Implementing RISC-V in Synchronous Message Exchange

Daniel Ramyar

Niels Bohr Institute
University of Copenhagen

MAX IV talk
December 3, 2019

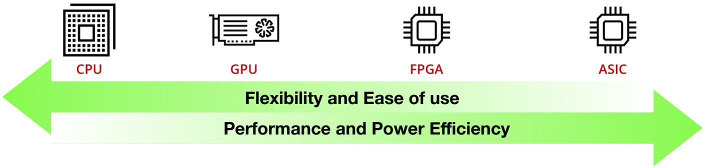
Overview

Motivation

- Most measurement instruments are based on Intel x86 CPU
- Limits the bandwidth at which data collection is possible
- Limits the possibility for custom solutions

Motivation

- RISC-V especially well suited embedding in scientific instruments
- Fully customizable = faster and more power efficient
- Could be implemented in a FPGA
- But FPGA programming is complicated

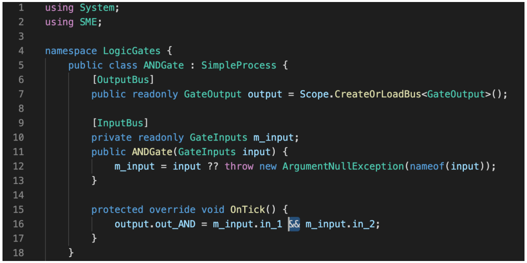


Synchronous Message Exchange

- Write FPGA applications within .Net framework[1]
- Enjoy productivity features of modern language
- Automatic VHDL conversion
- Based on Communicating Sequential Processes (CSP)[2]

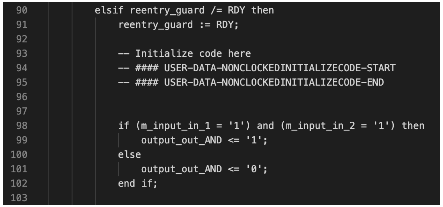
[1] Building Hardware from C# models, Kenneth Skovhede and Brian Vinter
[2] Communicating sequential processes, Charles Antony Richard Hoare

Synchronous Message Exchange



```
using System;
using SME;

namespace LogicGates {
    public class ANDGate : SimpleProcess {
        [OutputBus]
        public readonly GateOutput output = Scope.CreateOrLoadBus<GateOutput>();

        [InputBus]
        private readonly GateInputs m_input;
        public ANDGate(GateInputs input) {
            m_input = input ?? throw new ArgumentNullException(nameof(input));
        }

        protected override void OnTick() {
            output.out_AND = m_input.in_1 && m_input.in_2;
        }
    }
```

C# SME Code



```
            elsif reentry_guard /= RDY then
                reentry_guard := RDY;

                -- Initialize code here
                -- #### USER-DATA-NONCLOCKEDINITIALIZECODE-START
                -- #### USER-DATA-NONCLOCKEDINITIALIZECODE-END


                if (m_input_in_1 = '1') and (m_input_in_2 = '1') then
                    output_out_AND <= '1';
                else
                    output_out_AND <= '0';
                end if;

```

VHDL

Synchronous Message Exchange

- Processes are isolated no memory sharing
- Communicates with each other through channels
- Globally synchronous
- Perfect for implementing RISC-V!

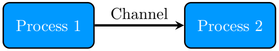


Figure: One to one

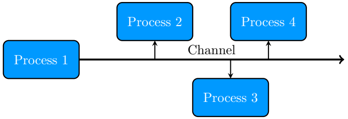
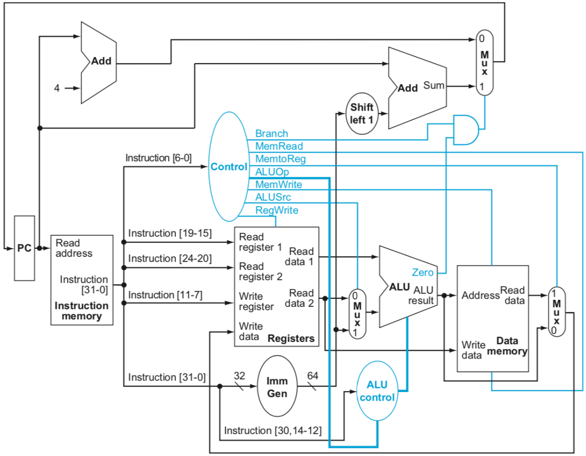


