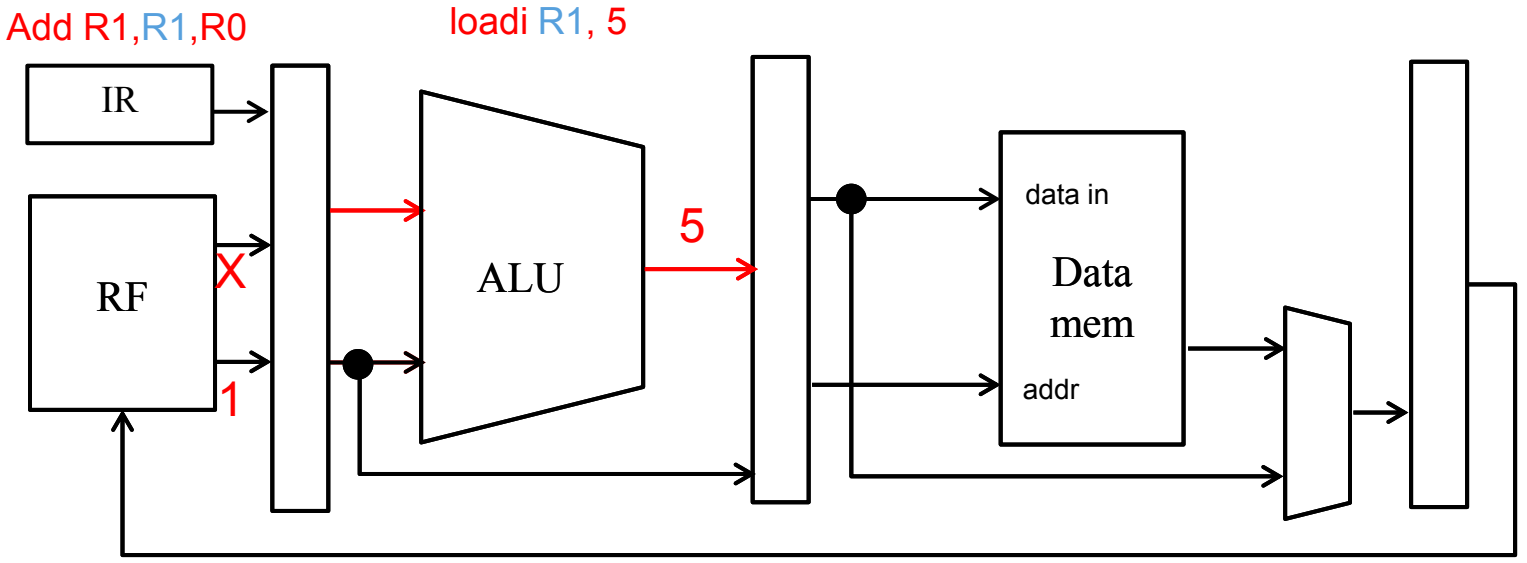
loadi Rd, imm

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

• Second solution: Data Forward

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

Data Hazard



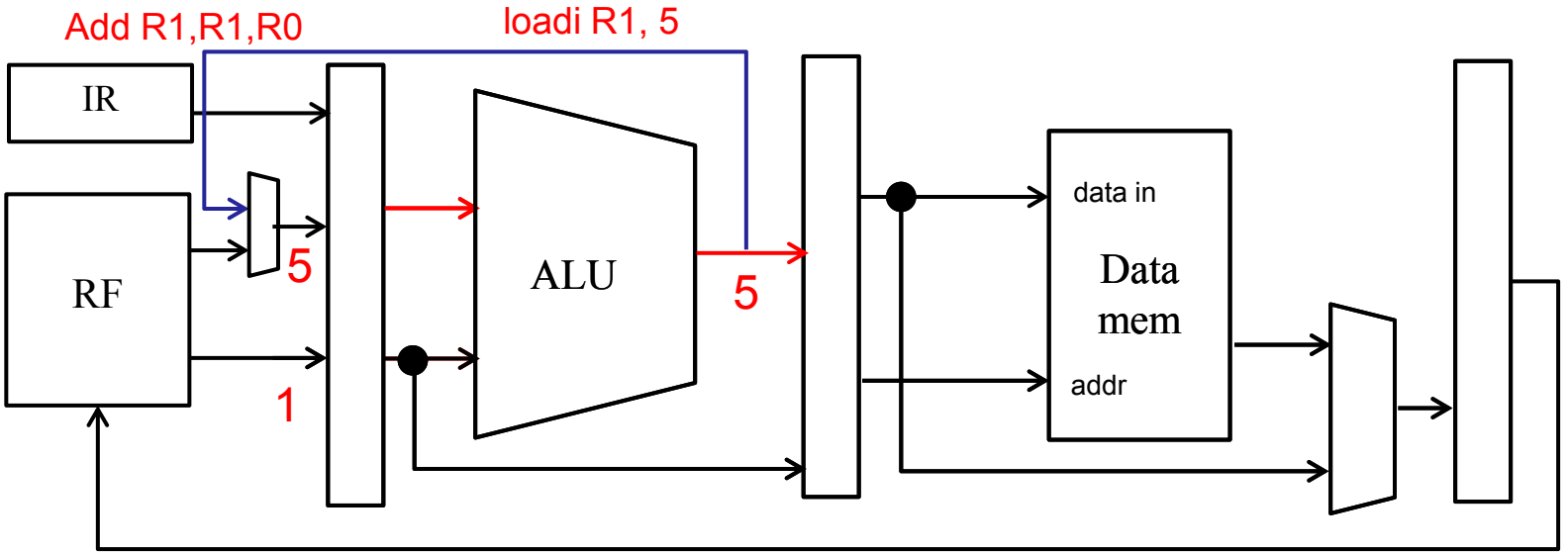
loadi Rd, imm

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)
```

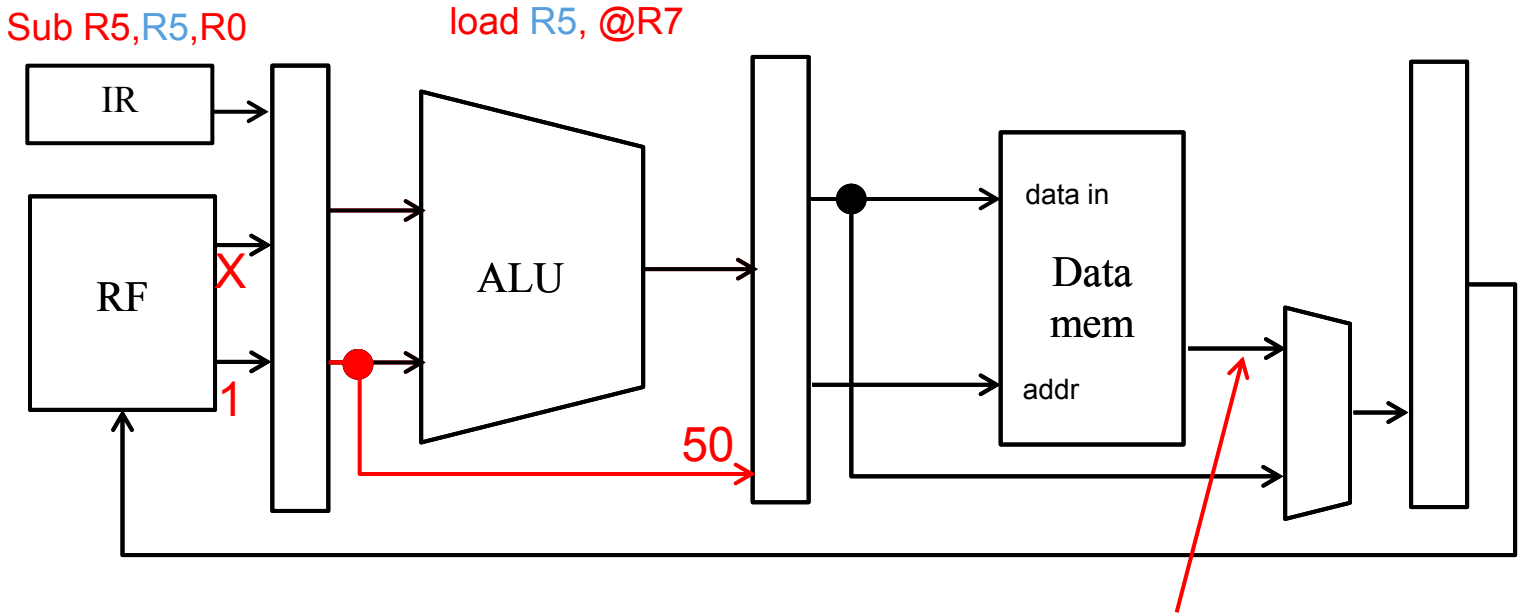
Data Hazard

