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

Devin

balddevin@outlook.com

2023/7/22

Content

1 Preface 4

2 Hardware Design 4

2.1 Basic Single Cycle Implementation 4

2.1.1 Deisng Diagram 4

2.1.2 The Building of Data Path 5

2.1.3 The Building of Control Path 5

2.1.4 How the Processor Works? 6

2.2 Basic Pilelined Implementation 7

2.2.1 Design Diagram 7

2.2.2 Data Hazard: Forwarding or Bypass 9

2.2.3 Control Hazard: Branch Prediction 10

2.2.4 Architecture Hazard: Stall 11

2.3 Support for RV32I and RV64I 11

2.3.1 Design Diagram 11

2.3.2 Modification Overview 12

2.3.3 ALU control unit 12

2.3.4 Branch and Jump 12

2.3.5 Strobe Unit 13

2.4 Support for RV32M and RV64M 15

2.4.1 Booth-Wallace Multiplier 15

2.4.2 SRT Divider 22

3 Functional Description 32

3.1 Files and Directory Structure 32

3.2 RV32I Module Design 32

3.3 RV32M Module Design 32

4 Appendices 33

4.1 Appendix 1: Support of Instruction Set 33

4.2 Appendix 2: Examples for Run 36

4.2.1 Example 1: Add and Store. 36

4.2.2 Example 2: Sum Less Than. 36

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

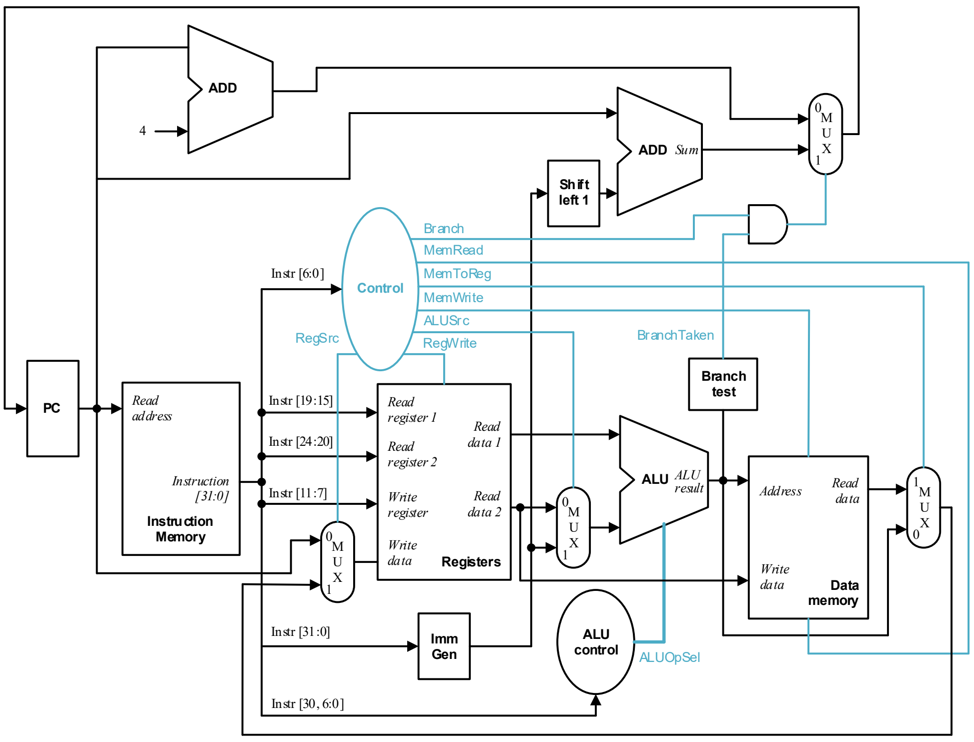


Figure 2.1 The Basic Single Cycle Implementation of CPU

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.

2.1.4 How the Processor Works?

The single cycle implementation is a good example to explain. As described, the processor contains both Data Path and Control Path, the Data Path is responsible for data exchange, and the Control Path generate nearly all the enable signals and selection signals and thus guiding the processor in the correct path.

Instruction Fetch: The processor fist fetch the instructions from the Instruction Memory, the instructions are the machine codes that the process could read itself. The instructions in RISC-V are all 32-bits long, there is no additional logic for supporting shorter instructions. The processor read the instructions one by one, and process them, give the corresponding responses.

Instruction Decode: As the processor obtains the instructions, the next action is to decode, that is, to analyze the instructions. The control unit is effective here, which generates control signals like Branch, MemRead, ALUSrc, etc. Immediate Generation unit is also effective here for sign expanding immediates for I-type instructions and U-type instructions so that the processor could use them as operands in the next stage.

Execution: ALU and ALU control unit are effective here, the ALU execute the arithmetic operation with the ALU control unit. Besides, the branch testing unit is also placed here for testing if branching of not in the next instruction fetch, that's because the branch action depends on the ALU result.

Memory Access: ALU result is feed into the address ports of the Data Memory for S-type store instructions and I-type load instructions. Only in this phase, the memory is accessed. The read or write action is decided by control units in ID phase.

Write Back: RISC-V do not allow direct accessing to memory to get operands, so does the write back operation. All the data related to memory must operate in registers. The write back phase write the memory read_data from load instruction or the alu result back into registers.

2.2 Basic Pilelined Implementation

2.2.1 Design Diagram

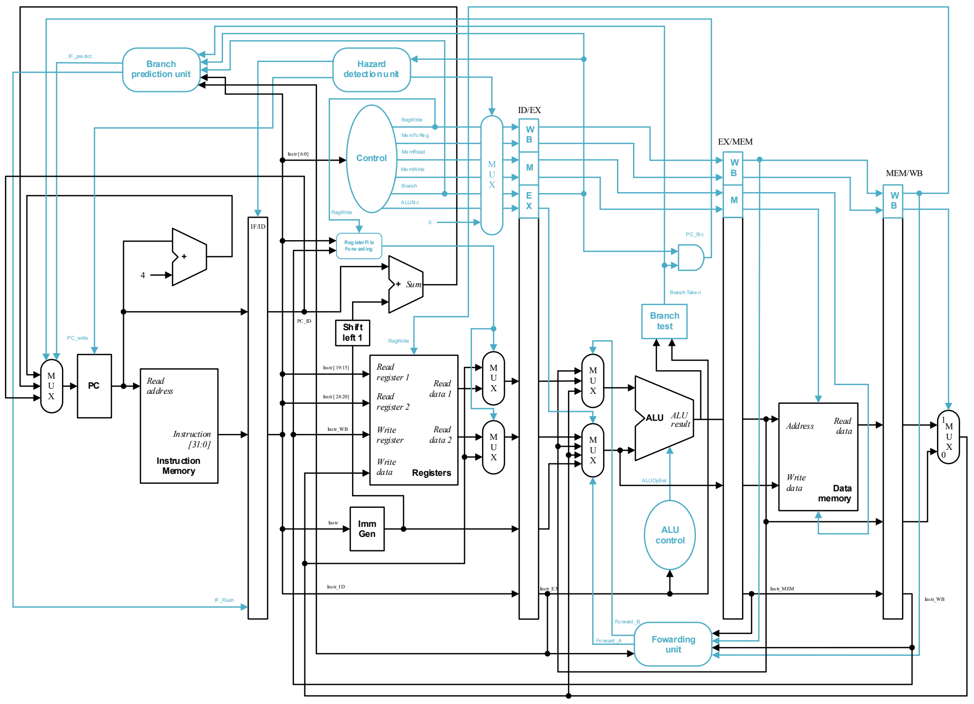


Figure 2.2 The basic pipelined implementation of CPU

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.

