QtRvSim – RISC-V Simulator for Computer Architectures Classes

Embedded World Conference 2022

Session 10.3 – System-on-Chip (SoC) Design RISC-V Development

Czech Technical University in Prague

More information: https://comparch.edu.cvut.cz/

QtRVSim https://github.com/cvut/qtrvsim

Jakub Dupák <dupakjak@fel.cvut.cz>

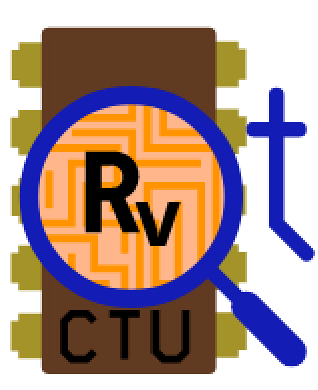
Pavel Píša <pisa@fel.cvut.cz>

Karel Kočí <cynerd@email.cz>

Michal Štepanovský <michal.stepanovsky@gmail.com>

Presentation Outline

- **Simulator Features**
- **Implementation Overview for Future Extending**
- **Example of our Teaching**



1. Simulator Features

QtRVSim – RISC-V Computer Architecture Simulator

Load

Run

Single Step

Make

Exceptions control

Registers

Terminal

Assembler

Editor

Peripherals

Code

CPU core view
- single cycle
- pipelined

Cache

Data memory

Basic view

File Machine Windows Help

1x 2x 5x 10x Unlimited Max

Registers

x0/zero	0x0	x1/ra	0x0	x2/sp	0xbffffec0	x3/gp	0x0	x4/tp	0x0	x5/t0	0x0
x6/t1	0x0	x7/t2	0x0	x8/s0	0x0	x9/s1	0x0	x10/a0	0x0	x11/a1	0x0
x12/a2	0x0	x13/a3	0x0	x14/a4	0x0	x15/a5	0x0	x16/a6	0x0	x17/a7	0x0
x18/s2	0x0	x19/s3	0x0	x20/s4	0x0	x21/s5	0x0	x22/s6	0x0	x23/s7	0x0
x24/s8	0x0	x25/s9	0x0	x26/s10	0x0	x27/s11	0x0	x28/t3	0x0	x29/t4	0x0

Control and Status Registers

mstatus	0x0	mtvec	0x100
mepc	0x0	mcause	0x0
mtval	0x0	mcycle	0x8
Compare	0x0		

Program

Follow fetch

Bp	Address	Code	Instruction
	0x000110b4	fc010113	addi x2, x2, -64
	0x000110b8	02112e23	sw x1, 60(x2)
	0x000110bc	02812c23	sw x8, 56(x2)
	0x000110c0	02912a23	sw x9, 52(x2)
	0x000110c4	03212823	sw x18, 48(x2)
	0x000110c8	03312623	sw x19, 44(x2)
	0x000110cc	03412423	sw x20, 40(x2)
	0x000110d0	03512223	sw x21, 36(x2)
	0x000110d4	03612023	sw x22, 32(x2)

0x000110b4

Core | simple-lw-sw-ia.S | template.S | template-os.S | template-os.S

sw x21, 36(x2) | sw x20, 40(x2) | sw x19, 44(x2) | sw x18, 48(x2) | sw x9, 52(x2)

Ready

Pipeline Visualization

Single Cycle Core

File Machine Windows Help

1x 2x 5x 10x Unlimited Max

Core | simple-lw-sw-ia.S | template.S | template-os.S | template-os.S

nop

NONE

Cycles: 0

Stalls: 0

BranchOutcome

Control Unit

MemToReg
MemWrite
MemRead
BranchBxx
BranchJal
BranchJalr
BranchVal
AluControl
AluMul
AluSrc
AuiPC

xor

Branch Jalr

Branch Jalx

MemWrite

MemRead

RegWrite

Instruction

00000000

PC

0xffffffff

Cache

Hit: 0

Miss: 0

Program Memory

rs1
rs2
rd

Registers

Immediate decode

zero

ALU

AluOut

WriteData

Peripherals

Terminal

Cache

Hit: 0

Miss: 0

Data Memory

PC

PC+4

rd

RegWriteData

BranchTarget

Ready

Integrated editor

File Machine Windows Help

1x 2x 5x 10x Unlimited Max

Core | simple-lw-sw-ia.S | template.S | template-os.S | template-os.S

Run external Makefile

Run internal assembler

New file | Open | Save | Close

```
.globl _start
.option norelax

.text

_start:
loop:
        // load the word from absolute address
        lw      x2, 0x400(x0)
        // store the word to absolute address
        sw      x2, 0x404(x0)

        // stop execution wait for debugger/user
        // break
        // ensure that continuation does not
        // interpret random data
        beq     x0, x0, loop
        nop
        nop
        ebreak

.data
.org 0x400

src_val:
```

Memory and Cache

File Machine Windows Help

1x 2x 5x 10x Unlimited Max

Core

Memory

Word | Direct

Address	+0	+4	+8	+12
0x00000400	00000000	00000000	00000000	00000000
0x00000410	00000000	00000000	00000000	00000000
0x00000420	00000000	00000000	00000000	00000000
0x00000430	00000000	00000000	00000000	00000000
0x00000440	00000000	00000000	00000000	00000000
0x00000450	00000000	00000000	00000000	00000000
0x00000460	00000000	00000000	00000000	00000000
0x00000470	00000000	00000000	00000000	00000000
0x00000480	00000000	00000000	00000000	00000000
0x00000490	00000000	00000000	00000000	00000000
0x000004a0	00000000	00000000	00000000	00000000
0x000004b0	00000000	00000000	00000000	00000000
0x000004c0	00000000	00000000	00000000	00000000
0x000004d0	00000000	00000000	00000000	00000000

0x00000400

Data Cache

Hit: 0
Miss: 0
Memory reads: 0
Memory writes: 0
Memory stall cycles: 0
Hit rate: 0.000%
Improved speed: 100%

Address 0x00000000
0000000 0 0

V Tag Data
0
0
0
0

V Tag Data
0
0
0
0

Different Cache Configuration

File Machine Windows Help

1x 2x 5x 10x Unlimited Max

Core

Memory

Word ▾ Direct ▾

Address	+0	+4	+8	+12
0x00000400	00000000	00000000	00000000	00000000
0x00000410	00000000	00000000	00000000	00000000
0x00000420	00000000	00000000	00000000	00000000
0x00000430	00000000	00000000	00000000	00000000
0x00000440	00000000	00000000	00000000	00000000
0x00000450	00000000	00000000	00000000	00000000
0x00000460	00000000	00000000	00000000	00000000
0x00000470	00000000	00000000	00000000	00000000
0x00000480	00000000	00000000	00000000	00000000
0x00000490	00000000	00000000	00000000	00000000
0x000004a0	00000000	00000000	00000000	00000000
0x000004b0	00000000	00000000	00000000	00000000
0x000004c0	00000000	00000000	00000000	00000000
0x000004d0	00000000	00000000	00000000	00000000

0x00000400

Data Cache

Hit:	0
Miss:	0
Memory reads:	0
Memory writes:	0
Memory stall cycles:	0
Hit rate:	0.000%
Improved speed:	100%

Ready

Configuration

Basic | Core | Memory | Program cache | Data cache | OS Emulation

Preset

- No pipeline no cache
- No pipeline with cache
- Pipelined without hazard unit and without cache
- Pipelined with hazard unit and cache
- (•) Custom

[✓] Reset at compile time (reload after make)

Elf executable: ie/viper/Dev/school/QtMips_support/qtmips-samples/fb-simple/a.out Browse

Basic | Core | Memory | Program cache | Data cache | OS Emulation

[✓] Pipelined

[✓] Delay slot

[✓] Hazard unit

- Stall when hazard is detected
- (•) Stall or forward when hazard is detected

Basic | Core | Memory | Program cache | Data cache | OS Emulation

[✓] Enable cache

Number of sets: 4

Block size: 2

Degree of associativity: 2

Replacement policy: Random

Basic | Core | Memory | Program cache | Data cache | OS Emulation

[✓] Enable emulation of operating system services

[✓] Stop on known system call

[✓] Stop on unknown system call

[✓] Stop on interrupt entry

[✓] Stop and step over exceptions (overflow, etc.)