Can we solve this hazard with forwarding mechanism?

- Second solution: Data Forward

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

Data Hazard

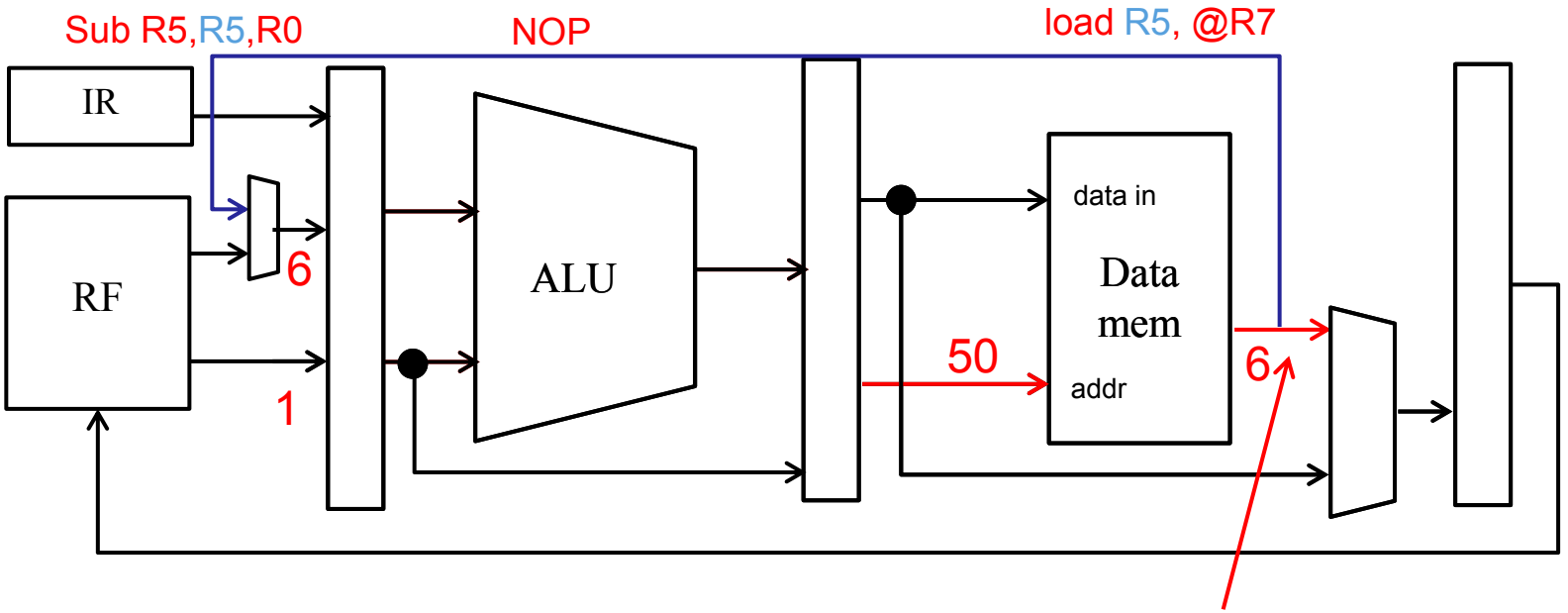


R5 is only available here!!! The pipeline has to be stalled

• Second solution: Data Forward

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

Data Hazard



- How to solve this hazard?

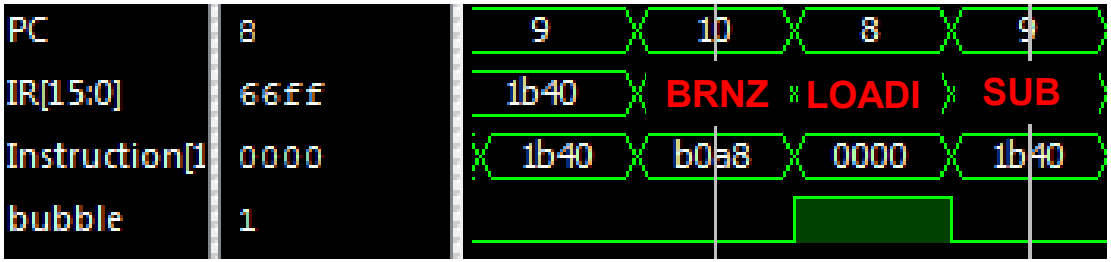
```
0    input   R0              (R0 = input = 1)
1    loadi   R2, 8           (R2 = 8)
2    loadi   R6, 0           (R6 = 0)
3    loadi   R7, 50          (R7 = 50)
4    loadi   R1, 5           (R1 = 5)
5    add     R1, R1, R0      (R1 = 5 + 1 = 6)
6    store   R1, @R7         (mem[50] = 6)
7    load    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 & 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

