RISC PROCESSOR DESIGN ON FPGA

Istanbul Technical University
Computer Engineering Department
Graduation Project

Tuğrul YATAĞAN
040100117

Advisor: Assoc. Prof. Dr. Mustafa Ersel Kamaşak

Outline

- Introduction
- Processor properties
- Processor organization
- Instruction set architecture
- Processor pipeline
- Pipeline hazards & solutions
- Assembler
- FPGA implementation
- Performance

Introduction

In this Project;

- A 16-Bit RISC soft processor is designed, simulated and implemented on Xilinx FPGA with Verilog hardware description language.
- An assembler is developed for soft processor with Python.

Advisor: Assoc. Prof. Dr. Mustafa Ersel Kamaşak

Former Advisor: Prof. Dr. Ahmet Coşkun Sönmez

Properties

- RISC architecture
- Mixture of THUMB and MIPS commercial architectures
- Harvard architecture so it has separate instruction and data memory
- Fixed length Load/Store instruction set architecture which means only load and store instructions can access to data memory
- Single cycle instruction execution capability, easier pipeline design

Organization - 1

- 16 Bit CPU, ALU, register and data bus architecture
- Address bus for data and instruction memory is 12 bit so maximum addressable instruction and data memory is 4K word and it is equal to 8KB
- 8 general purpose 16 bit registers in register file
- 16 bit instruction and condition code register
- 12 bit program counter and address register
- All general purpose registers are accessible
- PC can only be modifiable via branch instructions, its content is not accessible
- IR content is not accessible

Organization - 2

Condition code register

Bit	Equivalent
N	Negative/Less than
Z	Zero
C	Carry/Borrow
V	Overflow
IEN	Interrupt Enable
IRQ	Interrupt Request

Registers

Abbreviation	Equivalent	Bit
R0	General purpose register 0	16
R1	General purpose register 1	16
R2	General purpose register 2	16
R3	General purpose register 3	16
R4	General purpose register 4	16
R5	General purpose register 5	16
R6	General purpose register 6	16
R7	General purpose register 7	16
IR	Instruction register	16
PC	Program counter	12
AR	Address register	12
CCR	Condition code register	16

Instruction Set Format

- All instructions are16 Bit fixed length instruction.
- There are 11 type of instruction.
- Load/Store instruction set architecture, so only load and store instructions can access to data memory.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	X	X	X	X	X	X	X	X	X	Op		
2	0	0	0	1	Op			Off4				Rs		Rd		
3	0	0	1	Op			Off6					Rs		Rd		
4	0	1	0	Op				Off8						Rd		
5	0	1	1	Op				Rn			Rs			Rd		
6	1	0	0	X	X	X	Op	Ro			Rb			Rd		
7	1	0	1	Op			Off6				Rb			Rd		
8	1	1	0	Cond				Off9								
9	1	1	1	0	1			Off11								
10	1	1	1	0	1			Off11								
11	1	1	1	1				Off12								

Abbreviation	Equivalent	Bit
Rd	Destination Register	3
Rs	Source Register	3
Rb	Base Register	3
Ro	Offset Register	3
Off#	Immediate Offset	#
Op	Operation	X
Cond	Condition	X

Instruction Types

1. Inherent Instructions
2. Shift Instructions
3. Add/Subtract Register Immediate Instructions
4. Move/Compare/Add/Subtract Immediate Instructions
5. ALU Instructions
6. Load/Store with Register Offset Instructions
7. Load/Store with Immediate Offset Instructions
8. Conditional Branch Instructions
9. Unconditional Branch Instruction
10. Branch to Subroutine Instruction
11. Move Long Immediate

Instruction Set - 1

Assembly	Action	Updates
NOP	No operation	
ION	Interrupts on	I
IOF	Interrupts off	I
RTI	Return from interrupt	
RTS	Return from subroutine	
LSL Rd, Rs, #Off4	Logic shift left by #Off4	N Z C
LSR Rd, Rs, #Off4	Logic shift right by #Off4	N Z C
ASR Rd, Rs, #Off4	Arithmetic shift right by #Off4	N Z C
CSL Rd, Rs, #Off4	Circular shift left by #Off4	N Z C
ADD Rd, Rs, #Off6	Rd := Rs + #Off6	N Z C V
SUB Rd, Rs, #Off6	Rd := Rs - #Off6	N Z C V
MOV Rd, #Off8	Rd := #Off8	N Z
CMP Rd, #Off8	Rd - #Off8	N Z C V
ADD Rd, #Off8	Rd := Rd + #Off8	N Z C V
SUB Rd, #Off8	Rd := Rd - #Off8	N Z C V

1. Inherent
2. Shift
3. Add/Subtract Register Immediate
4. Move/Compare/Add/Subtract Immediate
5. ALU

AND Rd, Rs, Rn	Rd := Rs AND Rn	N Z
OR Rd, Rs, Rn	Rd := Rs OR Rn	N Z
XOR Rd, Rs, Rn	Rd := Rs XOR Rn	N Z
LSL Rd, Rs, Rn	Rd := Rs << Rn	N Z C
LSR Rd, Rs, Rn	Rd := Rs >> Rn	N Z C
ASR Rd, Rs, Rn	Rd := Rs ASR Rn	N Z C
CSL Rd, Rs, Rn	Rd := Rs CSL Rn	N Z C
ADD Rd, Rs, Rn	Rd := Rs + Rn	N Z C V
SUB Rd, Rs, Rn	Rd := Rs – Rn	N Z C V
NEG Rd, Rs	Rd := -Rs	N Z
NOT Rd, Rs	Rd := NOT Rs	N Z
CMP Rs, Rn	Rs – Rn	N Z C V
TST Rs, Rn	Rs AND Rn	N Z C V

Instruction Set - 2

Assembly	Action
LD Rd, [Rb, Ro]	Rd := MEM[Rb + Ro]
LD Rd, [Rb, #Off6]	Rd := MEM[Rb + #Off6]
STR [Rb, #Off6], Rd	MEM[Rb + #Off6] := Rd
BTS #Off11	Branch to subroutine
BAL #Off11	Branch always

6. Load/Store with Register Offset
7. Load/Store with Immediate Offset
8. Conditional Branch
9. Unconditional Branch
10. Branch to Subroutine
11. Move Long Immediate

BEQ #Off9	Branch if equal
BNE #Off9	Branch if not equal
BCS #Off9	Branch if unsigned higher or same
BCC #Off9	Branch if unsigned lower
BMI #Off9	Branch if negative
BPL #Off9	Branch if positive or zero
BVS #Off9	Branch if overflow
BVC #Off9	Branch if no overflow
BHI #Off9	Branch if unsigned higher
BLS #Off9	Branch if unsigned lower or same
BGE #Off9	Branch if signed greater than or equal
BLT #Off9	Branch if signed less than
BGT #Off9	Branch if signed greater than
BLE #Off9	Branch if signed less than or equal

