Fiscal Year 2024

Ver. 2024-12-12a

Course number: CSC.T433
School of Computing,
Graduate major in Computer Science

Advanced Computer Architecture

2. Instruction Set Architecture and single-cycle processor

www.arch.cs.titech.ac.jp/lecture/ACA/
Room No. W8E-308, Lecture (Face-to-face)
Mon 13:30-15:10, Thr 13:30-15:10

Kenji Kise, Department of Computer Science
kise _at_ c.titech.ac.jp

Two major ISA types: RISC vs CISC

An Instruction Set Architecture (ISA) is part of the abstract model of a computer that defines how the processor is controlled by the software. The ISA acts as an interface between the hardware and the software.

- RISC (**Reduced** Instruction Set Computer) philosophy
  - fixed instruction lengths
  - load-store instruction sets
  - limited addressing modes
  - limited operations
  - RISC: MIPS, Alpha, ARM, RISC-V, …
- CISC (**Complex** Instruction Set Computer) philosophy
  - ! fixed instruction lengths
  - ! load-store instruction sets
  - ! limited addressing modes
  - ! limited operations
  - CISC : DEC VAX11, Intel 80x86, …

MIPS, ARM, and RISC-V

https://en.wikipedia.org/wiki/MIPS_architecture

Wikipedia
The Free Encyclopedia

Main page
Contents
Current events
Random article
About Wikipedia
Contact us
Donate

Contribute
Help
Community portal
Recent changes
Upload file

Tools
What links here

Article Talk Read Edit View history

MIPS architecture

From Wikipedia, the free encyclopedia

MIPS (**Microprocessor without Interlocked Pipelined Stages**)[1] is a reduced instruction set computer (RISC) instruction set architecture (ISA)[2]:A-1[3]:19 developed by MIPS Computer Systems, now MIPS Technologies, based in the United States.

There are multiple versions of MIPS: including MIPS I, II, III, IV, and V; as well as five releases of MIPS32/64 (for 32- and 64-bit implementations, respectively). The early MIPS architectures were 32-bit only; 64-bit versions were developed later. As of April 2017, the current version of MIPS is MIPS32/64 Release 6.[4][5] MIPS32/64 primarily differs from MIPS I–V by defining the privileged kernel mode System Control Coprocessor in addition to the user mode architecture.

The MIPS architecture has several optional extensions. MIPS-3D which is a simple set of floating-point SIMD instructions dedicated to common 3D tasks,[6] MDMX (MaDMaX) which is a more extensive integer SIMD instruction set using the 64-bit floating-point registers, MIPS16e which adds compression to the instruction stream to make programs take up less room,[7] and MIPS MT, which adds multithreading capability.[8]

Computer architecture courses in universities and technical schools often study the MIPS architecture.[9] The architecture greatly influenced later RISC architectures such as Alpha.

arm

Arm "ABCD" building in Cherry Hinton, Cambridge, UK

ARM (Advanced RISC Machine)

RISC-V®

About RISC-V | Membership | RISC-V Exchange | Technical | News & Events | Community

RISC-V is an open standard Instruction Set (ISA) enabling a new era of processor in through open collaboration

RISC-V enables the community to share technical contribute to the strategic future, create more ra unprecedented design freedom, and substantially red innovation

About RISC-V
History of RISC-V
Board of Directors
Technical Steering Committee
RISC-V Staff
Guidelines
Frequently Asked Questions (FAQ)
Contact Us

ISC-V Are you ready to break free?

RISC-V International is the global non-profit home of the open standard RISC-V Instruction Set Architecture (ISA), related specifications, and stakeholder community

3,950 RISC-V members across 70 countries contribute and collaborate to define RISC-V open specifications as well as convene and govern related technical, industry, domain, and special interest groups.

Understanding the RISC-V ISA Open Standard

At the base level, the RISC-V ISA and extensions ratified by RISC-V International are royalty free and open base building blocks for anyone to build their own solutions and services on. The RISC-V ISA and ratified extensions are provided under globally accepted open licenses that are permanently open and remain available for all.

