

Computer Architectures

Session 1

Single cycle RISC-V processor

TAs :

Jun Yin (jun.yin@kuleuven.be)

Yuanyang Guo (yuanyang.guo@imec.be)

Xiaoling Yi (xiaoling.yi@kuleuven.be)

Yunzhu Chen (yunzhu.chen@imec.be)

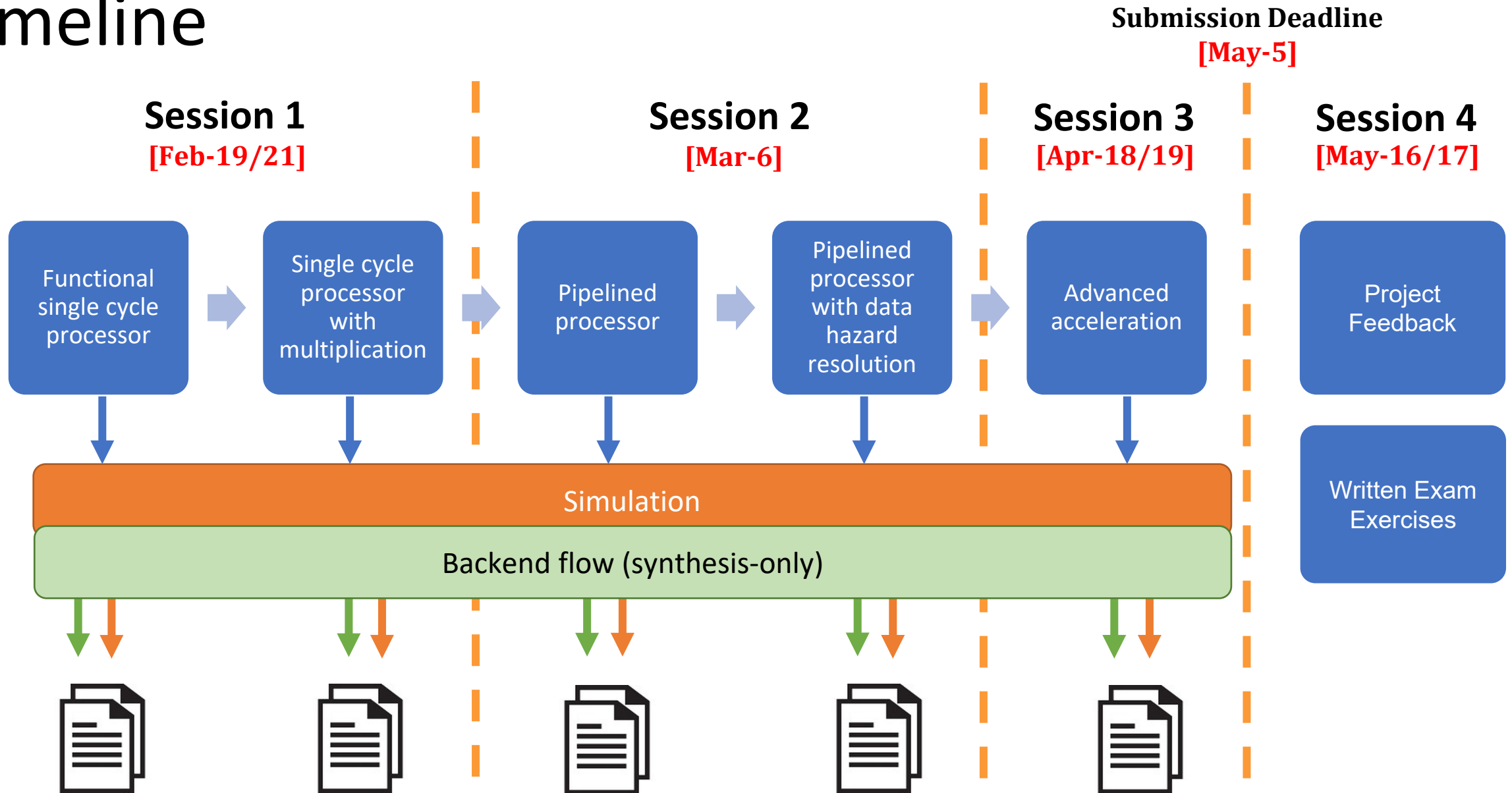
Objective

- RTL design of a RISC-V microprocessor
 - Simple Implementation
 - Pipelined implementation
 - Data Hazard resolution
 - Advanced Acceleration
 - Branch prediction
 - SIMD
 - ...

Chapter 4



Timeline



Session requirements and criteria:

- The assignments must be completed in groups of **≤2 people**.
- After completing each one of the scenarios:
 1. You have to copy-paste your RTL source code into the corresponding SOLUTION folder.
 2. You have to complete part of a small report to record performance readings and answer several related questions (see report.docx) .
- This project counts for **4 points** in the final course grade.
 1. The performance check in each design scenario.
 2. The report.

Grading

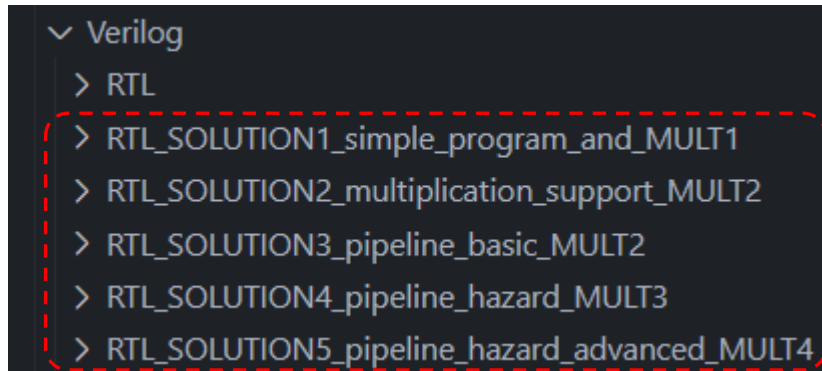
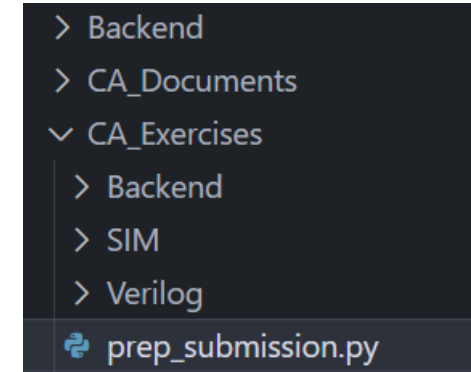
ITEM	Points
Functional pipelined MULT2	1.0
Functional pipelined MULT3	0.5
Functional MULT4	0.5
Functional MULT4 #cycles = baseline impl (844 cc)	0.5
Functional MULT4 #cycles < baseline impl (844 cc)	0.5
Report	1.0
Total	4.0