Example Instructions

- Inherent

```
NOP
```

- Shift

```
LSR  R2, R5, #10
```

- Add/Subtract Register Immediate

```
ADD  R1, R4, #60
```

- Move/Compare/Add/Subtract Immediate

```
MOV  R1, #130
```

- ALU

```
XOR  R2, R5, R3
```

- Load/Store with Register Offset

```
LD   R3, [R5, R2]
```

- Load/Store with Immediate Offset

```
STR  [R5, #26], R1
```

- Conditional Branch

```
BEQ  Label
```

- Unconditional Branch

```
BAL  Label
```

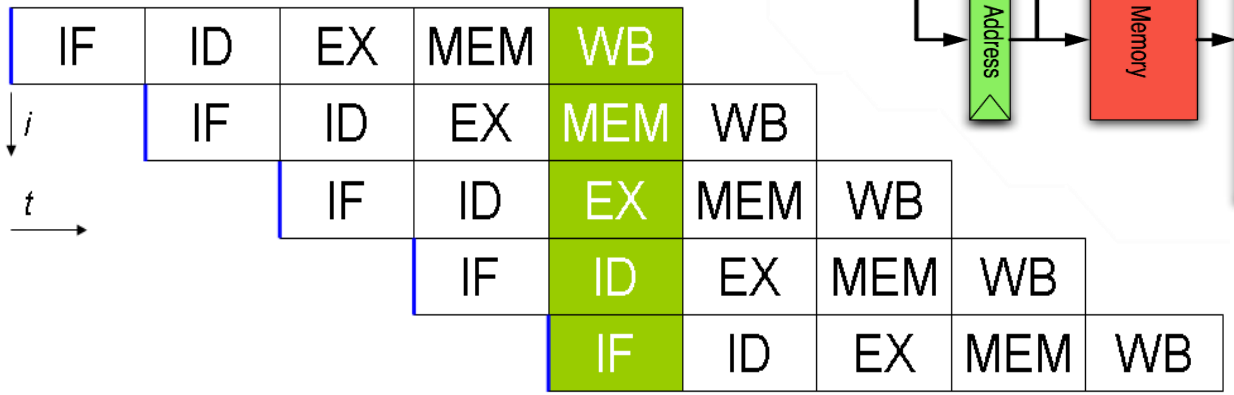
- Subroutine

```
BTS  Label
```

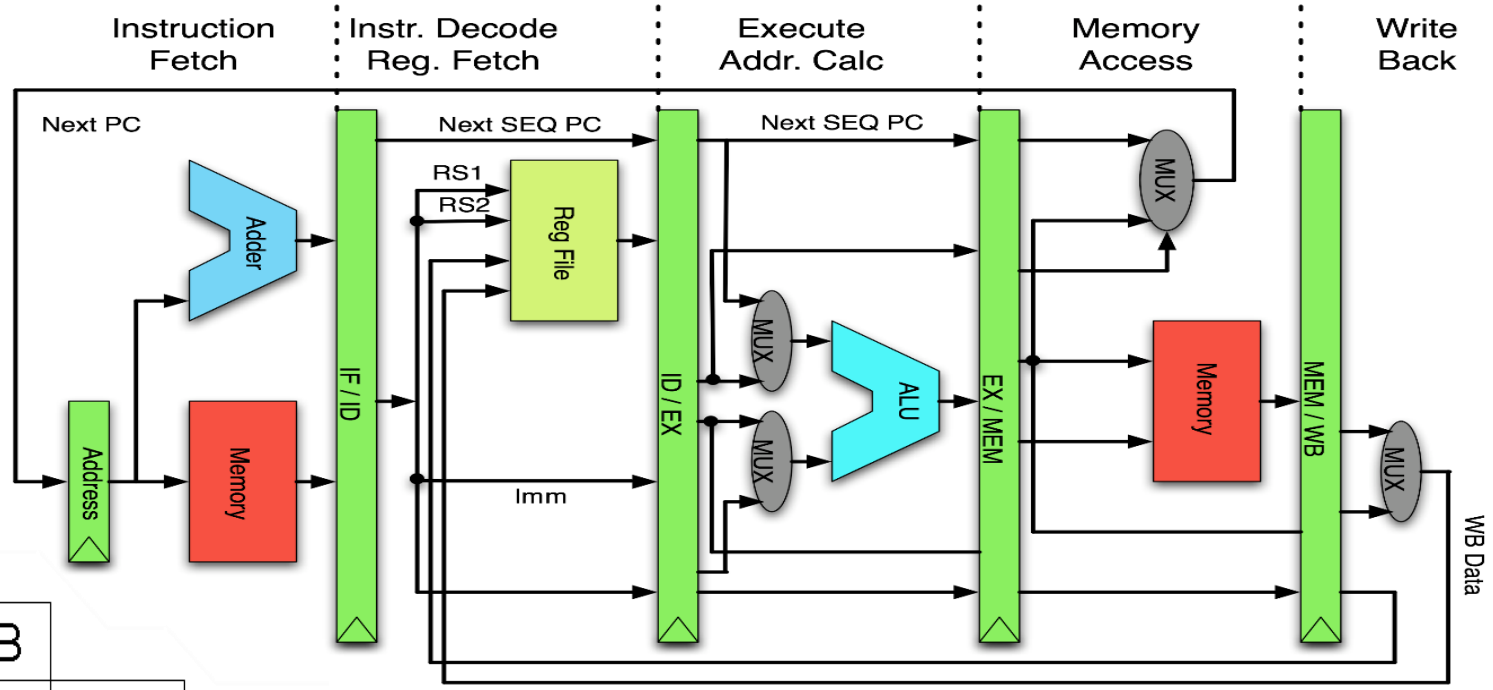
Pipeline

Classic RISC 5 stages pipeline architecture

- Instruction fetch
- Instruction decode
- Execute (ALU)
- Memory access
- Write back



5 stage classic RISC pipeline timing diagram [10]



RISC pipeline organization [10]