Filesystem root: Browse

Memory Mapped Peripherals Emulation

File Machine Windows Help

1x 2x 5x 10x Unlimited Max

Peripherals

LED RGB 1

00000000

LED RGB 2

00000000

Red Knob

0

Green Knob

0

Blue Knob

0

Word hexadecimal

00000000

Word decimal

0

Word binary

00000000000000000000000000000000

Core

Terminal

Input:

LCD Display

Motivation for Peripherals Emulation

- Educational kit for Computer Architectures Course
- Microzed XylinX Zynq 7Z010
- Set of basic peripherals implemented in VHDL – LEDs, knobs, LCD interface

https://cw.fel.cvut.cz/wiki/courses/b35apo/en/documentation/mz_apo/start

2. Implementation Overview

(Guide for Future Extensions)

Command Line Interface

```
>>> qtrvsim_cli --asm program.S --trace-fetch
Fetch: addi x1, x0, 17
Fetch: addi x2, x0, 34
Fetch: addi x3, x0, 51
Fetch: lw x4, 68(x0)
Fetch: addi x5, x4, 85
Fetch: beq x0, x0, 0x22c
Fetch: addi x21, x0, 17
Fetch: addi x22, x0, 34
Fetch: addi x23, x0, 51
Fetch: addi x24, x0, 68
Fetch: addi x25, x0, 85
Machine stopped on BREAK exception.
Fetch: ebreak
```

Instruction Decoding

size
2:0

32bit
7:2

load
15:12

op
26:25

system
21:20

ecall

ebreak

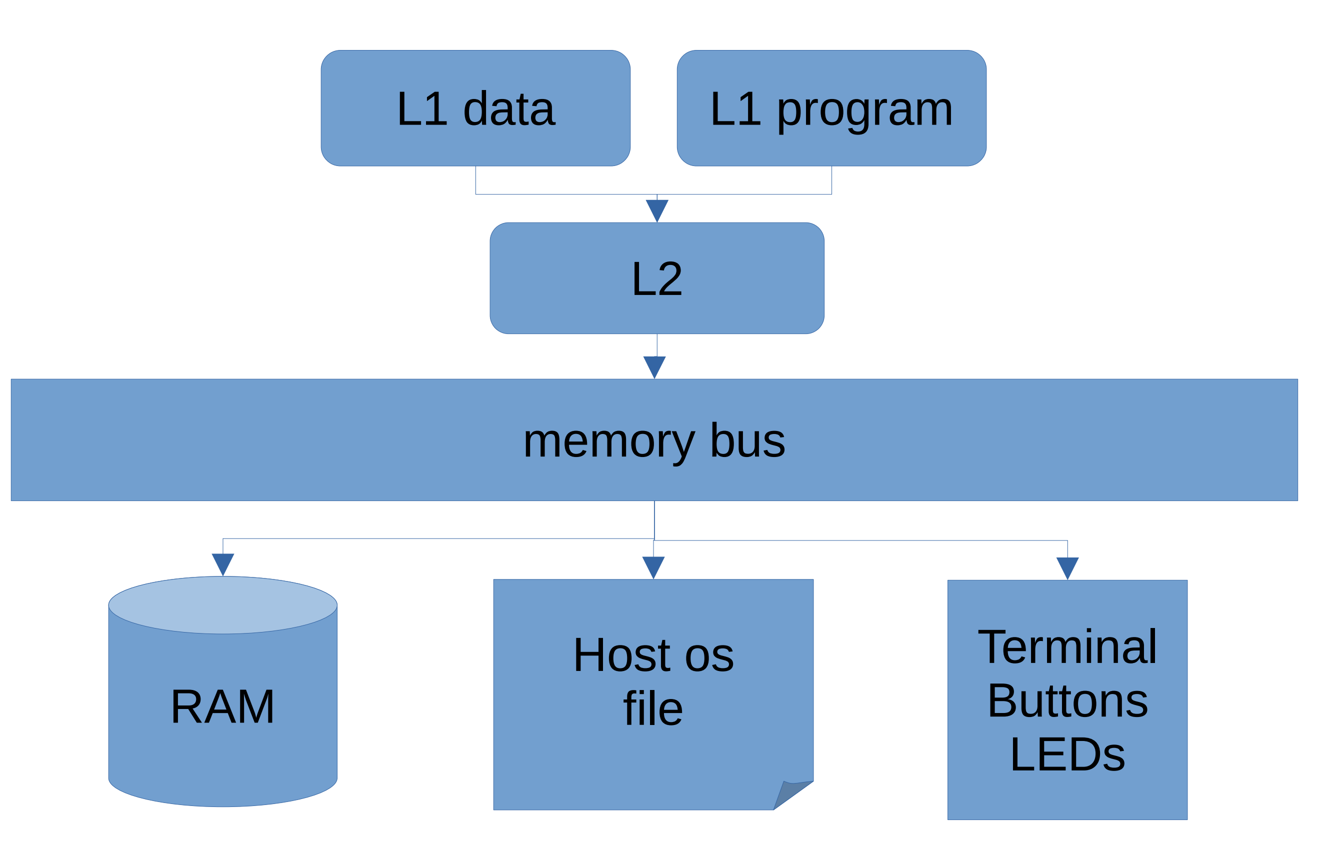
Instruction Decoding

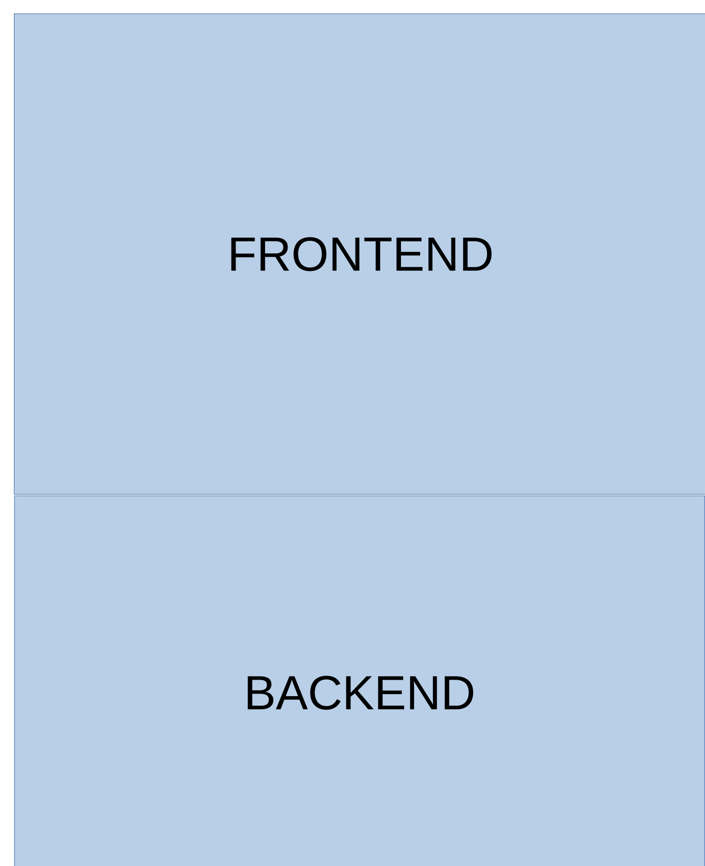
- Instruction map
  - Name — assembly/disasembly
  - Type — immediate handling
  - ALU operation — execution
  - Memory operation — execution
  - Subclass pointer — instruction hiearchy
  - Code mask, forbiden mask — validation
  - Assembler arguments — assembler parsing, to string
  - Fields — encoding/decoding

Instruction Decoding

```
{
    "addi", IT_I,
    { .alu_op=AluOp::ADD },
    NOMEM,
    nullptr,
    {"d", "s", "j"},
    0x00000013, 0x0000707f,
    { .flags = FLAGS_ALU_I }
}
```

Memory Model





Core View Implementation

ID: uWEq7-W_z53UrtFael4O-90

component: instruction-value

source: fetch

tags: forwarding simple pipeline

Enter Property Name

Add Property

3. Example of our Lectures

excerpts from B35APO course

Lecture 3 – Central Processing Unit (CPU)
Lecture 5 – CPU Pipelined Execution
Lecture 7 – Input/Output Subsystem 1

QtRVSim – Download

- Project Sources
  https://github.com/cvut/qtrvsim
- Windows, Linux, Mac
  https://github.com/cvut/qtrvsim/releases
- Ubuntu
  https://launchpad.net/~qtrvsimteam/+archive/ubuntu/ppa
- Suse, Fedora and Debian
  https://software.opensuse.org/download.html?project=home%3Ajdupak&package=qtrvsim
- Online version
  https://comparch.edu.cvut.cz/qtrvsim/app

QtRVSim – Origin and Development

- **MipsIt** used in past for Computer Architecture course at the Czech Technical University in Prague, Faculty of Electrical Engineering
- Switch to **QtMips** in the 2019 summer semester
  - Result of diploma theses of Karel Kočí
    Graphical CPU Simulator with Cache Visualization
    https://dspace.cvut.cz/bitstream/handle/10467/76764/F3-DP-2018-Koci-Karel-diploma.pdf
- Switch **QtRVSim** in the 2022 summer semester
  - Graphical RISC-V Architecture Simulator - Memory Model and Project Management
    Jakub Dupák; 2021; Bachelor Thesis
    https://dspace.cvut.cz/bitstream/handle/10467/94446/F3-BP-2021-Dupak-Jakub-thesis.pdf
  - Graphical RISC-V Architecture Simulator - Instructions Decode and Execution and OS Emulation
    Max Hollmann; 2021; Bachelor Thesis
    https://dspace.cvut.cz/bitstream/handle/10467/96707/F3-BP-2021-Hollmann-Max-thesis.pdf