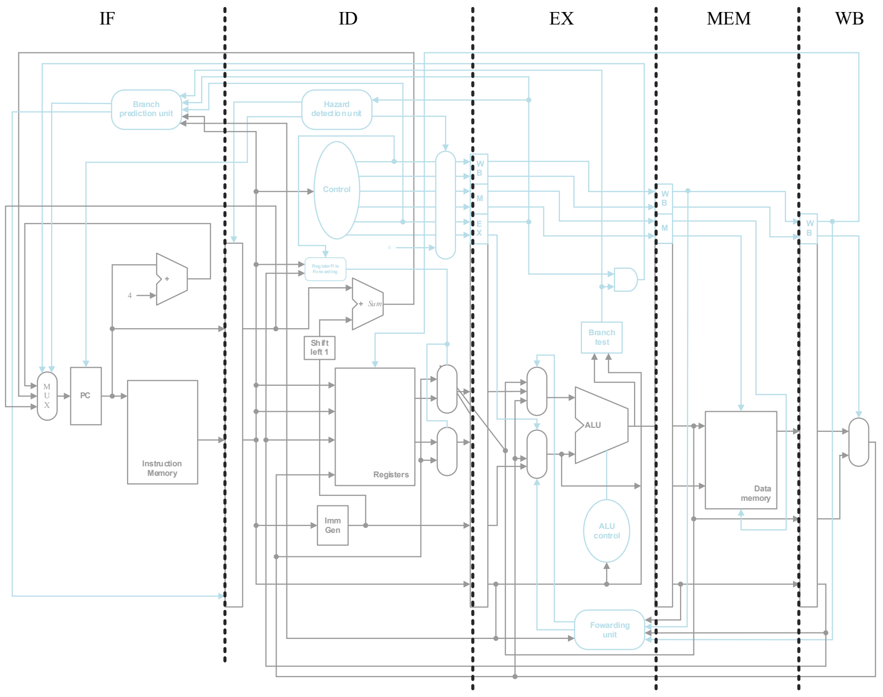


Figure 2.3 Five pipeline stages

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

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 (Execution): 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.

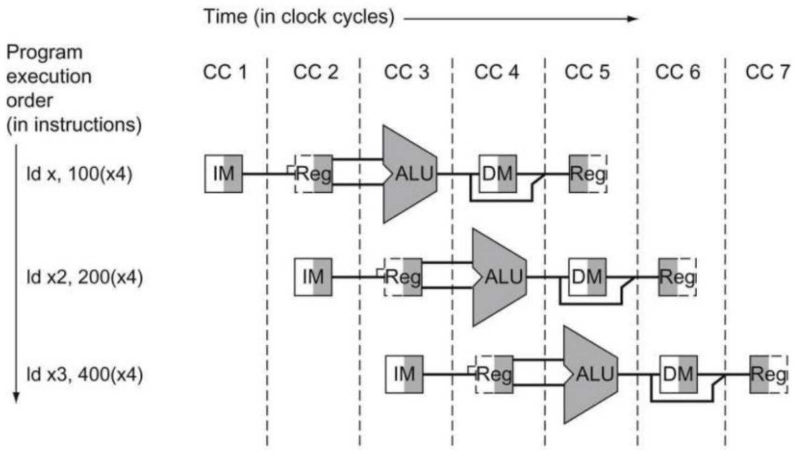


Figure 2.4 Pipeline sequential in this design

Different from the single cycle one, the pipelined architecture allow overlaps between these five phases, thus improves throughputs. The issues it brings are also clear, we will talk them in the later two sections.

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.

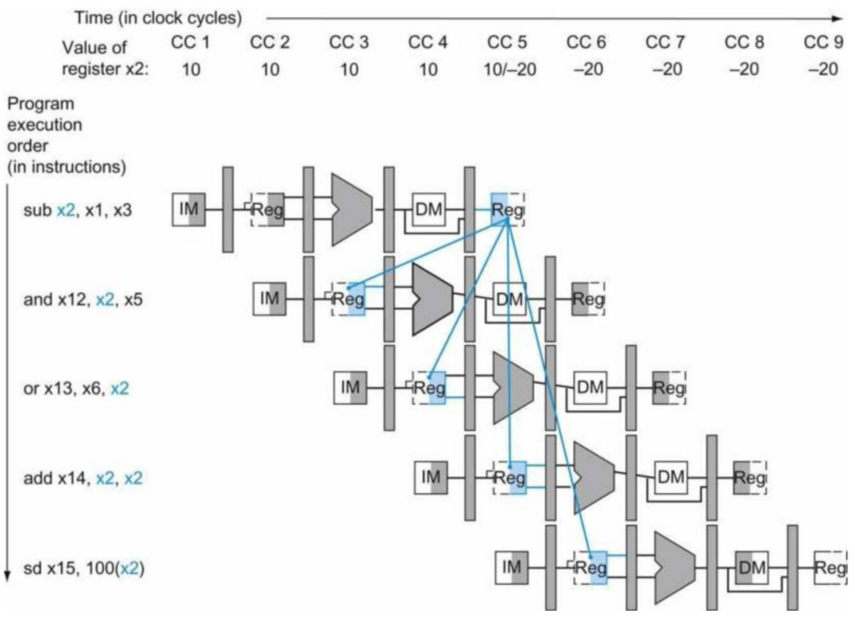


Figure 2.5 Data hazard in pipeline architecture

Figure is the example in *Computer Organization and Design*, the and instruction requires x2 as the operand for arithmetic operation, but the processor cannot supply x2 in time, the value of sub instruction is only available after write back. The next instruciont will also encounter the same issue.

The method to dealing with this issue is to add a Forwarding Unit, which forwards or bypass the alu results, or registered one to the alu input ports, so that the alu could obtain the operands in time.

We just simply inspect *rs1* in the EX stage, if *rd* in MEM stage equals to it, a forward is required, it's the case 1a in the book, the ALU result in MEM stage is forwarded to ALU operand 1. Else if *rd* in WB stage equals to it, a further registered one is required to be forwarded, it's the case 2a.

It's just the situation of ALU operand 1, the operand 2 has the similar situation, but a further condition is must. in RISC-V, I-type does not has the *rs2* filed, it will interfere the forwarding, so, if we detect current instruction is of I-type, no forward is taken, it's the additional condition.

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 Execution, 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.