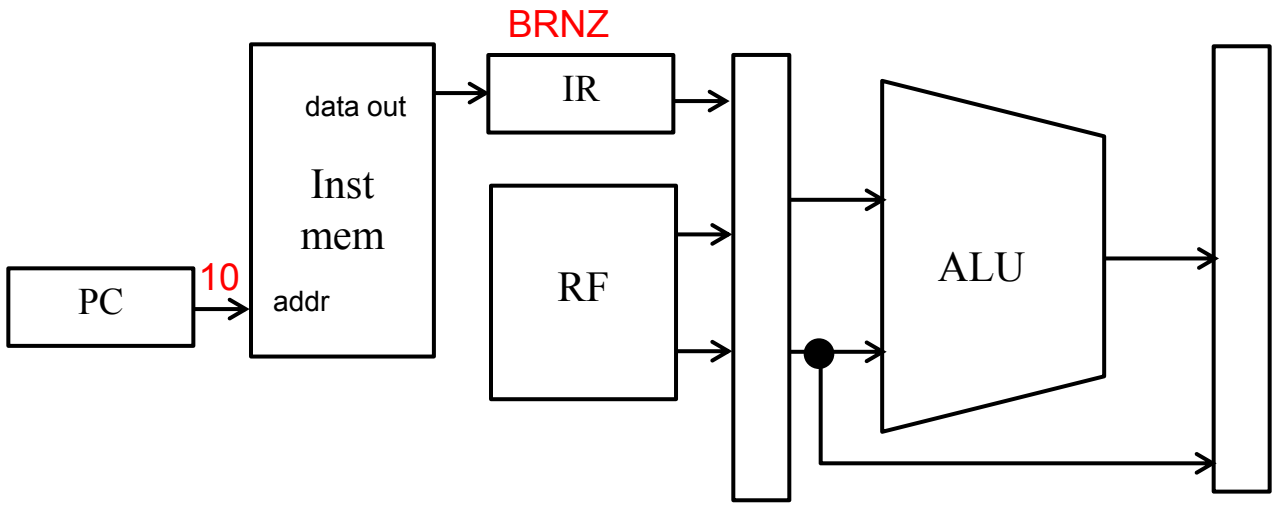


When the BRNZ instruction is fetched, the PC already points to 10, so in the next posedge Clk, the LOADI will be fetched.
However the instruction that should be executed is the SUB instruction, since the jump condition is true