Beyond RISC-V International, the community has opportunity to provide their own free or

https://riscv.org/

RISC-V base and extensions

Chapter 1

FE310-G002 Description

1.1 Features

- SiFive E31 Core Complex up to 320MHz.
- Flexible clocking options including internal PLL, free-running ring oscillator and external 16MHz crystal.
- 1.61 DMIPs/MHz, 2.73 Coremark/MHz
- RV32IMAC
- 8kB OTP Program Memory
- 8kB Mask ROM
- 16kB Instruction Cache
- 16kB Data SRAM
- 3 Independent PWM Controllers
- External RESET pin
- JTAG, SPI I2C, and UART interfaces.
- QSPI Flash interface.
- Requires 1.8V and 3.3V supplies.
- Hardware Multiply and Divide

1.2 Description

The FE310-G002 is the second Freedom E300 SoC. The FE310-G002 is built around the E31 Core Complex instantiated in the Freedom E300 platform.

The *FE310-G002 Manual* should be read together with this datasheet. This datasheet provides electrical specifications and an overview of the FE310-G002.

The FE310-G002 comes in a convenient, industry standard 6x6mm 48-lead QFN package (0.4mm pad pitch).

ISA base and extensions (20191213)

Name	Description	Version	Status[a]
Base			
RVWMO	Weak Memory Ordering	2.0	Ratified
RV32I	Base Integer Instruction Set, 32-bit	2.1	Ratified
RV32E	Base Integer Instruction Set (embedded), 32-bit, 16 registers	1.9	Open
RV64I	Base Integer Instruction Set, 64-bit	2.1	Ratified
RV128I	Base Integer Instruction Set, 128-bit	1.7	Open
Extension			
M	Standard Extension for Integer Multiplication and Division	2.0	Ratified
A	Standard Extension for Atomic Instructions	2.1	Ratified
F	Standard Extension for Single-Precision Floating-Point	2.2	Ratified
D	Standard Extension for Double-Precision Floating-Point	2.2	Ratified
G	Shorthand for the base integer set (I) and above extensions (MAFD)	N/A	N/A
Q	Standard Extension for Quad-Precision Floating-Point	2.2	Ratified
L	Standard Extension for Decimal Floating-Point	0.0	Open
C	Standard Extension for Compressed Instructions	2.0	Ratified
B	Standard Extension for Bit Manipulation	0.92	Open
J	Standard Extension for Dynamically Translated Languages	0.0	Open
T	Standard Extension for Transactional Memory	0.0	Open
P	Standard Extension for Packed-SIMD Instructions	0.2	Open
V	Standard Extension for Vector Operations	0.9	Open
N	Standard Extension for User-Level Interrupts	1.1	Open
H	Standard Extension for Hypervisor	0.4	Open
ZiCSR	Control and Status Register (CSR)	2.0	Ratified
Zifencei	Instruction-Fetch Fence	2.0	Ratified
Zam	Misaligned Atomics	0.1	Open
Ztso	Total Store Ordering	0.1	Frozen

RISC-V RV32I base and our target instructions

We do not support some system instructions (FENCE, ECALL, EBREAK) and 8-bit or 16-bit loads (LB, LH, LBU, LHU) and stores (SB, SH) of RV32I.

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
~~imm[11:0]~~		~~rs1~~	~~000~~	~~rd~~	~~0000011~~	~~LB~~
~~imm[11:0]~~		~~rs1~~	~~001~~	~~rd~~	~~0000011~~	~~LH~~
imm[11:0]		rs1	010	rd	0000011	LW
~~imm[11:0]~~		~~rs1~~	~~100~~	~~rd~~	~~0000011~~	~~LBU~~
~~imm[11:0]~~		~~rs1~~	~~101~~	~~rd~~	~~0000011~~	~~LHU~~
~~imm[11:5]~~	~~rs2~~	~~rs1~~	~~000~~	~~imm[4:0]~~	~~0100011~~	~~SB~~
~~imm[11:5]~~	~~rs2~~	~~rs1~~	~~001~~	~~imm[4:0]~~	~~0100011~~	~~SH~~
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
~~fm~~ ~~pred~~	~~succ~~	~~rs1~~	~~000~~	~~rd~~	~~0001111~~	~~FENCE~~
~~000000000000~~		~~00000~~	~~000~~	~~00000~~	~~1110011~~	~~ECALL~~
~~000000000001~~		~~00000~~	~~000~~	~~00000~~	~~1110011~~	~~EBREAK~~

