dv-cpu-rv: CPU design of RISC-V

Devin

balddevin@outlook.com

2023/7/15

Content

1 Preface 3

2 Hardware Design 3

2.1 Basic Single Cycle Implementation 3

2.1.1 Deisng Diagram 3

2.1.2 The Building of Data Path 3

2.1.3 The Building of Control Path 4

2.2 Basic Pilelined Implementation 4

2.2.1 Design Diagram 4

2.2.2 Data Hazard: Forwarding or Bypass 5

2.2.3 Control Hazard: Branch Prediction 5

3 Functional Description 5

3.1 Files and Directory Structure 5

3.2 Design of Modules in Details 6

3.2.1 ALU control unit 6

3.2.2 Branch and Jump 6

3.2.3 Strobe Unit 7

4 Appendices 9

4.1 Appendix 1: Support of Instruction Set 9

4.2 Appendix 2: Examples for Run 12

4.2.1 Example 1: Add and Store. 12

4.2.2 Example 2: Sum Less Than. 12

# Preface

The design of this CPU mainly refers to *Computer Organization and Design: The Hardware / Software Interface*: RISC-V Edition, David A.Patterson, John L. Hennessy.

The source code is also distributed to Github: [devindang/dv-cpu-rv](https://github.com/devindang/dv-cpu-rv). Please feel free to submit issue or pull request.

# Hardware Design

## Basic Single Cycle Implementation

### Deisng Diagram



Figure 1.1 The Basic Single Cycle Implementation of CPU

The corresponding hash code is: a7b05c264b7f45e27a81ddc02184c6dcee29fdf9

The figure above shows the implementation of the basic single cycle cpu, in which the hinted lines are signals of control path, while the others are signals of data path.

### The Building of Data Path

A data path is a unit used to operate on or hold data within a processor. In the ISC-V implementation, the datapath elements include the instruction and data memories, the register file, the ALU, and dders.

### The Building of Control Path

## Basic Pilelined Implementation

### Design Diagram



Figure 3.1 The basic pipelined implementation of CPU

The corresponding hash identifier is: 1d28ab2a485737b8bd90fa777fd550d5183b705c

The figure above shows the implementation of the basic pipelined cpu, in which the hinted lines are signals of control path, while the others are signals of data path. Compared to the single cycle implementation, additional units are required, they are:

i) 4 registers for pipelining, named IF/ID, ID/EX, EX/MEM, MEM/WB, separately;

ii) Forwarding unit for dealing with data hazard introduced by pipelining;

iii) Hazard detection unit for stalling the CPU in special cases;

iv) Branch prediction unit for accelerating the CPU, which saves the operation cycles;

v) Forwarding unit for RegisterFile, which solve the read/write hazard of register.



### Data Hazard: Forwarding or Bypass

Data hazards are obstacles to pipelined execution. The method to deal with this issue is adding a forwarding unit, which forwarding the data in the previous data flow, such as alu\_result, or registered ones, to current execution cycle, instead of waiting for the last instruction to write back.

### Control Hazard: Branch Prediction

# Functional Description

## Files and Directory Structure

The figure below shows the layout of the directories in the example system.

<home directory> Local git directory.

│ clean.pl Perl script for cleaning temporary files.

├─core/ The implementation of CPU core.

│ ├─bench/ Bench codes for CPU core.

│ ├─rtl/ RTL codes.

│ ├─sim/ VCS+Verdi simulation environment.

│ └─vsim/ Modelsim simulation environment.

└─docs/ Related documentation.

## Design of Modules in Details

### ALU control unit

To simplify the design of CPU, control path and data path are introduced. The ALU control unit control the ALU operation mode for all the instructions, the *alu\_op\_sel* signal in this module inherit the funct3 filed and instr[30] in R-type instruction in RV32I ISA.