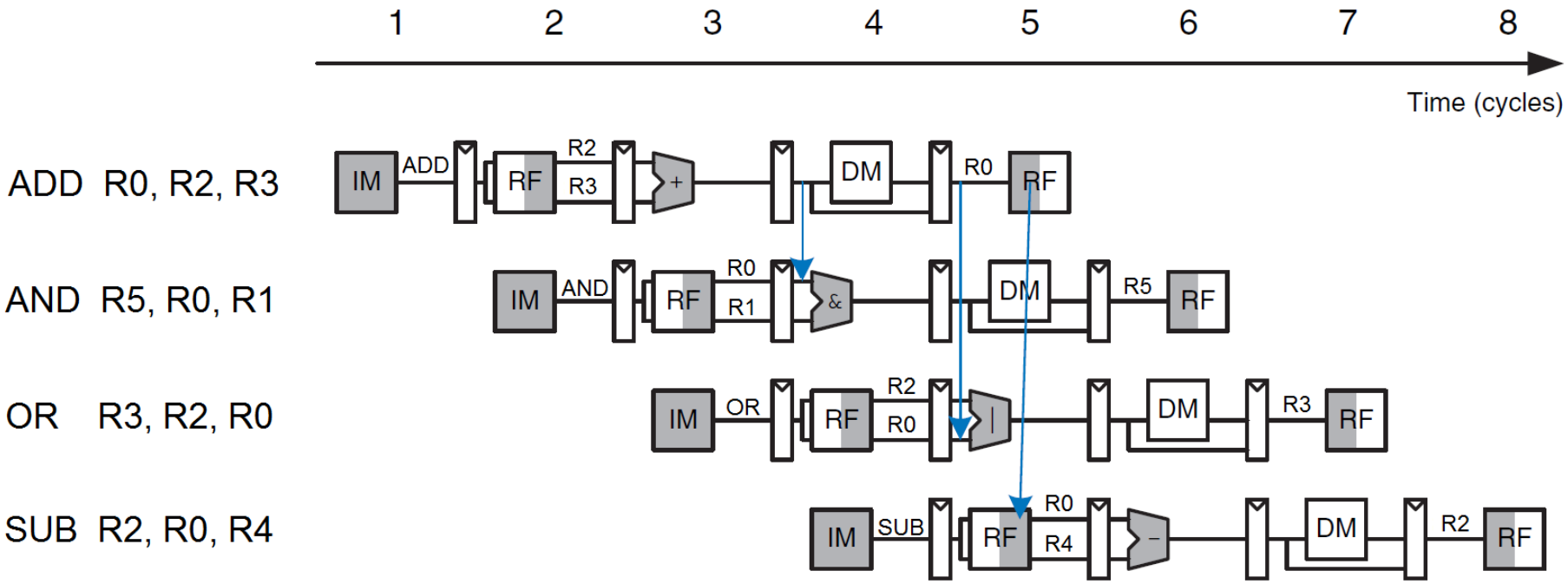
Pipeline Hazards

- Data hazards occurs if;
  - Register type operand needs to be read but previous instruction is not finished
  - Memory type instruction result need to be read by following register type instruction.
- Control hazards occurs if;
  - Branch is taken
  - Subroutine call
  - Interrupt

- Solutions;
  - Data forwarding
  - Pipeline stall
  - Pipeline flush

Pipeline Hazard Solutions: Data Forwarding

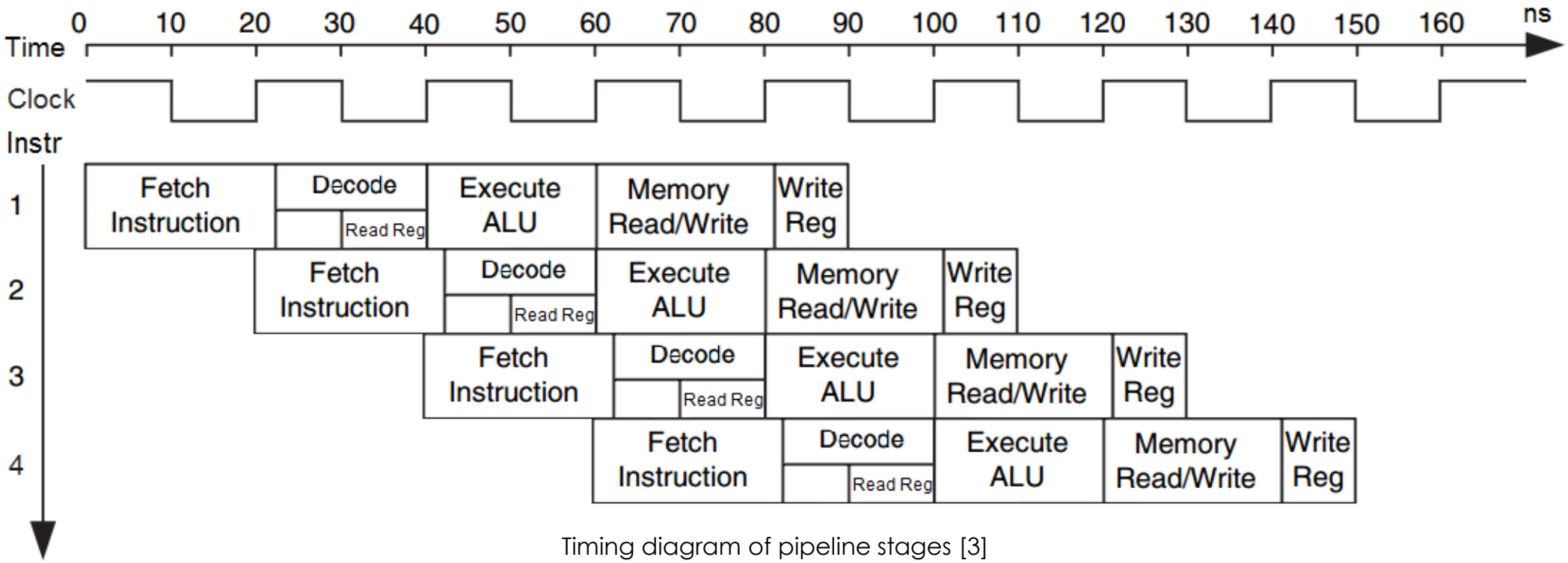
Data forwarding solves register operand data hazards.



Data forwarding timing diagram [3]

Pipeline Timing

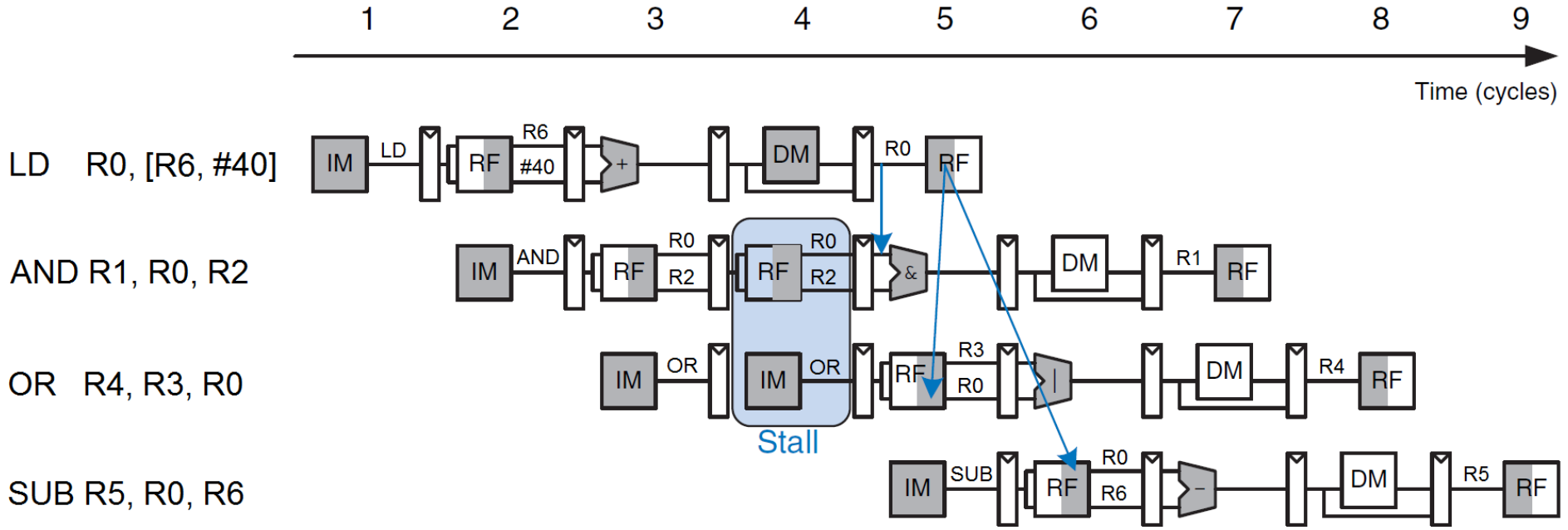
Decode, register read and register write operations are done in same clock cycle.