MIPS and RISC-V Education Simulator Alternatives

- SPIM/QtSPIM: A MIPS32 Simulator
  http://spimsimulator.sourceforge.net/
- MARS: IDE with detailed help and hints
  http://courses.missouristate.edu/KenVollmar/MARS/index.htm
- EduMIPS64: 1x fixed and 3x FP pipelines
  https://www.edumips.org/
- RARS – RISC-V Assembler and Runtime Simulator
  https://github.com/TheThirdOne/rars
- RIPES – A graphical processor simulator and assembly editor for the RISC-V ISA
  https://github.com/mortbopet/Ripes/

The main instruction cycle of the CPU

1) Initial setup/reset – set initial PC value, PSW, etc.
2) Read the instruction from the memory
   - PC → to the address bus
   - Read the memory contents (machine instruction) and transfer it to the IR
   - PC+l → PC, where l is length of the instruction
3) Decode operation code (opcode)
4) Execute the operation
   - compute effective address, select registers, read operands, pass them through ALU and store result
5) Check for exceptions/interrupts (and service them)
6) Repeat from the step 2

Compilation: C → Assembler → Machine Code

```
/* ffs as log2(x)*/
int x = 157;
int y = 0;

while(x != 0)
{
  x = x / 2;
  y = y + 1;
}
```

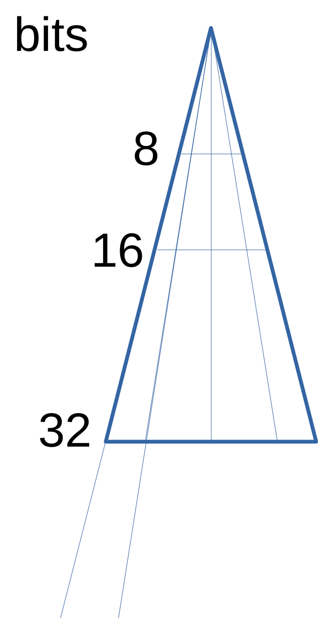
```
_start: addi   a0, zero, 157     // int x = 157;
        addi   t1, zero, -1      // int y = 0;
        beq    a0, zero, done    // while(x != 0) {
loop:   srli   a0, a0, 1         //   x = x / 2;
        addi   t1, t1, 1         //   y = y + 1;
        bne    a0, zero, loop    //}
done:   addi   a0, t1, 0
finish: ebreak
        beq    zero, zero, fini
```

```
0x00000200   09d00513   addi x10, x0, 157
0x00000204   fff00313   addi x6, x0, -1
0x00000208   00050863   beq x10, x0, 0x218
0x0000020c   00155513   srli x10, x10, 0x1
0x00000210   00130313   addi x6, x6, 1
0x00000214   fe051ce3   bne x10, x0, 0x20c
0x00000218   00030513   addi x10, x6, 0
0x0000021c   00100073   ebreak
```

https://gitlab.fel.cvut.cz/b35apo/stud-support/

seminaries/qtrvsim/ffs-as-log2

Instruction Encoding Constrains



- Encode 256 combinations in 8-bits
- Opcode and address to directly operate on megabytes of ram does not fit
- Use some limited number of fast accessible registers or use stack concept
- 8 registers, 3 bits to encode, for two operand operations (a += b, mem[a] = b), 6 bits to select registers → only 4 operations in total
- 16 bit (65536), 16 registers, 256 two operands or 16 three operands (4 + 3 * 4 bits)
- 32 bit, 32 registers, three operands (17 + 3 * 5)
- But immediate for arithmetic and address usually required (CISC followup words, RISC uses limited ranges and some other mechanism)

RISC-V – Instruction Length Encoding

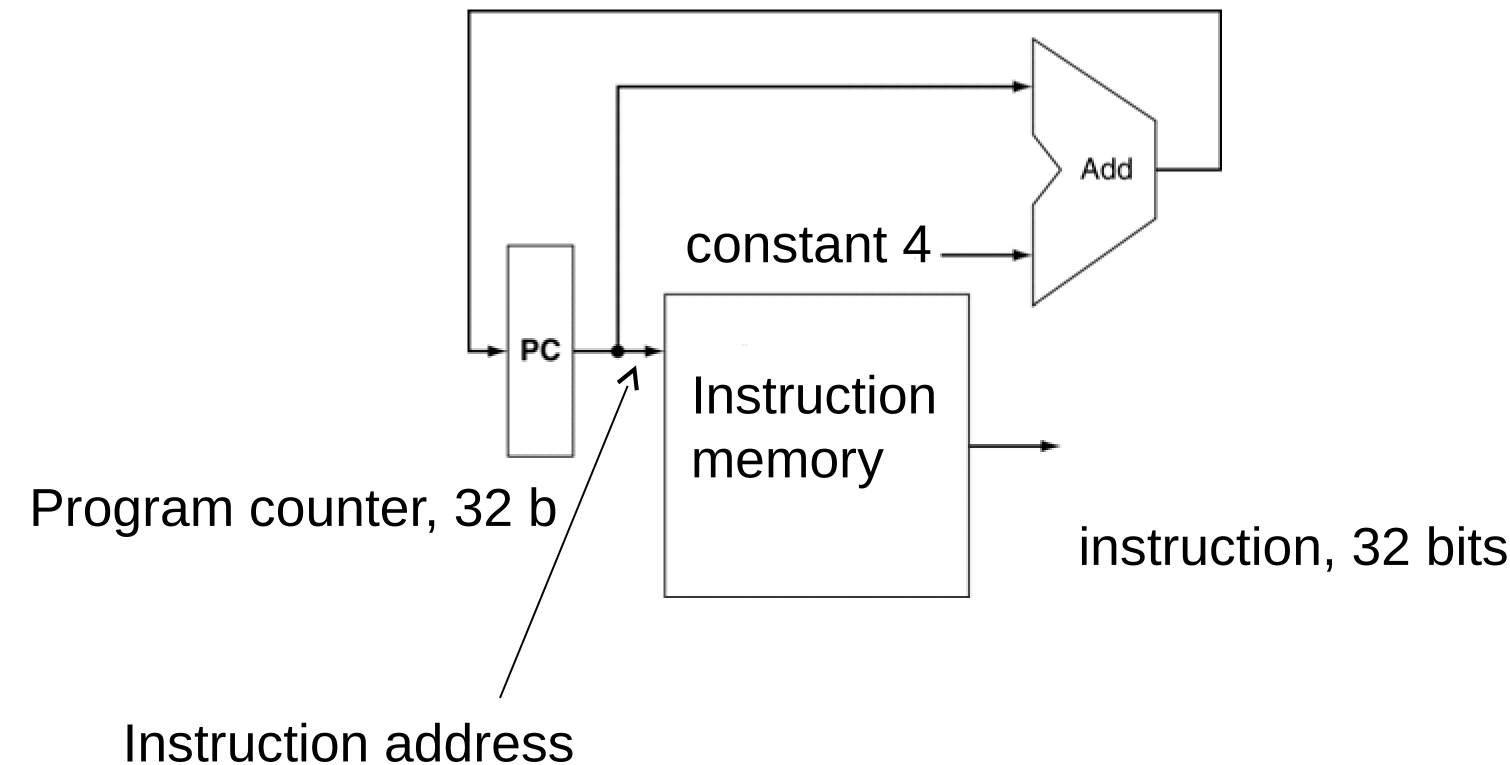
base+4	base+2	base	
		xxxxxxxxxxxxxxaa	16-bit (aa $\neq$ 11)
	xxxxxxxxxxxxxxxx	xxxxxxxxxxxbbb11	32-bit (bbb $\neq$ 111)
· · ·xxxx	xxxxxxxxxxxxxxxx	xxxxxxxxxx011111	48-bit
· · ·xxxx	xxxxxxxxxxxxxxxx	xxxxxxxxx0111111	64-bit
· · ·xxxx	xxxxxxxxxxxxxxxx	xnnnxxxxx1111111	(80+16*nnn)-bit, nnn $\neq$ 111
· · ·xxxx	xxxxxxxxxxxxxxxx	x111xxxxx1111111	Reserved for ≥192-bits

Address:
base+4 base+2 base

RISC-V Processor Registers

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
pc	pc	Program Counter	
f0–31		Floating point	
		Machine control and status	

Hardware Realization of Basic (main) CPU Cycle



Design of the Single Cycle CPU

- To understand the implementation of a simple computer consisting of CPU and separated instruction and data memory
- Our goal is to implement following instructions:
  - Read and write a value from/to the data memory
    `lw` – load word, `sw` – store word
  - Arithmetic and logic instructions: **add**, **sub**, **and**, **or**, **slt**
  - Immediate variants: **addi**, **ori**, upper bits loads **lui**, **auipc**
  - Program flow change/jump instruction **beq**
  - Subroutine call **jal**, **jalr** (provides even return from subroutine **jr ra**)
- CPU will consist of control unit and ALU (data path).
- Notes:
  - The implementation will be minimal (single cycle CPU – all operations processed in the single step/clock period)
  - The B35APO lecture 5 focuses on more realistic pipelined CPU implementation

