

HOMEWORK I

Due day: 3:00pm Oct. 09 (Wednesday), 2024

Introduction

This homework is to let you be familiar with **SystemVerilog** and CPU design.

After this homework, you will complete a simplified pipeline CPU with **49** instructions.

General rules for deliverables

- This homework needs to be completed by **INDIVIDUAL** student.
 - Compress all files described in the problem statements into one **tar** file.
 - Submit the compressed file to the course website before the due day.
- Warning!** AVOID submitting in the last minute. Late submission is not accepted.

Grading Notes

- **Important!** DO remember to include your SystemVerilog code. NO code, NO grades. Also, if your code can not be recompiled by TA successfully using tools in SoC Lab and commands in Appendix B, you will receive NO credit.
- Write your report seriously and professionally. Incomplete description and information will reduce your chances to get more credits.
- If extra works (like synthesis, post-simulation or additional instructions) are done, please describe them in your final report clearly for bonus points.
- Please follow course policy.
- Verilog and SystemVerilog generators aren't allowed in this course.

Deliverables

1. All SystemVerilog codes including components, testbenches and machine codes for each problem. NOTE: Please **DO NOT** include source codes in the report!
2. Write a homework report in MS word and follow the convention for the file name of your report: *N26XXXXXX.docx*. Please save as docx file format and replace N26XXXXXX with your student ID number. (Let the letter be uppercase.)
3. Organize your files as the hierarchy in Appendix A.

HOMEWORK I

Report Writing Format

- a. Use the **submission cover** which is already in provided *N26XXXXXX.docx*.
- b. **A summary in the beginning** to state what has been done.
- c. Report requirements from each problem.
- d. Describe the major problems you encountered and your resolutions.
- e. Lessons learned from this homework.

HOMEWORK I

Problem 1 (100/100)

1.1 Problem Description

In this course, we use the RISC-V instruction set architecture (ISA) to implement a pipelined CPU. The RISC-V ISA is composed of one base integer instruction set and some standard extensions. For simplicity of implementation, the homework of this course focuses on the RV32I base integer instruction set.

You need to implement a simplified pipeline CPU with the following features:

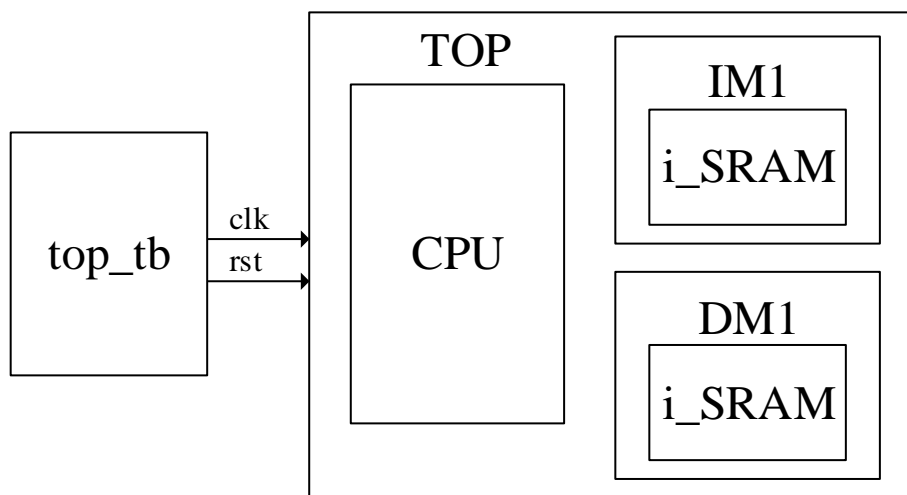
- The RISC-V ISA with the specified 49 instructions.
- The number of pipeline stage is 5.
- General register file size: 32×32-bit.
- Floating point register: 32×32-bit.
- Program counter with 32-bit.
- Mechanism to solve data hazard, control hazard and structural hazard.
- Instruction register with 64-bit
- Cycle register with 64-bit
- CSR with 2×64-bit

You also need to use two memories outside the CPU with specified size:

- Instruction memory size: 64KB.
- Data memory size: 64KB.