Timing diagram of pipeline stages [3]

Pipeline Hazard Solutions: Pipeline Stall

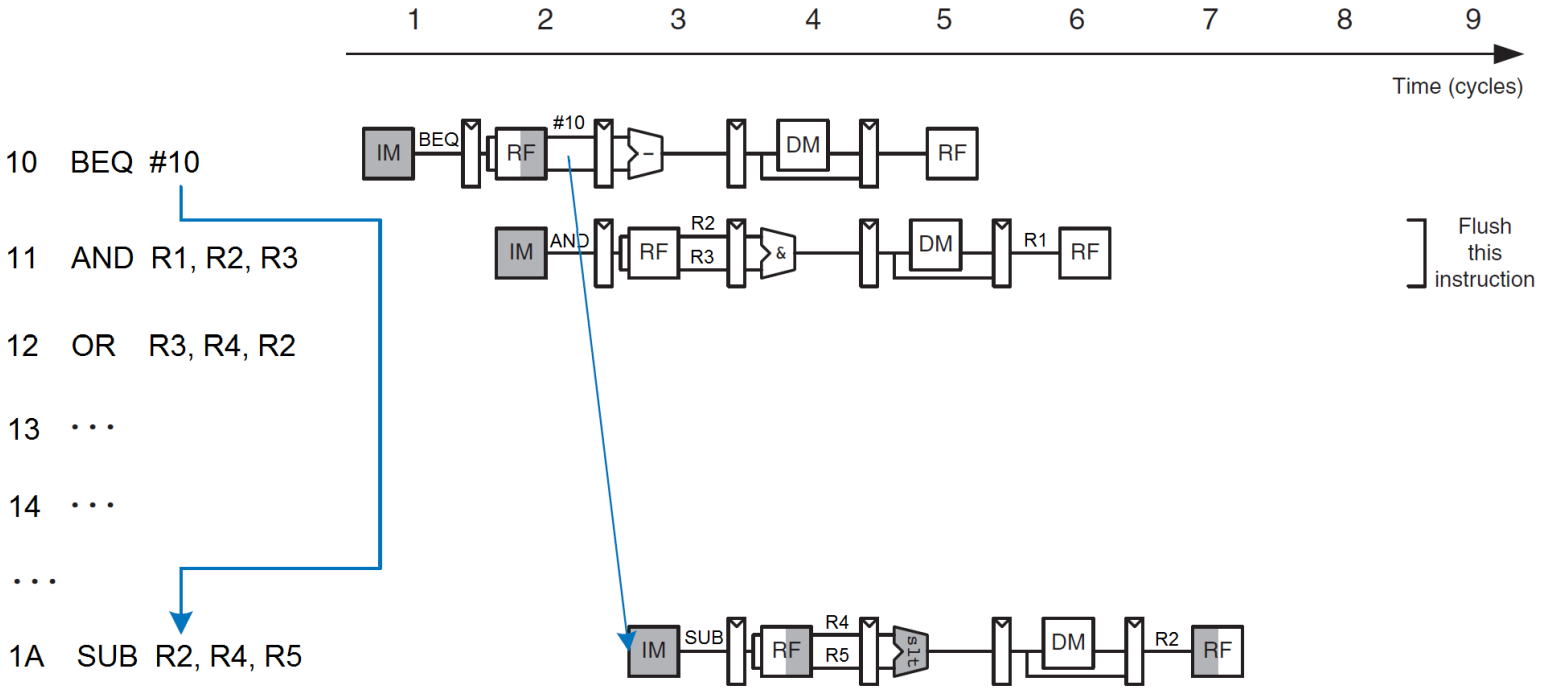
Pipeline stall operation solves memory type data hazard but solution has a one clock cycle pipeline penalty



Pipeline stall timing diagram [3]

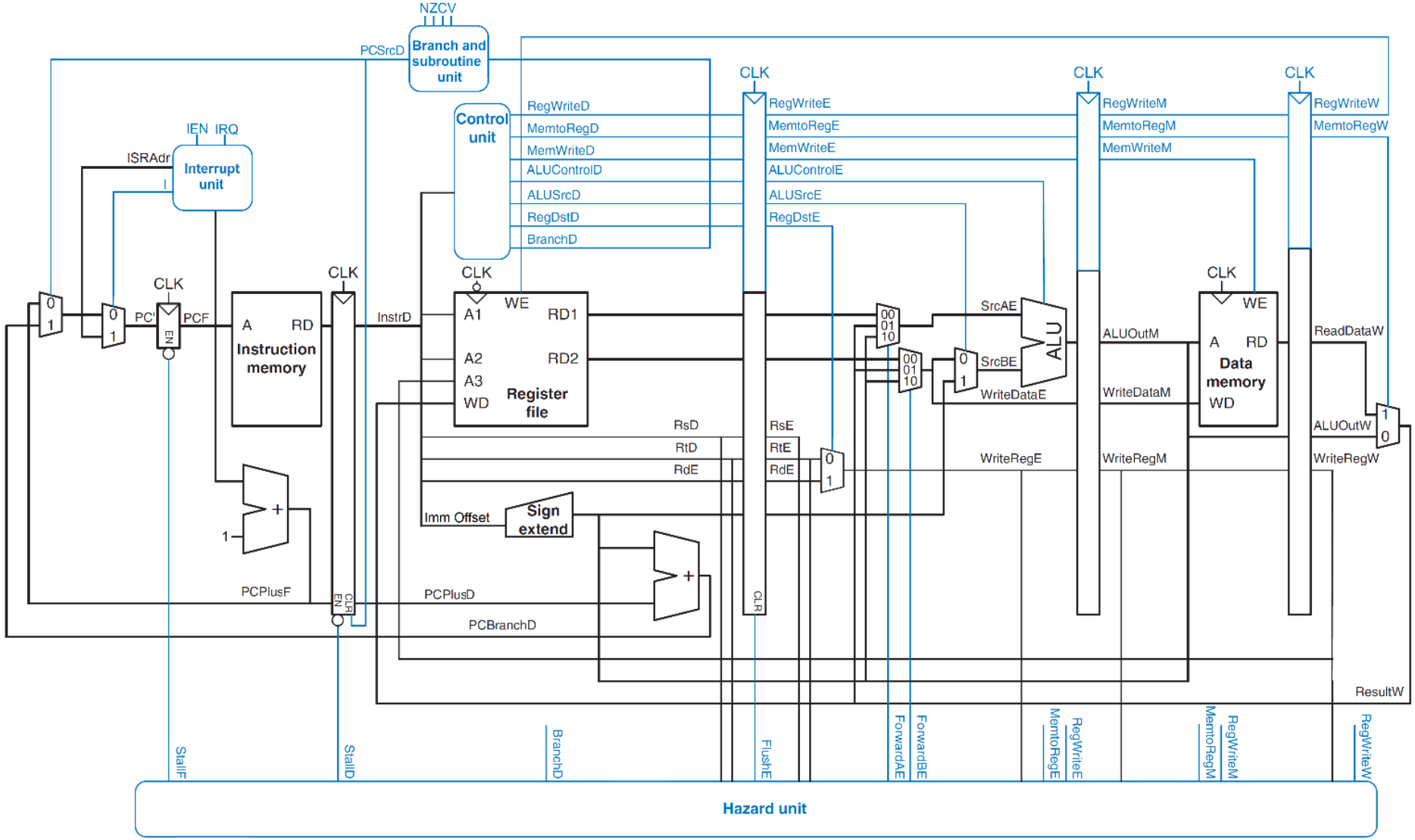
Pipeline Hazard Solutions: Pipeline Flush