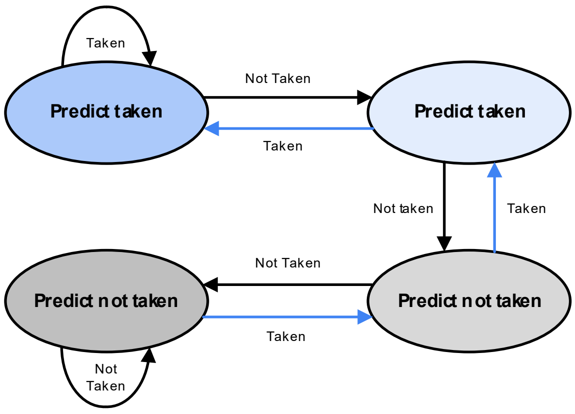


Figure 2.6 State transition for 2-bit dynamic prediction

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.

2.2.4 Architecture Hazard: Stall

2.3 Support for RV32I and RV64I

2.3.1 Design Diagram

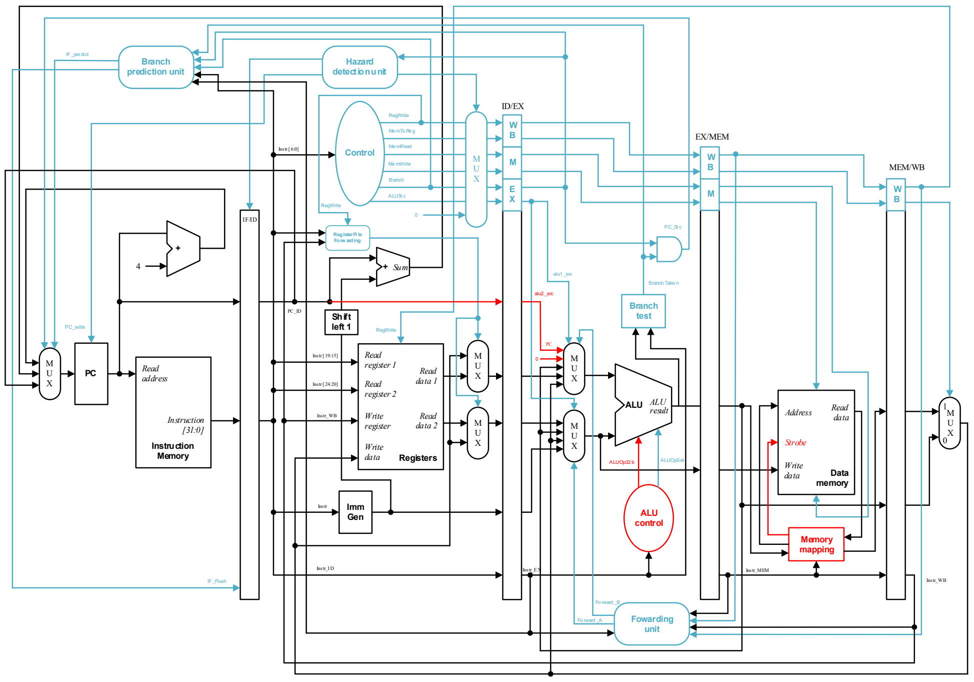


Figure 2.7 RV32I/RV64I design diagram

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].

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

Thus, the 20-bit address field in the jal instruction can encode a distance of ±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)
- 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.

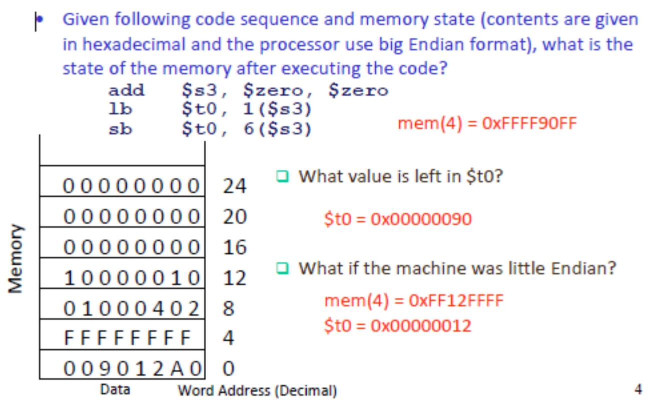
The LH and LHU instruction are defined analogously for 16-bit values, as are LB and LBU for 8-bit values. The SD, SW, SH, and SB instructions store 64-bit, 32-bit, 16-bit, and 8-bit values from the low bits of register rs2 to memory respectively.

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].



- Given following code sequence and memory state (contents are given in hexadecimal and the processor use big Endian format), what is the state of the memory after executing the code?

```
add   $s3, $zero, $zero
lb    $t0, 1($s3)
sb    $t0, 6($s3)
```

mem(4) = 0xFFFF90FF

Data	Word Address (Decimal)
00000000	24
00000000	20
00000000	16
10000010	12
01000402	8
FFFFFFFF	4
009012A0	0

Memory

- What value is left in $t0?

  $t0 = 0x00000090

- What if the machine was little Endian?

  mem(4) = 0xFF12FFFF

  $t0 = 0x00000012

4

```
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
```

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

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.

Signed Multiplication

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.

Consider the following example, -13*6=-78, where, -13 is multiplicand(X), and 6 is multiplier(Y). The last partial product is zero, it's omitted in the figure.

```
          |1 1 1 1 0 0 1 1
        x |0 0 0 0 0 1 1 0
      ----------------------
  + ... 1 1|1 1 1 0 0 1 1
  + ... 1 1|1 1 0 0 1 1
  -        |
  --------------------------
        1 1|1 0 1 1 0 0 1 0
```

The following example is 6*-13=-78, which has the same result.

```
                |0 0 0 0 1 1 0 1
              x |1 1 1 1 1 0 1 0
  ------------------------------
  + ... 0 0 0 0|0 0 0 0 1 1 0 1
  + ... 0 0 0 0|0 1 1 0 1
  + ... 0 0 0 0|1 1 0 1
  + ... 0 0 0 1|1 0 1
  + ... 0 0 1 1|0 1
  - ... 0 1 1 0|1
  ----------------------------------
        1 1 1 1|1 0 1 1 0 0 1 0
```

The differences from the unsigned multiplication algorithm are:

1). Each partial product is required to be sign expanded;

2). The last partial product is generated from subtraction instead of addition.

Radix-2 Booth Algorithm[3]

Booth multipliers transform the multiplications to

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

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

[3] 计算机体系结构基础, 胡伟武: P210

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

Where, $y_{-1} = 0$. After transformation, the regularity is revealed, no additional multiply -1 (subtraction) is required for MSB partial product. It's also called Radix-2 Booth Algorithm, for we use 2-bits to generate a corresponding 1-bit "multiplier" for the generation of each partial product.

The partial product is generated determined by "$y_i y_{i-1}$":

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

Take the following example into consideration, which calculate 13x6:

```
   0 0 0 0 1 1 0 1
 x 0 0 0 0 0 1 1 0 0
 ---------------------
   0 0 0 0 0 0 0 0 0  (00) null
   1 1 1 0 0 1 1      (10) -X
   0 0 0 0 0 0        (11) null
   0 1 1 0 1          (01) +X
 ---------------------
   0 0 0 0            (00) null
 ---------------------
 1 0 1 0 0 1 1 1 0
```

Note that, the grayed "1" is not the real carry, it's because I omitted the expanded sign bit of the partial product, the result must be possitive.

