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

Devin

balddevin@outlook.com

2023/7/15

Content

1 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. Please feel free to submit issue or pull request.

2 Hardware Design

These subsections of this section are not the final version, any of them is corresponded to a git commit, the hash identifiers are displayed, It mainly plays the role of recording and learning rather than technical documentation.

2.1 Basic Single Cycle Implementation

2.1.1 Deisng Diagram

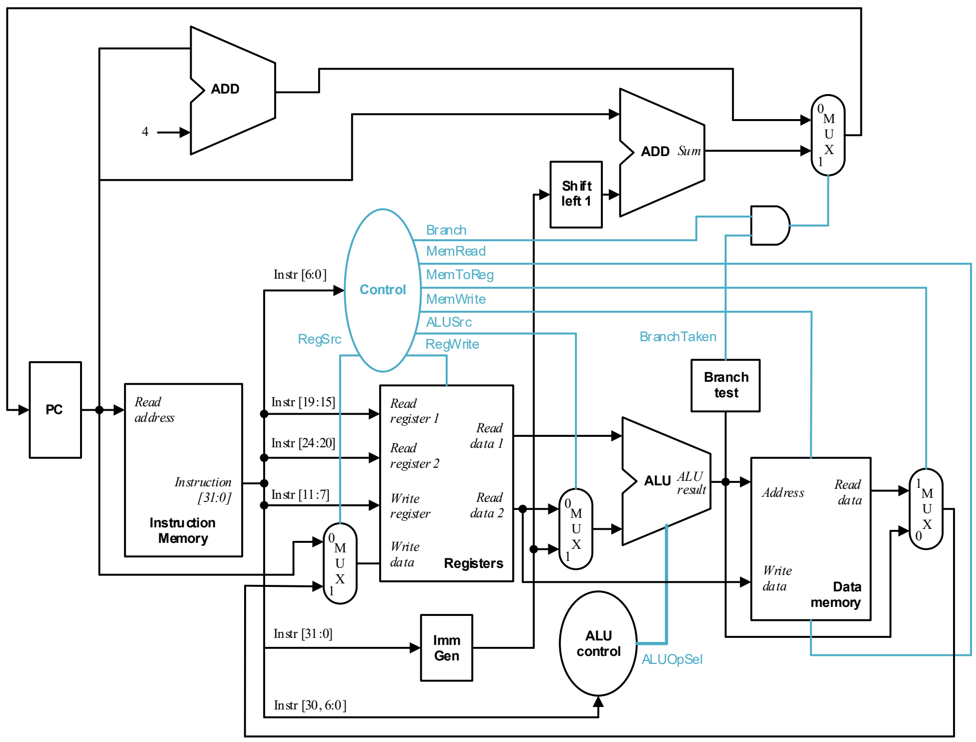


Figure 1.1 The Basic Single Cycle Implementation of CPU

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.

2.1.2 The Building of Data Path

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

Instruction Memory is the memory where instructions are stored, which is independent of Data Memory in Havard architecture computer. In this implementation, the instruction memory is a memory has the address line bits of 64, which is identical to the bit width of data processed in CPU and, has the data line of 32 bits, the bit width of all the RISC-V instructions.

PC (Program Counter) is designed to fetch the instructions of Instruction Memory, which increments itselves by 4 per clock cycle in most cases, 32 is 4 bytes, and addressing in RISC-V is in bytes. In some cases, PC will jump or branch to certain location of the Instruction Memory and fetch. Thus, there is a MUX in the top right.

Register File contains the all 32 registers defined in RISC-V, each is of 64 bits width, it’s designed to store constant 0, parameters, PC, subroutine entries, etc.

ALU (Algorithm Logic Unit) is the core of CPU, it’s responsible for almost all the algorithms like add, subtract, xor, or, and in I standard, and more broadly, multiply, division, floating in M extension and F extension. The ALU in this design has the data width of 64, of either the oprands or the result.

Data Memory is the memory to store rich data, interact with external components like DMA (Direct Memory Accessing). It determines the maximum data counts a computer could handle at the same time. For example, a data memory with data width of 1 Byte, and address width of 32, the maximum data counts it could handle is 2^32*1B=2^2*1GiB=4GiB. In this design, the Data memory has the data width of 64, which is 8B, so the physical address must be divided by 8.