Pipeline flush solves branch hazards but solution has a one clock cycle pipeline penalty



Pipeline stall timing diagram[3]

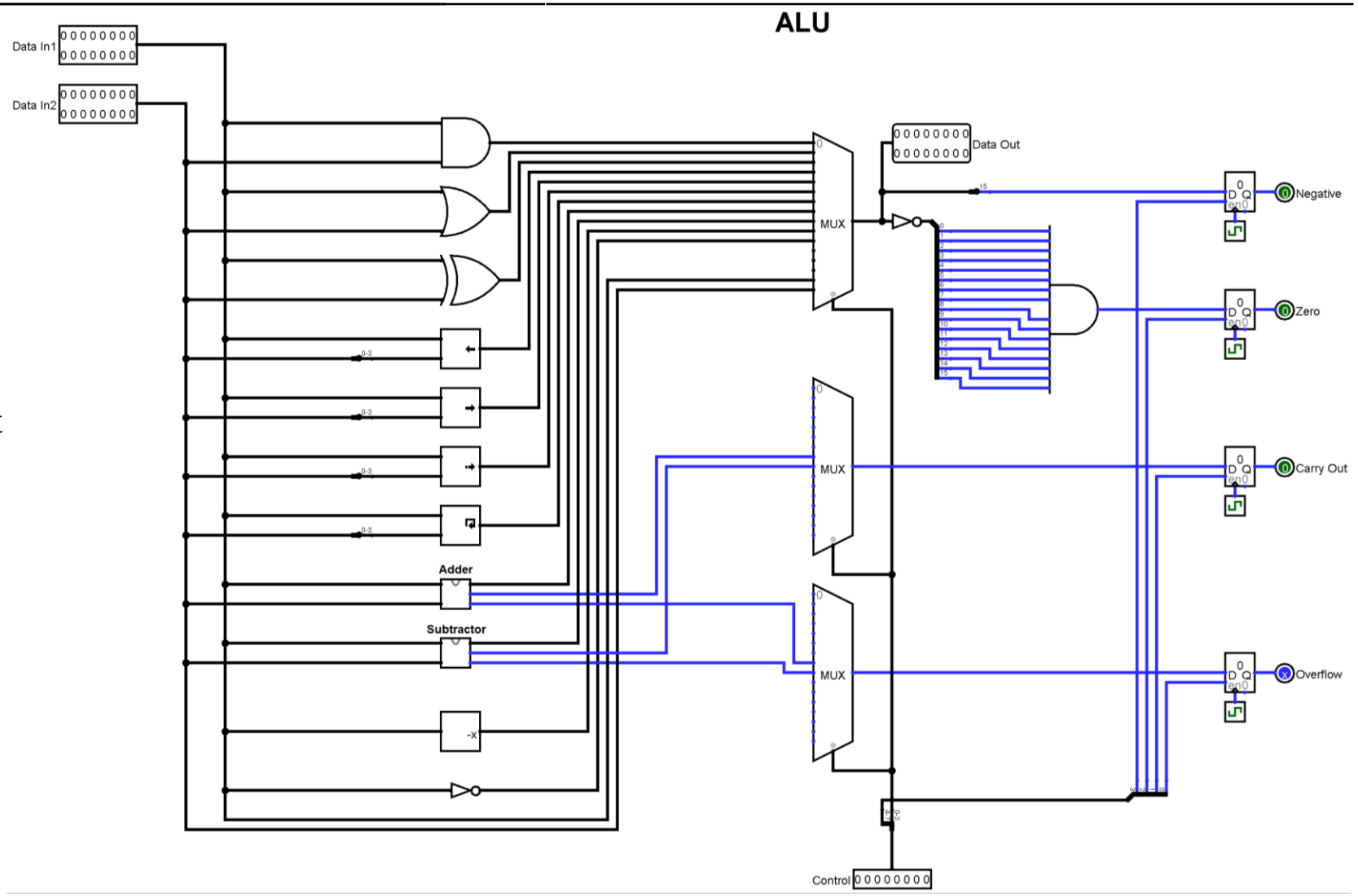
Structure of the Processor



Processor organization schematic with control and hazard units [3]