RISC-V general-purpose registers

XLEN = 32
for 32bit ISA

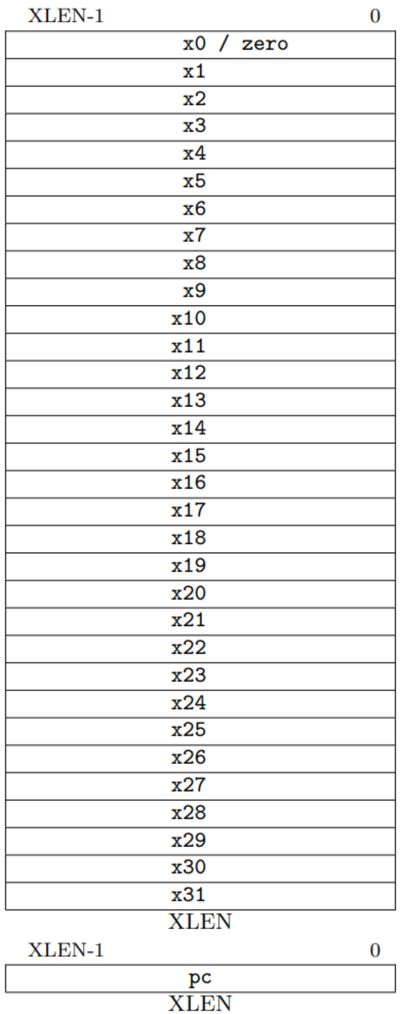


Figure 2.1: RISC-V base unprivileged integer register state.

ABI(Application Binary Interface) name

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	—
x5–7	t0–2	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

Table 18.2: RISC-V calling convention register usage.

RV32I does not have floating point regesters of f0 - f31.

RISC-V instruction length encoding

The RISC-V Instruction Set Manual
Volume I: Unprivileged ISA
Document Version 20191214-*draft*

Editors: Andrew Waterman[1], Krste Asanović[1,2]
[1]SiFive Inc.,
[2]CS Division, EECS Department, University of California, Berkeley
andrew@sifive.com, krste@berkeley.edu
November 12, 2021

We support 32-bit length instructions.
16-bit length instructions called compressed instructions are used in some embedded systems.

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

Figure 1.1: RISC-V instruction length encoding. Only the 16-bit and 32-bit encodings are considered frozen at this time.

RISC-V base instruction format

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

Figure 2.3: RISC-V base instruction formats showing immediate variants.

RISC-V Arithmetic Instructions

- RISC-V assembly language arithmetic statement

```
add  x7, x8, x9
sub  x7, x8, x9
```

- Each arithmetic instruction performs only one operation
- Each arithmetic instruction fits in 32 bits and specifies exactly three operands

```
destination <- source1 op source2
```

- Operand order is fixed (destination first)
- Those operands are all contained in the datapath's register file (x0, ..., x31)

Exercise 1

- Compiling a C assignment using registers

```
f = ( g + h ) - ( i + j );
```

- The variables f, g, h, i, and j are assigned to the registers s0, s1, s2, s3, and s4, respectively. What is the compiled RISC-V code?

```
s0 = ( s1 + s2 ) - ( s3 + s4 );
```

```
t0 = s1 + s2;
t1 = s3 + s4;
s0 = t0 - t1;
```

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	—
x5–7	t0–2	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

Table 18.2: RISC-V calling convention register usage.

(1) Machine Language - Add instruction (add)

- Instructions are 32 bits long
- Arithmetic Instruction Format (R-type):

add x7, x8, x9

funct7	rs2	rs1	funct3	rd	opcode

R-type

opcode 7-bits *opcode* that specifies the operation

rs1 5-bits register file address of the first source operand

rs2 5-bits register file address of the second source operand

rd 5-bits register file address of the result's destination

funct3 and **funct7** 10-bits select the type of operation (function)

(2) RISC-V Add immediate instruction (addi)

- Small constants are used often in typical code
- Possible approaches?
  - put "typical constants" in memory and load them
  - create hard-wired registers (like x0) for constants like 1
  - have special instructions that contain constants !

```
addi  x7, x8, -2      # x7 = x8 + (-2)
```

- Machine format (I format):

imm[11:0]	rs1	funct3	rd	opcode

I-type

- The constant is kept inside the instruction itself
  - Immediate format limits values to the range $+2^{11}-1$ to -2^{11}

RISC-V Memory Access Instructions

- RISC-V has two basic data transfer instructions for accessing memory
- `lw   x5, 24(x7)     # load word from memory`
- `sw   x3, 28(x9)     # store word to memory`

