

# Notes

Figure out a title for this chapter . . . . .	3
Rewrite this section . . . . .	3
Chapter sections are subject to change in name and order . . . . .	5
Chapter sections are subject to change in name and order . . . . .	6
Chapter sections are subject to change in name and order . . . . .	7
make a better title . . . . .	7
figure out a name for this subsection . . . . .	7
Need to figure out more sections to explain whole datapath . . . . .	7
figure out better naming for sections . . . . .	7

# Implementation of RISC-V in SME

Daniel Ramyar

December 26, 2019

# Contents

<b>1 Placeholder</b>	<b>3</b>
1.1 Communicating Sequential Processes . . . . .	3
1.2 Synchronous Message Exchange . . . . .	4
<b>2 Logic Design</b>	<b>5</b>
2.1 Boolean algebra . . . . .	5
2.1.1 Logic equations . . . . .	5
2.1.2 Truth tables . . . . .	5
2.1.3 Gates . . . . .	5
2.2 Combinational logic . . . . .	5
2.2.1 Decoder . . . . .	5
2.2.2 Multiplexor . . . . .	5
2.2.3 Two-level logic . . . . .	5
2.2.4 Programmable logic array . . . . .	5
<b>3 Introduction to RISC-V instructions</b>	<b>6</b>
3.1 RISC-V Assembly . . . . .	6
3.2 Operands . . . . .	6
3.2.1 Register . . . . .	6
3.2.2 Memory Format . . . . .	6
3.2.3 Const vs imm . . . . .	6
3.3 Numeral system of a computer . . . . .	6
3.3.1 base 2 . . . . .	6
3.3.2 signed unsigned . . . . .	6
3.4 Instruction representation in binary . . . . .	6
3.5 Operators . . . . .	6
<b>4 The RISC-V processor</b>	<b>7</b>
4.1 Single Cycle RISC-V Units . . . . .	7
4.1.1 Program Counter . . . . .	7
4.1.2 Instruction Memory . . . . .	7
4.1.3 incrementor? . . . . .	7
4.1.4 Register . . . . .	7
4.1.5 Arithmetic Logic Unit (ALU) . . . . .	7
4.1.6 Immediate generator . . . . .	7
4.1.7 Data Memory . . . . .	7
4.2 Designing the Control . . . . .	7
4.3 Single Cycle RISC-V datapath . . . . .	7
4.4 Improving the datapath . . . . .	7
4.4.1 RV64I Base Instructions Support . . . . .	8
4.4.2 Supporting R-Format . . . . .	8
4.4.3 Supporting I-Format . . . . .	8
4.4.4 Supporting S-Format . . . . .	8

4.4.5	Supporting B-Format . . . . .	8
4.4.6	Supporting U-Format . . . . .	8
4.4.7	Supporting J-Format . . . . .	8
4.5	Debugging the instructions . . . . .	8
4.5.1	Writing assembly to test instructions . . . . .	8
4.5.2	Writing simple C code to run on RISC-V . . . . .	8

# Chapter 1

## Placeholder

---

### 1.1 Communicating Sequential Processes

---

The problem with multiprocessor workloads is the sharing of memory. This creates a whole slew of problems. There are many different processes going on at once all having access to the same memory. Unless you got superpowers it is very hard to determine where in the program something goes wrong. It all boils down to the non-determinism.

For example if you are going to print multiple strings using multiple threads you don't know which string is going to be printed first it's gonna depend on the operating system not on anything in your code. That can create race conditions (meaning the behaviour in your code is dependent on the timing of different threads) which can cause unpredictable behaviour and therefore bugs which is undesirable.

This has been tried to be solved with mutexes or locks but this also has its downside in the form of deadlocks where multiple processes are waiting for each other and because these processes are non-deterministic it is very hard to reproduce errors in your code which in turn makes it hard to debug and therefore hard to make reliable software.

This is where Communicating Sequential Processes (CSP) comes in. CSP was an algebra first proposed by Hoare [1]. CSP is built on two very basic primitives one is the process (which should not be confused with operating system processes) which could be an ordered sequence of operations. These processes do not share any memory so one process cannot access a specific value in another process (which solves a lot of the problems we had with shared memory).

The other primitive is channels which is the way the processes communicate with each other. You can pass whatever you want through these channels and once you pass a value you lose access to it.