Your RTL code should comply with Superlint within 85% of your code. Besides, you should use programs listed in Section 1.5 to verify your design. Note that you **need to synthesize** your design. A more detailed description of this problem can be found in Section 1.4.

1.2 Block Overview



HOMEWORK I

Fig. 1-1: System block diagram

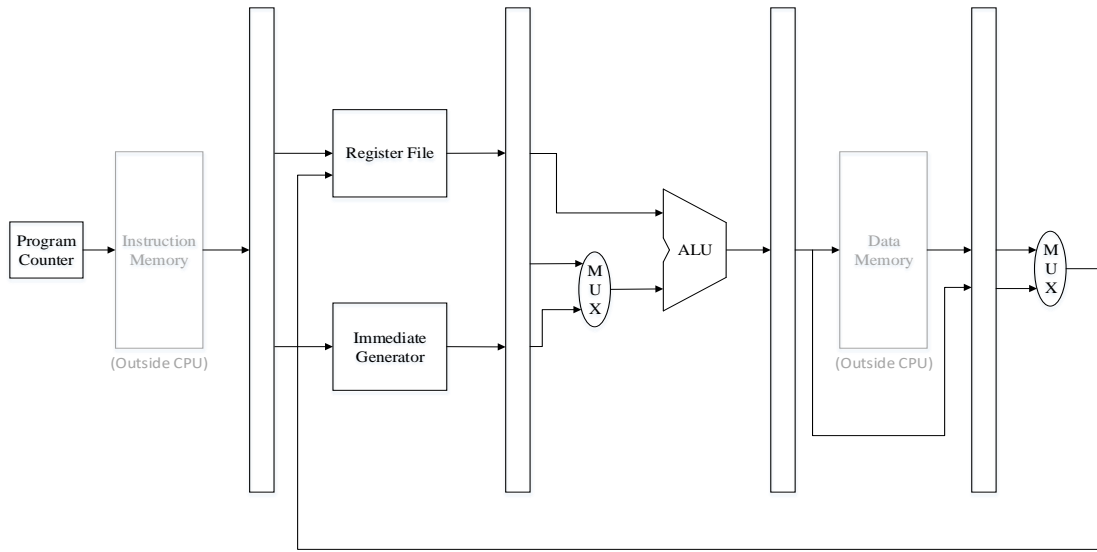


Fig. 1-2: A CPU architecture for reference

1.3 Module Specification

Table 1-1: Module naming rule

Category	Name			
	File	Module	Instance	SDF
RTL	top.sv	top	TOP	
RTL	SRAM_wrapper.sv	SRAM_wrapper	IM1	
RTL	SRAM_wrapper.sv	SRAM_wrapper	DM1	
RTL	TS1N16ADFPCLLLVT A512X45M4SWSHOD .sv	TS1N16ADFPCLLLVT A512X45M4SWSHOD	i_SRAM	

Table 1-2: Module signals

Module	Specifications			
top	Name	Signal	Bits	Function explanation
	clk	input	1	System clock
	rst	input	1	System reset (active high)
SRAM_wrapper	CLK	input	1	System clock
	CEB	input	1	Chip enable (active low)
	WEB	input	1	read:active high,write:active low

HOMEWORK I

	BWEB	input	32	Bit write enable (active low)
	A	input	14	Address
	DI	input	32	Data input
	DO	output	32	Data output
Module	Name	Signal	Bits	Function explanation
TS1N16ADFPCLLLVTA 512X45M4SWSHOD	Memory Space			
	MEMORY	logic	32	Size: [512][32]

1.4 Detailed Description

Fig. 1-1 is a system overview of this problem. You **SHOULD NOT** modify any port declarations or your design may have error when TA runs the hidden testbench.

Fig. 1-2 is a block diagram of a simplified 5-stage pipeline CPU. **Note that this diagram shows only some possible components and signals of a pipeline CPU, and you may need to add others, e.g. a controller and its signals, for your design.** You can also develop your own architecture. The only restriction is that your architecture **SHOULD** have 5 stages of pipeline.