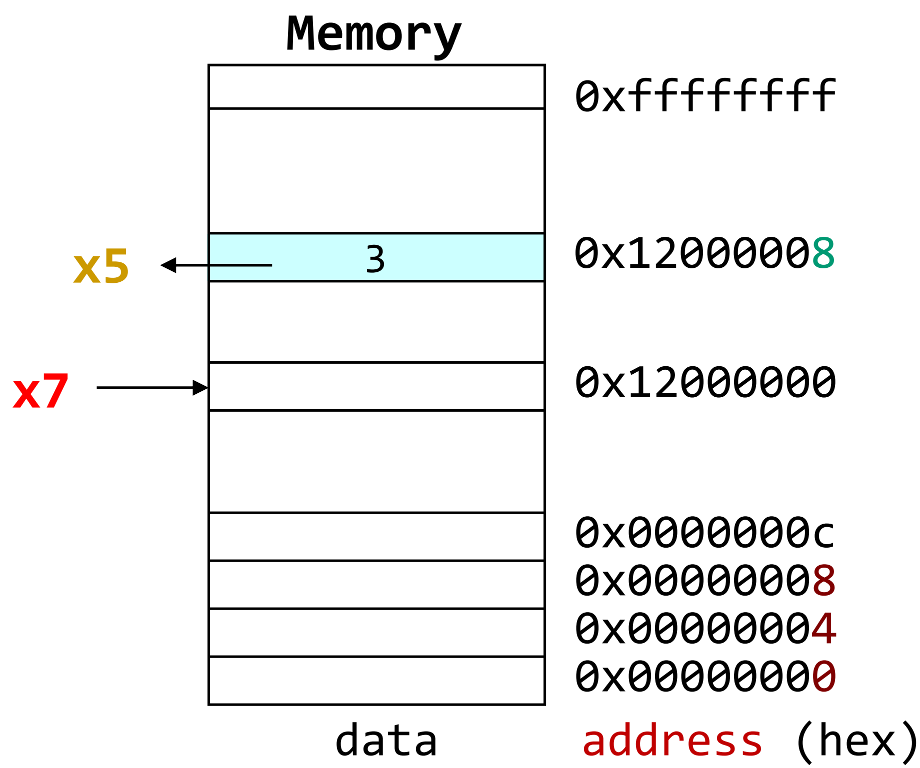
- The data is loaded into (lw) or stored from (sw) a register in the register file
- The memory address – a 32 bit address – is formed by adding the contents of the base address register to the offset value

(3) Machine Language - Load word instruction (lw)

- Load Instruction Format (I-type):

`lw x5, 8(x7)`

imm[11:0]	rs1	funct3	rd	opcode	I-type



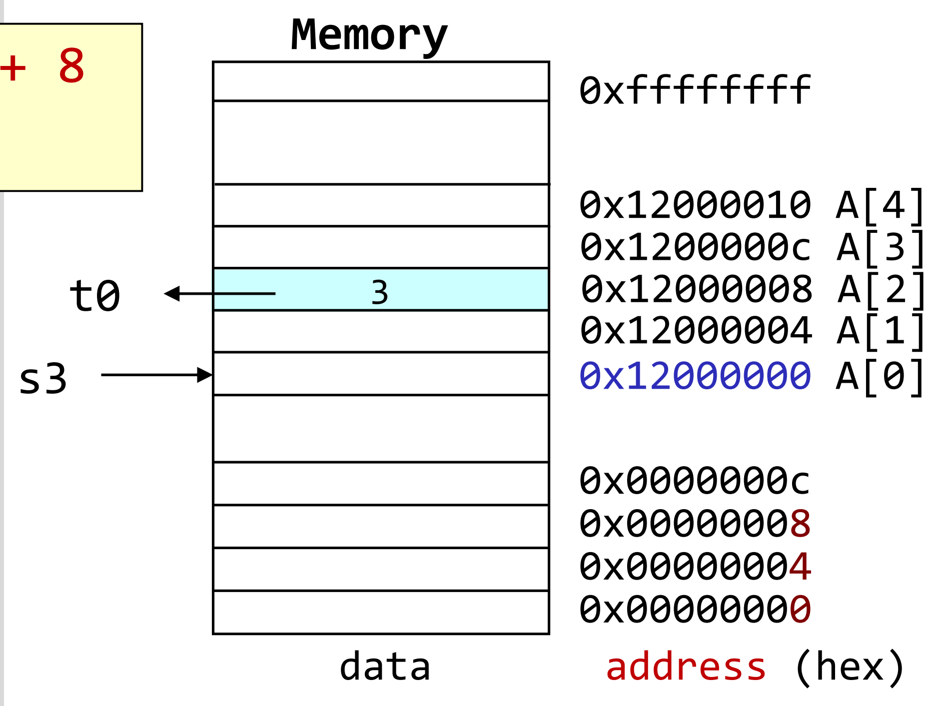
Exercise 2

- Compiling an assignment when an operand is in memory

```
g = h + A[2];
```

- Let's assume that A is an array of 100 words and the compiler has associated the variable g and h with the registers s1 and s2 as before. Let's also assume that the starting address, or base address, of the array is in s3. Compile this C code.

```
t0 = A[2];  # address is s3 + 8
s1 = s2 + t0;
```