Project handover

Deadline: **May 5th**

Session requirements and criteria:

- We release the same grading script as final.
 - Command: `python3 prep_submission.py`
 - This script works in standalone folders. But it is always wise to backup your ongoing **/RTL** codes before execution.
- Do not forget to perform at least one dry-run before submission.
 - **Make sure you have put the right version of source code into the right folder!**



```
Checking simpleprogram->simple_program ...
Checking mult1->MULT1 ...
Checking mult2->multiplication_support_MULT2 ...
Checking mult2->pipeline_basic_MULT2 ...
Checking mult3->pipeline_hazard_MULT3 ...
Checking mult4->pipeline_hazard_advanced_MULT4 ...
Done
Scoreboard:

Functional pipelined MULT2:      1.0 pts (9 cc, 2300 cc, 40 cc, 40 cc)

Functional pipelined MULT3:      0.5 pts (25 cc)

Functional pipelined MULT4:      1.0 pts (844 cc)

-----
Total Score:      2.5 pts

Do not forget to complete the GROUP_X folder before your submission!
Reference output when grading our baseline (2.5/3 pts).
```

prep_submission.py

- Evaluate your RTL_SOLUTION(s)
- Basic debug tracing if the program fails
- Create the structure for submission

```
GROUP_X/  
├── MULT4_content  
│   └── mult4_imem_content.txt  
└── RTL_SOLUTIONS  
    ├── RTL_SOLUTION1_simple_program_and_MULT1  
    ├── RTL_SOLUTION2_multiplication_support_MULT2  
    ├── RTL_SOLUTION3_pipeline_basic_MULT2  
    ├── RTL_SOLUTION4_pipeline_hazard_MULT3  
    └── RTL_SOLUTION5_pipeline_hazard_advanced_MULT4
```

```
Checking simpleprogram->simple_program_and_MULT1 ...  
Checking mult1->simple_program_and_MULT1 ...  
Checking mult2->pipeline_basic_MULT2 ...  
Verilog src folder not found. Please check file structures.  
Checking mult3->pipeline_hazard_MULT3 ...  
rm -rf xcelium.d xrun.* xmverilog.* *.vcd *.sim  
clear;  
xrun +sv -f files_verilog_grading.f -64bit -timescale 1ns/10ps -access +rwc -allowredefinition  
  
TOOL: xrun(64) 23.03-s002: Started on Jan 31, 2024 at 11:10:57 CET  
xrun(64): 23.03-s002: (c) Copyright 1995-2023 Cadence Design Systems, Inc.  
file: ../Verilog/cpu_tb.v  
module worklib.cpu_tb:v  
errors: 0, warnings: 0  
file: ../Verilog/skyl30_sram_2rw.v  
module worklib.skyl30_sram_2rw_32x128_32:v  
errors: 0, warnings: 0  
module worklib.skyl30_sram_2rw_64x128_64:v  
errors: 0, warnings: 0  
Caching library 'worklib' ..... Done  
Elaborating the design hierarchy:  
Caching library 'worklib' ..... Done  
  
cpu dut(  
|  
xmclab: *E,CUVMUR (../Verilog/cpu_tb.v,67|6): instance 'cpu_tb.dut' of design unit 'cp  
u' is unresolved in 'worklib.cpu_tb:v'.  
xrun: *E,ELBERR: Error (*E) or soft error (*SE) occurred during elaboration (status 1)  
, exiting.  
TOOL: xrun(64) 23.03-s002: Exiting on Jan 31, 2024 at 11:10:57 CET (total: 0  
0:00:00)  
make: *** [sim_grading] Error 1  
  
Errors occurred when simulating. Please check your source code structure.  
Checking mult4->pipeline_hazard_advanced_MULT4 ...  
Done  
Scoreboard:  
  
Functional pipelined MULT2: 0.0 pts (9 cc, 2300 cc, False cc)  
Functional pipelined MULT3: 0.0 pts (False cc)  
Functional pipelined MULT4: 1.0 pts (844 cc)  
  
-----  
Total Score: 1.0 pts
```

Debug tracing example.

Session requirements and criteria:

- The **report** and your **final processor design** need to be handed in through Toledo, following the below file structure:

```
GROUP_X/  
├── MULT4_content  
│   └── mult4_imem_content.txt  
├── README.txt  
├── report.docx  
└── RTL_SOLUTIONS  
    ├── RTL_SOLUTION1_simple_program_and_MULT1  
    ├── RTL_SOLUTION2_multiplication_support_MULT2  
    ├── RTL_SOLUTION3_pipeline_basic_MULT2  
    ├── RTL_SOLUTION4_pipeline_hazard_MULT3  
    └── RTL_SOLUTION5_pipeline_hazard_advanced_MULT4
```

Do not forget!

- Mult4 session (session3) is special so that additional files are submitted.**

- 1) Name your project as Group_X (X is your group number);
- 2) Put all the students' name and students' R-number of the group in README.txt;
- 3) You can also put anything that you think we need to know before running your project in the README.txt.
- 4) Pack your group folder into a zip file and submit.