You should implement the instructions in Table 1-3. The detailed instruction types and the immediate formats can be found in Appendix C. You can also study *The RISC-V Instruction Set Manual* posted on the course website.

Table 1-3: Instruction lists

R-type

31	25	24	20	19	15	14	12	11	7	6	0		
funct7		rs2		rs1		funct3		rd		opcode		Mnemonic	Description
0000000		rs2		rs1		000		rd		0110011		ADD	$rd = rs1 + rs2$
0100000		rs2		rs1		000		rd		0110011		SUB	$rd = rs1 - rs2$
0000000		rs2		rs1		001		rd		0110011		SLL	$rd = rs1_u \ll rs2[4:0]$
0000000		rs2		rs1		010		rd		0110011		SLT	$rd = (rs1_s < rs2_s)? 1:0$
0000000		rs2		rs1		011		rd		0110011		SLTU	$rd = (rs1_u < rs2_u)? 1:0$
0000000		rs2		rs1		100		rd		0110011		XOR	$rd = rs1 \wedge rs2$
0000000		rs2		rs1		101		rd		0110011		SRL	$rd = rs1_u \gg rs2[4:0]$
0100000		rs2		rs1		101		rd		0110011		SRA	$rd = rs1_s \gg rs2[4:0]$
0000000		rs2		rs1		110		rd		0110011		OR	$rd = rs1 \mid rs2$
0000000		rs2		rs1		111		rd		0110011		AND	$rd = rs1 \& rs2$

HOMEWORK I

0000001	rs2	rs1	000	rd	0110011	MUL	rd = lower 32 bits of (rs1 * rs2)
0000001	rs2	rs1	001	rd	0110011	MULH	rd = upper 32 bits of (rs1 * rs2) signed * signed
0000001	rs2	rs1	010	rd	0110011	MULHSU	rd = upper 32 bits of (rs1 * rs2) signed * unsigned
0000001	rs2	rs1	011	rd	0110011	MULHU	rd = upper 32 bits of (rs1 * rs2) unsigned * unsigned

I-type

31	20	19	15	14	12	11	7	6	0		
imm[11:0]		rs1		funct3		rd		opcode		Mnemonic	Description
imm[11:0]		rs1		010		rd		0000011		LW	rd = M[rs1+imm]
imm[11:0]		rs1		000		rd		0010011		ADDI	rd = rs1 + imm
imm[11:0]		rs1		010		rd		0010011		SLTI	rd = (rs1 _s < imm _s)? 1:0
imm[11:0]		rs1		011		rd		0010011		SLTIU	rd = (rs1 _u < imm _u)? 1:0
imm[11:0]		rs1		100		rd		0010011		XORI	rd = rs1 ^ imm
imm[11:0]		rs1		110		rd		0010011		ORI	rd = rs1 imm
imm[11:0]		rs1		111		rd		0010011		ANDI	rd = rs1 & imm
imm[11:0]		rs1		000		rd		0000011		LB	rd = M[rs1+imm] _{bs}
0000000	shamt	rs1		001		rd		0010011		SLLI	rd = rs1 _u << shamt
0000000	shamt	rs1		101		rd		0010011		SRLI	rd = rs1 _u >> shamt
0100000	shamt	rs1		101		rd		0010011		SRAI	rd = rs1 _s >> shamt
imm[11:0]		rs1		000		rd		1100111		JALR	rd = PC + 4 PC = imm + rs1 (Set LSB of PC to 0)
imm[11:0]		rs1		001		rd		0000011		LH	rd = M[rs1+imm] _{hs}
imm[11:0]		rs1		100		rd		0000011		LBU	rd = M[rs1+imm] _{bu}
imm[11:0]		rs1		101		rd		0000011		LHU	rd = M[rs1+imm] _{bu}

HOMEWORK I

☞ S-type

31	25	24	20	19	15	14	12	11	7	6	0		
imm[11:5]		rs2		rs1		funct3		imm[4:0]		opcode		Mnemonic	Description
imm[11:5]		rs2		rs1		010		imm[4:0]		0100011		SW	$M[rs1+imm] = rs2$
imm[11:5]		rs2		rs1		000		imm[4:0]		0100011		SB	$M[rs1+imm]_b = rs2_b$
imm[11:5]		rs2		rs1		001		imm[4:0]		0100011		SH	$M[rs1+imm]_b = rs2_h$