Radix-4 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	z0	z1	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	0

The column z0, z1, neg is the signals in the implementation. where, z0means that there is a 1-bit shift, z1 means that there is a 2-bits shift, neg means that there is a subtraction which is implemented by bitwise inverse then plus 1 (neg signal in this design).

The Boolean expression of this select signals can be inferred from the truth table, where,

$$z0 = y_i \oplus y_{i-1}\,,\quad z1 = (y_{i+1} \sim y_i \sim y_{i-1}) + (\sim y_{i+1} y_i y_{i-1})\,,\quad neg = y_{i+1} \cdot (\sim (y_i y_{i-1}))$$

The logic is implemented in the module *booth_encoder*, which generates these three signals for booth algorithm.

Once we obtain the encoded multipliers, the next thing we need to do is perform add / subtract / shift to generate the partial product which will be sended to Wallace tree. To achieve this, a moudule named *booth_selector* is designed, which generates a single bit of partial product. It all depends on the bits of multiplicands and the encoded multipliers.

The theory is boring, let's take an example!

The following example is the singed 13 times 6:

$$\begin{array}{lll}
 & 0\;0\;0\;0\;1\;1\;0\;1 & \\
\text{x} & 0\;\underline{0}\;0\;\underline{0}\;0\;\underline{1}\;1\;\underline{0}\;\;0 & \\
\hline
 & 1\;1\;1\;0\;0\;1\;0\;1^{+1} & (100)\ \text{-2X} \\
 & 0\;1\;1\;0\;1\;0 & (011)\ \text{+2X} \\
 & 0\;0\;0\;0 & (000)\ \text{null} \\
\hline
 & 0\;0 & (000)\ \text{null} \\
\hline
1 & 0\;1\;0\;0\;1\;1\;1\;0 &
\end{array}$$

Step1: Add trailing zero to multiplier 00000110(0);

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

Step2: Take 100 for booth encoding (the y_i s are all underlined), 100 means -2X, where z0=0, z1=1, neg=1;

Step3: Generate the 1st partial product, the multiplicand is shifted by 1, or in other words, the partial product use the lower 1 bits of multiplicand (it's the method implemented in this design), and obtain 00011010, and due to the neg sign is asserted, the partial product is inversed to 11100101, and there is required also plus 1 (neg), the neg value is stored into an register *n*, and sent to a ripple carry adder which generates the partial product with 2's complements.

```
        RCA #(65) u_rca(
            .a    (pp[u]),
            .b    ({64'd0,n[u]}),
            .cin  (1'b0),
            .sum  (pp2c[u]),
            .cout ()
        );
```

Step4: Inspect the next three bits, 011 measn +2X, no 2's complement transform here.

Step5: Go futher, and the next two partial multipliers are 000 and 000, which means no actions required.

Step6: Add the partial products, and we obtain 01001110 (8'd78)

You may notice that there's a solid line between 0000 and 00, that's what will talk in the next section.

The Booth encoding reduced the counts of addition operation from N into N/2, half of the addition is replaced by simple bit shift or NOP (null operation).

Wallace Tree[5]

Back to the example in the last section, the solid line is placed for Wallace tree multiplication. The Wallace tree takes every 3 inputs of the partial products of Booth algorithm, and outputs 2 result for the next stage. A kind of 32bits Wallace tree is illustrated in Figure 2.8, go on take this example, 11100101 is sent to the most right CSA, so was 01101000 and 00000000 (the 3rd partial product), 00000000 (the 4th partial product under the solid line) is sent to the right 2th CSA unit.

[5] 计算机体系结构基础, 胡伟武: P215

$$\begin{array}{ccccccccccl}
 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & & \\
\mathrm{x} & 0 & \underline{0} & 0 & \underline{0} & 0 & \underline{1} & 1 & \underline{0} & 0 & \\
\hline
 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & {\scriptstyle +1} & \text{(100) -2X} \\
 & 0 & 1 & 1 & 0 & 1 & 0 & & & & \text{(011) +2X} \\
 & 0 & 0 & 0 & 0 & & & & & & \text{(000) null} \\
\hline
 & 0 & 0 & & & & & & & & \text{(000) null} \\
\hline
1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & &
\end{array}$$

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

level	isolated	remain	in	out
0	0	2	10*3	10*2
1	1	1	7*3	7*2
2	1	0	5*3	5*2
3	0	1	3*3	3*2
4	0	1	2*3	2*2
5	0	2	1*3	1*2
6	1	1	1*3	1*2
7	1	0	1*3	1*2

The Wallace tree for multiplication has 3 inputs a, b, cin, and 2 outputs sum, cout. It use CSA (Carray Save Adder) instead of RCA (Ripple Carry Adder), where the carray is saved for futher addition, not rippled to the higher bits, it saves the latency.

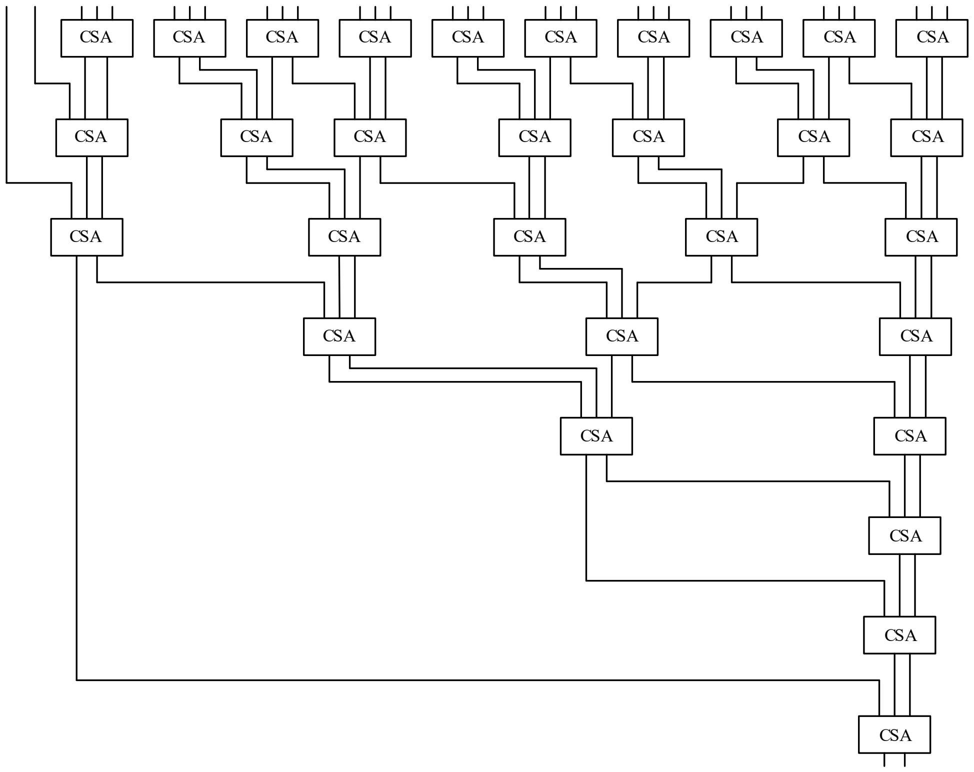


Figure 2.8 A 32bit-Wallace tree

The Wallace tree for this design is illustrated above, 3 registers are inserted into the output of second stage(reserve enough setup slack for booth encoder), fifth stage, and the last stage for pipelining, if the timing is not met, the pipelining states should be increased, I don't have a clear idea about that currently.