There is a lot of ways the processes and channels can be arranged the most simple one

Figure out a title for this chapter

Rewrite this section

can be found in figure 1.1 which illustrates process 1 which passes a value onto a channel which process 2 takes as input. Some different configurations can be found in figures 1.2-1.4

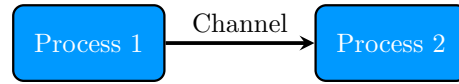


Figure 1.1: CSP one to one

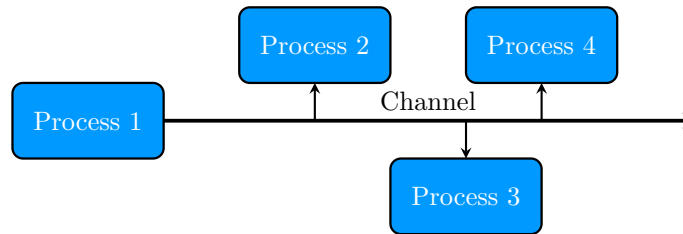


Figure 1.2: CSP one to many

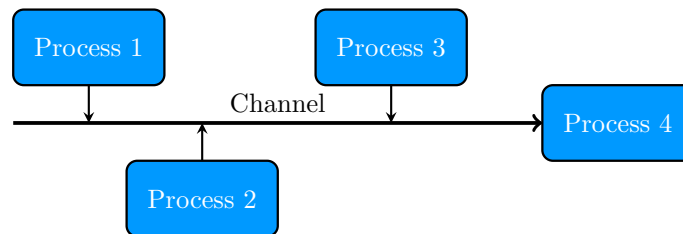


Figure 1.3: CSP many to one

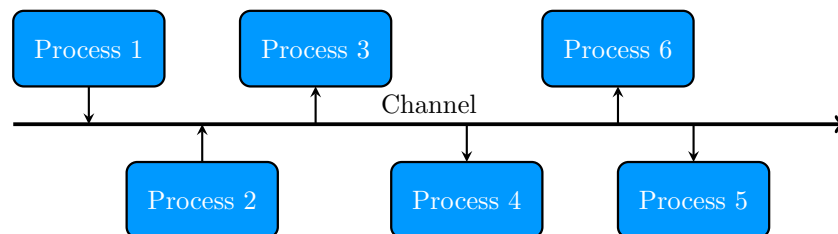


Figure 1.4: CSP many to many

## 1.2 Synchronous Message Exchange

Vinter and Skovhede [3] Vinter and Skovhede [4]

# Chapter 2

## Logic Design

---

This chapter aims to introduce the reader to the basics of logic design. Based on Appendix A in [2]

Chapter sections are subject to change in name and order

### 2.1 Boolean algebra

lorem ipsum

#### 2.1.1 Logic equations

#### 2.1.2 Truth tables

#### 2.1.3 Gates

### 2.2 Combinational logic

#### 2.2.1 Decoder

#### 2.2.2 Multiplexor

#### 2.2.3 Two-level logic

#### 2.2.4 Programmable logic array

## Chapter 3

# Introduction to RISC-V instructions

---

This chapter aims to introduce the reader to the basics of machine language. Based on chapter 2 in [2]

Chapter sections are subject to change in name and order

### 3.1 RISC-V Assembly

### 3.2 Operands

#### 3.2.1 Register

#### 3.2.2 Memory Format

#### 3.2.3 Const vs imm

### 3.3 Numeral system of a computer

#### 3.3.1 base 2

#### 3.3.2 signed unsigned

### 3.4 Instruction representation in binary

### 3.5 Operators



# Chapter 4

## The RISC-V processor

This chapter aims to introduce the reader to the basics of machine language. Based on chapter 4 in [2]

Chapter sections are subject to change in name and order

### 4.1 Single Cycle RISC-V Units

make a better title

#### 4.1.1 Program Counter

#### 4.1.2 Instruction Memory

#### 4.1.3 incrementor?

figure out a name for this subsection

#### 4.1.4 Register

#### 4.1.5 Arithmetic Logic Unit (ALU)

#### 4.1.6 Immediate generator

#### 4.1.7 Data Memory

Need to figure out more sections to explain whole datapath

### 4.2 Designing the Control

### 4.3 Single Cycle RISC-V datapath

### 4.4 Improving the datapath

figure out better naming for sections

#### 4.4.1 RV64I Base Instructions Support

#### 4.4.2 Supporting R-Format

#### 4.4.3 Supporting I-Format

#### 4.4.4 Supporting S-Format

#### 4.4.5 Supporting B-Format

#### 4.4.6 Supporting U-Format

#### 4.4.7 Supporting J-Format

### 4.5 Debugging the instructions

#### 4.5.1 Writing assembly to test instructions

#### 4.5.2 Writing simple C code to run on RISC-V

# Risc V Reference Card

## Instruction Formats

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

\* This is a special case of the RV64I I-type format used by slli, srli and srai instructions where the lower 6 bits in the immediate are used to determine the shift amount (shamt). If slliw, srlw and sraiw are used it should generate an error if  $\text{imm}[6] \neq 0$