☞ B-type

31	25	24	20	19	15	14	12	11	7	6	0		
imm[12 10:5]		rs2		rs1		funct3		imm[4:1 11]		opcode		Mnemonic	Description
imm[12 10:5]		rs2		rs1		000		imm[4:1 11]		1100011		BEQ	PC = (rs1 == rs2)? PC + imm: PC + 4
imm[12 10:5]		rs2		rs1		001		imm[4:1 11]		1100011		BNE	PC = (rs1 != rs2)? PC + imm: PC + 4
imm[12 10:5]		rs2		rs1		100		imm[4:1 11]		1100011		BLT	PC = (rs1 _s < rs2 _s)? PC + imm: PC + 4
imm[12 10:5]		rs2		rs1		101		imm[4:1 11]		1100011		BGE	PC = (rs1 _s ≥ rs2 _s)? PC + imm: PC + 4
imm[12 10:5]		rs2		rs1		110		imm[4:1 11]		1100011		BLTU	PC = (rs1 _u < rs2 _u)? PC + imm: PC + 4
imm[12 10:5]		rs2		rs1		111		imm[4:1 11]		1100011		BGEU	PC = (rs1 _u ≥ rs2 _u)? PC + imm: PC + 4

☞ U-type

31	12	11	7	6	0		
imm[31:12]				rd	opcode	Mnemonic	Description
imm[31:12]				rd	0010111	AUIPC	rd = PC + imm
imm[31:12]				rd	0110111	LUI	rd = imm

☞ J-type

31	12	11	7	6	0		
imm[20 10:1 11 19:12]				rd	opcode	Mnemonic	Description
imm[20 10:1 11 19:12]				rd	1101111	JAL	rd = PC + 4 PC = PC + imm

HOMEWORK I

F-type

31	20	19	15	14	12	11	7	6	0		
imm[11:0]		rs1		funct3		frd		opcode		Mnemonic	Description
imm[11:0]		rs1		010		frd		0000111		FLW	$frd = M[rs1+imm]$

31	25	24	20	19	15	14	12	11	7	6	0		
imm[11:5]		frs2		rs1		funct3		imm[4:0]		opcode		Mnemonic	Description
imm[11:5]		frs2		rs1		010		imm[4:0]		0100111		FSW	$M[rs1+imm] = frs2$

31	27	26	25	24	20	19	15	14	12	11	7	6	0		
funct5		fmt		frs2		frs1		rm		frd		opcode		Mnemonic	Description
00000		00		frs2		frs1		111		frd		1010011		FADD.S	$frd = frs1 + frs2$
00001		00		frs2		frs1		111		frd		1010011		FSUB.S	$frd = frs1 - frs2$

- Note1 : The fmt field represents floating-point format, and in this lab, we only use **32-bit single-precision**.
- Note2 : The rm field represents rounding mode, and in this lab, we only use **the Round to Nearest, ties to Even** mode.
- Note3 : You don't need to consider overflow or underflow situations in this lab.

CSR instruction

imm[11:0]	rs1	funct3	rd	opcode	Mnemonic	Description
110010000010	00000	010	rd	1110011	RDINSTRETH	$rd = instret[63:32]$
110000000010	00000	010	rd	1110011	RDINSTRET	$rd = instret[31:0]$
110010000000	00000	010	rd	1110011	RDCYCLEH	$rd = cycle[63:32]$
110000000000	00000	010	rd	1110011	RDCYCLE	$rd = cycle[31:0]$

The data size of your design **SHOULD BE 32 BITS**. You need to implement a register file with 32 registers. There are 31 general registers x1–x31. **Note that the general register x0 is hardwire to the constant 0.** There are 32 floating point registers f0–f31. In addition to the register file, you also need to implement a 32-bit program counter. Every memory address should be **four-byte aligned**, i.e., the program counter and the load/store address should be multiple of 4. You should implement the mechanism to avoid hazards, e.g. forwarding or hazard detection.