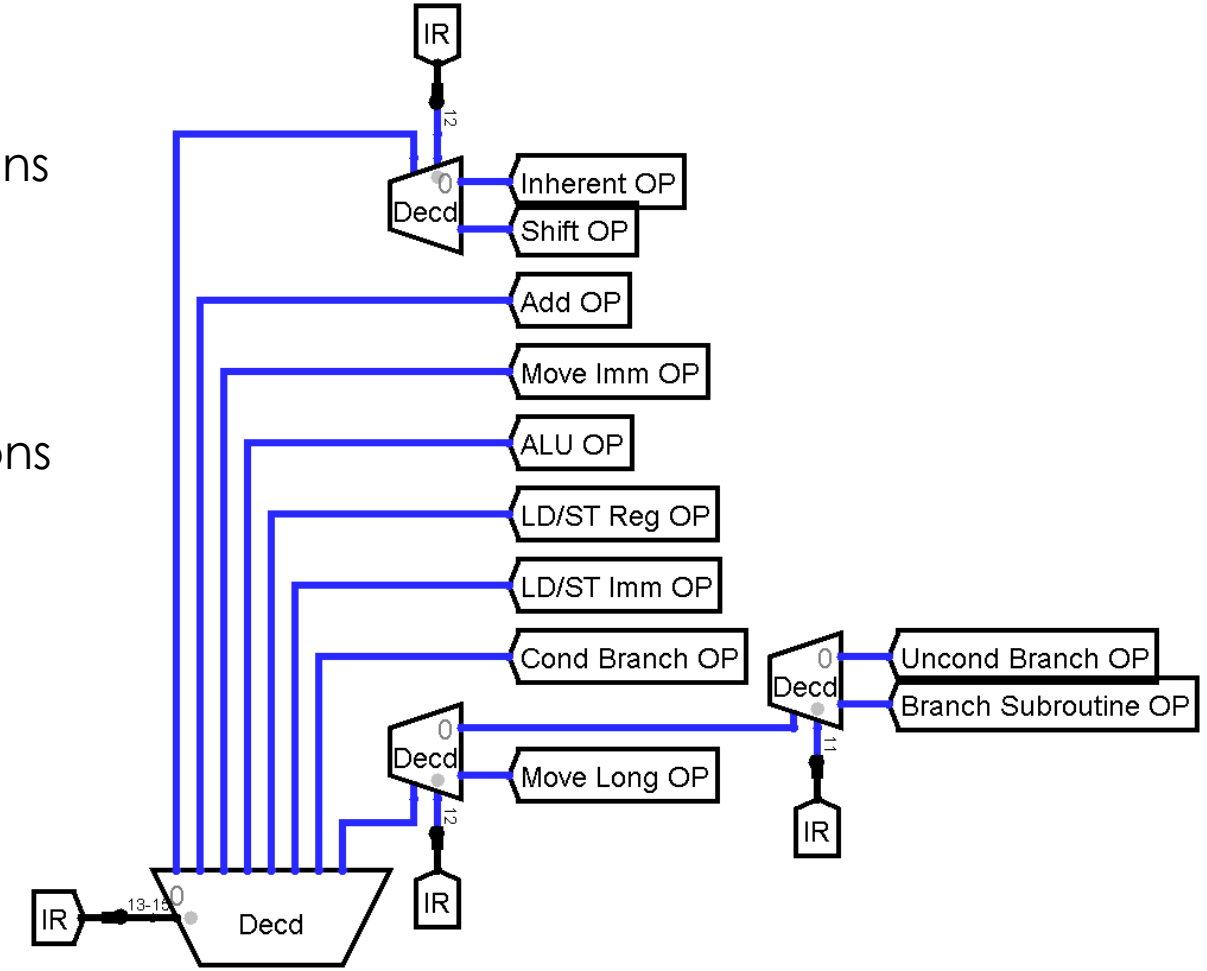
ALU

- Logical AND
- Logical OR
- Logical XOR
- Logical left shift
- Logical right shift
- Arithmetic shift right
- Circular shift left
- Arithmetic add
- Arithmetic subtract
- Arithmetic negation
- Logical NOT



Decode

1. Inherent Instructions
2. Shift Instructions
3. Add/Subtract Register Immediate Instructions
4. Move/Compare/Add/Subtract Immediate Instructions
5. ALU Instructions
6. Load/Store with Register Offset Instructions
7. Load/Store with Immediate Offset Instructions
8. Conditional Branch Instructions
9. Unconditional Branch Instruction
10. Branch to Subroutine Instruction
11. Move Long Immediate



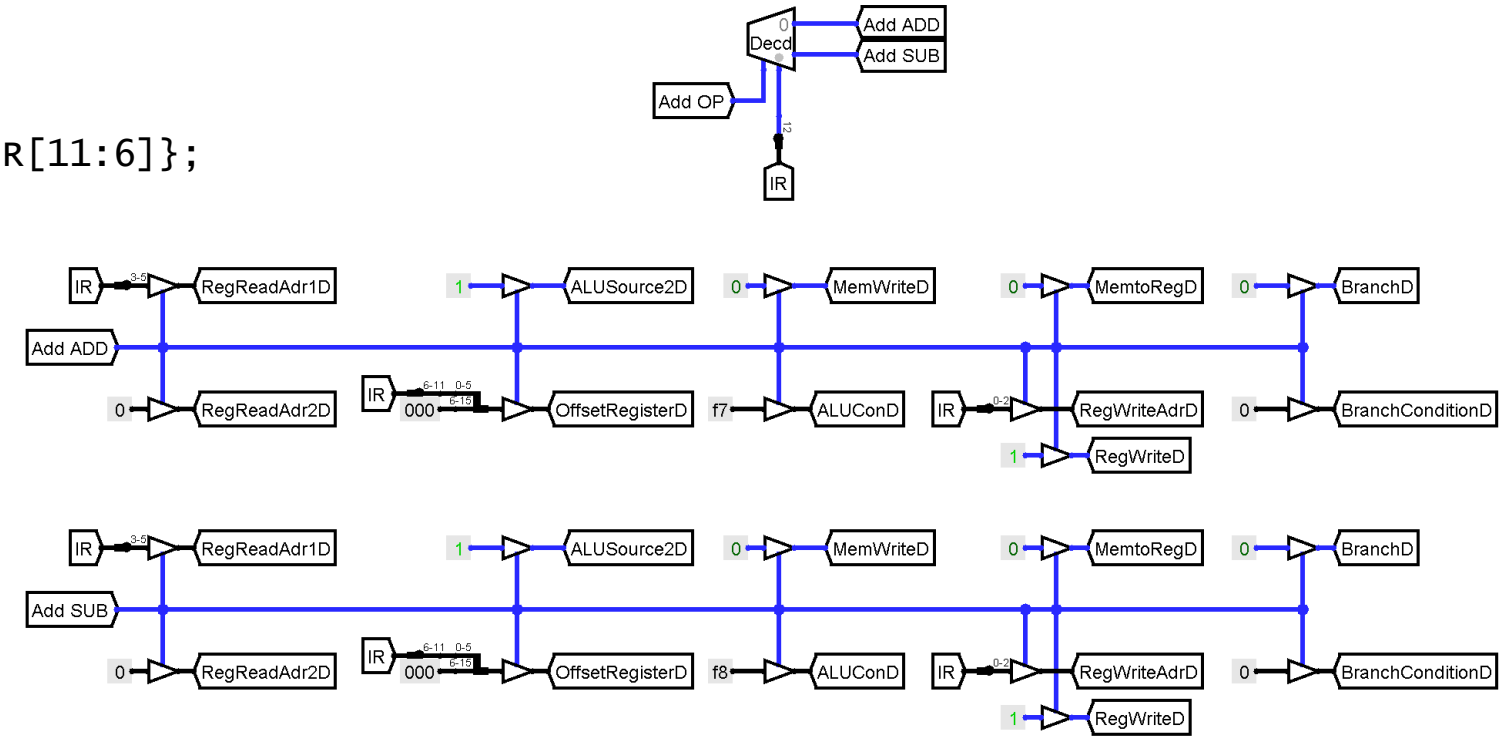
Decode Example

Verilog

```
4'b001x: begin
        reg_read_adr1_d = IR[5:3];
        reg_read_adr2_d = 3'h0;
        reg_write_adr_d = IR[2:0];
        reg_write_d = 1;
        ALU_source2_d = 1;
        offset_register_d = {10'h000, IR[11:6]};
        mem_write_d = 0;
        mem_to_reg_d = 0;
        branch_d = 0;
        branch_condition_d = 4'h0;
        IEN_d = 0;
        IOF_d = 0;
        RTI_d = 0;
        subroutine_call_d = 0;
        subroutine_return_d = 0;
        case (IR[12])
                0: ALU_con_d = 8'hf7;
                1: ALU_con_d = 8'hf8;
        endcase
end
```

Logisim

Add/Subtract Register Immediate



Assembler

- A Python script is written for assemble instructions and create binary machine instructions.
- Assembler uses simple regular expressions to parse instruction mnemonics.
- After generating instruction binaries, these are converted to Logisim memory input file format, ISim memory input file format and Xilinx FPGA mem input file format.