Full Adder

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

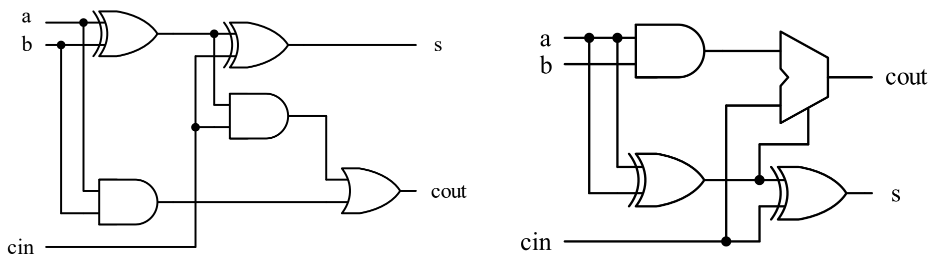


Figure 2.9 Full adder using logic gates

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

$\overline{a \oplus b} = a \cdot b + \bar{a} \cdot \bar{b}$, 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

Overview

This part mainly refers to *Computer arithmetic algorithms*, Koren, Israel.

SRT division is a kind of improved non-restoring division algorithm, which is a slow division method, say, generate only 1-bit or several fixed bits in an iteration. The slow division method use the reccurence formula for generating residual remainder, adjusting the quotient bit to make the reccurence converge.

I will cut into this section from Restoring Dvision which is the basic reccuerence division, then talk about Non-restoring Division, then talk about the Radix-2 SRT Division, and finally introduce the Radix-4 SRT Division which is adopted as the divisior module of the CPU.

Before the beginning of this section, it's necessary to introduce the basic concept and the recurrence formula.

We perform the following division,

$$X/D = (Q, R)$$

where, X is the dividend, D is the divisor, Q is the quotient, R is the Remainder. And we define q_i, r_i as the i-th most significant quotient digit, and residual remainder (procedure remainder), separately.

The reccurence formula is defined as follows:

$$r_i = \beta r_{i-1} - q_i \cdot D$$

where, β is the base, which is 2 in radix-2 algorithm and becomes to 4 in radix-4 algorithm.

Restoring Division[6]

The final r_i is the remainder we want, and the initial r_i is X, consider $\beta = 2$,

[6] https://en.wikipedia.org/wiki/Division_algorithm

$$R = 2r_{n-1} - q_n \cdot D = 2^2 r_{n-2} - 2^1 q_{n-1} D - q_n D = \cdots = 2^n X - QD$$

Before the recurrence, we must shift left D for n bits, and after the recurrence, we should also shift right R for n bits, this is a very important key point which will be talked further in the SRT division algorithm.

The algorithm is described as follows:

```
R := X
D := D << n              -- R and D need twice the word width of N and Q
for i := n - 1 .. 0 do   -- For example 31..0 for 32 bits
  R := 2 * R - D           -- Trial subtraction from shifted value
(multiplication by 2 is a shift in binary representation)
  if R >= 0 then
    q(i) := 1            -- Result-bit 1
  else
    q(i) := 0            -- Result-bit 0
    R := R + D           -- New partial remainder is (restored) shifted value
  end
end
-- Where: X = dividend, D = divisor, n = #bits, R = partial remainder, q(i) =
bit #i of quotient from MSB
```

The selection of quotient digit can be described as follows, which has the digit set of {0,1}:

$$q_i = \begin{cases} 1 & , if\ 2r_{i-1} \geq D \\ 0 & , if\ 2r_{i-1} < D \end{cases}$$

Examine the following example which calculate 13/5=(2,3):

r0		0 0 0 0 1 1 0 1		
D		0 1 0 1 0 0 0 0		
r1	2r0	0 0 0 1 1 0 1 0	<D	q0=0
r2	2r1	0 0 1 1 0 1 0 0	<D	q1=0
	2r2	0 1 1 0 1 0 0 0	>=D	q2=1
r3	-D	0 0 0 1 1 0 0 0		
r4	2r3	0 0 1 1 0 0 0 0	<D	q3=0
R		0 0 0 0 0 0 1 1		

Thus, we obtain Q=0010, and R=0110, that's the correct result. Note that the MSB of quotient is first produced.

The term restoring means that we should restore residual remainder if q equals to zero.

Non-restoring Division

The Non-restoring Division uses the digit set {-1,1} instead of {0,1}, the advantage is that when implemented in hardware there is only one decision and addition/subtraction per quotient bit, no restoring step is required. The basic radix 2 Non-restoring Division algorithm is:

```
R := X
D := D << n              -- R and D need twice the word width of N and Q
for i = n - 1 .. 0 do    -- for example 31..0 for 32 bits
  if R >= 0 then
    q(i) := +1
    R := 2 * R - D
  else
    q(i) := -1
    R := 2 * R + D
  end if
end
-- Note: X=dividend, D=divisor, n=#bits, R=partial remainder, q(i)=bit #i of
quotient from MSB.
```

The selection of quotient digit can be described as follows, which has the digit set of {-1,1}:

$$q_i = \begin{cases} 1 \ , if \ 2r_{i-1} \geq D \\ -1 \ , if \ 2r_{i-1} < D \end{cases}$$

Following this algorithm, the quotient is in a non-standard form consisting of digits of −1 and +1. This form needs to be converted to binary to form the final quotient. Example:

0. Start: $Q = 111\bar{1}1\bar{1}1\bar{1}$

1. Form the positive terms: $P = 11101010$

2. Mask the negative terms: $M = 00010101$

3. Subtract: P-M=Q=11010101

The quotient computed by this algorithm is always odd, and the remainder is in the range $-D \leq R \leq D$, for example 5/2=(3,-1). Sometimes, we should convert the remainder to positive, just add D to R, and subtract 1 from Q (for unsigned algorithm).

Radix-2 SRT Division

The SRT Division algorithm allows 0 to be the quotient digit for which no add/subtract operation is needed, the digit set become {-1,0,1}. The quotient digit selecting rule is changed into:

$$q_i = \begin{cases} 1 \ , if \ 2r_{i-1} \geq D \\ 0 \ , if \ -D \leq 2r_{i-1} < D \\ -1 \ , if \ 2r_{i-1} < -D \end{cases}$$

In the Restoring and Non-restoring algorithm, the total iteration is N, the bit width of the dividend. But the case in SRT algorithm is different, we do not shift N bits for the initial residual remainder, most of the iteration is pointless.

In the SRT algorithm, the fist step is to normalize Dividend and Divisor, which accelerates the reccurence process, the total iteration stages is determined by this step.