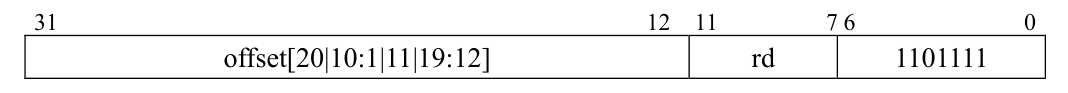
|  |  |
| --- | --- |
| ***alu\_op\_sel*** | **function** |
| 0000 | Add |
| 1000 | Subtract |
| 0001 | Shift Left Logical |
| 0010 | Set Less Than |
| 0011 | Set Less Than Unsigned |
| 0100 | Exclusive Or |
| 0101 | Shift Right Logical |
| 1101 | Shift Right Arithmetic |
| 0110 | Or |
| 0111 | And |

### Branch and Jump

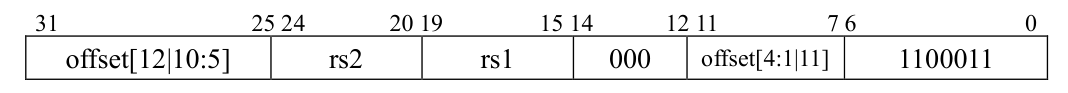
RISC-V instructions are 4 bytes long, the RISC-V branch instructions are designed to stretch their reach by having the PC-relative address refer to the number of words between the branch and the target instruction, rather than the number of bytes. However, the RISC-V architects wanted to support the possibility of instructions that are only 2 bytes long, so the branch instructions represent the number of halfwords between the branch and the branch target[[1]](#footnote-1).

Thus, the 20-bit address field in the jal instruction can encode a distance of ±219 halfwords, or ±1 MiB from the current PC. Similarly, the 12-bit field in the conditional branch instructions is also a halfword address, meaning that it represents a 13-bit byte address.

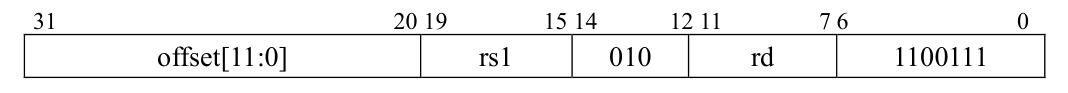
**jal**



**beq**



**jalr**



The term offset in RISC-V is in Bytes, so the offset filed in jal and beq instruction use [12:1] rather than [11:0], which means that the content in the offset filed concatenated with 0 becomes the true offset in Bytes.

Note that JALR has the offset in Bytes instead of Halfword, the PC souces has the following

* Increment (+4 in physical address)
* Branch target (require shift left 1)
* JAL (require shift left)
* JALR (no shift required)

### Strobe Unit

Strobe unit is designed to implement instructions about Load and Store, they are:

* LB, LH, LW, LBU, LHU, SB, SH, SW in RV32I,
* LD, LWU, SD in RV64I.

The LD instruction loads a 64-bit value from memory into register rd for RV64I.

The LW instruction loads a 32-bit value from memory and sign-extends this to 64 bits before storing it in register rd for RV64I.

The LWU instruction, on the other hand, zero-extends the 32-bit value from memory for RV64I.

The LH and LHU instruction are defined analogously for 16-bit values, as are LB and LBU for 8-bit values. The SD, SW, SH, and SB instructions store 64-bit, 32-bit, 16-bit, and 8-bit values from the low bits of register rs2 to memory respectively.

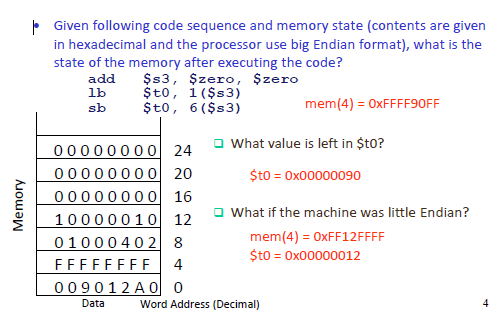
The offset filed is in Bytes. for SD or LD instruction, the offset must be the multiplies of 8 (A doubleword is 8 bytes), for example,

ld x9, 240(x10) // Temporary reg x9 gets A[30]

lw x10, 240(x10) // Temporary reg x9 gets A[30][31:0]

It’s important that the offset must be aligned for simplicity, unaligned access of memory costs additional clock cycles thus slows the CPU.