Figure: One to many

Single Cycle RISC-V

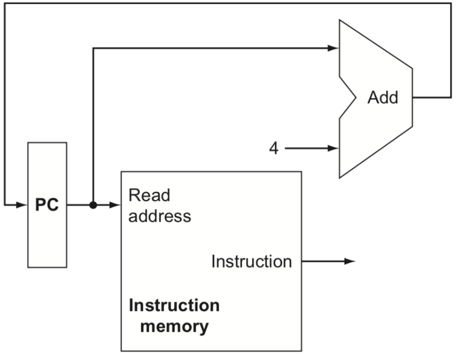
Full datapath for single cycle RISC-V



Taken from *Computer organization and design RISC V*

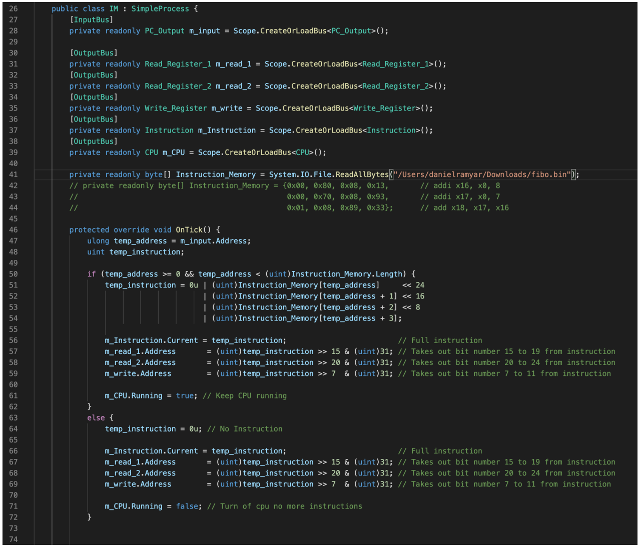
Fetching instruction and increment

We need memory for instructions and a way to remember current instruction



Taken from *Computer organization and design RISC V*

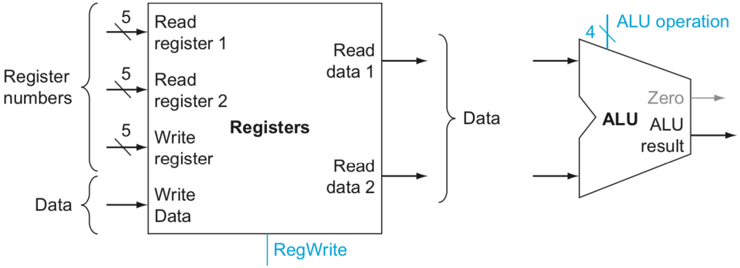
Fetching instruction and increment



```csharp
public class IM : SimpleProcess {
    [InputBus]
    private readonly PC_Output m_input = Scope.CreateOrLoadBus<PC_Output>();

    [OutputBus]
    private readonly Read_Register_1 m_read_1 = Scope.CreateOrLoadBus<Read_Register_1>();
    [OutputBus]
    private readonly Read_Register_2 m_read_2 = Scope.CreateOrLoadBus<Read_Register_2>();
    [OutputBus]
    private readonly Write_Register m_write = Scope.CreateOrLoadBus<Write_Register>();
    [OutputBus]
    private readonly Instruction m_Instruction = Scope.CreateOrLoadBus<Instruction>();
    [OutputBus]
    private readonly CPU m_CPU = Scope.CreateOrLoadBus<CPU>();

    private readonly byte[] Instruction_Memory = System.IO.File.ReadAllBytes("/Users/danielramyar/Downloads/fibo.bin");
    // private readonly byte[] Instruction_Memory = {0x00, 0x80, 0x08, 0x13,      // addi x16, x0, 8
    //                                               0x00, 0x70, 0x08, 0x93,      // addi x17, x0, 7
    //                                               0x01, 0x08, 0x89, 0x33};     // add x18, x17, x16

    protected override void OnTick() {
        ulong temp_address = m_input.Address;
        uint temp_instruction;

        if (temp_address >= 0 && temp_address < (uint)Instruction_Memory.Length) {
            temp_instruction = 0u | (uint)Instruction_Memory[temp_address]     << 24
                                  | (uint)Instruction_Memory[temp_address + 1] << 16
                                  | (uint)Instruction_Memory[temp_address + 2] << 8
                                  | (uint)Instruction_Memory[temp_address + 3];

            m_Instruction.Current = temp_instruction;                         // Full instruction
            m_read_1.Address       = (uint)temp_instruction >> 15 & (uint)31; // Takes out bit number 15 to 19 from instruction
            m_read_2.Address       = (uint)temp_instruction >> 20 & (uint)31; // Takes out bit number 20 to 24 from instruction
            m_write.Address        = (uint)temp_instruction >> 7  & (uint)31; // Takes out bit number 7 to 11 from instruction

            m_CPU.Running = true; // Keep CPU running
        }
        else {
            temp_instruction = 0u; // No Instruction

            m_Instruction.Current = temp_instruction;                         // Full instruction
            m_read_1.Address       = (uint)temp_instruction >> 15 & (uint)31; // Takes out bit number 15 to 19 from instruction
            m_read_2.Address       = (uint)temp_instruction >> 20 & (uint)31; // Takes out bit number 20 to 24 from instruction
            m_write.Address        = (uint)temp_instruction >> 7  & (uint)31; // Takes out bit number 7 to 11 from instruction

            m_CPU.Running = false; // Turn of cpu no more instructions
        }