Let's consider an example, which calculates 11/3=(3,2), both dividend and divisor are of 8 bits, 8'd11=00001011, 8'd3=00000011.

```
r0    0. 0 0 1 0 1 1 0    n= 1
D     0. 1 1 0 0 0 0 0    m= 5
```

There is 4 leading zeros of the dividend, 11, and 6 leading zeros of the divisor. We just shift 5 bits for divisor to obtain 0.110000, and shift 1 bit for the dividend and obtain 0.0010110. The total iteration is 5-1=4, which is the shift bits of divisor subtracted by the shift bits of dividend. If the previous one is smaller than the later one, it means that the dividend is smaller than the divisor, no iteration is needed.

After normalized, the comparator will be more simply, full width comparator is not needed any more. That's because we constrain D to [1/2,1), which is 0.1xxxx, and 1.0xxxx(neg), thus, only 2 bits are required to compare. In some cases, for example, the dividend X is larger than 1/2, the shifted remainder has an extra integer bit, thus, 3 bits are required.

```
r0        0. 0 0 1 0 1 1 0    n= 1
D         0. 1 1 0 0 0 0 0    m= 5
-----------------------------------------
r1  2r0   0. 0 1 0 1 1 0 0    <1/2    q1=0
```

The second step is the reccurence, recall the recurrence formula: $r_i = 2r_{i-1} - q_i \cdot D$. The iteration steps is illustrated following.

```
r0        0. 0 0 1 0 1 1 0    n= 1
D         0. 1 1 0 0 0 0 0    m= 5
-----------------------------------------
r1  2r0   0. 0 1 0 1 1 0 0    <1/2    q1=0
    2r1   0. 1 0 1 1 0 0 0    >=1/2   q2=1
r2  -D    1. 1 1 1 1 0 0 0
r3  2r2   1. 1 1 1 0 0 0 0    >=-1/2  q3=0
r4  2r3   1. 1 1 0 0 0 0 0    >=-1/2  q4=0
```

The quotient has no -1 as its digit, so there is no conversion to the standard form. Notice that the remainder is negative, which is not what we want, so we plus D to r4, and subtract 1 from quotient:

r0		0. 0 0 1 0 1 1 0	n= 1	
D		0. 1 1 0 0 0 0 0	m= 5	
r1	2r0	0. 0 1 0 1 1 0 0	<1/2	q1=0
	2r1	0. 1 0 1 1 0 0 0	>=1/2	q2=1
r2	-D	1. 1 1 1 1 0 0 0		
r3	2r2	1. 1 1 1 0 0 0 0	>=-1/2	q3=0
r4	2r3	1. 1 1 0 0 0 0 0	>=-1/2	q4=0
R	+D	0. 1 0 0 0 0 0 0	q'=0100-1=0011	

Do not forget shift right R for m-bits, which is 5 in this example, and we obtain 00000010 (8'd2) as the final remainder, 00000011 (8'd3) as the final quotient.

As for the signed case, the quotient is selected by the following rule:

$$q_i = \begin{cases} 0 \quad if\ |2r_{i-1}| < 1/2 \\ 1 \quad if\ |2r_{i-1}| \geq 1/2\ ,\ \ r_{i-1}\ and\ D\ have\ the\ same\ sign. \\ -1 \quad if\ |2r_{i-1}| \geq 1/2,\ \ r_{i-1}\ and\ D\ have\ opposite\ sign. \end{cases}$$

Radix-4 SRT Division

Radix-4 SRT Division is similar to radix-2, the only several differences are:

1. Normalization.

2. The selection of quotient digit.

3. On-the-fly conversion.

The radix-4 SRT use every 2 bits for iteration, the iteration is also determined by the shifted bits. Use m to denote the shifted bits for the divisor, and n for the dividend. In radix-2, the total iteration is (m-n), but for radix-4, the total iteration is (m-n)/2. Besides, m and n must be both odd or both even or there will be unpredictable events.

Assume the dividend is 23, and the divisor is 5, which is an unsigned case, the shift will be:

r0	0 0. 0 0 1 0 1 1 1	n= 0
D	0 0. 1 0 1 0 0 0 0	m= 4

Where, n=0, it's different from the case in radix-2, n will be 1 in radix-2. The residual remainder should be sign expanded, if the previous residual remainder r_{i-1} is larger than 1/4, which is 0.01xxxx, then $4r_{i-1}$ will be 1.xxxx, the digit 1 masks the sign bit.

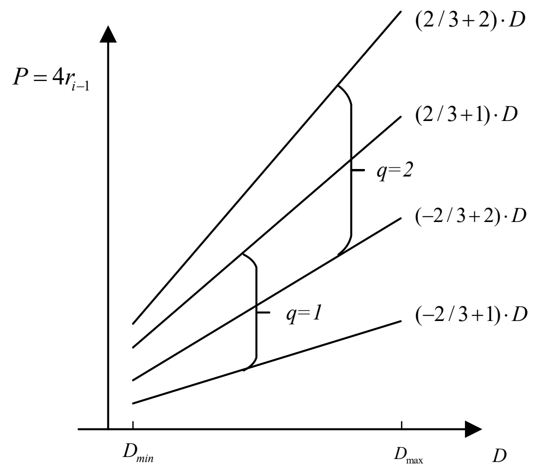
The quotient digit selection is more complex than the radix-2 algorithm, a large QDS(quotient digit selection) table is used to determine the quotient digit. Let's talk about the selection rules, and build the QDS table for implementation.

The digit set we choose for Radix-4 SRT division is {-2,-1,0,1,2}, which means that redundancy is k=2/(4-1)=2/3. k is the redundancy factor, which reduce the size of allowable region for the partial remainder. The partial remainder is limited to $-k \cdot D \leq r_i \leq k \cdot D$.

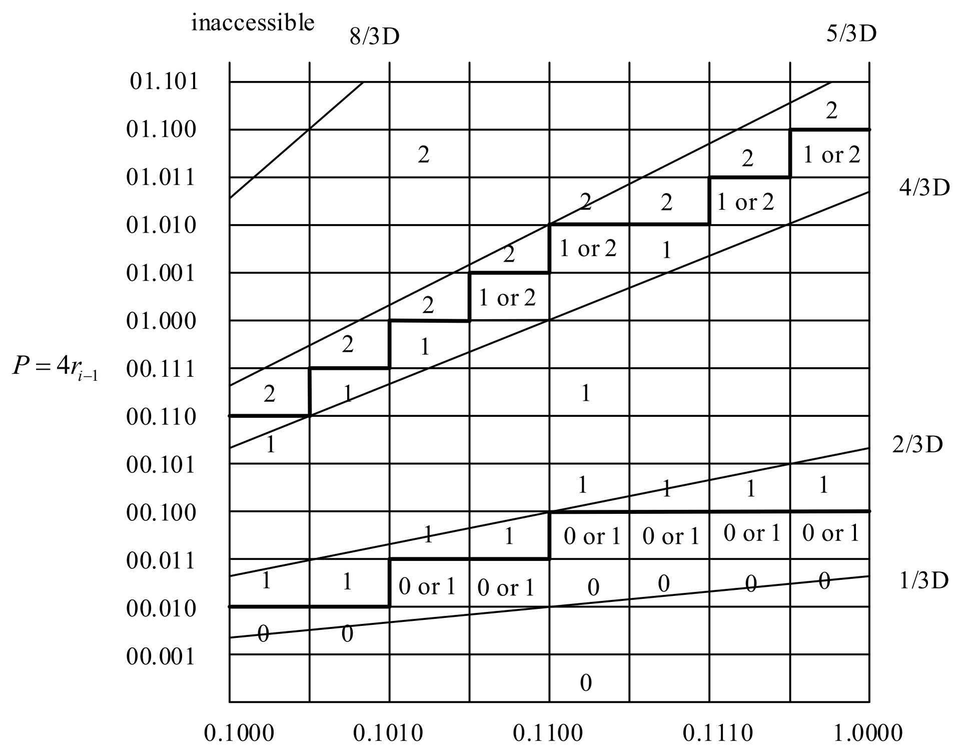
We use P to denote the previous partial remainder $4r_{i-1}$, $P = r_i + q \cdot D$, thus,

$$(-k + q) \cdot D \leq P \leq (k + q) \cdot D$$

Plot the relationship between P and D, the graph is called P-D plot, the x and y coordinates are both inputs to the QDS table, the quotient digit is the only output.