Subset of RISC-V Instructions to Implement

Instruction	Executed operation
lw rd, imm12(rs1)	[rd] ← Mem[[rs1] + imm12];
sw rs2, imm12(rs1)	Mem[[rs1] + imm12] ← [rs2];
addi rd, rs1, imm12	[rd] ← [rs1] + imm12;
add rd, rs1, rs2	[rd] ← [rs1] + [rs2];
sub rd, rs1, rs2	[rd] ← [rs1] – [rs2];
and rd, rs1, rs2	[rd] ← [rs1] & [rs2];
or rd, rs1, rs2	[rd] ← [rs1] | [rs2];
slt rd, rs1, rs2	[rd] ← [rs1] < [rs2];
beq rs1, rs2, imm12	if [rs1] == [rs2] go to [PC]+(imm12<<1); else go to [PC]+4;
jal rd, imm20	[rd] ← [PC]+4; go to [PC]+(imm20<<1);
jalr rd, rs1, imm12	[rd] ← [PC]+4; go to [rs1]+imm12;
lui rd, imm20	[rd] ← imm20<<12
auipc rd, imm20	[rd] ← PC+(imm20<<12)

Remark, all immediate operands are signed

The RISC-V Instruction Format and Instruction Types

- The six basic formats of the instructions are considered:

31	30 25	24 21	20	19 15	14 12	11 8	7	6 0	
funct7		rs2		rs1	funct3	rd		opcode	R-type
imm[11:0]				rs1	funct3	rd		opcode	I-type
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type

- opcode – specify operation, func 3 and func 7 variant
- rs1 – source 1, rs2 – source 2, rd – destination
- 5 bits allows to encode 32 GPRs ($0 is hardwired to 0/discard)

RISC-V Immediate Operand Encoding

- There are five options how to form immediate operand

31	30 20	19 12	11	10 5	4 1	0	
— inst[31] —				inst[30:25]	inst[24:21]	inst[20]	I-immediate
— inst[31] —				inst[30:25]	inst[11:8]	inst[7]	S-immediate
— inst[31] —			inst[7]	inst[30:25]	inst[11:8]	0	B-immediate
inst[31]	inst[30:20]	inst[19:12]	— 0 —				U-immediate
— inst[31] —		inst[19:12]	inst[20]	inst[30:25]	inst[24:21]	0	J-immediate

- goal: keep same position in the instruction for corresponding bit (compare U and J, B and S, S and I)
- sign in instruction bit 31 (needs fan-out to drive multiple bits)

Opcode encoding

The primary selection of operation is **opcode**

Opcode	The group, operation	Actual operations for our subset
0110011	R-type (see funct7 and funct3)	add, sub, slt, or, and
0010011	ALU-imm (see funct3)	addi
0000011	Memory load (funct3 = 010)	lw
0100011	Memory store (funct3)	sw
1100011	Branch (see funct3)	beq
1101111	jal	jal
1100111	jalr	jalr
0000111	lui	lui

func7	fun3	instruction/ALU operation
0000000	000	add
0100000	000	sub
0000000	010	slt
0000000	110	or
0000000	111	and

CPU Building Blocks

CLK
PC'
PC
32
32
32
A RD
Instr.
Memory
(ROM)
32
CLK
WE3
5 A1 RD1 32
5 A2 RD2 32
5 A3
32 WD3
Reg.
File
CLK
WE
32 A RD 32
Data
Memory
32 WD
0
1

Write at the rising edge of CLK when WE = 1

Multiplexer

Read after “enough time” for data propagation

4 ALU operation
Zero
ALU
ALU
result

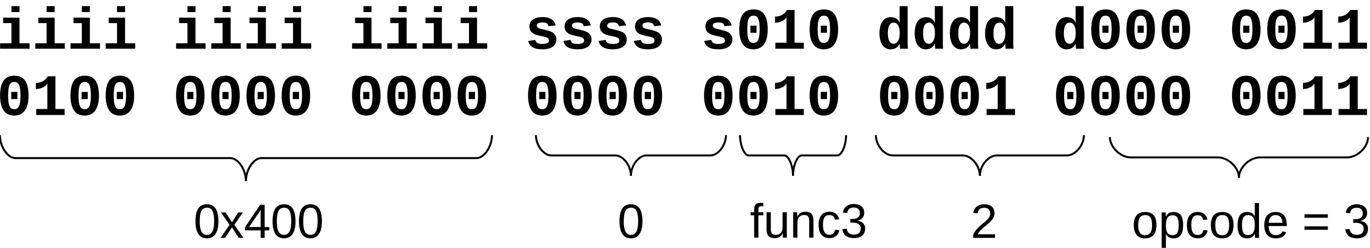
State element 1
Combinational logic
State element 2
Clock cycle

The Load Word Instruction

`lw` – **load word** – load word from data memory into a register

Description	A word is loaded into a register from the specified address
Operation:	[rd] ← Mem[[rs1] + imm12];
Syntax:	lw rd, imm12(rs1)
Encoding:	`iiii iiii iiii ssss s010 dddd d000 0011`

Example: Read word from memory address 0x400 into register number 2:
lw x2, 0x400(x0)

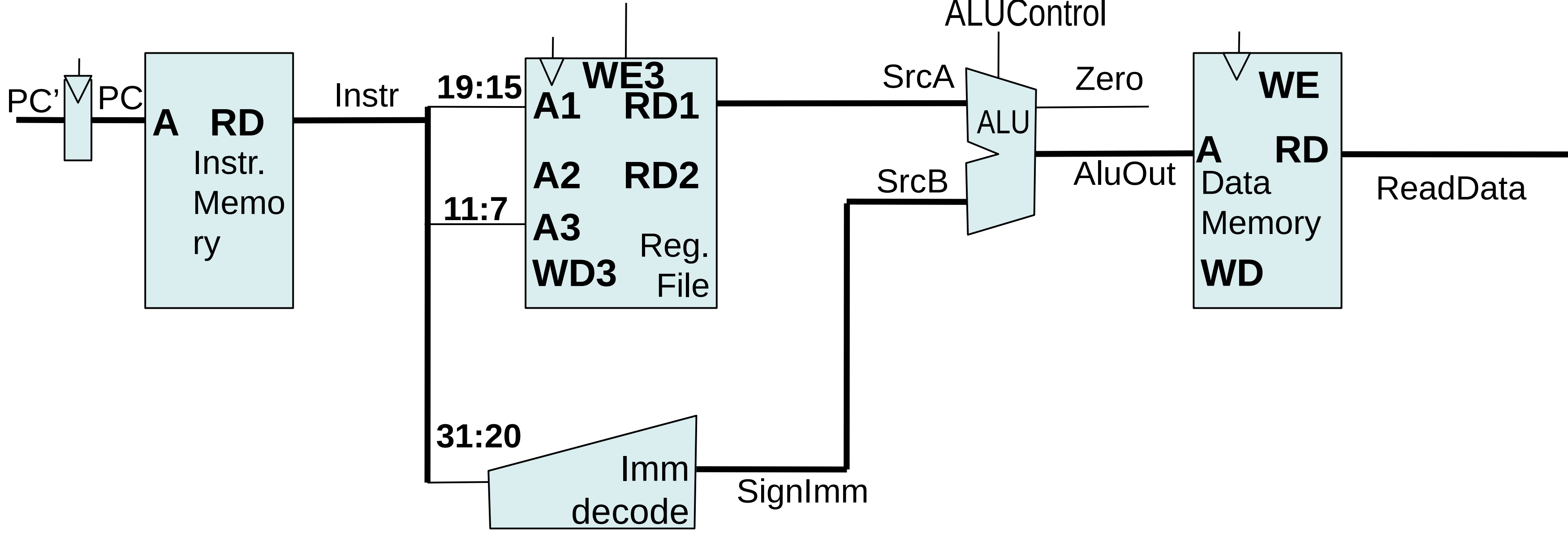


0x 40 00 21 03 – machine code for instruction lw x2, 0x400(x0)
Note: Register x0 is hardwired to the zero

Implementation of the load instruction

`lw`: rs1 – base address, imm12 – offset, rd – register where to store fetched data

I	**imm(12)**, 31:20	**rs1(5)**, 19:15	**func3**, 14:12	**rd**(5), 11:7	**opcode**(7), 6:0



Implementation of the load instruction

lw: rs1 – base address, imm12 – offset, rd – register where to store fetched data

I	**imm(12)**, 31:20	**rs1(5)**, 19:15	**func3**, 14:12	**rd**(5), 11:7	**opcode**(7), 6:0