The size of the instruction memory is **16K**×4-byte (64KB), and the size of the

HOMEWORK I

data memory is **16K**×4-byte (64KB). You should use SRAM_wrapper as instruction memory (IM1) and data memory (DM1). You **SHOULDN'T** modify Verilog code in SRAM_wrapper and SRAM.

Your RTL code needs to comply with Superlint within 85% of your code, i.e., the number of errors & warnings in total shall not exceed 15% of the number of lines in your code. HINT: You can use the command in Appendix B to get the number of lines in *src* and *include* directories.

1.5 Verification

You should complete following programs and use the commands in Appendix B to verify your design.

- a. Use *prog0* to perform verification for the functionality of instructions.
- b. Write a program defined as *prog1* to perform a sort algorithm. The number of sorting elements is stored at the address named *array_size* in “.rodata” section defined in *data.S*. The first element is stored at the address named *array_addr* in “.rodata” section defined in *data.S*, others are stored at adjacent addresses. The maximum number of elements is 64. All elements are **signed 4-byte integers** and you should sort them in **ascending order**. Rearranged data should be stored at the address named *_test_start* in “_test” section defined in *link.ld*.
- c. Use *prog2* to perform multiplication. The multiplicand is stored at the address named *mul1* in “.rodata” section defined in *data.S*. The multiplier is stored at the address named *mul2* in “.rodata” section defined in *data.S*. The multiplicand and the multiplier are **signed 4-byte integers**. Their product is **signed 8-byte integers** and should be stored at the address named *_test_start* in “_test” section defined in *link.ld*.
- d. Write a program defined as *prog3* to perform greatest common divisor(GCD). The first number is stored at the address named *div1* in “.rodata” section defined in *data.S*. The second data is stored at the address named *div2* in “.rodata” section defined in *data.S*. These two numbers are **unsigned 4-byte integers**. The result should be stored at the address named *_test_start* in “_test” section defined in *link.ld*. The values of the quotient and the remainder should follow C99 specification. “When integers are divided, the result of the / operator is the algebraic quotient with any fractional part discarded. If the quotient **a/b** is representable, the expression **(a/b)*b + a%b** shall equal **a**.”

HOMEWORK I

- e. Use *prog4* to perform verification for the functionality of *rdinstret*, *rdinstreth*, *rdcycle*, *rdcycleh*.
- f. Use *prog5* to perform multiplication. This multiplication use `MUL[[S]U]` instructions to finish the computation.
- g. Use *prog6* to perform floating point computation.

Don't forget to return from *main* function to finish the simulation in each program. Save your assembly code or C code as *main.S* or *main.c* respectively. You should also explain the result of this program in the report. In addition to these verification, **TA will use another program to verify your design.** Please make sure that your design can execute the listed instructions correctly.

1.6 Report Requirements

Your report should have the following features:

- a. Proper explanation of your design is required for full credits.
- b. Block diagrams shall be drawn to depict your designs.
- c. Show your snapshots of **the waveforms and the simulation results on the terminal** for the different test cases in your report and **illustrate** the correctness of your results.
- d. Report the number of lines of your RTL code, the final results of running Superlint and 3~5 most frequent warning/errors in your code. Describe how you modify your code to comply with the Superlint.

HOMEWORK I

Appendix

A. File Hierarchy Requirements

All homework **SHOULD** be uploaded and follow the file hierarchy and the naming rules, especially letter case, specified below. You should create a main folder named your student ID number. It contains your homework report and other files. The names of the files and the folders are labeled in **red color**, and the specifications are labeled in black color. Filenames with * suffix in the same folder indicate that you should provide one of them. Before you submit your homework, you can use Makefile macros in Appendix B to check correctness of the file structure.