- The **deadline** of the project is **May. 5th, 2024.**

Course material

- Project resources

```
CA_processor_design_2024/
├── CA_Documents
│   ├── Jupyter_VNC_Manual.pdf
│   ├── reading_backend_report.pdf
│   ├── report.docx
│   ├── Session1.pdf
│   ├── Session2.pdf
│   ├── Session3.pdf
│   └── session_guide.pdf
├── CA_Exercises
│   ├── Backend
│   ├── prep_submission.py
│   ├── SIM
│   └── Verilog
```

- Extra resources

- Computer organization and design. RISC-V edition.

[See Toledo](#)

- RISC-V specification

<https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf>

- RISC-V green card

<https://inst.eecs.berkeley.edu/~cs61c/fa17/img/riscv-card.pdf>

- Verilog tutorial

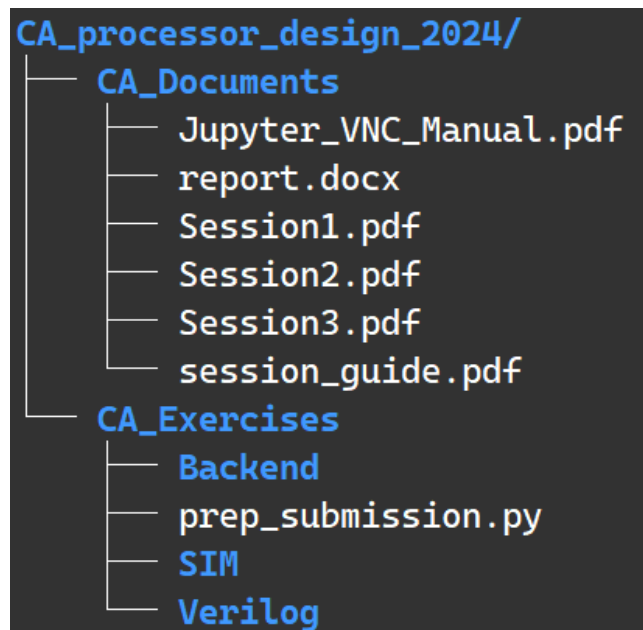
<https://www.asic-world.com/verilog/veritut.html>

- RISC-V machine code generator *

<https://github.com/Kritagya-Agarwal/Assembly-To-Machine-Code-RISC-V>

* assembles RV64 but only simulates RV32, **only use it for assembly to machine code translation**

Project resources



- **Verilog**

- RTL
RTL source code
- sky130_sram_2rw.v
Memory behaviour model
- cpu_tb.v
Testbench (**modification not allowed**)

- **Backend**

- CA_RISCV.ipynb
Python file to run backend flow
- Figs
- OpenRAM_output
- Setup.sh

- **prep_submission.py**

Python script to dry-run the grading before submission

- **SIM**

Simulation related resources

- data
Instruction and data memory sources
 - testcode_m
Imem and dmem contents used for different scenarios
 - dmem_content.txt
Data memory to be loaded in testbench
 - imem_content.txt
Instruction memory to be loaded in testbench
- files_verilog.f
RTL simulation filelist
- Makefile
simulation-related commands
- xcelium_23.03.rc
simulator

Please refer to the **CA_Documents** for more information:

- **session_guide**
- **Jupyter_VNC_Manual**

Remarks: Keep your project safe!



Please keep all 5 versions of your processor (RTL), we will grade based on them!

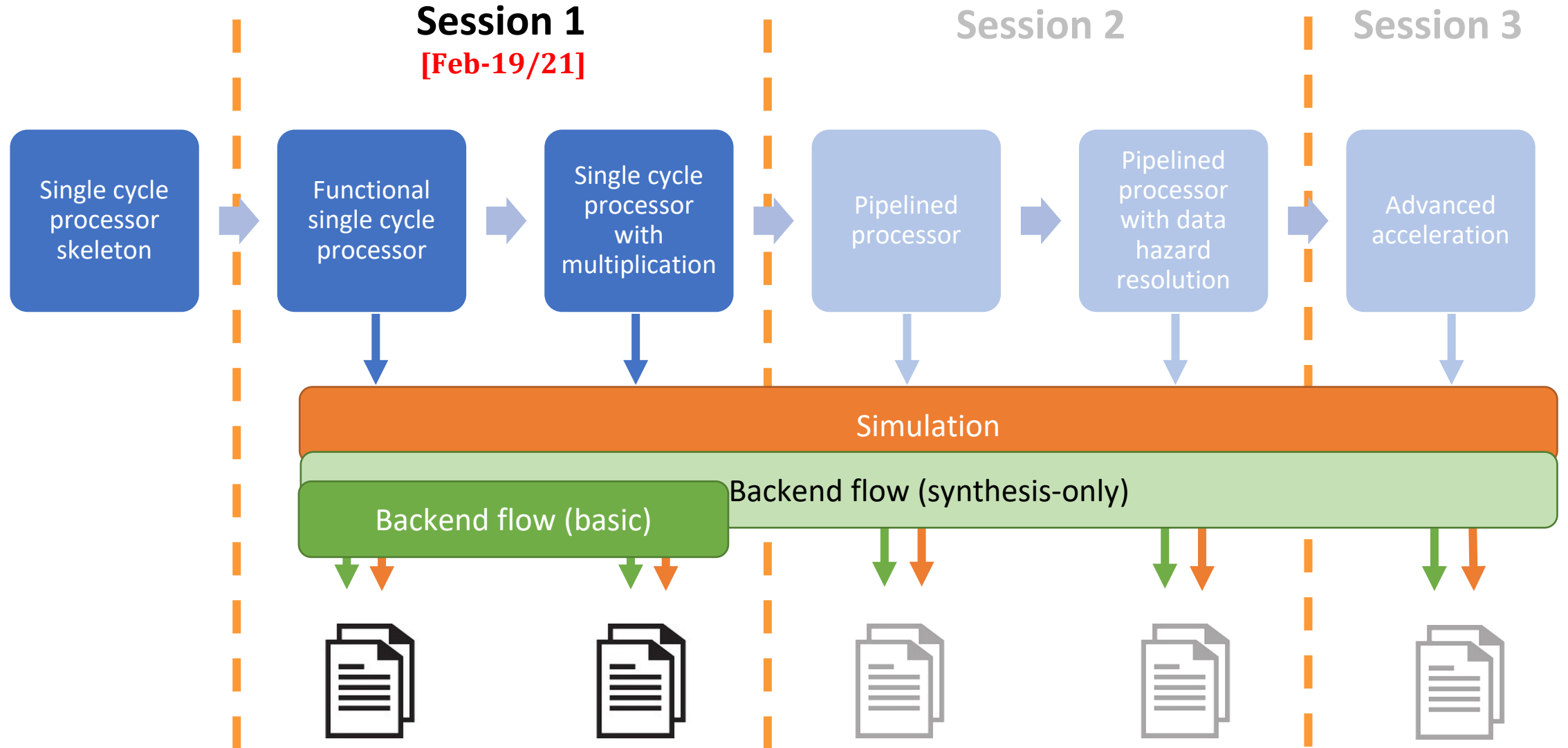
We highly suggest you to use ***git*** as a versioning tool.