Write at the rising edge of the clock

RegWrite = 1

ALUControl

PC' PC

A RD Instr. Memory

Instr

19:15

11:7

31:20

WE3 A1 RD1 A2 RD2 A3 WD3 Reg. File

SrcA SrcB ALU Zero AluOut

WE A RD Data Memory WD

ReadData

Imm decode

SignImm

Implementation of the load instruction – PC += 4

lw: rs1 – base address, imm12 – offset, rd – register where to store fetched data

I	**imm(12)**, 31:20	**rs1(5)**, 19:15	**func3**, 14:12	**rd**(5), 11:7	**opcode**(7), 6:0

RegWrite = 1
ALUControl
PC'
PC
A
RD
Instr.
Memory
Instr
19:15
11:7
WE3
A1
RD1
A2
RD2
A3
WD3
Reg.
File
SrcA
SrcB
ALU
Zero
AluOut
WE
A
RD
Data
Memory
WD
ReadData
+
4
PCPlus4
31:20
Imm
decode
SignImm

The Store Word Instruction

sw – store word – store word in a register to data memory

Description	Stores a value in register rs2 to given address in memory.
Operation:	Mem[[rs1] + imm12] = [rs2]
Syntax:	sw rs2,imm12(rs1)
Encoding:	`iiii iiit tttt ssss s010 iiii i000 0011`

Example: Store word in register 2 to memory address computed as addition of value in register 5 and constant 0x404, bit symbol **t** used for rs2:
sw x2, 0x404(x5)

```
iiii iiit tttt ssss s010 iiii i000 0011
0100 0000 0010 0010 1010 0010 0010 0011
```

0x40_ 2 5 func3 0x_04 opcode = 35

0x 40 22 a2 23 – machine code for instruction sw x2, 0x404(x5)

The Store Word Instruction

sw: rs1 – base address, imm12 – offset, rs2 – select register to store into memory

S	**imm(12)**, 31:25, 11:7	**rs2(5)**, 24:20	**rs1(5)**, 19:15	**func3**, 14:12	**opcode**(7), 6:0

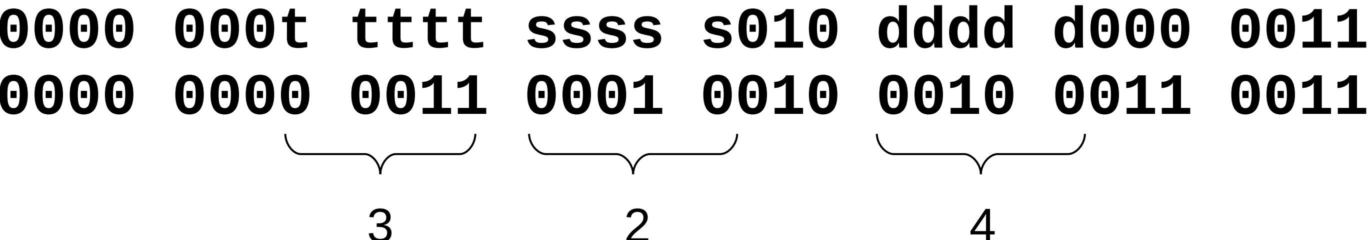
RegWrite = 0
ALUControl
MemWrite = 1
PC'
PC
A **RD**
Instr. Memory
Instr
19:15
24:20
11:7
WE3
A1 **RD1**
A2 **RD2**
A3
WD3
Reg. File
SrcA
SrcB
ALU
Zero
AluOut
WE
A **RD**
Data Memory
WD
ReadData
WriteData
+
4
PCPlus4
31:25, 24:20
11:7
Imm decode
SignImm

Instruction for Two Registers Addition

add – **addition** – add content of two registers and store it to destination one

Description	Add together values in two registers (rs1 + rs2) and stores the result in register rd.
Operation:	[rd] = [rs1] + [rs2]
Syntax:	add rd, rs1, rs2
Encoding:	`0000 000t tttt ssss s010 dddd d000 0011`

Example: Add values in registers 1 and 2 and store result into register 3:
add x4, x2, x3



0x 00 31 02 33 – machine code for instruction add x4, x2, x3

Instruction for Two Registers Addition

add: type R, rs, rt – source, rd – destination, funct – select ALU operation = add

R	**func(7)**,31:25	**rs2(5)**,24:20	**rs1(5)**,19:15	**func3**, 14:12	**rd(5)**,11:7	**opcode**(7), 6:0

ALUSrc = 0
MemToReg = 0
RegWrite = 1
ALUControl
PC' PC
A RD Instr. Memory
Instr
19:15
24:20
11:7
WE3
A1 RD1
A2 RD2
A3
WD3
Reg. File
SrcA
SrcB
0
1
ALU
Zero
AluOut
WE
A RD
Data Memory
WD
WriteData
ReadData
0
1
Result
+
4
PCPlus4
31:25, 24:20
11:7
Imm decode
SignImm

Single cycle CPU – sub, and, or, slt

Only difference is another ALU operation selection (ALUcontrol). The data path is the same as for **add** instruction

ALUSrc = 0
MemToReg = 0
RegWrite = 1
ALUControl
PC'
PC
A
RD
Instr.
Memory
Instr
19:15
24:20
11:7
WE3
A1
RD1
A2
RD2
A3
WD3
Reg.
File
SrcA
SrcB
ALU
Zero
AluOut
0
1
WriteData
WE
A
RD
Data
Memory
WD
ReadData
0
1
Result
+
4
PCPlus4
31:25, 24:20
11:7
Imm
decode
SignImm

Single cycle CPU – addi, ori, andi

addi – add immediate; [rd] = [rs1] + imm12

I	**imm(12)**, 31:20	**rs1(5)**, 15:19	**func3**, 14:12	**rd**(5), 7:11	**opcode**(7), 6:0

```
addi rd, rs, imm12
```

ALUSrc = 1

MemToReg = 0

RegWrite = 1

ALUControl

PC' PC A RD Instr. Memory Instr 19:15 24:20 11:7 WE3 A1 A2 A3 WD3 RD1 RD2 Reg. File SrcA SrcB 0 1 ALU Zero AluOut WE A RD Data Memory WD WriteData ReadData 0 1 Result + 4 PCPlus4 31:25, 24:20 11:7 Imm decode SignImm

ori, andi instructions do not sign extend the immediate operand

Single cycle CPU – implementation of beq

beq – branch if equal; imm–offset; PC´ = PC + SignImm

S	**imm(12)**, 31:25, 11:7	**rs2(5)**, 24:20	**rs1(5)**, 19:15	**func3**, 14:12	**opcode**(7), 6:0

Single Cycle CPU Throughput: IPS = IC / T = $IPC_{str}.f_{clk}$

- What is the maximal possible frequency of the CPU?
- It is given by latency on the critical path – it is `lw` instruction in our case:

$$T_c = t_{PC} + t_{Mem} + t_{RFread} + t_{ALU} + t_{Mem} + t_{Mux} + t_{RFsetup}$$

Non-pipelined Execution

0
1
PC'
PC
A RD
Instr.
Memory
Instr
19:15
24:20
11:7
15:0, 24:0
11:7
WE3
A1 RD1
A2 RD2
A3
WD3
Reg.
File
Imm
decode
SignImm
4
+
PCPlus4
SrcA
SrcB
0
1
ALU
Zero
AluOut
WriteData
WriteReg
PC
+
PCBranch
WE
A RD
Data
Memory
WD
ReadData
0
1
Result