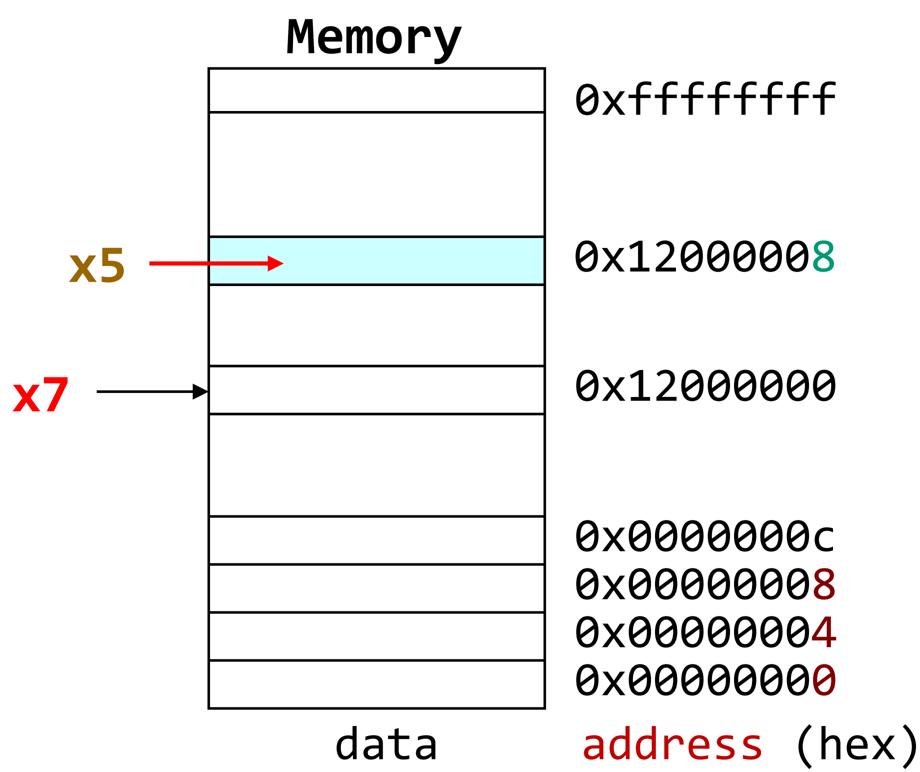


(4) Machine Language - Store word instruction (sw)

- Load Instruction Format (S-type):

`sw x5, 8(x7)`

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type



Exercise 3

- Compiling using load and store

```
A[1] = h + A[2];
```

- Assume variable h is associated with register s2 and base address of the array A is in s3. What is the RISC-V assembly code for the C program?

```
t0 = A[2];  # address is s3 + 8
t1 = s2 + t0;
A[1] = t1;  # address is s3 + 4
```

(5) RISC-V branch if not equal instructions (bne)

- RISC-V conditional branch instructions (bne, branch if not equal) :

```
 bne x4, x5, Lbl  # go to Lbl if x4!=x5
```

```
Ex:    if (i==j) h = i + j;

       bne x4, x5, Lbl1   # if (i!=j) goto Lbl1
       add x6, x4, x5     # h = i + j;
Lbl1:  ...
```

- Instruction Format (B-type):

imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode	B-type

- How is the branch destination address specified?