It helps you to:

- keep track of modifications
- automatically sync the project between group members
- allow TAs to support you more easily

Please have a look at the dedicated guide.

Today's session: Single cycle processor



Basic architecture

- Testbench

- cpu_tb.v

- RTL

- alu_control.v

- alu.v

- branch_unit.v

- control_unit.v

- cpu.v

- immediate_extend_unit.v

- mux_2.v

- pc.v

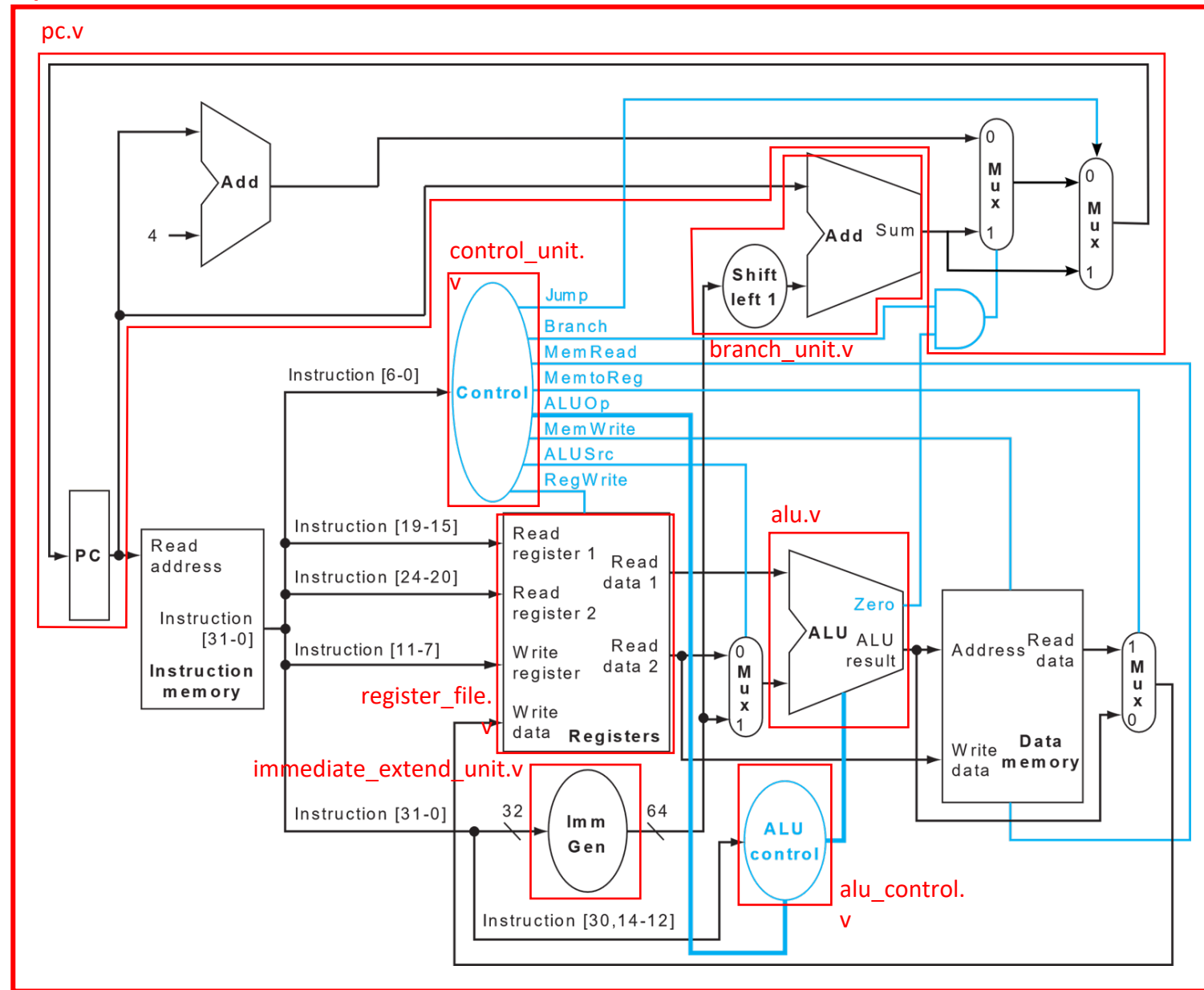
- register_file.v

- sram.v

- IP Library

- sky130_sram_2rw.v

cpu.v



Simulation Tool

- Simulation (Cadence Xcelium)
 - Tool to simulate the behaviour of the RTL
- In the ./SIM folder, set up the environment

```
source xcelium_23.03.rc
```