Pipelined Execution

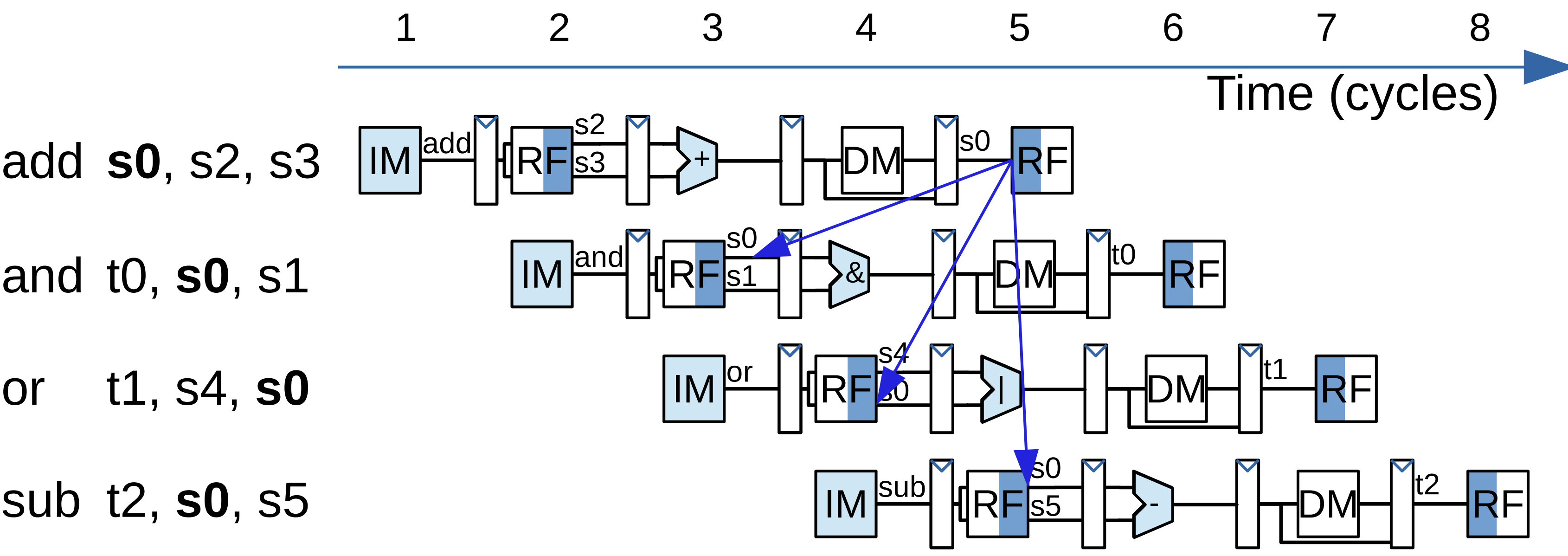
0 1 PC' PC
A RD
Instr.
Memory
Instr
19:15
24:20
WE3
A1 RD1
A2 RD2
A3
WD3
Reg.
File
11:7
15:0, 24:0
11:7
Imm
decode
4
+
PCPlus4
PCF
PCD
SrcA
0 1 SrcB
ALU
Zero
WriteDataE
WriteRegE
SignImm
PCE
+
PCBranch
AluOutM
WriteDataM
WriteRegM
WE
A RD
Data
Memory
WD
AluOutW
ReadData
WriteRegW
0 1 Result

Fetch Decode Execute Memory WriteBack

Pipelined Execution with Control Unit

Control unit
Op
Funct
6:0
31:25, 14:12
RegWriteD
MemToRegD
MemWriteD
ALUControlD
ALUSrcD
TypeRISBUJ
BranchD
RegWriteE
MemToRegE
MemWriteE
ALUControlE
ALUSrcE
BranchE
RegWriteM
MemToRegM
MemWriteM
BranchM
PCSrcM
RegWriteW
MemTo RegW
PC´
PC
A RD
Instruction Memory
InstrD
19:15
24:20
WE3
A1 RD1
A2 RD2
A3 Reg.
WD3 File
11:7
31:25, 24:20
11:7
Imm decode
SignImmD
WriteRegD 4:0
SrcAE
SrcBE
ALU
Zero
WriteDataE
ALUOutM
WriteDataM
WE
A RD
Data Memory
WD
ReadDataW
ALUOutW
WriteRegE 4:0
WriteRegM 4:0
WriteRegW 4:0
SignImmE
PCBranch
4
+
PCPlus4F
PCD
PCE
ResultW

Cause of the Data Hazards



Register File – access from two pipeline stages (Decode, WriteBack) – actual write occurs at the first half of the clock cycle, the read in the second half ⇒ there is no hazard for sub **s0** input operand

- RAW (Read After Write) hazard – and (or) requires **s0** in 3 (4)
- How can such hazard be prevented without pipeline throughput degradation?

QtRVSim – Resolve Data Hazard by Stall

or x6, x20, **x8**	and x5, **x8**, x9	add **x8**, x18, x19	addi x21, x21, 1365	nop
or t1, s4, **s0**	and t1, **s0**, s1	add **s0**, s2, s3	addi s5, s5, 1365	nop

Cycles: 23
Stalls: 11
Hazard Unit: NORMAL

QtRVSim https://github.com/cvut/qtrvsim
https://gitlab.fel.cvut.cz/b35apo/stud-support/-/blob/master/seminaries/qtrvsim/hazards-from-lecture/alu-hazards.S

QtRVSim – Resolve Data Hazard by Stall

or x6, x20, **x8**	and x5, **x8**, x9	nop	nop	add **x8**, x18, x19
or t1, s0, **s0**	and t0, **s0**, s1	stall	stall	add **s0**, s2, s3

Cycles: 25

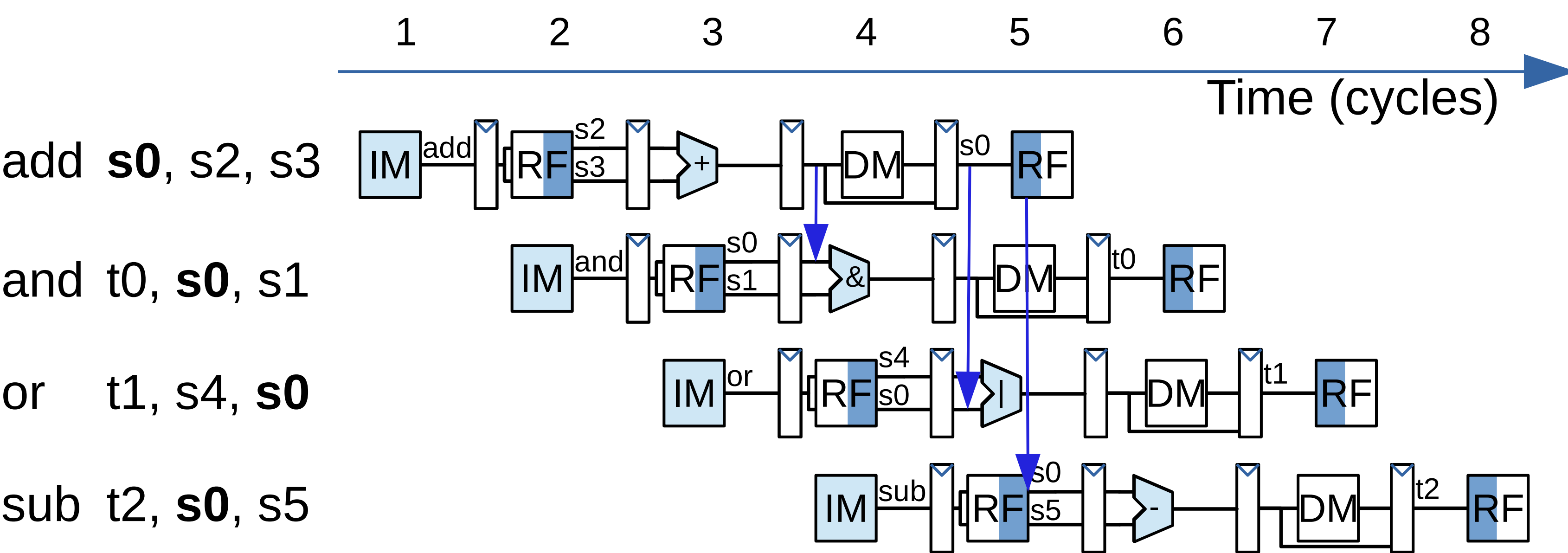
Stalls: 12

Hazard Unit: STALL

QtRVSim https://github.com/cvut/qtrvsim

https://gitlab.fel.cvut.cz/b35apo/stud-support/-/blob/master/seminaries/qtrvsim/hazards-from-lecture/alu-hazards.S

Forwarding to Avoid Data Hazards



- If a result is available (computed) before subsequent instruction(s) requires the value then data hazard can be avoided by forwarding
- Hazard case is indicated when some of source registers in EX stage is the same as destination register in stage MEM or WB
- The register numbers are fed to the Hazard Unit
- The RegWrite signal from MEM and WB stage has to be monitored as well to check that register number on WriteReg lines takes effect – lw / sw

Data Hazards Solved by Forwarding

Control unit
Op
Funct
6:0
31:25, 14:12
RegWriteD
MemToRegD
MemWriteD
ALUControlD
ALUSrcD
TypeRISBUJ
BranchD
RegWriteE
MemToRegE
MemWriteE
ALUControlE
ALUSrcE
BranchE
RegWriteM
MemToRegM
MemWriteM
BranchM
PCSrcM
Zero
RegWriteW
MemTo RegW
0 1
PC´
PC
A RD
Instruction Memory
InstrD
19:15
24:20
WE3
A1 RD1
A2 RD2
A3 Reg.
WD3 File
00 01 10
00 01 10
0 1
SrcAE
SrcBE
ALU
WriteDataE
WriteDataM
ALUOutM
WE
A RD
Data Memory
WD
ReadDataW
ALUOutW
1 0
11:7
WriteRegD 4:0
WriteRegE 4:0
WriteRegM 4:0
WriteRegW 4:0
31:25, 24:20
11:7
SignImmD
Imm decode
SignImmE
4
+
PCPlus4F
19:15
24:20
Rs1D
Rs2D
Rs1E
Rs2E
+
PCBranch
PCD
PCE
ResultW
Forward AE
Forward BE
RegWriteM
RegWrite W
Hazard unit