```

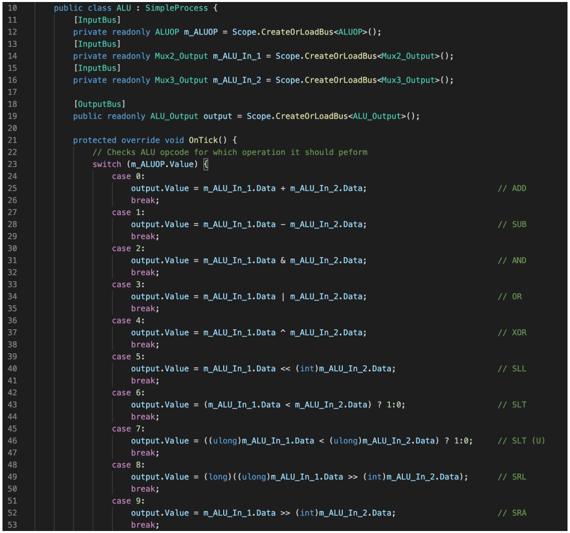
Instruction decode and execution

Storage for data we currently work with and unit for execution for instructions



Taken from *Computer organization and design RISC V*

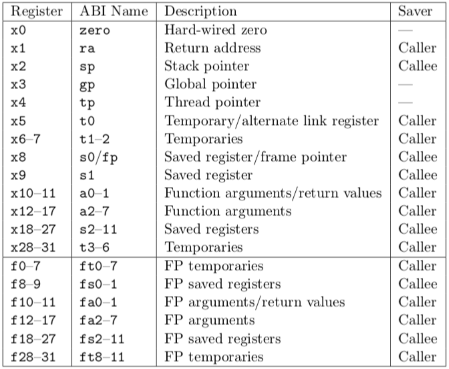
Instruction decode and execution



```
public class ALU : SimpleProcess {
    [InputBus]
    private readonly ALUOP m_ALUOP = Scope.CreateOrLoadBus<ALUOP>();
    [InputBus]
    private readonly Mux2_Output m_ALU_In_1 = Scope.CreateOrLoadBus<Mux2_Output>();
    [InputBus]
    private readonly Mux3_Output m_ALU_In_2 = Scope.CreateOrLoadBus<Mux3_Output>();

    [OutputBus]
    public readonly ALU_Output output = Scope.CreateOrLoadBus<ALU_Output>();

    protected override void OnTick() {
        // Checks ALU opcode for which operation it should peform
        switch (m_ALUOP.Value) {
            case 0:
                output.Value = m_ALU_In_1.Data + m_ALU_In_2.Data;                           // ADD
                break;
            case 1:
                output.Value = m_ALU_In_1.Data - m_ALU_In_2.Data;                           // SUB
                break;
            case 2:
                output.Value = m_ALU_In_1.Data & m_ALU_In_2.Data;                           // AND
                break;
            case 3:
                output.Value = m_ALU_In_1.Data | m_ALU_In_2.Data;                           // OR
                break;
            case 4:
                output.Value = m_ALU_In_1.Data ^ m_ALU_In_2.Data;                           // XOR
                break;
            case 5:
                output.Value = m_ALU_In_1.Data << (int)m_ALU_In_2.Data;                     // SLL
                break;
            case 6:
                output.Value = (m_ALU_In_1.Data < m_ALU_In_2.Data) ? 1:0;                   // SLT
                break;
            case 7:
                output.Value = ((ulong)m_ALU_In_1.Data < (ulong)m_ALU_In_2.Data) ? 1:0;     // SLT (U)
                break;
            case 8:
                output.Value = (long)((ulong)m_ALU_In_1.Data >> (int)m_ALU_In_2.Data);      // SRL
                break;
            case 9:
                output.Value = m_ALU_In_1.Data >> (int)m_ALU_In_2.Data;                     // SRA
                break;
```