Assembler has some special words for create predefined variables.

- Assembler variables can resides between "**_initial**" and "**_end**" block.
- " **$** " character assigns decimal variable.
- " **&** " character gives starting address of variable.
- " **;** "character makes afterwards a comment in a line.

Example Assembly Program

Assembler input:

```
; I/O test
_initial
    $proc_in = 4080
    $proc_out = 4081
_end

    MOV R2, #4
    MOV R3, #3
H:  CSL R3, R3, #1
    SUB R2, R2, #1
    BNE H
    ADD R1, R1, #1
    MOVL R0, $proc_out
    STR [R0], R3

F:  BAL F
```

C equivalent:

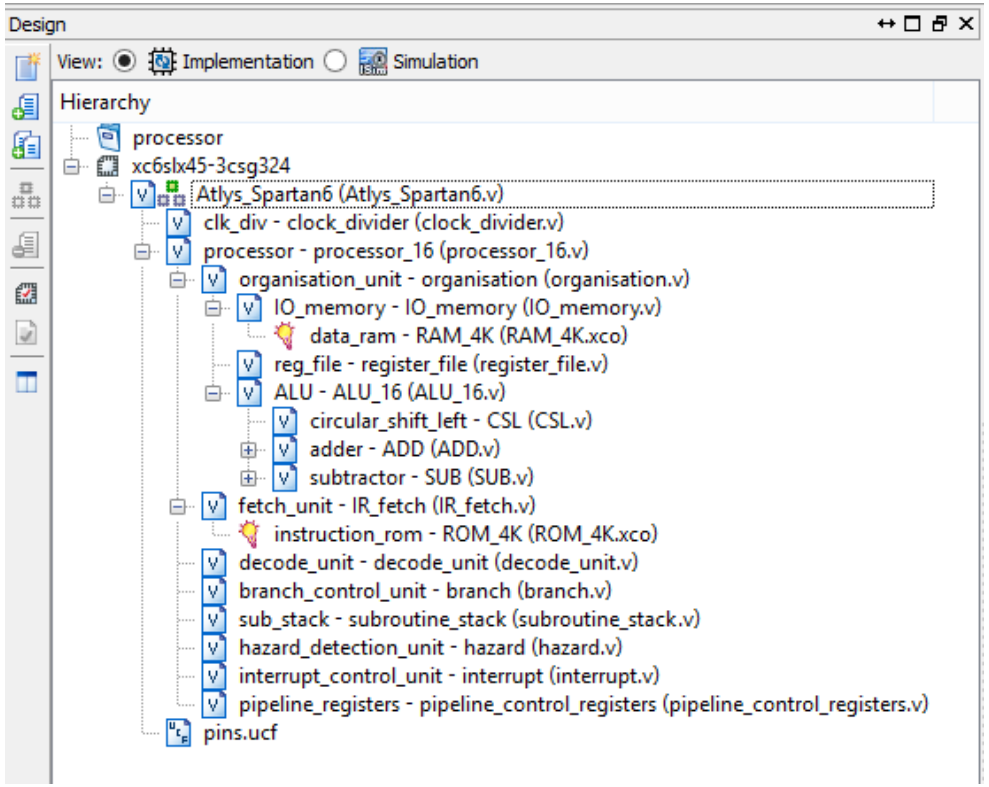
```
int a = 3;
for (int i = 4; i > 0; i--)
    a = a << 1;
```

Assembler output:

```
010   0100001000000100 4204    MOV R2 #4
011   0100001100000011 4303    MOV R3 #3
012   0001110001011011 1C5B    CSL R3 R3 #1
013   0011000001010010 3052    SUB R2 R2 #1
014   1100001111111101 C3FD    BNE H
015   0010000001001001 2049    ADD R1 R1 #1
016   1111111111110001 FFF1    MOVL R0 #4081
017   1011000000000011 B003    STR [R0] R3
018   1110011111111111 E7FF    BAL F
```

Processor Verilog Modules

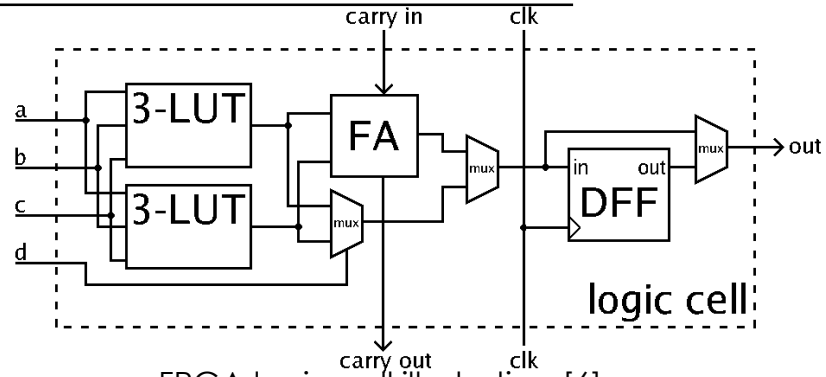
- Processor [processor]
  - Organization unit [organization_unit]
    - I/O and data memory unit [IO_memory]
    - Register file [register_file]
    - ALU [ALU]
  - Fetch unit [fetch_unit]
    - Instruction memory [instruction_rom]
  - Decode unit [decode_unit]
  - Branch control unit [branch_control_unit]
  - Subroutine stack [subroutine_stack]
  - Hazard detection unit [hazard_detection_unit]
  - Interrupt control unit [interrupt_control_unit]
  - Pipeline control registers [pipeline_registers]



FPGA Implementation

Xilinx Spartan-6 XC6SLX45 FPGA [1]

