Group 19 - Project Report

Nishant Misal, Ulises Chavarria

Santa Clara University

CSEN 210 Computer Architecture

Yan Cui

December 6, 2024

Table of Contents

Objective

Design a 32-bit pipelined CPU for the given SCU Instruction Set Architecture (SCU ISA). The SCU ISA is described below.

Register file size: 64 registers, each register has 32 bits.

• PC: 32 bits

• 2's complement is assumed.

• Instruction format: Each instruction is 32-bit wide, and consists of five fields: opcode, rd, rs, rt, and unused. The format is as follows.

Opcode (4 bits)	rd (6 bits)	rs (6 bits)	rt (6 bits)	unused(10 bits)

The 13 instructions are defined below.

Instruction	Symbol	Opcode	rd	rs	rt	Function
No operation	NOP	0000	×	×	×	No operation
Save PC	SVPC rd, y	1111	rd	y		xrd ← PC+y
Load	LD rd, rs	1110	rd	rs	×	xrd ← M[xrs]
Store	ST rt, rs	0011	×	rs	rt	M[xrs] ← xrt
Add	ADD rd, rs, rt	0100	rd	rs	rt	xrd ← xrs + x rt
Increment	INC rd, rs, y	0101	rd	rs	y	xrd ← xrs + y
Negate	NEG rd, rs	0110	rd	rs	×	x rd ← - xrs
Subtract	SUB rd, rs, rt	0111	rd	rs	rt	x rd ← xrs – x rt
Jump	J rs	1000	×	rs	×	PC ← xrs
Branch if zero	BRZ rs	1001	×	rs	×	PC ← xrs, if Z = 1
Jump memory	JM rs	1010	×	rs	×	PC ← M[xrs]
Branch if negative	BRN rs	1011	×	rs	×	PC ← xrs, if N = 1
MAX	MAX, rd, rs, rt	0001	rd	rs	rt	See *

*xrd = Max{memory[xrs], memory[xrs + 1], …, memory[xrs + xrt -1]},
×: don't care

Assembly Code using SCU ISA

C Code

C Code was used to translate to SCU ISA.

```
int findmax(int arr[], int size){
        int max = arr[0];
        for(int i = 1; i < size; i++){
                if (arr[i] - max >= 0){
                max = arr[i];
                }
        }
        return max;
}
```

Software Loop

Assumptions and Restrictions

- x10 = &arr[0]
- x11 = size
- At any given instruction, the PC holds the address of the next instruction
- No labels allowed
- # are comments

```
SVPC    x1, 0x0
INC     x12, x1, 0x18     # x12 = &Loop
INC     x13, x1, 0x30     # x13 = &LoopIncrement
INC     x14, x1, 0x3c     # x14 = &LoopEnd
INC     x5, x0, 0x1       # x5 => i = 1
LD      x6, x10           # x6 => max = arr[0]
INC     x7, x10, 0x4      # x7 => &arr[i] => aka &arr[1]


#Loop
SUB     x15, x11, x5      # x15 => size - i
BRZ     x14               # if (i == size) => end of array, so exit loop
LD      x15, x7           # x15 = arr[i]
SUB     x16, x15, x6      # x16 = arr[i] - max
BRN     x13               # when arr[i] < max, go to next iteration of loop
ADD     x6, x0, x15       # x6 => max = arr[i]
```

```
#LoopIncrement

INC     x5, x5, 0x1     # x5 => i++

INC     x7, x7, 0x4     # x7 => &arr[i++]

J       x12             # start next iteration of Loop


#LoopEnd

ADD     x10, x6, x0     # store max in return register
```

Software Loop Code Explanation

Initialization and Address Saving:

SVPC x1, 0x0

- Saves the Program Counter (PC) into register x1

- Enables capturing memory addresses of code sections

Label Address Preparation:

INC x12, x1, 0x18, **INC x13, x1, 0x30**, **INC x14, x1, 0x3c**

- Stores precise addresses for different code sections and loop control

Loop Variable Setup:

INC x5, x0, 0x1, **LD x6, x10**, **INC x7, x10, 0x4**

- Initializes loop counter (i = 1)

- Loads initial array element as the first max value

- Sets up pointer to next array element

Loop Condition Check:

SUB x15, x1, x5, **BRZ x14**

- Calculates remaining array elements (size - current index)

- Branches to loop end when all elements are processed

Element Comparison:

LD x15, x7, **SUB x16, x15, x6**, **BRN x13**

- Loads current array element

- Compares current element with existing max value

- Skips max replacement if current element is smaller

Max Value Update:

ADD x6, x0, x15

- Updates max value when current element is greater or equal

Loop Increment:

INC x5, x5, 0x1, **INC x7, x7, 0x4**, **J x12**

- Increments loop counter

- Advances array element pointer

- Jumps back to loop start

Result Preparation:

ADD x10, x6, x0

- Moves final max value to return register (x10)

Hardware Loop

- This program finds the max integer in an array “arr” of length “size”
- Assumptions:
    - x10 = &arr[0]
    - x11 = size

```
MAX x10, x10, x11
```

Datapath & Control: 4-Stage Pipeline Overview

Pipeline Configuration

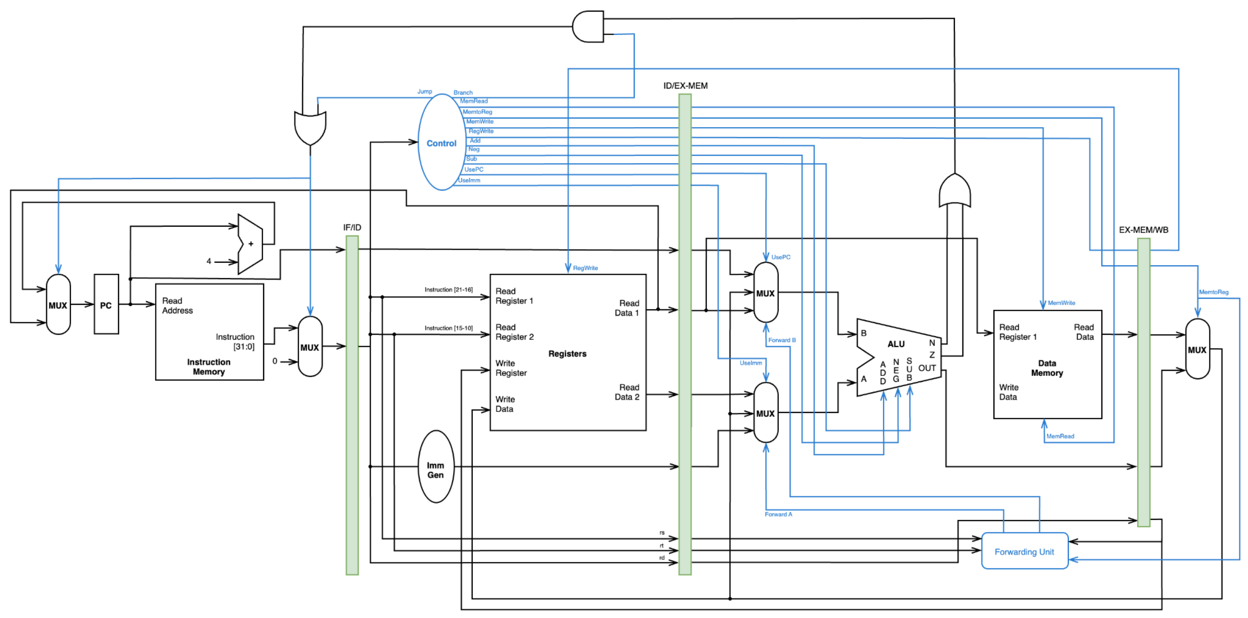
1. Reduced traditional 5-stage pipeline to 4 stages
2. Combined Execute (EX) and Memory (MEM) stages
3. Rationale: SCU ISA software limitations
4. Data Memory Unit used only for LD instruction
5. LD instruction directly moves data from Memory to Register without ALU dependency

Hazard Management Strategy

1. Data Hazards
    a. Minimal instruction interactions in software code
    b. Handled primarily through Forwarding Unit
    c. Two primary data hazard scenarios identified:
2. First Data Hazard
    a. Instructions: SVPC x1, 0x0 → INC x12, x1, 0x18
    b. Problem: SVPC writes to x1, which INC subsequently uses
    c. Solution: Forward new x1 value from Write-Back (WB) stage to Execute-Memory (EX-MEM) stage
3. Second Data Hazard
    a. Instructions: LD x15, x7 → SUB x16, x15, x6
    b. Problem: LD writes to x15, which SUB subsequently uses
    c. Solution: Forward new x15 value from Write-Back (WB) stage to Execute-Memory (EX-MEM) stage