Instruction decode and execution

Conventions according to RISC-V spec sheet

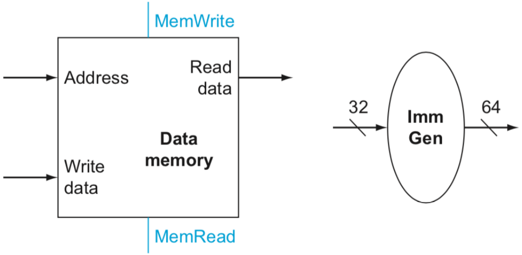


Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Taken from *The RISC-V Instruction Set Manual - Volume I: User-Level ISA*

Memory access

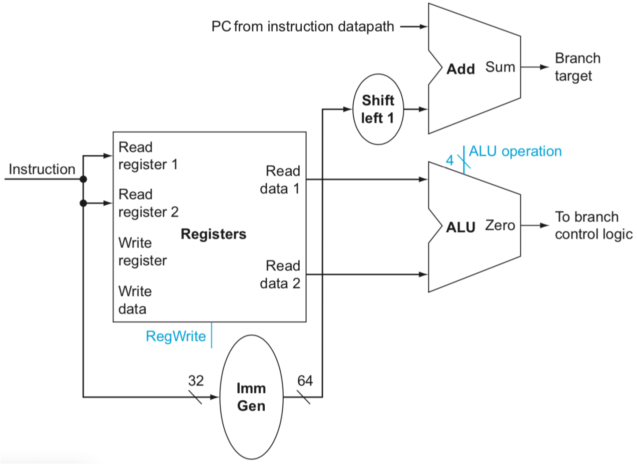
We need memory unit for storing data and immediate generation unit for calculating memory address



Taken from *Computer organization and design RISC V*

Branching

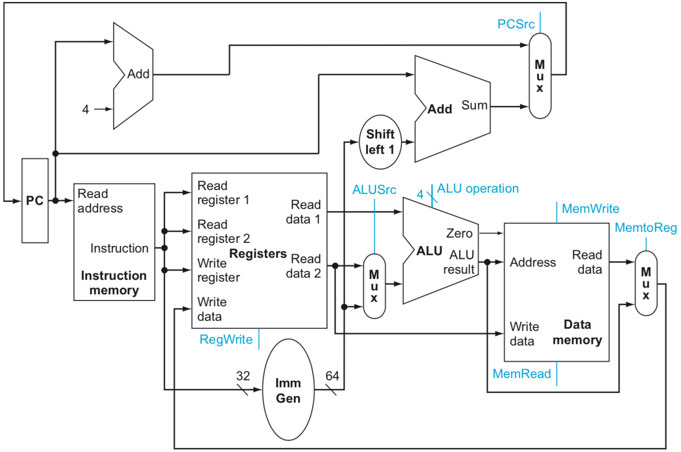
Additional unit needed for branching instructions



Taken from *Computer organization and design RISC V*

Simple datapath

Wiring it up we have our simple datapath



Taken from *Computer organization and design RISC V*

Instructions

Instruction Formats

31 … 26	25	24 … 20	19 … 15	14 … 12	11 … 7	6 … 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]	imm[5]	imm[4:0]	rs1	funct3	rd	opcode	I-type*
imm[11:5]		rs2	rs1	imm[4:0]	rd	opcode	S-type
imm[12|10:5]		rs2	rs1	imm[4:1|11]	rd	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

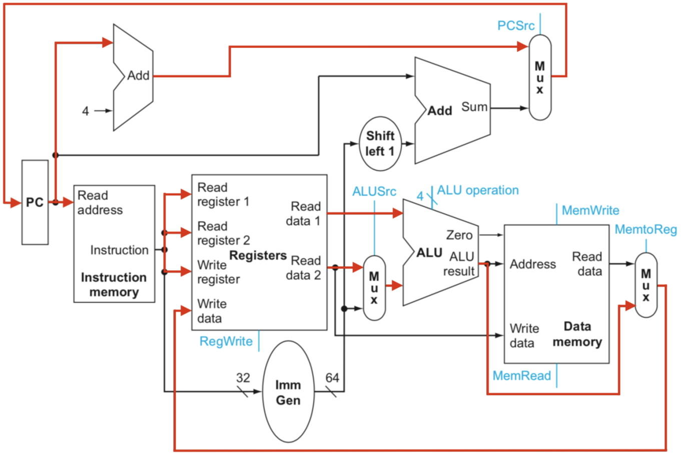
* This is a special case of the RV64I I-type format used by slli, srli and srai instructions where the lower 6 bits (imm[5] and imm[4:0]) are used to determine the shift amount (shamt). If slliw, srliw and sraiw are used it should generate an error if imm[5] $\neq 0$