Exercise 4

- Compiling using add, addi, and bne

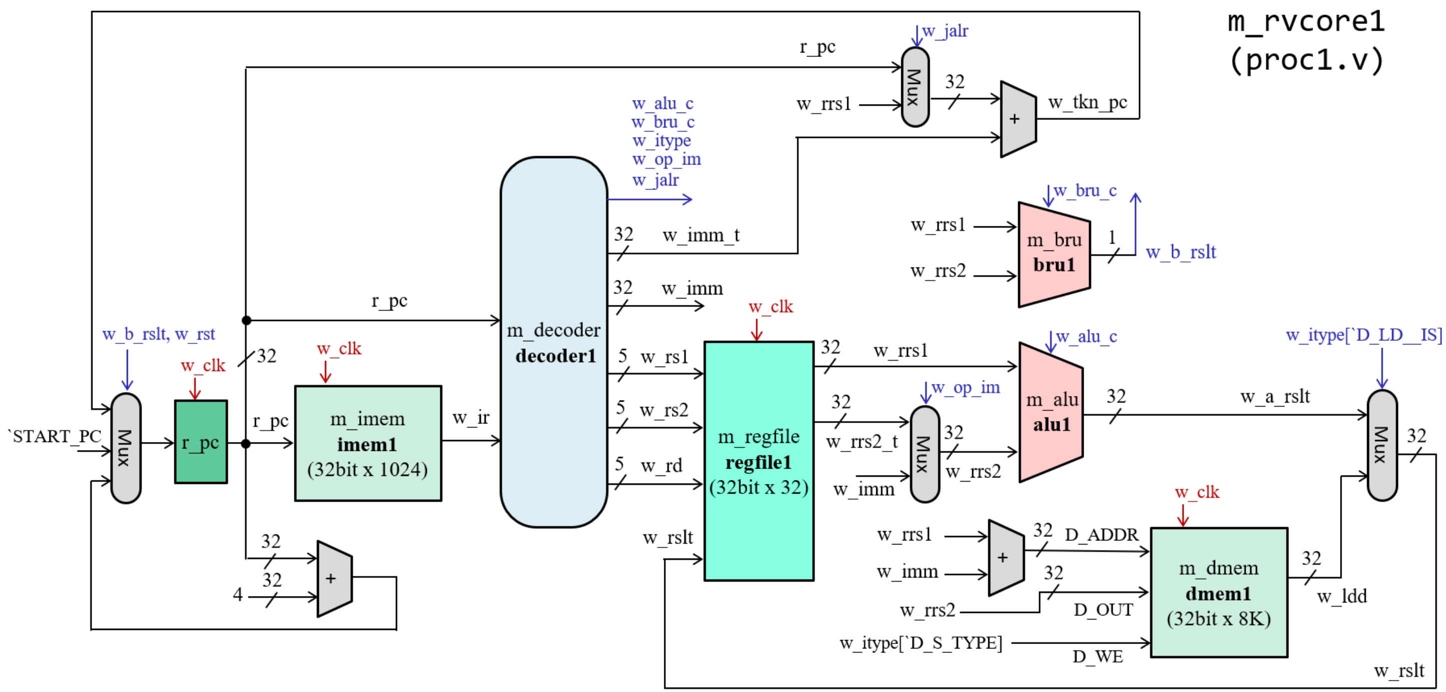
```
void main(){
    int i, sum=0;
    for(i=1; i<11; i++) sum = sum + i;
}
```

- What is the RISC-V assembly code for the C program?

```
void main(){
    int s2, s3=11, s4=0;
    for(s2=1; s2<s3; s2++) s4 = s4 + s2;
}
```

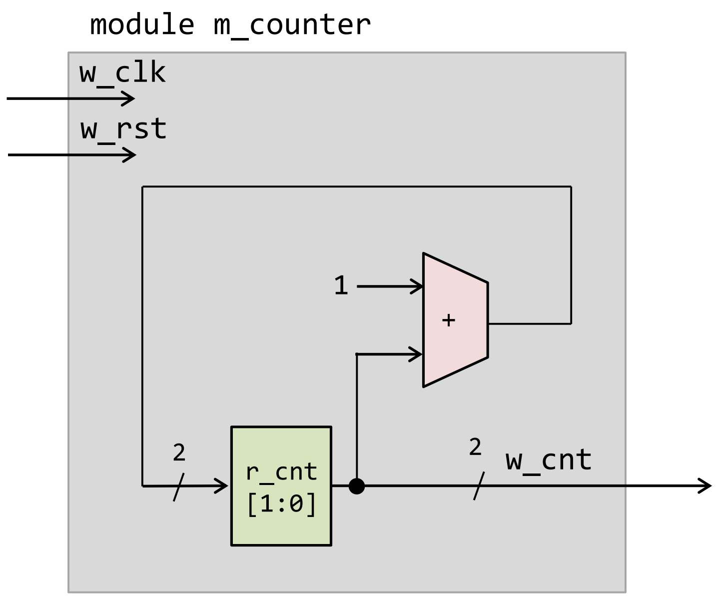
Single-cycle implementation of processors

- Single-cycle implementation also called single clock cycle implementation is the implementation in which an instruction is executed in one clock cycle.
  While easy to understand, it is too slow to be practical.
  It is useful as a baseline for lectures.