Immediate Generation is the unit to sign expand the immediate, and transfer to ALU or, act as an offset to PC to control procedure. It decodes the instruction fetched, retrieve the corresponding immediate according to the opcode filed, funct7 filed, or funct3 filed, and then perform sign expand to 64 bits.

2.1.3 The Building of Control Path

A control path is a unit to control the data transfer between the data path units according the decoding result of the instruction. In this RISC-V implementation, the control path generate the following signals to control the process.

Note that these control signals are only applied to the single path situation, only in this commit, the further update introduced several complicated control signals, it will be talked in the later sections.

Control Lanes	Deasseted	Asserted
Branch	None.	combined with branch testing to determine whether to branch.

MemRead	None.	Read memory.
MemToReg	The value fed to register write data input comes from the ALU result.	The value fed to register write data input comes from the data memory.
MemWrite	None.	Write memory.
ALUSrc	The operand 2 of ALU comes from the register read data 2.	The operand 2 of ALU comes from the immediate generation unit output.
RegWrite	None.	Write data to register.
RegSrc	X	X

RegSrc is a failed set, it will deprecated further, it's initially created to select PC as register write data for instructions like JAL, JALR.

With these control signals, the CPU could process simply calculation, it will process only one instruction per signal cycle. There is an example in ./ docs/assembly.md , which is marked as 1, it will perform an addition calculation and obtain 29 as the result, which will be stored into memory. This is the only test cases adapted to this commit.