RV64I Base Instructions

Name	Fmt	Opcode	Funct3	Funct7/ imm[11:5]	Assembly	Description (in C)
Add	R	0110011	000	0000000	add rd, rs1, rs2	rd = rs1 + rs2
Subtract	R	0110011	000	0100000	sub rd, rs1, rs2	rd = rs1 − rs2
AND	R	0110011	111	0000000	and rd, rs1, rs2	rd = rs1 & rs2
OR	R	0110011	110	0000000	or rd, rs1, rs2	rd = rs1 | rs2
XOR	R	0110011	100	0000000	xor rd, rs1, rs2	rd = rs1 ˆ rs2

Simple datapath

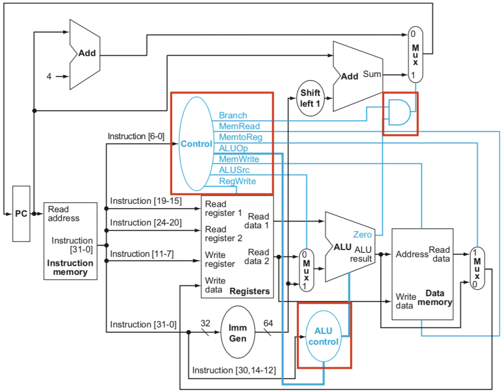
The R-Type instruction datapath would look like



Taken from *Computer organization and design RISC V*

Control

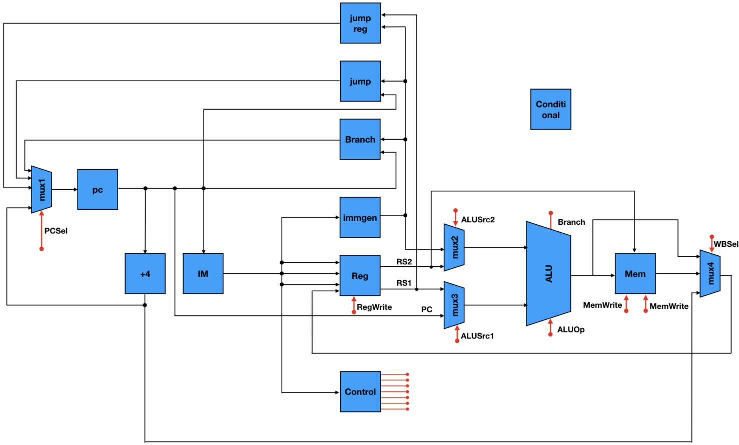
We need control unit to determine path for instructions. Main Control and ALU control are separated to simplify complexity of system.



Taken from *Computer organization and design RISC V*

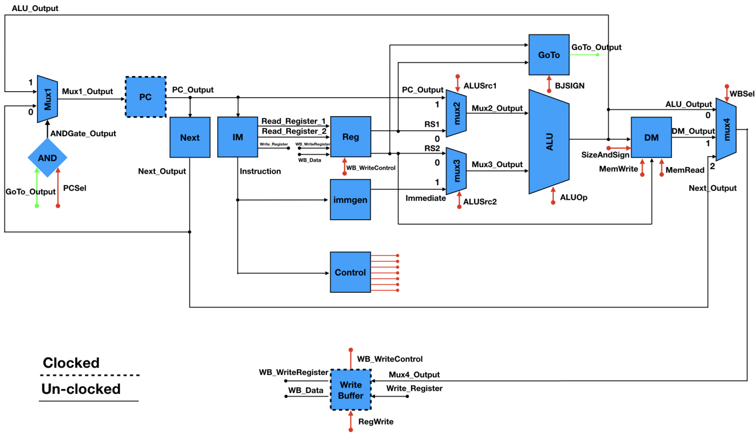
Single Cycle RISC-V version 2.0

Previous datapath not capable of running all standard instructions.
So we add "missing" units.



Single Cycle RISC-V version 2.0

We notice many units have almost same function.
So datapath can be significantly simplified.



Further work

- Test on a FPGA
- Run Linux on it
- Pipeline the datapath for improved performance