QtRVSim – Resolve Data Hazard by Forwarding

or x6, x20, **x8**	and x5, **x8**, x9	add **x8**, x18, x19	addi x21, x21, 1365	lui x21, 0x55555
or t1, s4, **s0**	and t1, **s0**, s1	add **s0**, s2, s3	addi s5, s5, 1365	lui s5, 0x55555

QtRVSim https://github.com/cvut/qtrvsim

https://gitlab.fel.cvut.cz/b35apo/stud-support/-/blob/master/seminaries/qtrvsim/hazards-from-lecture/alu-hazards.S

QtRVSim – Resolve Data Hazard by Forwarding

sub x7, **x8**, x21	or x6, x20, **x8**	and x5, **x8**, x9	add **x8**, x18, x19	addi x21, x21, 136
sub t2, **s0**, s5	or t1, s4, **s0**	and t0, **s0**, xs1	add **s0**, s2, s3	addi s5, s5, 1365

QtRVSim https://github.com/cvut/qtrvsim
https://gitlab.fel.cvut.cz/b35apo/stud-support/-/blob/master/seminaries/qtrvsim/hazards-from-lecture/alu-hazards.S

QtRVSim – Resolve Data Hazard by Forwarding

unknown	sub x7, **x8**, x21	or x6, x20, **x8**	and x5, **x8**, x9	add **x8**, x18, x19
unknown	sub t2, **s0**, s5	or t1, s4, **s0**	and t0, **s0**, s1	add **s0**, s2, s3

QtRVSim https://github.com/cvut/qtrvsim
https://gitlab.fel.cvut.cz/b35apo/stud-support/-/blob/master/seminaries/qtrvsim/hazards-from-lecture/alu-hazards.S

QtRVSim – Simple I/O Peripherals

base of SPILED port region

SPILED_REG_BASE 0xffffc100

RGB LED 1 color components – 8 bits each

SPILED_REG_LED_RGB1 0xffffc110

SPILED_REG_LED_RGB1_o 0x0010

RGB LED 2 color components – 8 bits each

SPILED_REG_LED_RGB2 0xffffc114

SPILED_REG_LED_RGB2_o 0x0014

Three 8 bit knob values

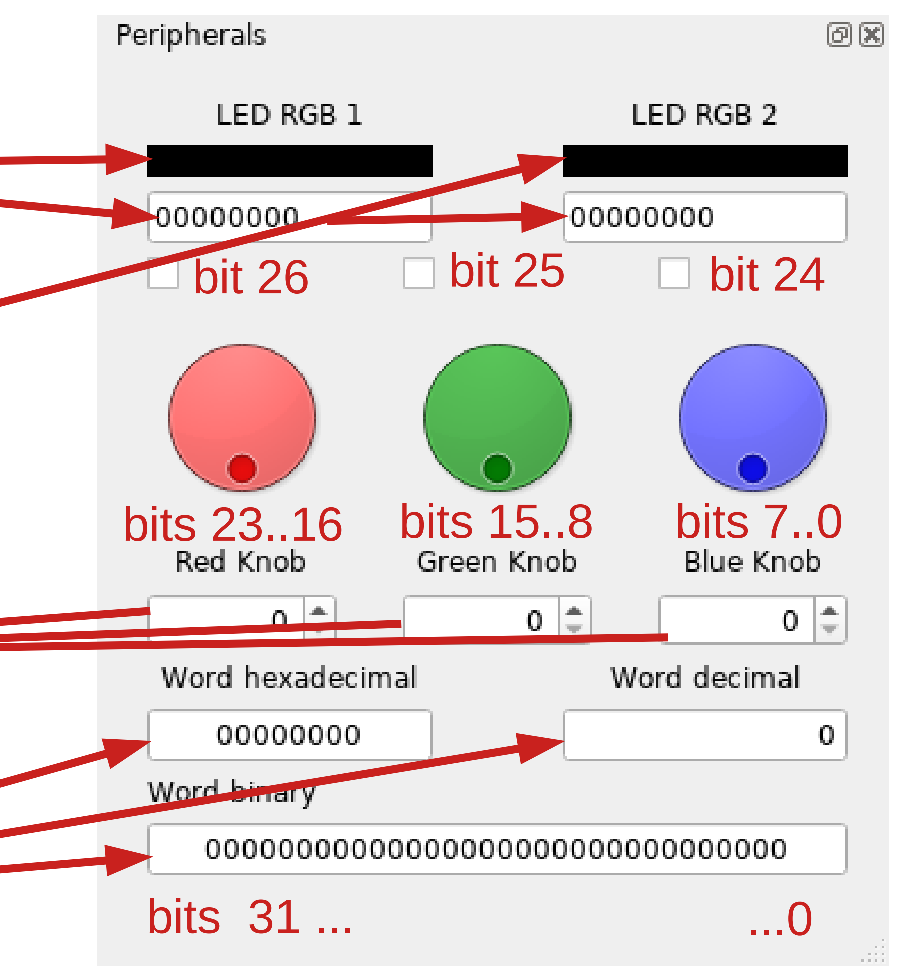
SPILED_REG_KNOBS_8BIT 0xffffc124

SPILED_REG_KNOBS_8BIT_o 0x0024

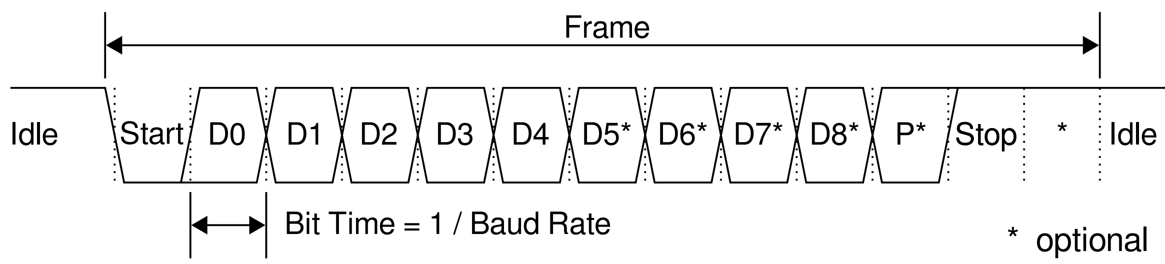
line of 32 LEDs for binary value representation

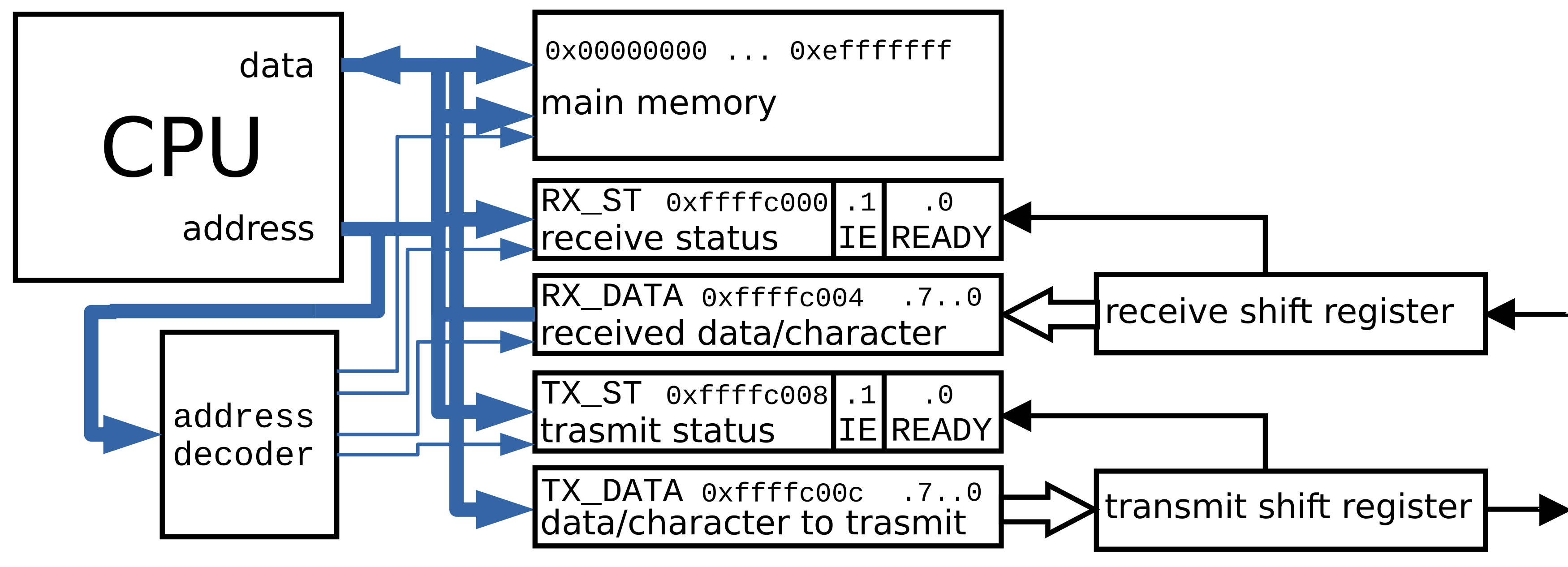
SPILED_REG_LED_LINE 0xffffc104

SPILED_REG_LED_LINE_o 0x0004



Serial Port – UART





QtRVSim Serial Port Addresses

SERIAL_PORT_BASE 0xffffc000
base address of QtRVSim serial port, mirrors on 0xffff0000 for MARS and QtSpim