## RV64I Base Instructions

Name	Fmt	Opcode	Funct3	Funct7/ imm[11:5]	Assembly	Description (in C)
Add	R	0110011	000	0000000	add rd, rs1, rs2	rd = rs1 + rs2
Subtract	R	0110011	000	0100000	sub rd, rs1, rs2	rd = rs1 - rs2
AND	R	0110011	111	0000000	and rd, rs1, rs2	rd = rs1 & rs2
OR	R	0110011	110	0000000	or rd, rs1, rs2	rd = rs1   rs2
XOR	R	0110011	100	0000000	xor rd, rs1, rs2	rd = rs1 ^ rs2
Shift Left Logical	R	0110011	001	0000000	sll rd, rs1, rs2	rd = rs1 << rs2
Set Less Than	R	0110011	010	0000000	slt rd, rs1, rs2	rd = (rs1 < rs2)?1:0
Set Less Than (U)*	R	0110011	011	0000000	sltu rd, rs1, rs2	rd = (rs1 < rs2)?1:0
Shift Right Logical	R	0110011	101	0000000	srl rd, rs1, rs2	rd = rs1 >> rs2
Shift Right Arithmetic†	R	0110011	101	0100000	sra rd, rs1, rs2	rd = rs1 >> rs2
Add Word	R	0111011	000	0000000	addw rd, rs1, rs2	rd = rs1 + rs2
Subtract Word	R	0111011	000	0100000	subw rd, rs1, rs2	rd = rs1 - rs2
Shift Left Logical Word	R	0111011	001	0000000	sllw rd, rs1, rs2	rd = rs1 << rs2
Shift Right Logical Word	R	0111011	101	0000000	srlw rd, rs1, rs2	rd = rs1 >> rs2
Shift Right Arithmetic Word†	R	0111011	101	0100000	sraw rd, rs1, rs2	rd = rs1 >> rs2
Add Immediate	I	0010011	000		addi rd, rs1, imm	rd = rs1 + imm
AND Immediate	I	0010011	111		andi rd, rs1, imm	rd = rs1 & imm
OR Immediate	I	0010011	110		ori rd, rs1, imm	rd = rs1   imm
XOR Immediate	I	0010011	100		xori rd, rs1, imm	rd = rs1 ^ imm
Shift Left Logical Immediate	I	0010011	001	0000000	slli rd, rs1, shamt	rd = rs1 << shamt
Shift Right Logical Immediate	I	0010011	101	0000000	srlw rd, rs1, shamt	rd = rs1 >> shamt
Shift Right Arithmetic Immediate†	I	0010011	101	0100000	sraiw rd, rs1, shamt	rd = rs1 >> shamt
Set Less Than Immediate	I	0010011	010		slti rd, rs1, imm	rd = (rs1 < imm)?1:0
Set Less Than Immediate (U)*	I	0010011	011		sltiu rd, rs1, imm	rd = (rs1 < imm)?1:0
Add Immediate Word	I	0011011	000		addiw rd, rs1, imm	rd = rs1 + imm
Shift Left Logical Immediate Word	I	0011011	001	0000000	slliw rd, rs1, shamt	rd = rs1 << shamt
Shift Right Logical Immediate Word	I	0011011	101	0000000	srlw rd, rs1, shamt	rd = rs1 >> shamt
Shift Right Arithmetic Imm Word†	I	0011011	101	0100000	sraiw rd, rs1, shamt	rd = rs1 >> shamt
Load Byte	I	0000011	000		lb rd, rs1, imm	rd = M[rs1+imm][0:7]
Load Half	I	0000011	001		lh rd, rs1, imm	rd = M[rs1+imm][0:15]
Load Word	I	0000011	010		lw rd, rs1, imm	rd = M[rs1+imm][0:31]
Load Doubleword	I	0000011	011		ld rd, rs1, imm	rd = M[rs1+imm][0:63]
Load Byte (U)*	I	0000011	100		lbu rd, rs1, imm	rd = M[rs1+imm][0:7]
Load Half (U)*	I	0000011	101		lhu rd, rs1, imm	rd = M[rs1+imm][0:15]
Load Word (U)*	I	0000011	110		ldu rd, rs1, imm	rd = M[rs1+imm][0:31]
Store Byte	S	0100011	000		sb rs1, rs2, imm	M[rs1+imm][0:7] = rs2[0:7]
Store Half	S	0100011	001		sh rs1, rs2, imm	M[rs1+imm][0:15] = rs2[0:15]
Store Word	S	0100011	010		sw rs1, rs2, imm	M[rs1+imm][0:31] = rs2[0:31]
Store Doubleword	S	0100011	011		sd rs1, rs2, imm	M[rs1+imm][0:63] = rs2[0:63]
Branch If Equal	B	1100011	000		beq rs1, rs2, imm	if(rs1 == rs2) PC += imm
Branch Not Equal	B	1100011	001		bne rs1, rs2, imm	if(rs1 != rs2) PC += imm
Branch Less Than	B	1100011	100		blt rs1, rs2, imm	if(rs1 < rs2) PC += imm
Branch Greater Than Or Equal	B	1100011	101		bge rs1, rs2, imm	if(rs1 ≥ rs2) PC += imm
Branch Less Than (U)*	B	1100011	110		bltu rs1, rs2, imm	if(rs1 < rs2) PC += imm
Branch Greater Than Or Equal (U)*	B	1100011	111		bgeu rs1, rs2, imm	if(rs1 ≥ rs2) PC += imm
Load Upper Immediate	U	0110111			lui rd, imm	rd = imm << 12
Add Upper Immediate To PC	U	0010111			auipc rd, imm	rd = PC + (imm << 12)
Jump And Link	J	1101111			jal rd, imm	rd = PC + 4; PC += imm
Jump And Link Register	I	1100111	000		jalr rd, rs1, imm	rd = PC + 4; PC = rs1 + imm