Assume that the data accessed is aligned in memory, they are stored or fetched in special rules, just take the following example to understand[[2]](#footnote-2).



add $s3, $zero, $zero

This performs the addition $s3 = 0 + 0, effectively setting the register $s3 to a value of zero.

lb $t0, 1($s3)

This loads a byte from a location in memory into the register $t0. The memory address is given by 1($s3), which means the address $s3+1. This would be the 0+1=1st byte in memory. Since we have a big-endian architecture, we read bytes the 4-byte chunks "big end first".

byte: 0 1 2 3

00 90 12 A0

The 0th byte is 00, and the 1st byte is 90. So we load the byte 90 into $t0.

sb $t0, 6($s3)

This stores a byte from the register $t0 into a memory address given by 6($s3). Again this means the address $s3+6.

byte: 4 5 6 7

FF FF FF FF

becomes

byte: 4 5 6 7

FF FF 90 FF

Now, what if the architecture was little-endian? (Which is the endian sequence of RISC-V) This would mean bytes are arranged "little end first" in memory, so the effect of the 2nd and 3rd instructions change.

lb $t0, 1($s3)

This loads the byte in memory address 1 into register $t0. But now the addresses are "little end first", so we read 12 into the register instead.

byte: 3 2 1 0

00 90 12 A0

Next...

sb $t0, 6($s3)

This stores the byte in register $t0, which is 12 into a memory address 6. Again with little-endian architecture:

byte: 7 6 5 4

FF FF FF FF

becomes

byte: 7 6 5 4

FF 12 FF FF

The strobe unit is designed to read 0x90 from the bit range of [23:16], the second byte in big-endian, of the slice of physical address 0 of memory and, to write 0x90 to the correct part of the correct address.

For sb, lb instruction, the offset must be the multiply of 1, the lower 3 bits are used to determine the location of byte in 64 bits to store and fetch. For sh, lh, lhu instruction, the offset must be the multiply of 2, the lower 2 bits are used to determine the location of halfword in 64 bits. For sw, lw, lwu instruction, the occasion becomes 4, and 1 bit. For sd, ld instruction, the whole 64 bits are used, the offset is the multiply of 8.

# Appendices

## Appendix 1: Support of Instruction Set

The regularity of opcode:

The example of RISC-V pseudo instructions can be found in *risc-v specification* v2.2 p109, from which we can deal with the assembly codes.

**RV32I Base Instruction Set**

Total: 47

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Type** | **Order** | **Instruction** | **Description** | **Compatibility** |
| R-type | 1 | ADD | Add. | YES |
| 2 | SUB | Subtract. | YES |
| 3 | SLL | Shift Left Logical. | YES |
| 4 | SLT | Set Less Than. | YES |
| 5 | SLTU | Set Less Than Unsigned. | YES |
| 6 | XOR | Exclusive or. | YES |
| 7 | SRL | Shift Right Logical. | YES |
| 8 | SRA | Shift Right Arithmetic. | YES |
| 9 | OR | Or. | YES |
| 10 | AND | And. | YES |
| I-type | 11 | JALR | Jump And Link Register. |  |
| 12 | LB | Load Byte. | YES |
| 13 | LH | Load Halfword. | YES |
| 14 | LW | Load Word. | YES |
| 15 | LBU | Load Byte Unsigned. | YES |
| 16 | LHU | Load Halfword Unsigned. | YES |
| 17 | ADDI | Add Immediate. | YES |
| 18 | SLTI | Set Less Than Immediate. | YES |
| 19 | SLTIU | Set Less Than Immediate Unsigned. | YES |
| 20 | XORI | Exclusive Or Immediate. | YES |
| 21 | ORI | Or Immediate. | YES |
| 22 | ANDI | And Immediate. | YES |
| 23 | SLLI | Shift Left Logic Immediate. | YES |
| 24 | SRLI | Shift Right Logic Immediate. | YES |
| 25 | SRAI | Shift Right Arithmeic Immediate. | YES |
| S-type | 26 | SB | Store Byte. | YES |
| 27 | SH | Store Halfword. | YES |
| 28 | SW | Store Word. | YES |
| B-type | 29 | BEQ | Branch if Equal. | YES |
| 30 | BNE | Branch Not Equal. | YES |
| 31 | BLT | Branch Less Than. | YES |
| 32 | BGE | Branch Greater or Equal. | YES |
| 33 | BLTU | Branch Less Than Unsigned. | YES |
| 34 | BGEU | Branch Greater or Equal Unsigned. | YES |
| U-type | 35 | LUI | Load Upper Immediate. | YES |
| 36 | AUIPC | Add Upper Immediate to PC. |  |
| J-type | 37 | JAL | Jump And Link. |  |
| other | 38 | FENCE |  | NO |
| 39 | FENCE.I |  | NO |
| 40 | ECALL |  | NO |
| 41 | EBREAK |  | NO |
| 42 | CSRRW |  | NO |
| 43 | CSRRS |  | NO |
| 44 | CSRRC |  | NO |
| 45 | CSRRWI |  | NO |
| 46 | CSRRSI |  | NO |
| 47 | CSRRCI |  | NO |