2.2 Basic Pilelined Implementation

2.2.1 Design Diagram

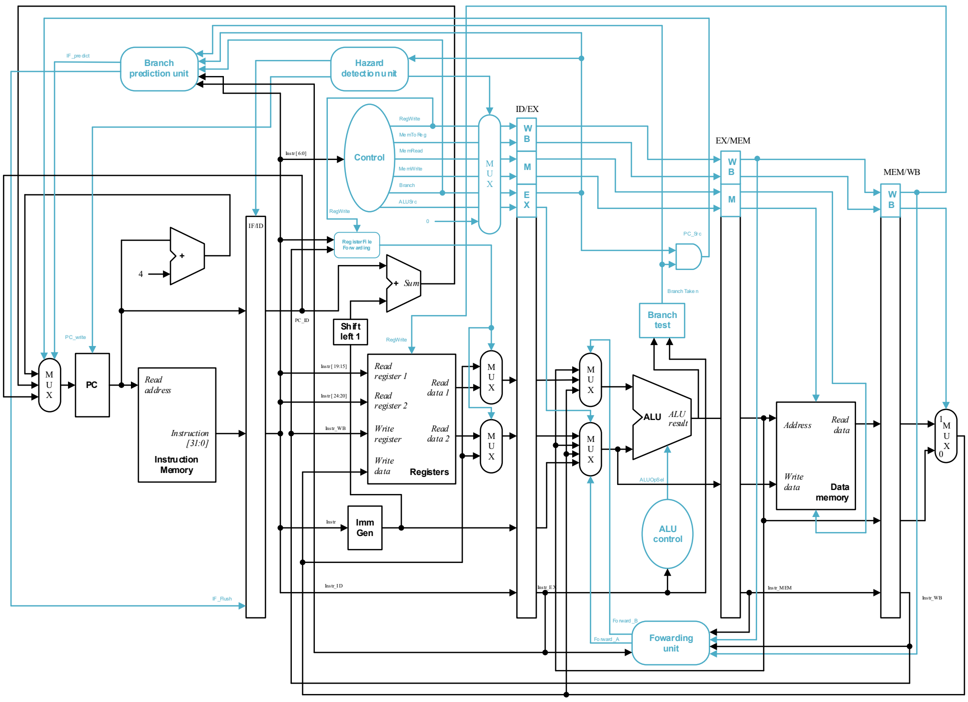


Figure 3.1 The basic pipelined implementation of CPU

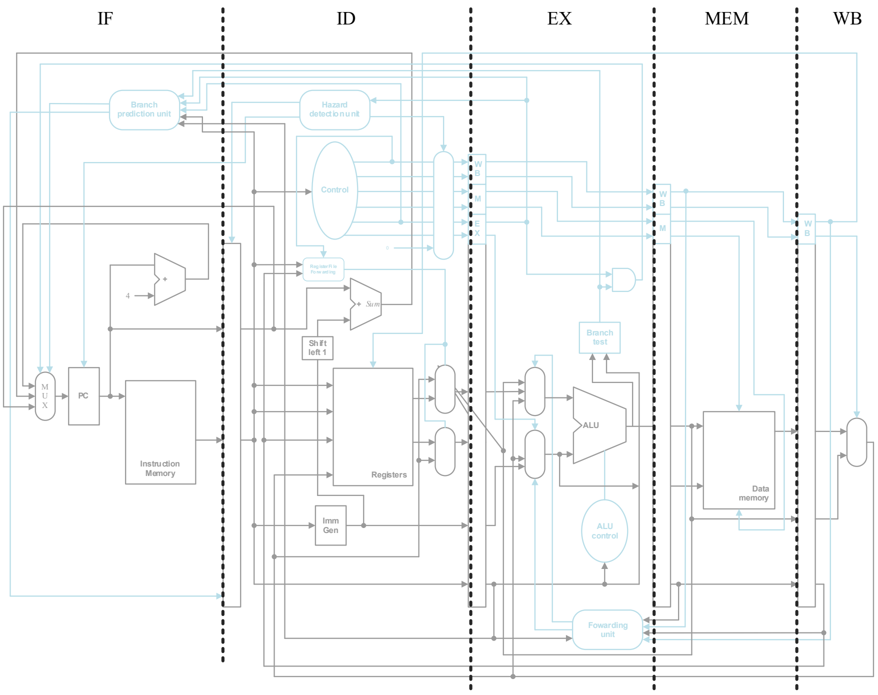
The corresponding hash identifier is: 1d28ab2a485737b8bd90fa777fd550d5183b705c

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.

The details of these units will be talked in the later section: Support for RV32I and RV64I.



The figure above is the partition of the 5-stage pipelined CPU, the five stages and their functions are listed:

1. IF (Instruction Fetch): Fetch instructions from the Instruction Memory with PC as memory address. Besides, branch prediction unit is placed here to control PC.
2. ID (Instruction Decode): Decode instructions to generate control signals, sign expand immediates, control the operation of register files. Besides, Hazard detect unit is placed here to stall the pipeline.
3. EX (Executation): Execute algorithm calculation, and perform branch testing. Besides, forwarding unit is placed here to forwarding result of the pipeline stages output.
4. MEM (Memory Access): Access memory for load and store instructions.
5. WB (Write Back): Write back data to register files.

2.2.2 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.

2.2.3 Control Hazard: Branch Prediction

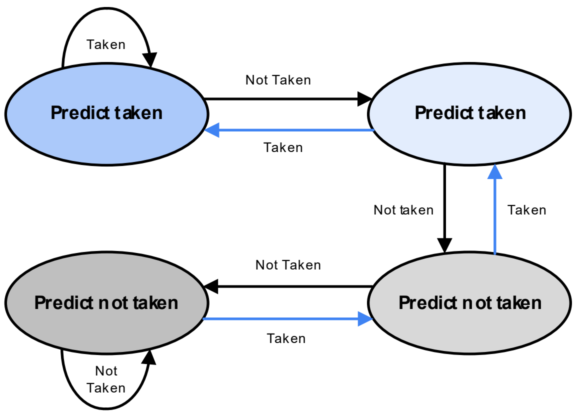
Branch prediction is a technique used in CPU design that attempts to guess the outcome of a conditional operation and prepare for the most likely result. A digital circuit that performs this operation is known as a branch predictor. It is an important component of modern CPU architectures.

Let's take the process without branch prediction. The branch result is available only when the process get the result from branch testing unit at the phase of Executation, there is 3 cycles delay after PC get changed (PC=>IF=>ID=>EX). As the branch instruction was fill into the pipeline, and the branch result is not available, the instructions below will also fill into the pipeline, regardless if the branch will be taken or not. That's the case that we predict the branch will always not be taken, additional cycles will be wasted if the branch is taken.

The basic branch prediction scheme is 1-bit prediction buffer which store the history branch result for corresponding address or low-order address.