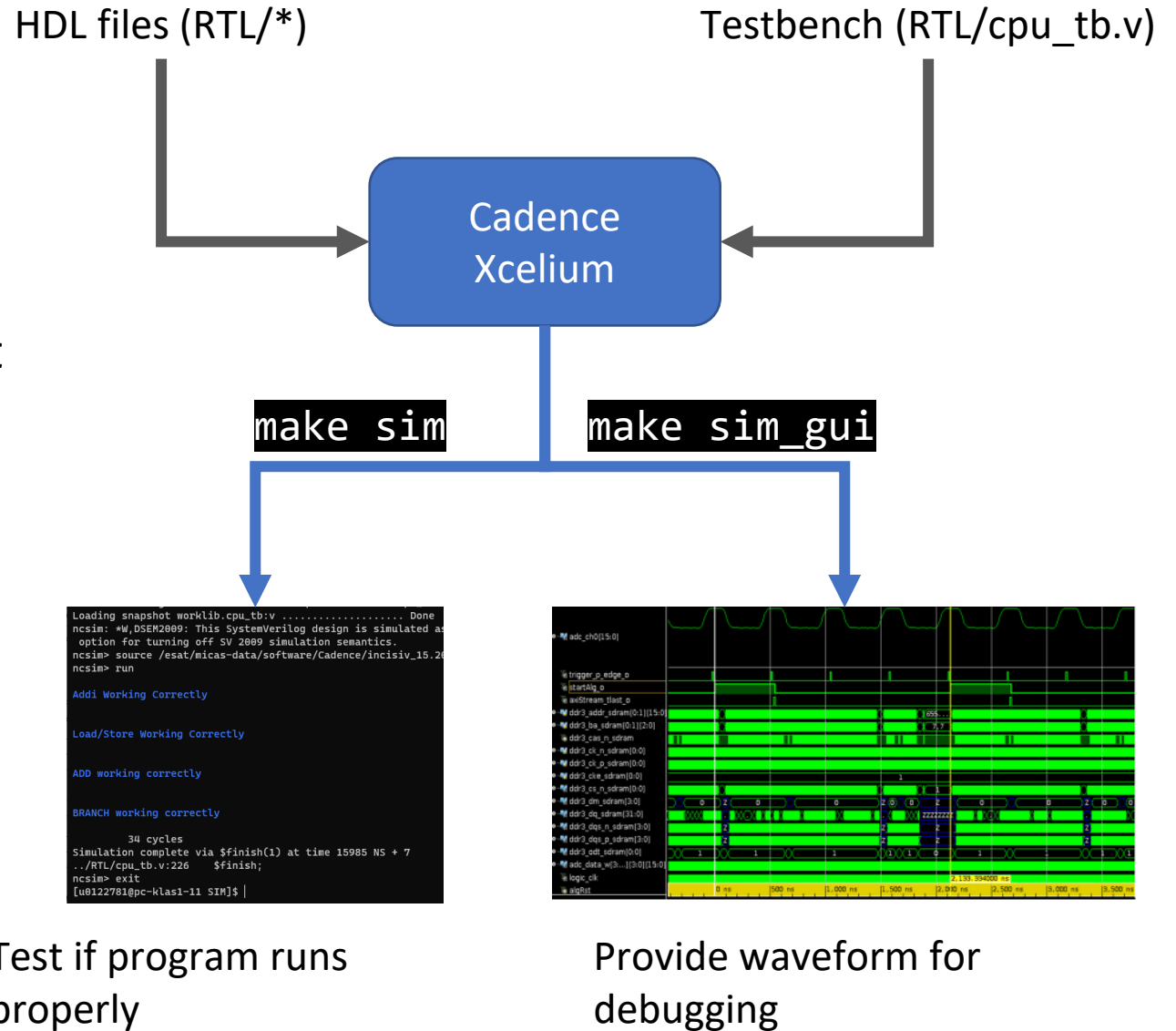
- Make simulation

```
make sim
```

uses `cpu_tb.v`, and the content of `SIM/data/imem_content` and `SIM/data/dmem_content` to verify that the current RTL is able to run the program.

```
make sim_gui
```

same but invokes waveform for debugging (interactive)



Design Browser 1 - SimVision

File Edit View Select Explore Simulation Windows Help

TimeA = 0

fs

Search Times: Value

0 + 0

Scope: All Available Data

cpu_tb

cnt_and_wait

debug_regfile

dut

alu

alu_ctrl

alu_operand_mux

branch_unit

control_unit

data_memory

immediate_extend_u

instruction_memory

program_counter

regfile_data_mux

register_file

load_dmem

load_imem

test_basic

test_mult_1

test_mult_2

Find: String

Show contents: In the signal list area

Objects

Methods

immediate_extended[63:0]

instruction[31:0]

jump

jump_pc[63:0]

mem_2_reg

mem_data[63:0]

mem_read

mem_write

rdata_ext[31:0]

rdata_ext_2[63:0]

reg_dst

reg_write

regfile_rdata_1[63:0]

regfile_rdata_2[63:0]

regfile_waddr[4:0]

regfile_wdata[63:0]

ren_ext

ren_ext_2

updated_pc[63:0]

wdata_ext[31:0]

wdata_ext_2[63:0]

Value

'd x

'h xxxxxxxx

x

'h xxxxxxxx_xxxxxxxxxx

x

'h xxxxxxxx_xxxxxxxxxx

x

x

'h xxxxxxxx

'h xxxxxxxx_xxxxxxxxxx

x

x

'h xxxxxxxx_xxxxxxxxxx

'h xxxxxxxx_xxxxxxxxxx

'h zz

'h xxxxxxxx_xxxxxxxxxx

x

x

'h xxxxxxxx_xxxxxxxxxx

'h xxxxxxxx

'h xxxxxxxx_xxxxxxxxxx

x

x

x

instruction[31:0]

Bookmark This View

Expand Options

Expand

Cut

Copy

Paste

Delete

Trace Drivers in Sidebar

Trace Loads in Sidebar

Send to Waveform Window

Send to Watch Window

Send to Source Browser

Send to Schematic Tracer

Send to Memory Viewer

Send to Design File Search

Send to New

Break on Change

Create Force...