Fig. A-1 File hierarchy

- ❏ **N260XXXXX.tar** (**Don't** add version text in filename, e.g. *N260XXXXX_v1.tar*)
 - ❏ **N260XXXXX** (Main folder of this homework)
 - 📄 **N260XXXXX.docx** (Your homework report)
 - 📄 **StudentID** (Specify your student ID number in this file)
 - 📄 **Makefile** (You shouldn't modify it)
 - ❏ **src** (Your RTL code with *sv* format)
 - 📄 **top.sv**
 - 📄 **SRAM_wrapper.sv**
 - 📄 Other submodules (*.sv)
 - ❏ **include** (Your RTL definition with *svh* format, optional)
 - 📄 Definition files (*.svh)
 - ❏ **syn** (Your synthesized code and timing file, optional)
 - 📄 **top_syn.v**
 - 📄 **top_syn.sdf**
 - ❏ **script** (Any scripts of verification, synthesis or place and route)
 - 📄 Script files (*.sdc, *.tcl or *.setup)
 - ❏ **sim** (Testbenches and memory libraries)
 - 📄 **top_tb.sv** (Testbench. You can only modify CYCLE in tb)
 - 📄 **CYCLE** (Specify your clock cycle time in this file)
 - 📄 **MAX** (Specify max clock cycle number in this file)
 - ❏ **SRAM** (SRAM libraries and behavior models)
 - 📄 Library files (*.lib, *.db, *.lef or *.gds)
 - 📄 **TS1N16ADFPCLLLVTA512X45M4SWSHOD.sv** (**SRAM behavior model**)

HOMEWORK I

- 📁 *prog0* (Subfolder for Program 0)
 - 📄 *Makefile* (Compile and generate memory content)
 - 📄 *main.S* (Assembly code for verification)
 - 📄 *setup.S* (Assembly code for testing environment setup)
 - 📄 *link.ld* (Linker script for testing environment)
 - 📄 *golden.hex* (Golden hexadecimal data)
- 📁 *prog1* (Subfolder for Program 1)
 - 📄 *Makefile* (Compile and generate memory content)
 - 📄 *main.S* * (Assembly code for verification)
 - 📄 *main.c* * (C code for verification)
 - 📄 *data.S* (Assembly code for testing data)
 - 📄 *setup.S* (Assembly code for testing environment setup)
 - 📄 *link.ld* (Linker script for testing environment)
 - 📄 *golden.hex* (Golden hexadecimal data)
- 📁 *prog2* (Subfolder for Program 2)
 - 📄 *Makefile* (Compile and generate memory content)
 - 📄 *main.c* (C code for verification)
 - 📄 *data.S* (Assembly code for testing data)
 - 📄 *setup.S* (Assembly code for testing environment setup)
 - 📄 *link.ld* (Linker script for testing environment)
 - 📄 *golden.hex* (Golden hexadecimal data)
- 📁 *prog3* (Subfolder for Program 3)
 - 📄 *Makefile* (Compile and generate memory content)
 - 📄 *main.S* * (Assembly code for verification)
 - 📄 *main.c* * (C code for verification)
 - 📄 *data.S* (Assembly code for testing data)
 - 📄 *setup.S* (Assembly code for testing environment setup)
 - 📄 *link.ld* (Linker script for testing environment)
 - 📄 *golden.hex* (Golden hexadecimal data)

HOMEWORK I

- 📁 *prog4* (Subfolder for Program 4)
 - 📄 *Makefile* (Compile and generate memory content)
 - 📄 *main.c* (C code for verification)
 - 📄 *data.S* (Assembly code for testing data)
 - 📄 *setup.S* (Assembly code for testing environment setup)
 - 📄 *link.ld* (Linker script for testing environment)
 - 📄 *golden.hex* (Golden hexadecimal data)
- 📁 *prog5* (Subfolder for Program 5)
 - 📄 *Makefile* (Compile and generate memory content)
 - 📄 *main.c* (C code for verification)
 - 📄 *data.S* (Assembly code for testing data)
 - 📄 *setup.S* (Assembly code for testing environment setup)
 - 📄 *link.ld* (Linker script for testing environment)
 - 📄 *golden.hex* (Golden hexadecimal data)
- 📁 *prog6* (Subfolder for Program 6)
 - 📄 *Makefile* (Compile and generate memory content)
 - 📄 *main.c* (C code for verification)
 - 📄 *data.S* (Assembly code for testing data)
 - 📄 *setup.S* (Assembly code for testing environment setup)
 - 📄 *link.ld* (Linker script for testing environment)
 - 📄 *golden.hex* (Golden hexadecimal data)