We don't know, in fact, if the prediction is the right one—it may have been put there by another conditional branch that has the same low-order address bits. However, this doesn't affect correctness. Prediction is just a hint that we hope is correct, so fetching begins in the predicted direction. If the hint turns out to be wrong, the incorrectly predicted instructions are deleted, the prediction bit is inverted and stored back, and the proper sequence is fetched and executed.

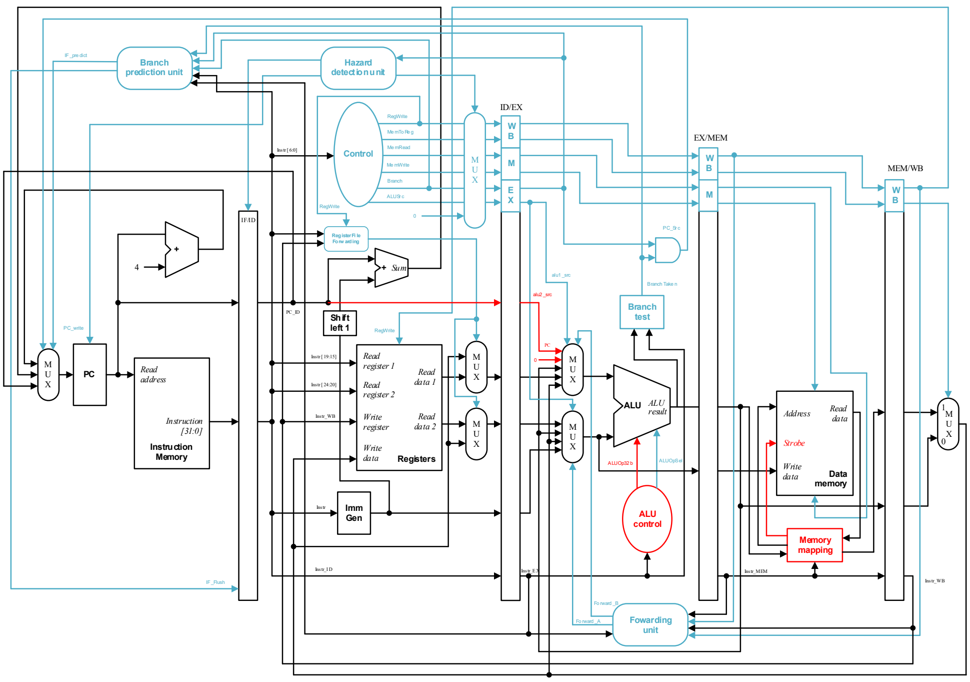
For most cases, the iteration use the same address (offset in RISC-V) for branching, and repeat considerable times. Branch prediction mechanism will save clock cycles in this situation.



The figure above shows the state transition diagram for 2-bit branch prediction, only after 2 branch taken action, the prediction unit assert a branch indication. It's impelented by using a simple 2-bit counter, the counter value 0,1 deassert the prediction, while the counter value 2,3 assert the prediction.

2.3 Support for RV32I and RV64I

2.3.1 Design Diagram



The red hinted lines are logic units introduced to fully support the instructions in RV32I and RV42I. It will be talked later.

2.3.2 Modification Overview

There are modified units and additional units for fully supporting RV32I and RV64I. They are:

i) ALUOp32b for ALU control unit, it's designed for RV64I to support word operation;

ii) Memory mapping unit for Byte, Halfword, Word operation of data memory, more details , refer to 3.2.3 Strobe Unit;

iii) ALU op1 MUX logic, PC and 0 are added to the MUX inputs, LUI instruction use 0 as op1, and AUIPC instruction use PC as op1.

iv) Besides, ImmGen, Control, ALU, Memory, and BranchTest units are required to be updated for wider instructions, the details of them will be talked in the later section – Functional Description.

2.3.3 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

2.3.4 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].

Thus, the 20-bit address field in the jal instruction can encode a distance of $\pm 2^{19}$ 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

31 12	11 7	6 0
offset[20|10:1|11|19:12]	rd	1101111

beq

31 25	24 20	19 15	14 12	11 7	6 0
offset[12|10:5]	rs2	rs1	000	offset[4:1|11]	1100011

jalr

31 20	19 15	14 12	11 7	6 0
offset[11:0]	rs1	010	rd	1100111

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)

[1] Computer Organization and Design RISC-V Edition: P264

- Branch target (require shift left 1)
- JAL (require shift left)
- JALR (no shift required)

2.3.5 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.