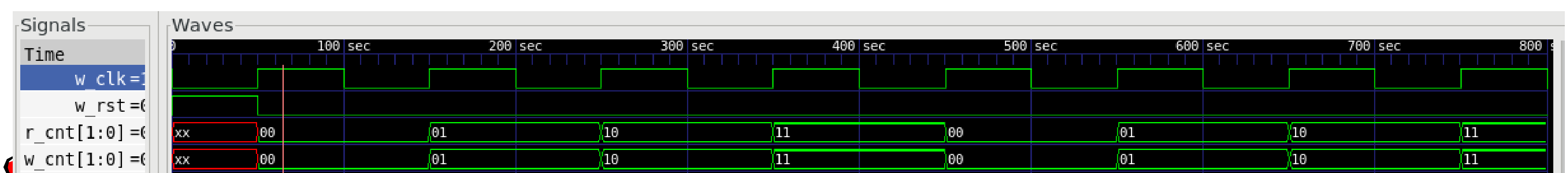
Sample circuit 2

- 2-bit counter as a simple sequential circuit



circuit2.v

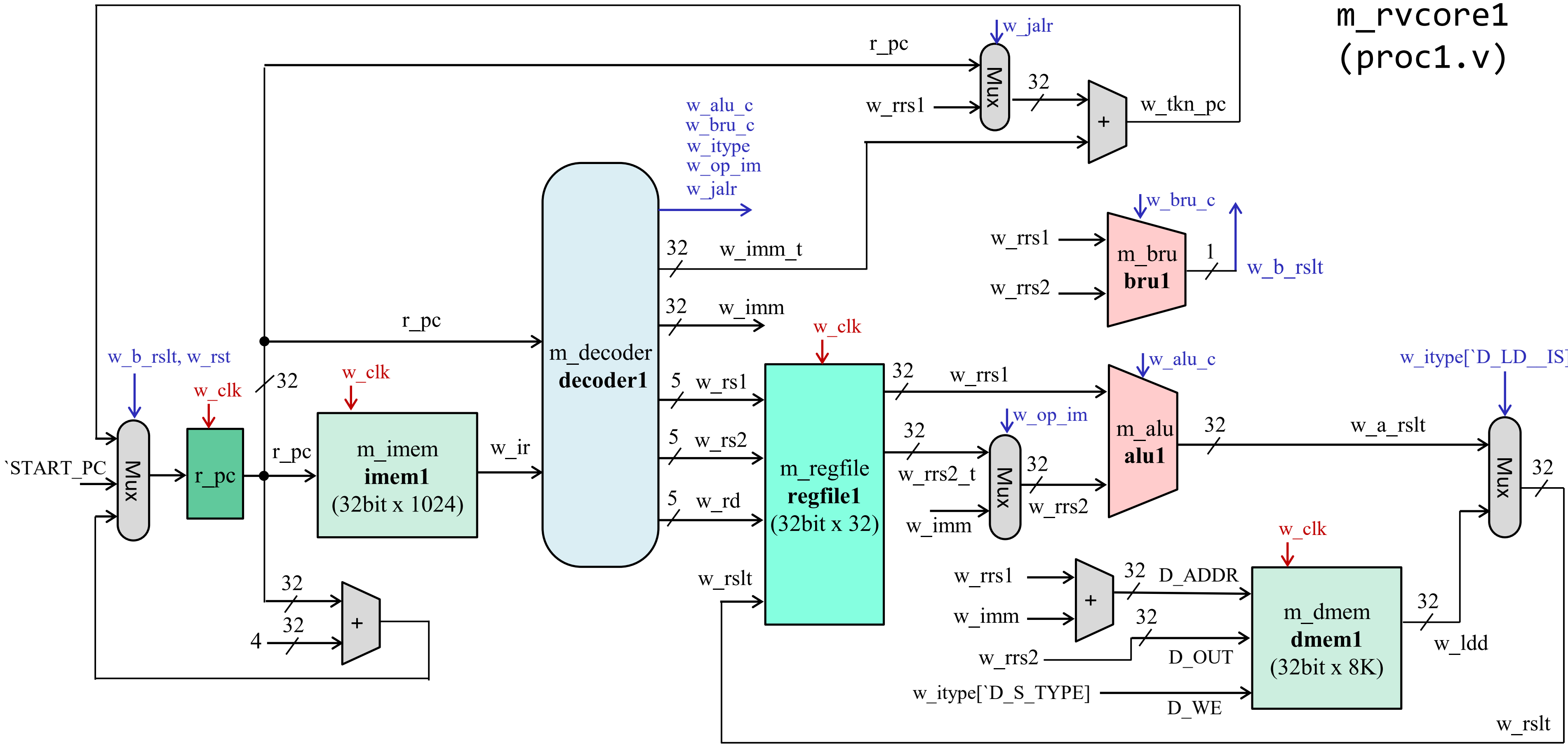
```
module top();
  reg r_clk = 0;
  always #50 r_clk = ~r_clk;
  reg r_rst = 1;
  always @(posedge r_clk) r_rst <= 0;
  wire [1:0] w_cnt;
  m_counter m1 (r_clk, r_rst, w_cnt);
  initial begin $dumpfile("dump.vcd"); $dumpvars(0); end
  initial #800 $finish;
endmodule

module m_counter (
  input wire w_clk,
  input wire w_rst,
  output wire [1:0] w_cnt
);
  reg [1:0] r_cnt;
  always@(posedge w_clk) r_cnt <= (w_rst) ? 0 : r_cnt + 1;
  assign w_cnt = r_cnt;
endmodule
```