\* Assumes values are unsigned integers and zero extends † Fills in with sign bit during right shift and msb (most significant bit) extends

## RV64M Standard Extension Instructions

Name	Fmt	Opcode	Funct3	Funct7	Assembly	Description (in C)
Multiply	R	0110011	000	0000001	mul rd, rs1, rs2	$rd = (rs1 \cdot rs2)[63:0]$
Multiply Upper Half	R	0110011	001	0000001	mulh rd, rs1, rs2	$rd = (rs1 \cdot rs2)[127:64]$
Multiply Upper Half Sign/Unsigned <sup>†</sup>	R	0110011	010	0000001	mulhsu rd, rs1, rs2	$rd = (rs1 \cdot rs2)[127:64]$
Multiply Upper Half (U) <sup>*</sup>	R	0110011	011	0000001	mulhu rd, rs1, rs2	$rd = (rs1 \cdot rs2)[127:64]$
Divide	R	0110011	100	0000001	div rd, rs1, rs2	$rd = rs1 / rs2$
Divide (U) <sup>*</sup>	R	0110011	101	0000001	divu rd, rs1, rs2	$rd = rs1 / rs2$
Remainder	R	0110011	110	0000001	rem rd, rs1, rs2	$rd = rs1 \% rs2$
Remainder (U) <sup>*</sup>	R	0110011	111	0000001	remu rd, rs1, rs2	$rd = rs1 \% rs2$
Multiply Word	R	0111011	000	0000001	mulw rd, rs1, rs2	$rd = (rs1 \cdot rs2)[63:0]$
Divide Word	R	0111011	100	0000001	divw rd, rs1, rs2	$rd = rs1 / rs2$
Divide Word (U) <sup>*</sup>	R	0111011	101	0000001	divuw rd, rs1, rs2	$rd = rs1 / rs2$
Remainder Word	R	0111011	110	0000001	remw rd, rs1, rs2	$rd = rs1 \% rs2$
Remainder Word (U) <sup>*</sup>	R	0111011	111	0000001	remuw rd, rs1, rs2	$rd = rs1 \% rs2$

<sup>\*</sup> Assumes values are unsigned integers and zero extends <sup>†</sup> Multiply with one operand signed and the other unsigned

# Bibliography

- [1] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978. ISSN 1557-7317.
- [2] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software/Interface*. Morgan Kaufmann, 2017. ISBN 9780128122754.
- [3] B. Vinter and K. Skovhede. Synchronous message exchange for hardware designs. *Communicating Process Architectures*, pages 201–212, 2014.
- [4] B. Vinter and K. Skovhede. Bus centric synchronous message exchange for hardware designs. 2015.