SERP_RX_ST_REG	**0xffffc000**	Receiver status register
SERP_RX_ST_REG_o	0x0000	Offset of RX_ST_REG
SERP_RX_ST_REG_READY_m	0x1	Data byte is ready to be read
SERP_RX_ST_REG_IE_m	0x2	Enable Rx ready interrupt
SERP_RX_DATA_REG	**0xffffc004**	Received data byte in 8 LSB bits
SERP_RX_DATA_REG_o	0x0004	Offset of RX_DATA_REG
SERP_TX_ST_REG	**0xffffc008**	Transmitter status register
SERP_TX_ST_REG_o	0x0008	Offset of TX_ST_REG
SERP_TX_ST_REG_READY_m	0x1	Transmitter can accept next byte
SERP_TX_ST_REG_IE_m	0x2	Enable Tx ready interrupt
SERP_TX_DATA_REG	**0xffffc00c**	Write word to send 8 LSB bits to terminal
SERP_TX_DATA_REG_o	0x000c	Offset of TX_DATA_REG

QtRVSim Peripherals Documentation

- All peripherals supported by QtRVSim are described at QtRVSim project page (README.md file)
  https://github.com/cvut/qtrvsim/#peripherals
- The simple 16-bit per pixel (RGB565) framebuffer
  The size corresponds to MZ_APO 480 x 320 pixel display
  components
  - bits 11 .. 15 red
  - bits 5 .. 10 green
  - bits 0 .. 4 blue

  Frame buffer starts at address
  - **LCD_FB_START** 0xffe00000
  - **LCD_FB_END** 0xffe4afff

Example of Code and Cache Optimization Task

```
// apo-sort.S file template, rename and implement the algorithm
// Test algorithm in qtrvsim_gui program
// Select the CPU core configuration with delay-slot
// This setups requires (for simplicity) one NOP instruction after
// each branch and jump instruction (more during lecture about pipelining)
// The code will be compiled and tested by external riscv64-unknown-elf-gcc
// compiler by teachers, you can try make in addition, but testing
// by internal assembler should be enough

// copy directory with the project to your repository to
// the directory work/apo-sort
// critical is location of the file work/apo-sort/apo-sort.S
// and cache parameters work/apo-sort/d-cache.par
// which is checked by the scripts
```

Example of Code and Cache Optimization Task

```
// The file d-cache.par specifies D cache parameters in the form
//   <policy>,<#sets>,<#words in block>,<#ways>,<write method>
// The example is
//   lru,1,1,1,wb
// The cache size is limited to 16 words maximum.

// Directives to make interesting windows visible
#pragma qtrvsim show registers
#pragma qtrvsim show memory

.option norelax

.globl    array_size
.globl    array_start
```

Example of Code and Cache Optimization Task

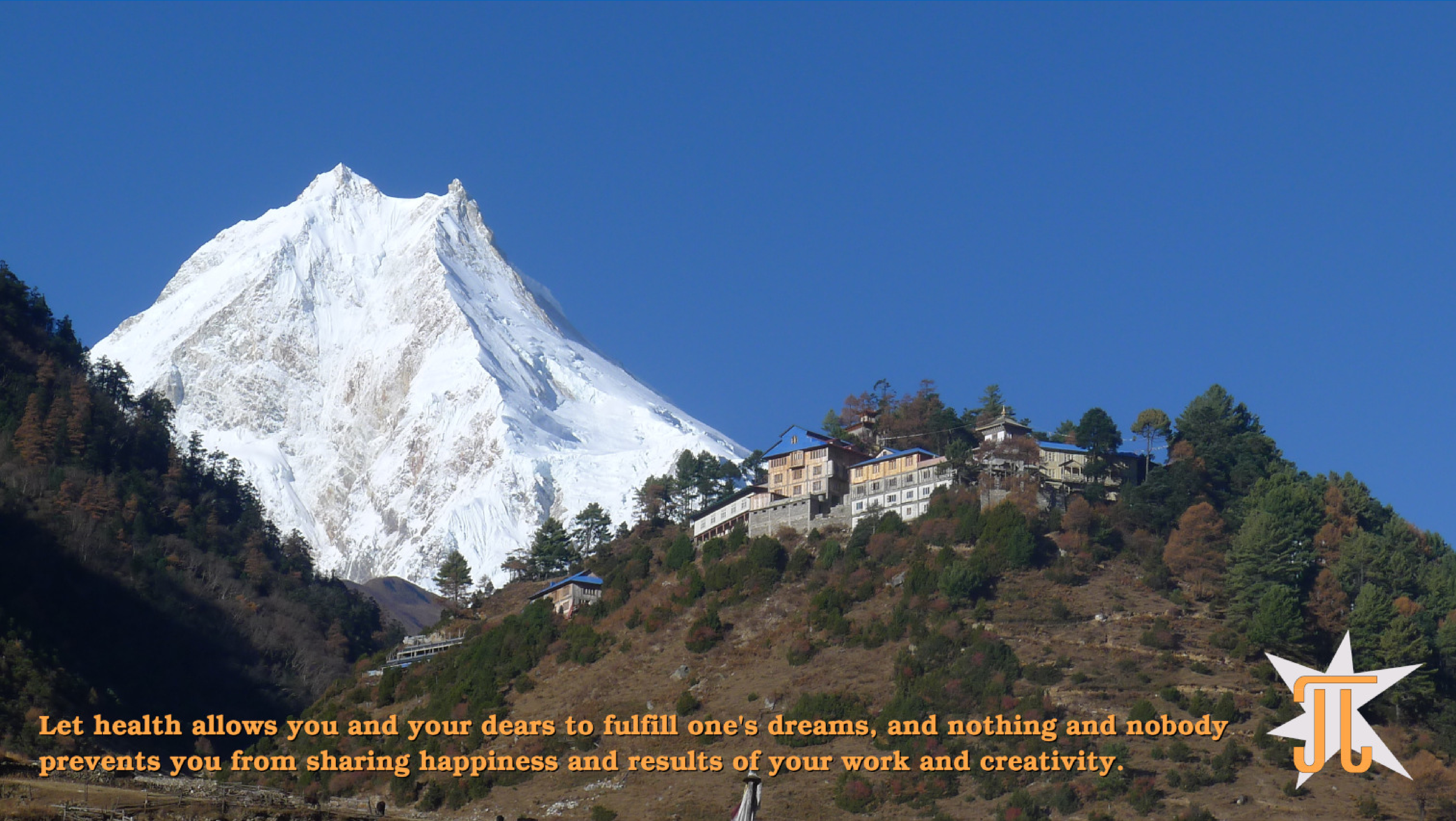
```
.text
.globl _start
_start:
        la   a0, array_start
        la   a1, array_size
        lw   a1, 0(a1) // number of elements in the array

//Insert your code there

//Final infinite loop
end_loop:
        fence           // flush cache memory
        ebreak          // stop the simulator
        j end_loop
.data
// .align    2 // not supported by qtrvsim yet

array_size:
.word   15
array_start:
.word   5, 3, 4, 1, 15, 8, 9, 2, 10, 6, 11, 1, 6, 9, 12

// Specify location to show in memory window
#pragma qtrvsim focus memory array_size
```

- Can be tuned in QtRVSim GUI
- Makefile provided for automatic compile and run in qtrvsim_cli
- Options
  - dump-cache-stats
  - load-range
  - dump-range array
- Data can be injected into compiled student program and memory with results dumped for check
- Complete example
  https://gitlab.fel.cvut.cz/b35apo/stud-support/-/tree/master/seminaries/qtrvsim/apo-sort

Links and Resources

- CTU Computer Architectures resources https://comparch.edu.cvut.cz/
- QtRVSim https://github.com/cvut/qtrvsim
- Peripheries match MZ_APO (Zynq ARM) documentation at B35APO pages
- LinuxDays 2019 – QtMips (https://www.youtube.com/watch?v=fhcdYtpFsyw)
- Student work: CAN/TWAI driver for NuttX (in mainline) RISC-V ESP32C3
- You can meet us at the Embedded World Conference 2022.

  Talk QtRvSim – RISC-V Simulator for Computer Architectures Classes is scheduled June 21, 2022 Session 10.3 – System-on-Chip (SoC) Design RISC-V Development (16:00 – 16:30)
- Open Technologies Research Education and Exchange Services – Wiki https://gitlab.fel.cvut.cz/otrees/org/-/wikis/home


Let health allows you and your dears to fulfill one's dreams, and nothing and nobody
prevents you from sharing happiness and results of your work and creativity.