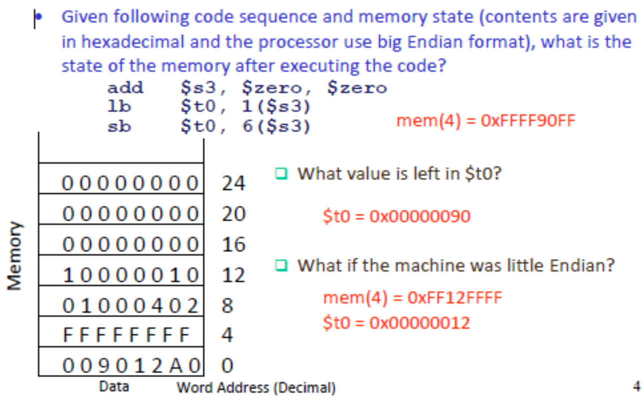
The offset filed is in Bytes. for SD or LD instruction, the offset must be the multiplies of 8 (A doubleword is 8 bytes), for example,

```
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].

[2] https://stackoverflow.com/questions/28707615/loading-and-storing-bytes-in-mips




```
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.

2.4 Support for RV32M and RV64M

2.4.1 Booth-Wallace Multiplier

This part mainly refers to 《计算机体系结构基础》https://github.com/foxsen/archbase, which compiles with CC BY-NC 4.0, thanks to the contributors.

Booth Algorithm[3]

Suppose there is a 8x8 multiplication, the result for signed multiplication is

$$\begin{aligned}[X\cdot Y]_{2c} &= -X\cdot y_7\cdot 2^7 + X\cdot y_6\cdot 2^6 + \cdots + X\cdot y_1\cdot 2^1 + X\cdot y_0\cdot 2^0 \\ &= [X]_{2c}\cdot(-y_7\cdot 2^7 + y_6\cdot 2^6 + \cdots + y_1\cdot 2^1 + y_0\cdot 2^0)\end{aligned}$$

Where, 2c means 2's complement, the sign will be omitted in the subsequent expression, only signed multiplication is talked at present.

Booth multipliers transform the multiplications to

$$[X\cdot Y]_{2c} = [X]_{2c}\cdot(-y_7\cdot 2^7 + y_6\cdot 2^6 + \cdots + y_1\cdot 2^1 + y_0\cdot 2^0)$$

[3] 计算机体系结构基础, 胡伟武: P210

$$= -y_7 \cdot 2^7 + y_6 \cdot 2^6 - y_6 \cdot 2^6 + \cdots + y_1 \cdot 2^2 - y_1 \cdot 2^1 + y_0 \cdot 2^1 - y_0 \cdot 2^0$$

$$= (y_6 - y_7) \cdot 2^7 + (y_5 - y_6) \cdot 2^6 + \cdots + (y_0 - y_1) \cdot 2^1 + (y_{-1} - y_0) \cdot 2^0$$

Where, $y_{-1} = 0$. After transformation, the regularity is revealed, no additional multiply -1 is required for MSB partial product. It's also called Radix-2 Booth Algorithm.

Let's talk about Radix-4 Booth Algorithm, which is the most common implementation in CPU, and DSP unit.

$$[X \cdot Y]_{2c} = (y_5 + y_6 - 2y_7) \cdot 2^6 + (y_3 + y_4 - 2y_5) \cdot 2^4 + \cdots + (y_{-1} + y_0 - 2y_1) \cdot 2^0$$

Rather than divde 2 partial product together, Radix-4 Booth Algorith divide every 3 partial product together and perform sum every 2 bits, or say, shift 2 bits for the multipliers. Radix-4 means that, it set the multipliers as a set of digits 0–3 instead of just 0,1 (basic array multiplications, or radix-2 booth algorithm). This cuts the number of partial products in half because you are multiplying by two binary bits at once. However, it requires multiplying by 3 which is difficult. (Multiplication by 0, 1, or 2 is trivial because they only involve simple shifts. Three is the hard one[4].) To avoid multiplying by 3, we use Booth's observation and recode the digit set to be 2, 1, 0, －1, and －2.