To solve this hazard a bubble is introduced

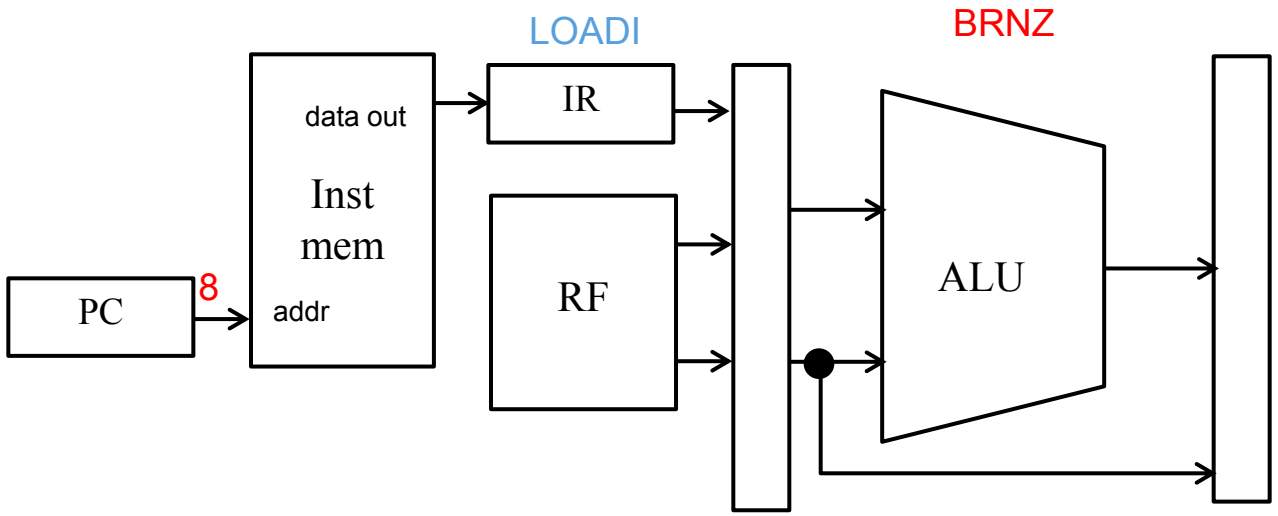
```
8    sub     R5, R5, R0      (R5 = R5 - 1)
9    brnz    R5, @R2         (if(R5!=0) PC = 8)
10   loadi   R3, 255         (R3 = 255)
```

Control Hazard



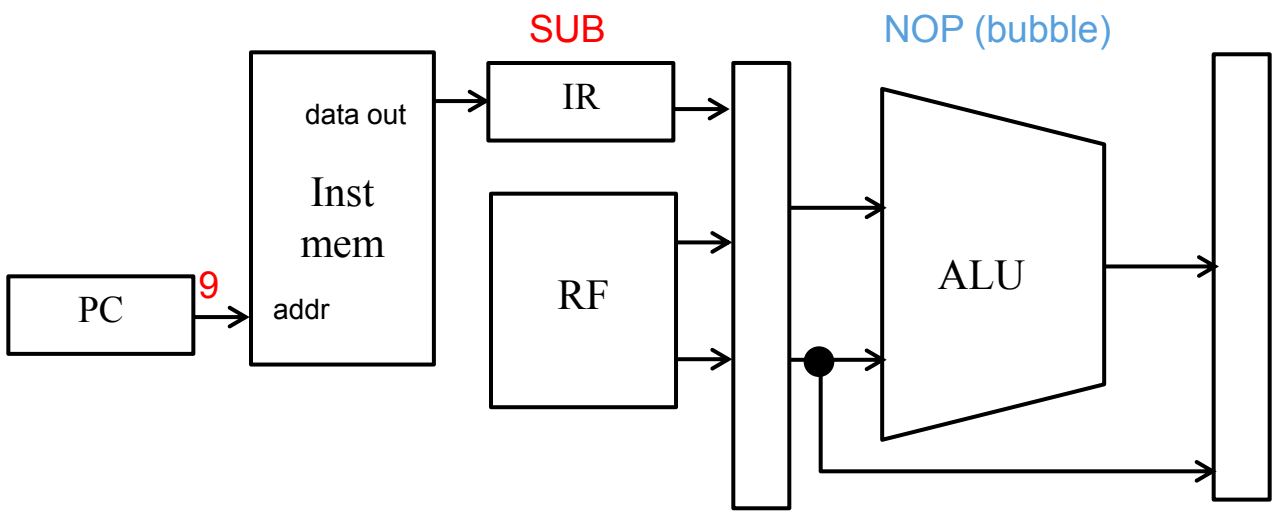
```
8    sub     R5, R5, R0      (R5 = R5 - 1)
9    brnz    R5, @R2         (if(R5!=0) PC = 8)
10   loadi   R3, 255         (R3 = 255)
```