**RV64I Base Instruction Set (in addition to RV32I)**

Total: 15

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Type** | **Order** | **Instruction** | **Description** | **Compatibility** |
| R-type | 1 | ADDW | Add Word. |  |
| 2 | SUBW | Subtract Word. |  |
| 3 | SLLW | Shift Left Logical Word. |  |
| 4 | SRLW | Set Less Than Word. |  |
| 5 | SRAW | Set Less Than Unsigned Word. |  |
| I-type | 6 | LWU | Load Word Unsigned. | YES |
| 7 | LD | Load Doubleword. | YES |
| 8 | SLLI | Shift Left Logic Immediate. |  |
| 9 | SRLI | Shift Right Logic Immediate. |  |
| 10 | SRAI | Shift Right Arithmetic Immediate. |  |
| 11 | ADDIW | Add Immediate Word. |  |
| 12 | SLLIW | Shift Left Logic Immediate Word. |  |
| 13 | SRLIW | Shift Right Logic Immediate Word. |  |
| 14 | SRAIW | Shift Right Arithmetic Immediate Word. |  |
| S-type | 15 | SD | Store Doubleword. | YES |

## Appendix 2: Examples for Run

### Example 1: Add and Store.

**Date**: 2023/7/23

**Hash**: a7b05c264b7f45e27a81ddc02184c6dcee29fdf9

**Description**: Given two numbers, add them and store into memory.

C code

#include "stdio.h"

int main() {

int a = 14;

int b = 15;

int c;

c = a + b;

return 0;

}

Assembly code

addi x2 x0 14; //0// 00000000111000000000000100010011

addi x3 x0 15; //1// 00000000111100000000000110010011

add x1 x2 x3; //2// 00000000001100010000000010110011

sd x1 8(x2); //3// 00000000000100010011010000100011

### Example 2: Sum Less Than.

**Date**: 2023/7/29

**Hash**: 1d28ab2a485737b8bd90fa777fd550d5183b705c

**Description**: Given a non-zero natural number N, calculate the sum of natural numbers less than N.

C code

#include "stdio.h"

int main() {

int N = 10;

int sum = 0;

for(int i=1; i<N; i++){

sum = sum+i;

}

return 0;

}

Assembly code

addi x1 x0 10; //0// 00000000101000000000000010010011

addi x2 x0 1; //4// 00000000000100000000000100010011

addi x3 x0 0; //8// 00000000000000000000000110010011

add x3 x2 x3; //12// 00000000001100010000000110110011

addi x2 x2 1; //16// 00000000000100010000000100010011

blt x2 x1 A12; //20// 11111110000100010100110011100011

sd x3 8(x1); //24// 00000000001100001011010000100011

1. Computer Organization and Design RISC-V Edition: P264 [↑](#footnote-ref-1)
2. https://stackoverflow.com/questions/28707615/loading-and-storing-bytes-in-mips [↑](#footnote-ref-2)