Release Force

CA_Exercises > Verilog > cpu_tb.v

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

);

1. Navigate to the correct module instance in the Design Browser

2. Right click on the wire you want to inspect

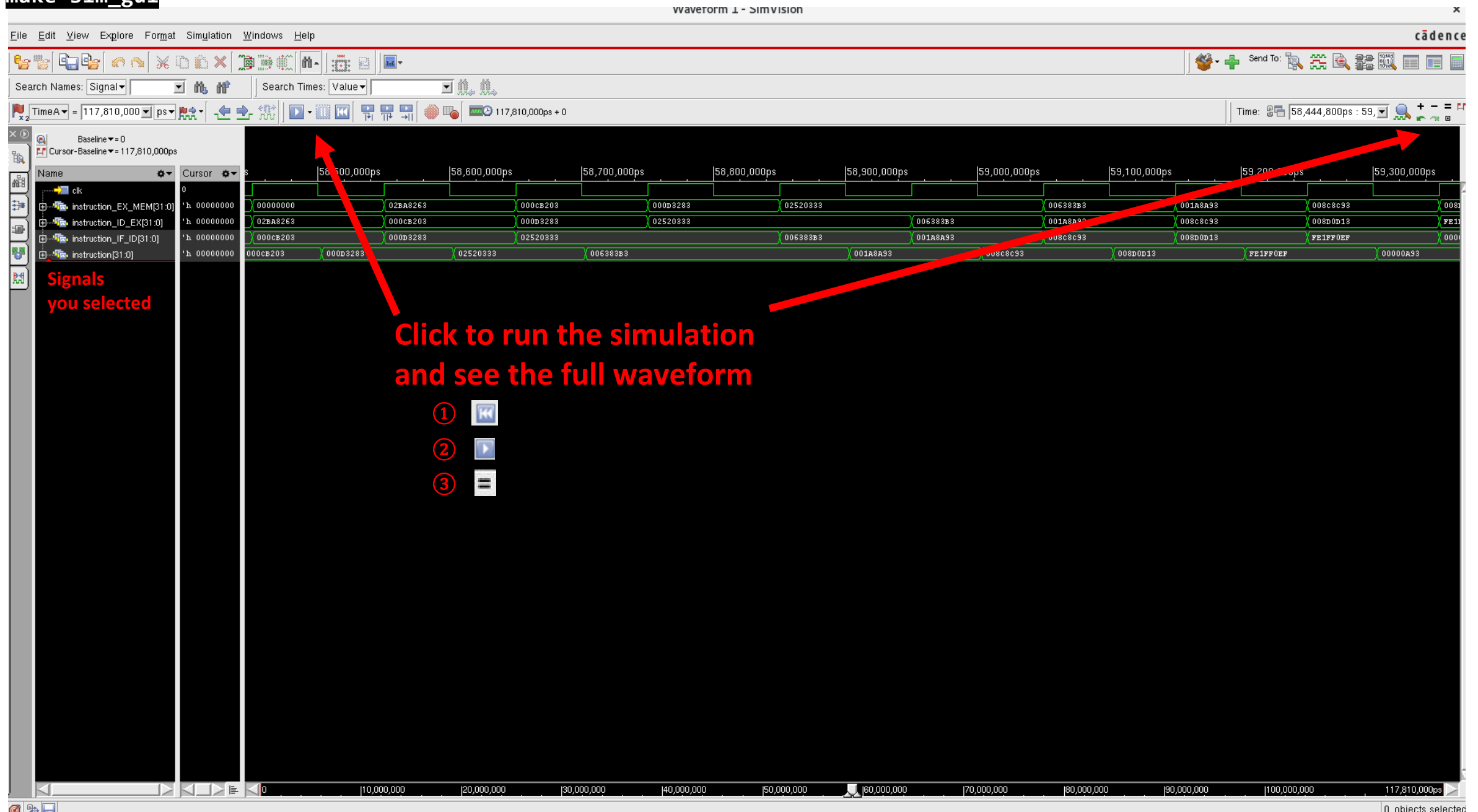
3. Send to Waveform Window

Design hierarchy (module instance names)

simulator::cpu_tb.dut.instruction[31:0]