We can construct such a QDS table as follow. Thanks to the normalization, we can only examine only several bits to determine the quotient digits!



In this PD plot, the quotient digit is selected associated with the coordinate point that is on its lower-left, for example, if D=0.1010, and P=00.011, we should choose 1 as quotient digit, however if D=0.1010, and P=00.010, we will choose 0.

The comment that 0or1, or 1or2 means that the quotient digit can be selected as either 0 or 1, the bold solid line is one kind of selection scheme, which allows more 0 and 1, less 2, thus simplified the operation.

Once we obtain the quotient digit, we can keep iterating.

```
r0        0 0. 0 0 1 0 1 1 1   n= 0
D         0 0. 1 0 1 0 0 0 0   m= 4
-----------------------------------------
     4r0  0 0. 1 0 1 1 1 0 0        q1= 1
r1  -D    0 0. 0 0 0 1 1 0 0
     4r1  0 0. 0 1 1 0 0 0 0        q2= 1
r2  -D    1 1. 1 1 0 0 0 0 0
-----------------------------------------
R   +D    0 0. 0 1 1 0 0 0 0  q'= 0101-1=0100
```

The quotient is selected from the QDS table instead of comparation to 1/2. The first reccurence, the index of P is 00.101, and index of D is 0.1010, so we choose 1 as the quotient digit and thus subtract D from 4r0 to obtain r1.

Conver the quotient to normal form:

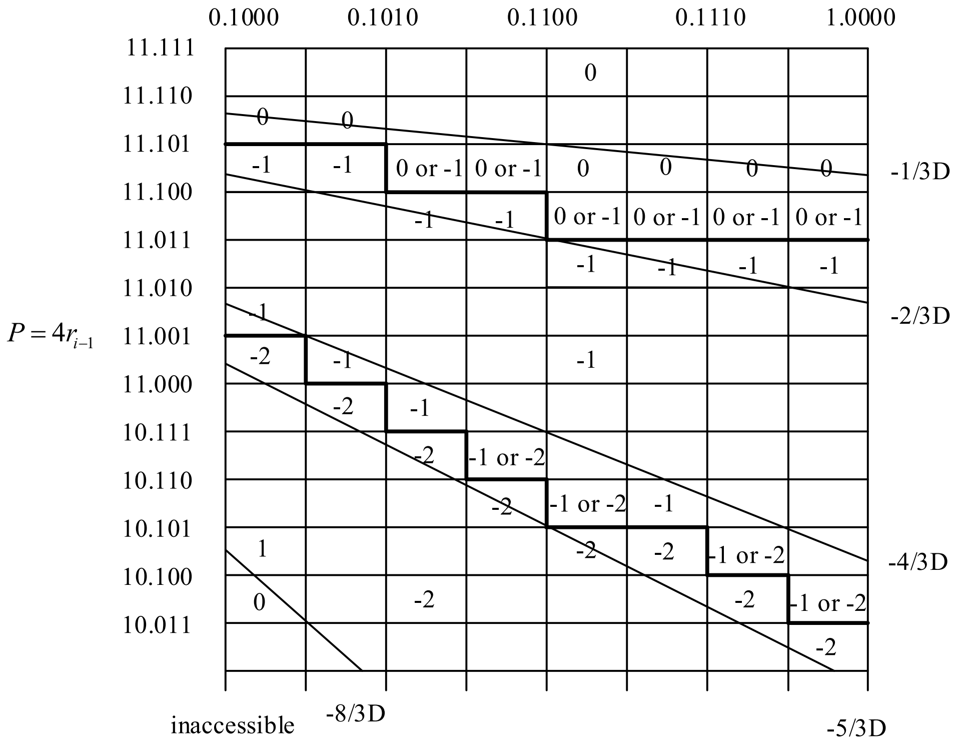
11 in radix-4 is 0101 in radix-2, there is no negative items, no subtraction is needed.

The remainder r2 is negative, so we should plus D to it, and we obtain 00.0110000 as final remainder, 0100 as final quotient. The final remainder 00.0110000 should shift right m=4, and we obtain 00000011=3. The equation is 23/5=(4,3).

Is this section finished? We haven't discuss the negative case. Consider 23/-5=(-4,3),

```
r0       0 0. 0 0 1 0 1 1 1     n= 0
D        1 1. 0 1 1 0 0 0 0     m= 4
--------------------------------------
r1  4r0  0 0. 1 0 1 1 1 1 0 0        q1= ?
```

How do we select quotient digit? We should first convert D into positice, which is 0.1010, and then use the negative QDS table.



Instead of with its lower-left, the upper-left coordinate that quotient digit is associated, for example, if D=0.1010, and P=11.101, we should choose -1 as quotient digit, in the positive PD plot, if D=0.1010, and P=00.011, we choose 1.

The following example is 23/-5=(-4,3), the process is similar to the unsigned SRT.

```
D'       0 0. 1 0 1 0 0 0 0
r0       0 0. 0 0 1 0 1 1 1 1    n= 0
D        1 1. 0 1 1 0 0 0 0 0    m= 4
-------------------------------------------
    4r0  0 0. 1 0 1 1 1 1 0 0          q1= -1
r1   +D  0 0. 0 0 0 1 1 1 0 0
    4r1  0 0. 0 1 1 1 0 0 0 0          q2= -1
r2   +D  1 1. 1 1 0 0 0 0 0 0
-------------------------------------------
R    -D  0 0. 0 1 1 0 0 0 0 0    q'= -5+1=-4
```

Note that, due to the divisor is negative, q should plus 1, and R should subtract D(which is negative).

And the example for -23/5=(-5,2):

```
r0       1 1. 1 1 0 1 0 0 1      n= 0
D        0 0. 1 0 1 0 0 0 0      m= 4
-------------------------------------------
    4r0  1 1. 0 1 0 0 1 0 0          q1= -1
r1   +D  1 1. 1 1 1 0 1 0 0
r2  4r1  1 1. 1 0 1 0 0 0 0          q2= 0
     +D  0 0. 0 1 0 0 0 0 0
-------------------------------------------
R        0 0. 0 1 0 0 0 0 0      q'= -4-1=-5
```

Note that, the sign of remainder has no cleart definition, for example, -29/-26, Matlab give -3 as remainder, which keeps the remainder the same sign of dividend, while in python, it keeps the remainder the same sign of divisor, and in some case, the remainder is set to be positive. In the implementation, the remainder is always positive, which will give the result 23, -26*2+23=-29.