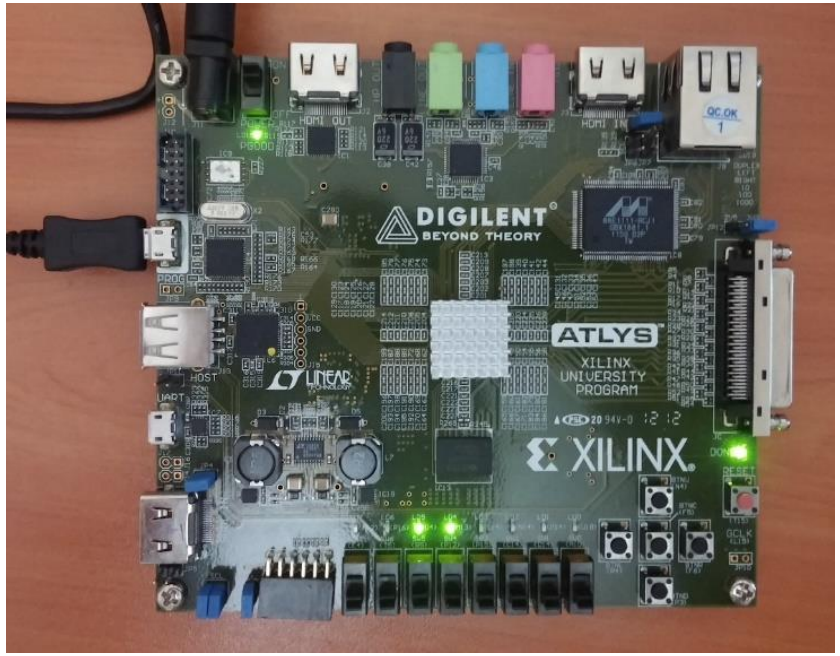
- 6,822 slices, each containing four 6-input LUTs and eight flip-flops
- 2,088 Kb block ram capacity
- 50MHz clock frequency



FPGA logic cell illustration [6]

Xilinx ISE synthesis place and route report:

```
Device Utilization Summary:
Slice Logic Utilization:
  Number of Slice Registers:    415 out of   54,576    1%
  Number of Slice LUTs:         623 out of   27,288    2%
    Number used as logic:       579 out of   27,288    2%
    Number used as Memory:       25 out of    6,408    1%
```



Demo Assembly Program

Assembler input:

```
; I/O test
_initial
    $proc_in = 4080
    $proc_out = 4081
_end

    MOV R2, #4
    MOV R3, #3
H:  CSL R3, R3, #1
    SUB R2, R2, #1
    BNE H
    ADD R1, R1, #1
    MOVL R0, $proc_out
    STR [R0], R3

F:  BAL F
```

C equivalent:

```
int a = 3;
for (int i = 4; i > 0; i--)
    a = a << 1;
```

Assembler output:

```
010   0100001000000100 4204    MOV R2 #4
011   0100001100000011 4303    MOV R3 #3
012   0001110001011011 1C5B    CSL R3 R3 #1
013   0011000001010010 3052    SUB R2 R2 #1
014   1100001111111101 C3FD    BNE H
015   0010000001001001 2049    ADD R1 R1 #1
016   1111111111110001 FFF1    MOVL R0 #4081
017   1011000000000011 B003    STR [R0] R3
018   1110011111111111 E7FF    BAL F
```

Performance

- With the help of pipeline, processor can achieve instruction level parallelism.
- Nominal cycles per instruction (CPI) is 1. Without pipelining CPI would be 5.
- Penalty for stall and flush instructions is 1 clock cycle.
- Pipeline stall and flush operations increase cycle per instruction. But in long execution run, CPI converges to 1. [4]

$$\text{CPI} \cong 1 \quad \text{Effective processor performance} = \frac{\textbf{Clock frequency}}{\textbf{CPI} \times 1000000} = \frac{50000000}{1 \times 1000000} \cong 50 \text{ MIPS}$$

- Execution time for a given instruction count can be calculated as [4]:

$$\text{Execution time } (T) = \frac{\textbf{CPI} \times \textbf{Instruction count}}{\textbf{Clock frequency}}$$

- Instruction count for example program is 24. Execution time for example program:

$$\text{Execution time } (T) = \frac{1 \times 24}{50000000} \cong 0.48 \text{ ms}$$

Demo Assembly Program - 2

```
; factorial
_initial
    $proc_in = 4080
    $proc_out = 4081
_end

D:  BTS P
    BTS R

    ; if n == 0
    CMP R2, #0
    BNE C1
    MOV R1, #1
    BAL D

    ; if n == 1
C1: CMP R2, #1
    BNE C2
    MOV R1, #1
    BAL D
```

```
    ; if n == 2
C2: CMP R2, #2
    BNE C3
    MOV R1, #2
    BAL D

    ; if n > 2
C3: MOV R3, R2
    MOV R1, R2
S:  SUB R3, R3, #1
    CMP R3, #1
    BEQ D
    MUL R1, R1, R3
    BAL S
```

```
    ; SUBROUTINES

    ; print
P:  MOVL R0, $proc_out
    STR [R0], R1
    RTS

    ; read input
R:  MOVL R0, $proc_in
    LD   R2, [R0]
    RTS
```

Demo Assembly Program - 3

```
; Knight Rider Kitt
_initial
    $proc_in = 4080
    $proc_out = 4081
    $delay = 30
_end

    MOV R1, #1
```

```
    ; left
L:  BTS P
    BTS W
    LSL R1, R1, #1
    CMP R1, #128
    BCC L

    ; right
R:  BTS P
    BTS W
    ASR R1, R1, #1
    CMP R1, #1
    BHI R
    BAL L
```

```
    ; SUBROUTINES

    ; print
P:  MOVL R0, $proc_out
    STR [R0], R1
    RTS

    ; wait
W:  MOVL R0, $delay
D2: MOV R7, #0
D1: SUB R7, R7, #1
    BNE D1
    SUB R0, R0, #1
    BNE D2
    RTS
```

References

[1] Berger, A. S. (2005). *Hardware and Computer Organization: The Software Perspective.* Burlington: Elsevier Inc.

[2] Flynn, M. J. (1995). *Computer architecture: Pipelined and parallel processor design.* Boston, MA: Jones and Bartlett.

[3] Harris, D. & Harris, S. (2012). *Digital Design and Computer Architecture* (2nd ed.). San Francisco: Morgan Kaufmann Publishers Inc.

[4] Hennessy, J. L. & Patterson, D. A. (2011). *Computer Architecture: A Quantitative Approach* (5th ed.). San Francisco: Morgan Kaufmann Publishers Inc.

[5] Hennessy, J. L., & Patterson, D. A. (1998). *Computer organization and design: The hardware/software interface.* San Francisco, Calif: Morgan Kaufmann Publishers.

[6] Lawlor, O. (2014). *CS 441: System Architecture* [Lecture Note]. Retrieved from UAF Computer Science Department website: https://www.cs.uaf.edu/courses/cs441/notes/encoding/

[7] The McGraw-Hill Companies Inc. (2009). *Schaum's Outline of Theory and Problems of Computer Architecture*. Retrieved from http://www.dauniv.ac.in/downloads/CArch_PPTs/CompArchCh04L07InstructionSetFeatures.pdf

[8] Microprocessor Design. (n.d.). In *Wikibooks*. Retrieved from http://en.wikibooks.org/wiki/Microprocessor_Design/Print_Version

[9] University of Waterloo. (2012). *THUMB Instruction Set*. Retrieved from University of Waterloo website: https://ece.uwaterloo.ca/~ece222/ARM/ARM7-TDMI-manual-pt3.pdf

[10] Weatherspoon, H. (2011). *CS 3410: Computer System Organization and Programming* [Lecture Slides]. Retrieved from Cornell University website: http://www.cs.cornell.edu/courses/CS3410/2011sp/schedule.html

THANK YOU FOR LISTENING