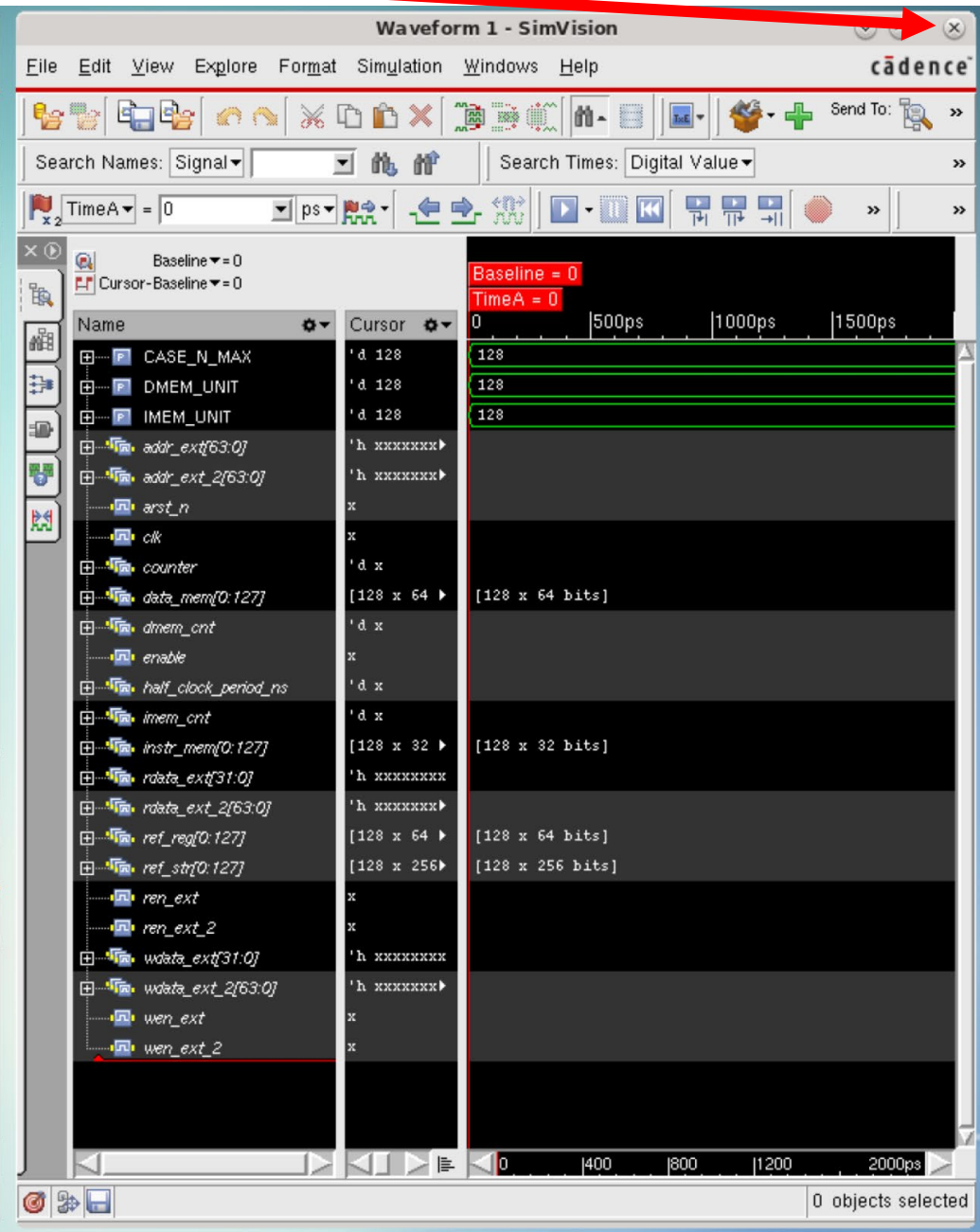
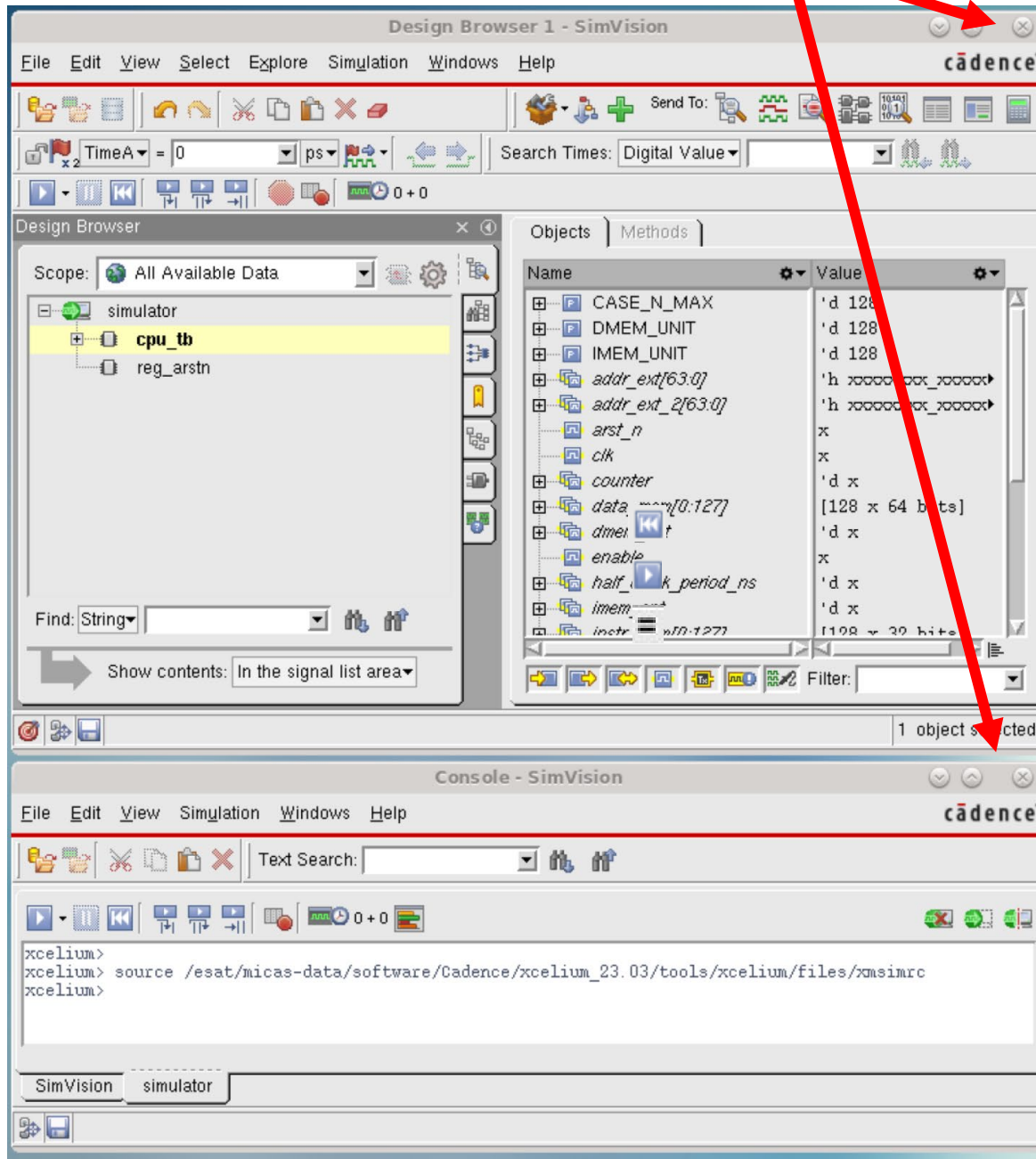
1 object selected