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)
```

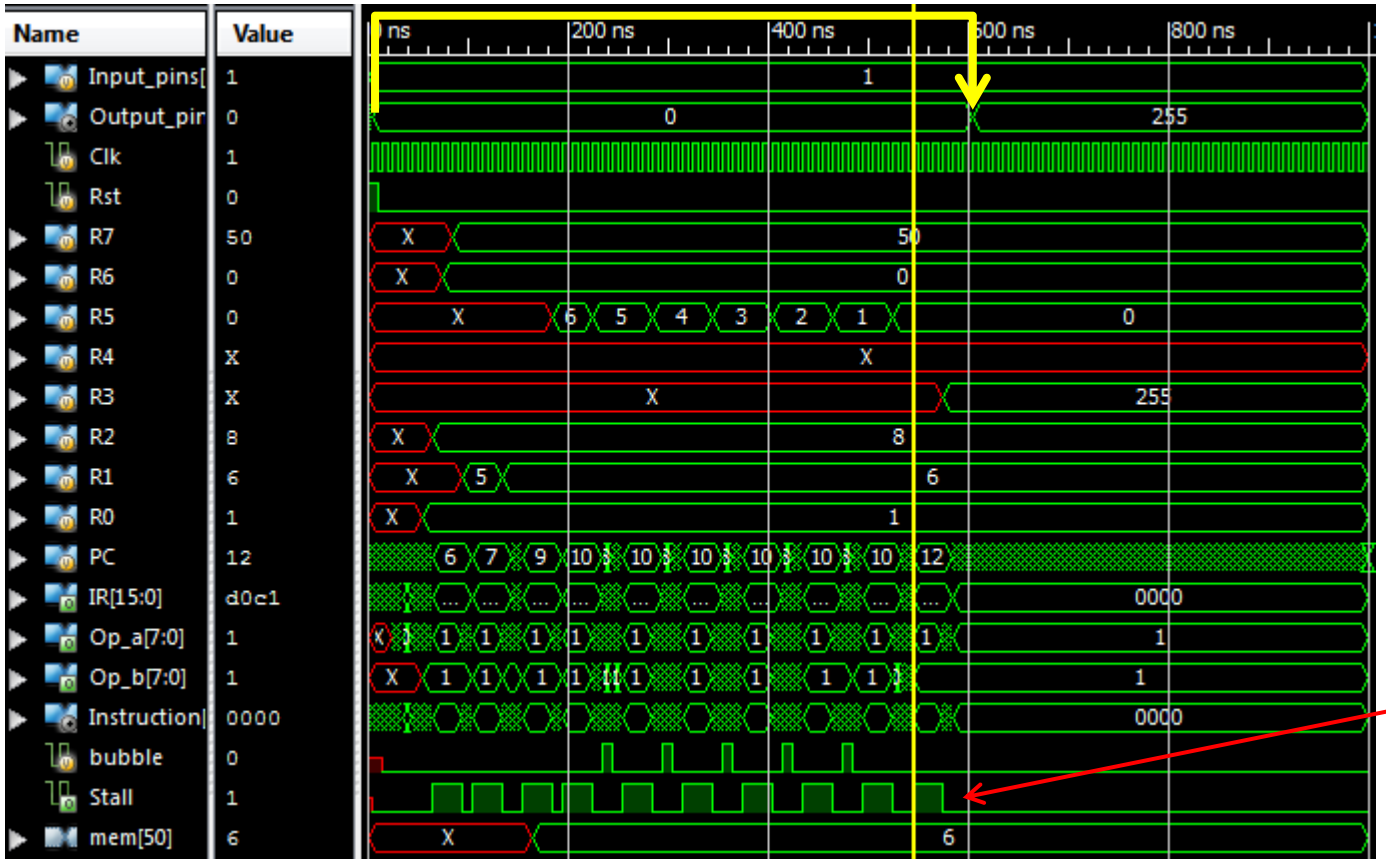
Data Hazard

Also solved with data forwarding!

Results

Embedded Systems Research Group

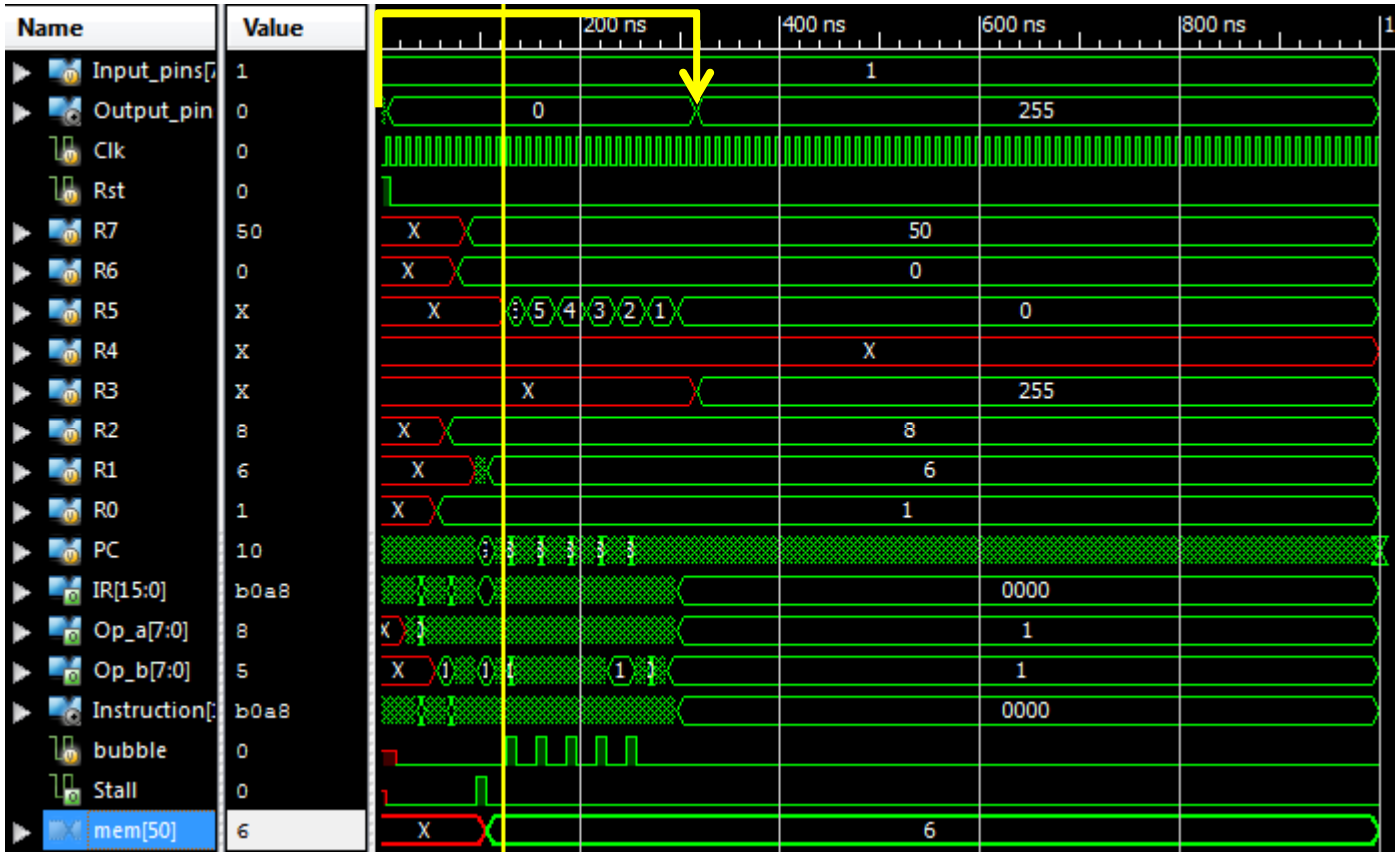
- Pipeline without data forwarding executes this code in 605ns



Results

Embedded Systems Research Group

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



Future work

Embedded Systems Research Group

- Extend the microprocessor
  - Add more instructions
    - Mov between Registers
      - MOV R3, R4
    - Other branch conditions
      - BRN, BRC
    - Halt
      - Stop execution
  - Add support to Jump and Link
- Develop Assembler

- THE END