On-the-fly Conversion[7]

The redundant digit set has negative digit, and the quotient is not represent in standard form, we should convert the non-standard form quotient to the standard one at the final step. But it will cost additional cycles and chip area: for the subtraction, a full width carry adder is required, the latency will be as large as N if ripple carray adder is applied, rr we can sacrifice chip area and use a carry look-ahead adder.

On-the-fly conversion is designed to obtain the real-time conversion result. It uses only 2 Flip-flop and several combination logic units.

[7] Ercegovac, and Lang. "On-the-fly conversion of redundant into conventional representations." IEEE Transactions on Computers 100.7 (1987): 895-897.

The real Q can be represented as: $Q[j] = \sum_{i=1}^{j} q_i r^{-i}$, and the updata equation: $Q[j+1] = Q[j] + q_{j+1} r^{-(j+1)}$, where, $Q[j]$ means the real Q of the j-th iteration, q_i is the quotient digit, which is the redundatnt representation. Since q_{j+1} can be negative:

$$Q[j+1] = \begin{cases} Q[j] + q_{j+1} r^{-(j+1)} & , \ if\ q_{j+1} \geq 0 \\ Q[j] - r^{-j} + (r - |q_{j+1}|) r^{-(j+1)}, & if\ q_{j+1} < 0 \end{cases}$$

It has the disadvantage that a subtraction is required, the propagation from a carry is too slow, thus we define another register $QM[j] = Q[j] - r^{-j}$, where QM means *Quotient of Minus*.

$$Q[j+1] = \begin{cases} Q[j] + q_{j+1} r^{-(j+1)} & , \ if\ q_{j+1} \geq 0 \\ QM[j] + (r - |q_{j+1}|) r^{-(j+1)}, & if\ q_{j+1} < 0 \end{cases}$$

Subtraction is replaced by sampling the register QM, the QM can be updated by:

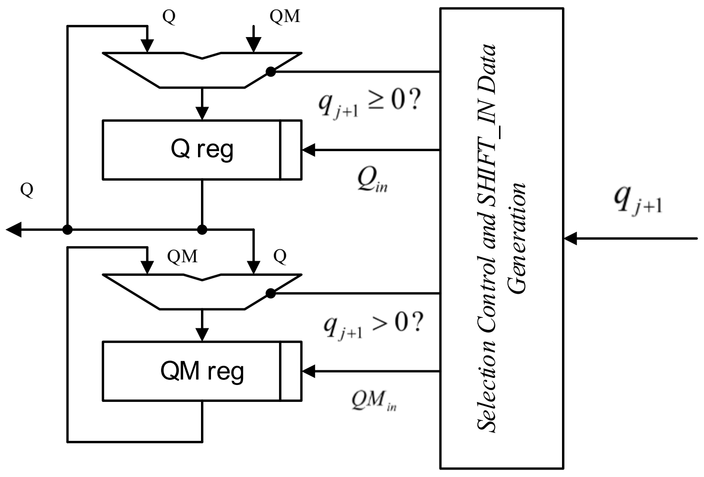
$$QM[j+1] = \begin{cases} Q[j] + (q_{j+1} - 1) r^{-(j+1)} & , \ if\ q_{j+1} > 0 \\ QM[j] + \left((r-1) - |q_{j+1}|\right) r^{-(j+1)}, & if\ q_{j+1} \leq 0 \end{cases}$$

The item $r^{-(j+1)}$ can be implemented by concatenating q_{j+1} behind the register Q or QM. Thus, we obtain the On-the-fly Conversion updating formula:

$$Q[j+1] = \begin{cases} \{Q[j], q_{j+1}\} & , \ if\ q_{j+1} \geq 0 \\ \{QM[j], (r - |q_{j+1}|)\}, & if\ q_{j+1} \leq 0 \end{cases}$$

$$QM[j+1] = \begin{cases} \{Q[j], q_{j+1} - 1\} & , \ if\ q_{j+1} > 0 \\ \{QM[j], ((r-1) - |q_{j+1}|)\}, & if\ q_{j+1} \leq 0 \end{cases}$$

The hardware implementation flow is described as follows:



The initialization condition is:

$$Q = QM = \begin{cases} all\ 0s, & if\ quotient\ positive \\ all\ 1s, & if\ quotient\ negative \end{cases}$$

We can simply deduce the sign of quotient just by the sign of dividend and divisor: if the dividend and divisor has the opposite sign bit, we just initialize Q and QM to all 1s, or initialize them to all 0s.

The generation data of SHIFT_IN can be deduced simply from the truth table, for the Radix-4 case (Radix-2 is much simple):

$$Q[j+1] = \begin{cases} \{Q[j], q\} & , \ if\ q \geq 0 \\ \{QM[j], 1'b1, q[0]\}, & if\ q \leq 0 \end{cases}$$

$$QM[j+1] = \begin{cases} \{Q[j], 1'b0, q[1]\}, & if\ q > 0 \\ \{QM[j], \sim q\} & , \ if\ q \leq 0 \end{cases}$$

There is also a Radix-2 example for the conversion, it convert 1101(-1)00 into 1100100, which is 1101000-0000100.

j	q_j	***Q[j]***	***QM[j]***
0		0	0
1	1	0.1	0.0
2	1	0.11	0.10
3	0	0.110	0.101
4	1	0.1101	0.1100
5	-1	0.11001	0.11000
6	0	0.110010	0.110001
7	0	0.1100100	0.1100011

That's all for SRT division.

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 Logical Immediate.	YES
	9	SRLI	Shift Right Logical Immediate.	YES
	10	SRAI	Shift Right Arithmetic Immediate.	YES
	11	ADDIW	Add Immediate Word.	YES
	12	SLLIW	Shift Left Logical Immediate Word.	YES
	13	SRLIW	Shift Right Logical Immediate Word.	YES

	14	SRAIW	Shift Right Arithmetic Immediate Word.	YES
S-type	15	SD	Store Doubleword.	YES

RV32M Standard Extension

Total: 8

Type	**Order**	**Instruction**	**Description**	**Compatibility**
R-type	1	MUL	Multiply.	
	2	MULH	Multiply High.	
	3	MULHSU	Multiply High Signed Unsigned.	
	4	MULHU	Multiply High Unsigned.	
	5	DIV	Divide.	
	6	DIVU	Divide Unsigned.	
	7	REM	Remainder.	
	8	REMU	Remainder Unsigned.	

RV64M Standard Extension (in addition to RV32M)

Total: 5

Type	**Order**	**Instruction**	**Description**	**Compatibility**
R-type	1	MULW	Multiply Word.	
	2	DIVW	Divide Word.	
	3	DIVUW	Divide Unsigned Word.	
	4	REMW	Remainder Word.	
	5	REMUW	Remainder Unsigned Word.	

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
```