4. 2. Control Hazards
    a. Handled through pipeline flushing mechanism
    b. Triggered by branch and jump instructions
    c. Uses logic gate sequence to insert 1-cycle bubble
5. Branch Hazard Resolution
    a. Detection: Occurs when branch instruction is in Instruction Decode (ID) stage
    b. Previous Instruction: In Execute-Memory (EX-MEM) stage (e.g., SUB)
    c. Resolution Process:
        i. OR the N and Z flags from ALU
        ii. AND result with branch flag from Control Unit
        iii. If branch condition met:
            1. First multiplexer passes new branch address
            2. Second multiplexer inserts zero into IF/ID pipeline register
6. Jump Instruction Handling
    a. Automatic 1-cycle flush when jump control unit flag is active
    b. Implemented via final OR gate in control logic

Datapath



Control

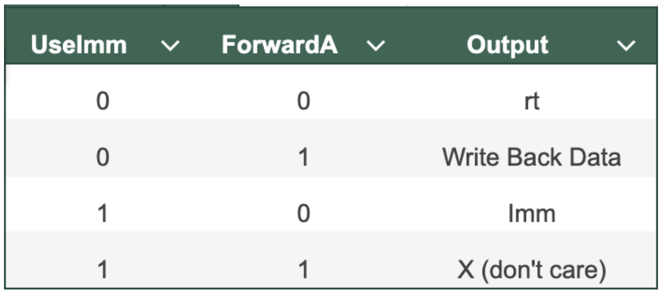
Control Unit Truth Table

Instruction	Opcode	Jump	Branch	MemRead	MemtoReg	MemWrite	RegWrite	Add	Neg		UsePC	UseImm
SVPC	1111	0	0	0	0	0	1	1	0	0	1	1
LD	1110	0	0	1	1	0	1	x	x	x	x	x
ADD	0100	0	0	0	0	0	1	1	0	0	0	0
INC	0101	0	0	0	0	0	1	1	0	0	0	1
SUB	0111	0	0	0	0	0	1	0	0	1	0	0
J	1000	1	0	0	x	0	0	x	x	x	x	x
BRZ	1001	0	1	0	x	0	0	x	x	x	x	x
BRN	1011	0	1	0	x	0	0	x	x	x	x	x

Multiplexer B Truth Table

UsePC	ForwardB	Output
0	0	rs
0	1	Write Back Data
1	0	PC
1	1	X (don't care)

Multiplexer A Truth Table



UseImm	ForwardA	Output
0	0	rt
0	1	Write Back Data
1	0	Imm
1	1	X (don't care)

RISC-V Emulation

RISC-V	**SCU ISA**
ADD rd, rs1, rs2	ADD rd, rs1, rs2
SUB rd, rs1, rs2	SUB rd, rs1, rs2
ADDI rd, rs1, imm	INC rd, rs, imm
LW rd, imm(rs1)	INC rs1, rs1, imm LD rd, rs1
SW rs2, imm(rs1)	INC rs1, rs1, imm ST rs2, rs1
AND rd, rs1, rs2	SUB rd, rd, rd SVPC x12, 16 ADD rs1, rs1, rs2 BRZ x12 INC rd, rs1, -1 NOP
OR rd, rs1, rs2	SUB rd, rd, rd SVPC x12, 16 ADD rs1, rs1, rs2 BRZ x12 INC rd, rd, 1 NOP
NOR rd, rs1, rs2	SUB rd, rd, rd INC rd, rd, 1 SVPC x12, 16 ADD rs1, rs1, rs2 BRZ x12 INC rd, rd, -1 NOP
ANDI rd, rs1, imm	SUB rd, rd, rd SVPC x12, 16 ADD rs1, rs1, imm BRZ x12 INC rd, rd, -1 NOP
BEQ rs1, rs2, imm	SVPC x6, imm SUB x7, rs1, rs2

	BRZ x6
JAL rd, imm	SVPC rd, 4 SVPC x6, imm J x6
JALR rd, rs1, imm	INC rt, rd, imm J x6

Performance Evaluation

Instruction Count

Calculation: 9 instructions in the loop + 8 outside the loop

Formula: 9n + 8, where n is the number of array elements

Cycle Time

Datapath: 4-stage pipeline

Longest Delay: 3 ns

Clock Cycle Time: 3 ns

Cycles Per Instruction (CPI)

Formula: (9n + 12) / (9n + 8)

Example (3-element array): (9 * 3 + 12) / (9 * 3 + 8) = 1.114

Execution Time

Formula: (Instruction Count) * (CPI) * (Clock Cycle Time)

Detailed Calculation: (9n + 8) * [(9n + 12) / (9n + 8)] * 3 ns

Example (3-element array):

(9 * 3 + 8) * 1.114 * 3 ns = 117 ns

Notes:

Calculations assume a general array size of n elements

Performance metrics demonstrate slight overhead due to loop control instructions