y_{i+1}	y_i	y_{i-1}	z_i	comment	z1	z2	neg
0	0	0	0	No add required.	0	0	0
0	0	1	1	+X (2's complement)	1	0	0
0	1	0	1	+X (2's complement)	1	0	0
0	1	1	2	+2X (2's complement)	0	1	0
1	0	0	-2	-2X (2's complement)	0	1	1
1	0	1	-1	-X (2's complement)	1	0	1
1	1	0	-1	-X (2's complement)	1	0	1
1	1	1	0	No add required.	0	0	1

The column z1, z2, neg is the signals in the implementation. where, z1 means that there is a 1-bit shift, z2 means that there is a 2-bits shift, neg means that there is a subtraction which is implemented by

Wallace Tree[5]

The Wallace tree is build by the following reasoning, assuming already booth encoded.

level	isolated	remain	in	out
0	0	0	11*3	11*2
1	0	1	7*3	7*2
2	1	0	5*3	5*2
3	0	1	3*3	3*2

[4] https://www.brown.edu/Departments/Engineering/Courses/En164/BoothRadix4.pdf

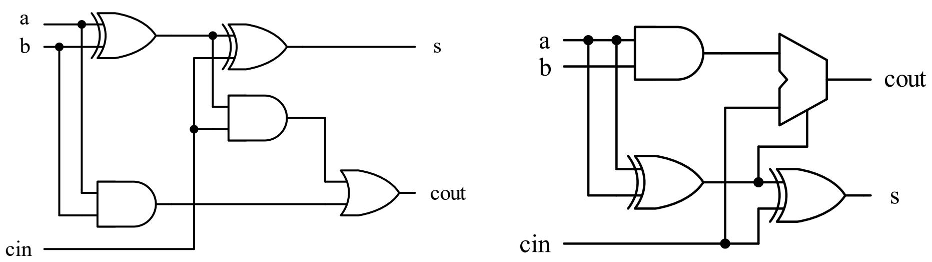
[5] 计算机体系结构基础, 胡伟武: P215

4	0	1	2*3	2*2
5	0	2	1*3	1*2
6	0	1	1*3	1*2
7	0	0	1*3	1*2

Add register to every two stages?

Full Adder

The logic architecture chosen to implement full adder is talked below.



The left side one is the basic connection, where,

$s = a \oplus b \oplus cin, \quad cout = a \cdot b + (a \oplus b) \cdot c$

While the right side is the improved implementation, where

$s = a \oplus b \oplus cin, \quad cout = (a \oplus b)\ ?\ cin : (a \cdot b) = a \cdot b + (a \oplus b) \cdot cin$

It takes the advantage of the feature that

$\overline{a \oplus b} = a \cdot b + \bar{a} \cdot \bar{b}, \quad \text{and} \;\; 1 + A = 1$

Consider the perfomance, the left implementation has a logic stage of 3, while the right one also has it of 3 (the MUX takes 2).

Consider the area, the left implementation consume the MOSFET counts of 8+8+6+6+6=34, while the right one consume it of 8+8+6+14=36.

In Xilinx FPGA, there are proprietary MUX units, the CARRY4 Chain adopts the right one to save LUTs resources.

2.4.2 SRT Divider

3 Functional Description

3.1 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.

3.2 RV32I Module Design

3.3 RV32M Module Design

4 Appendices

4.1 Appendix 1: Support of Instruction Set

The regularity of opcode:

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 (>32b)
00	LOAD	LOAD-FP	custom-0	MISC-MEM	OP_IMM	AUIPC	OP-IM-M32	48b
01	STORE	STORE-FP	custom-1	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	reserved	custom-2/rv128	48b
11	BRANCH	JALR	reserved	JAL	SYSTEM	reserved	custom-3/rv128	>=80b

Besides, inst[1:0]=11.

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.	YES

	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.	YES
J-type	37	JAL	Jump And Link.	YES
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.	YES

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

4.2 Appendix 2: Examples for Run

4.2.1 Example 1: Add and Store.

Date: 2023/7/23

Hash: a7b05c264b7f45e27a81ddc02184c6dcee29fdf9

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

C code

```c
#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
```

4.2.2 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

```c
#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
```