m_rvcore (RV32I, single-cycle processor)

- around 40MHz operating frequency for Arty A7 FPGA board
- lb, lbu, lh, lhu, sb, sh are not supported

m_rvcore1
(proc1.v)



m_rvcore (RV32I, single-cycle processor)

```
/***** subset of RV32I where LB, LH, LBU, LHU, SB, SH are not supported          *****/
/**************************************************************************************/
module m_rvcore ( ///// RVCore Simple Version
     input  wire        w_clk,  // clock signal
     input  wire        w_rst,  // reset signal
     output wire [31:0] D_ADDR, // data memory, address
     output wire [31:0] D_OUT,  // data memory, output data
     output wire        D_WE    // data memory, write enable
);
     reg  [31:0] r_pc;
     wire        w_jalr, w_op_im, w_b_rslt;
     wire [9:0]  w_itype;
     wire [10:0] w_alu_c;
     wire [6:0]  w_bru_c;
     wire [4:0]  w_rs1, w_rs2, w_rd;
     wire [31:0] w_ir, w_rrs1, w_rrs2_t, w_rrs2, w_imm_t, w_imm;
     wire [31:0] w_a_rslt, w_rslt, w_tkn_pc, w_ldd;

     m_imem imem1 (w_clk, r_pc, w_ir);

     m_decoder decoder1 (r_pc, w_ir, w_rd, w_rs1, w_rs2,
                         w_op_im, w_itype, w_jalr, w_alu_c, w_bru_c, w_imm_t, w_imm);

     m_regfile regfile1 (w_clk, w_rs1, w_rs2, w_rrs1, w_rrs2_t, w_rd, w_rslt);
     assign w_rrs2 = (w_op_im) ? w_imm : w_rrs2_t;

     m_alu alu1 (w_rrs1, w_rrs2, w_alu_c, w_a_rslt);
     m_bru bru0 (w_rrs1, w_rrs2, w_bru_c, w_b_rslt);

     assign D_ADDR = w_rrs1 + w_imm;
     assign D_OUT  = w_rrs2;
     assign D_WE   = w_itype[`D_S_TYPE];
     m_dmem dmem1 (w_clk, D_WE, D_ADDR, D_OUT, w_ldd);

     assign w_rslt = (w_itype[`D_LD__IS]) ? w_ldd : w_a_rslt;

     assign w_tkn_pc = ((w_jalr) ? w_rrs1 : r_pc) + w_imm_t;
     always @(posedge w_clk) r_pc <= (w_rst) ? `START_PC : (w_b_rslt) ? w_tkn_pc : r_pc + 4;
endmodule
```

Simulation in the ACRi room environment

```
$ cd
$ mkdir aca
$ cd aca
$ cp /home/tu_kise/aca/circuit1.v .
$ iverilog circuit1.v
$ ./a.out

$ /usr/bin/gtkwave dump.vcd
```

```
$ cd
$ cd aca
$ cp -r /home/tu_kise/aca/rvcore1 .
$ cd rvcore1
$ make
$ make run
```