B. Simulation Setting Requirements

You **SHOULD** make sure that your code can be simulated with specified commands in Table B-1. **TA will use the same command to check your design under SoC Lab environment. If your code can't be recompiled by TA successfully, you receive NO credit.**

Table B-1: Simulation commands

Simulation Level	Command
Problem1	
RTL	<i>make rtl_all</i>
Post-synthesis (optional)	<i>make syn_all</i>

TA also provide some useful Makefile macros listed in Table B-2. Braces {}

HOMEWORK I

means that you can choose one of items in the braces. X stands for 0,1,2,3..., depend on which verification program is selected.

Table B-2: Makefile macros

Situation	Command
RTL simulation for progX	<code>make rtlX</code>
Post-synthesis simulation for progX	<code>make synX</code>
Dump waveform (no array)	<code>make {rtlX,synX} FSDB=1</code>
Dump waveform (with array)	<code>make {rtlX,synX} FSDB=2</code>
Open nWave without file pollution	<code>make nWave</code>
Open Superlint without file pollution	<code>make superlint</code>
Open DesignVision without file pollution	<code>make dv</code>
Synthesize your RTL code (You need write <i>synthesis.tcl</i> in <i>script</i> folder by yourself)	<code>make synthesize</code>
Delete built files for simulation, synthesis or verification	<code>make clean</code>
Check correctness of your file structure	<code>make check</code>
Compress your homework to <i>tar</i> format	<code>make tar</code>

You can use the following command to get the number of lines:

```
wc -l src/* include/*
```

HOMEWORK I

C. RISC-V Instruction Format

Table C-1: Instruction type

R-type

31	25	24	20	19	15	14	12	11	7	6	0
funct7		rs2		rs1		funct3		rd		opcode	

I-type

31	20	19	15	14	12	11	7	6	0
imm[11:0]		rs1		funct3		rd		opcode	

S-type

31	25	24	20	19	15	14	12	11	7	6	0
imm[11:5]		rs2		rs1		funct3		imm[4:0]		opcode	

B-type

31	30	25	24	20	19	15	14	12	11	8	7	6	0
imm[12]	imm[10:5]		rs2		rs1		funct3		imm[4:1]		imm[11]		opcode

U-type

31	12	11	7	6	0
imm[31:12]		rd		opcode	

J-type

31	30	21	20	19	12	11	7	6	0
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	

F-type(FLW)

31	20	19	15	14	12	11	7	6	0
imm[11:0]		rs1		funct3		frd		opcode	

F-type(FSW)

31	25	24	20	19	15	14	12	11	7	6	0
imm[11:5]		frs2		rs1		funct3		imm[4:0]		opcode	

F-type(FADD.S,FSUB.S)

31	27	26	25	24	20	19	15	14	12	11	7	6	0
funct5		00		frs2		frs1		111		frd		opcode	

Table C-2: Immediate type

I-immediate

31	11	10	5	4	1	0
— inst[31] —		inst[30:25]		inst[24:21]		inst[20]

HOMEWORK I

☞ S-immediate

31	11	10	5	4	1	0
— inst[31] —		inst[30:25]		inst[11:8]		inst[7]

☞ B-immediate

31	12	11	10	5	4	1	0
— inst[31] —		inst[7]	inst[30:25]		inst[11:8]		0

☞ U-immediate

31	30	20	19	12	11	0
Inst[31]	inst[30:20]		inst[19:12]		— 0 —	

☞ J-immediate

31	20	19	12	11	10	5	4	1	0
— inst[31] —		inst[19:12]		inst[20]	inst[30:25]		inst[24:21]		0

☞ FLW-immediate

31	11	10	5	4	1	0
— inst[31] —		inst[30:25]		inst[24:21]		inst[20]

☞ FSW-immediate

31	11	10	5	4	1	0
— inst[31] —		inst[30:25]		inst[11:8]		inst[7]

☞ CSR instruction

31	20	19	15	14	12	11	7	6	0
— imm[11:0] —		00000		010		rd		opcode	

“— X —” indicates that all the bits in this range is filled with X.