14

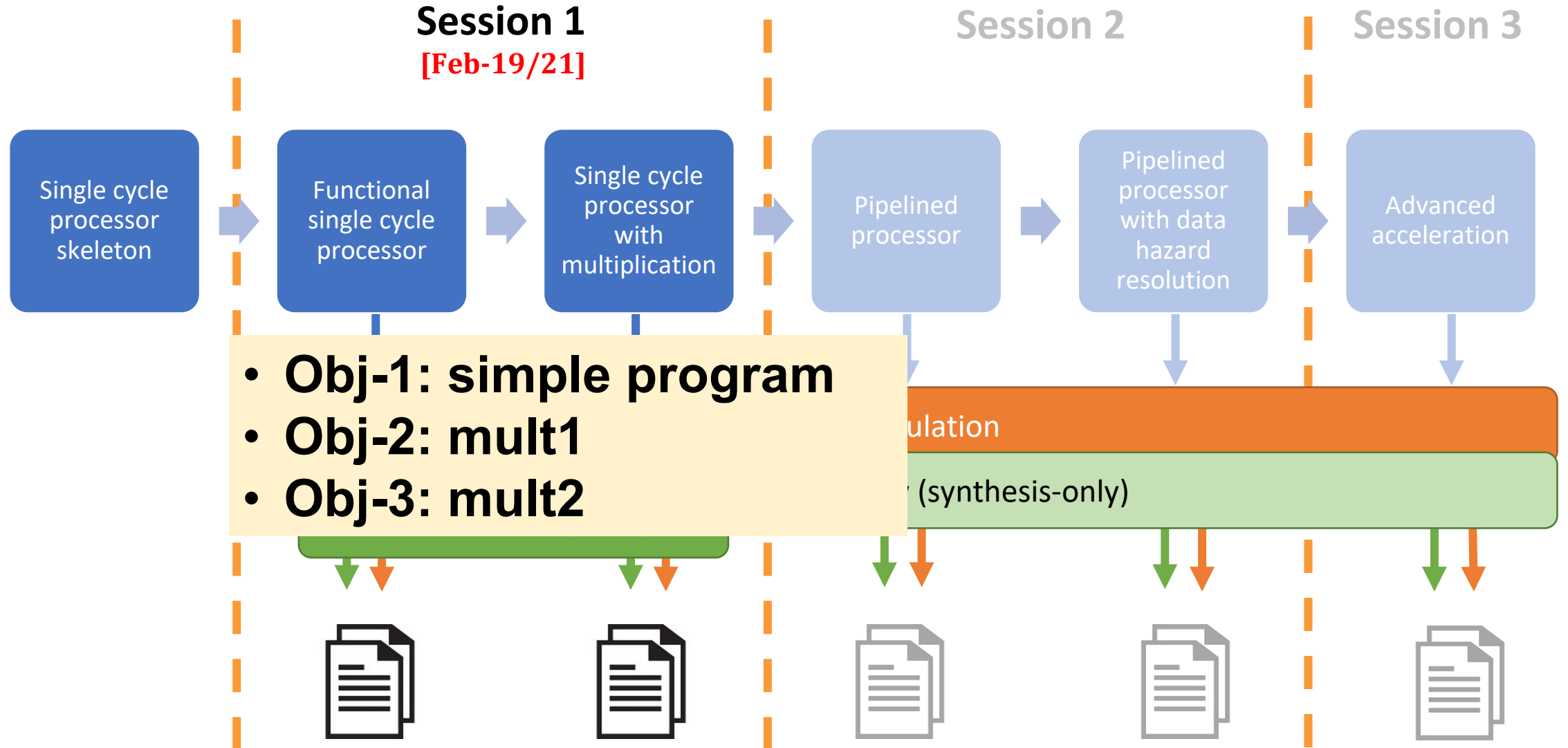


make sim_gui

Terminate: close all three GUI windows



Today's session: Single cycle processor



Obj-1: simple program

Single cycle
processor
skeleton



Functional
single cycle
processor

- Functional single-cycle processor
 - Complete the `RTL/control_unit.v` to support BEQ, JUMP, LD, SD, ADDI and R-type ALU instructions (ALU_R)
 - Follow the `RUN CYCLE-ACCURATE SIMULATION` in `session_guide.pdf`
 - Run **SIMPLE_PROGRAM**
 - Get to folder `/SIM`
 - Overwrite the instruction & data memory for the testbench to imem and dmem

```
cat data/testcode_m/simpleprogram_imem_content.txt > data/imem_content.txt
```

```
cat data/testcode_m/simpleprogram_dmem_content.txt > data/dmem_content.txt
```
 - Run simulation with `make sim` or `make all`

Simple Program

```
addi x8, x0, 7
addi x9, x8, 2
sd x9, 0(x0)
ld x17, 0(x0)
ld x18, 8(x0)
add x19, x17, x18
beq x9, x17, FINAL
add x20, x8, x9
FINAL: add x20, x18, x19
sll x21, x18, x8
STOP
```

STOP is a "fake" instruction used for the `cpu_tb` to recognize the end of the program (Check line 305 of `cpu_tb.v`)

Obj-2: mult1

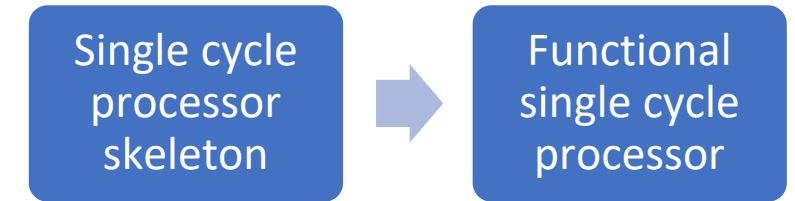
- Multiplication by addition loops

Algorithm 1:

1. $Acc = 0$
2. $N = operand1$
3. Traverse each bit of operand2 (LSB \rightarrow MSB)
 - a. If the bit of operand2 is 1, accumulate. $Acc = Acc + N$.
 - b. If the bit is 0, $Acc = Acc$.
 - c. Left shift N by 1 position.

			1	1	0
			1	0	1
<hr/>					
			1	1	0
		0	0	0	
	1	1	0		
<hr/>					
	1	1	1	1	0

- **MULT1**
 - Multiply adjacent operand in pairs (5 multiplications) using Algorithm 1
 - Sums results together
- Run **MULT1** with your processor



```
# init
addi x16, x0, 80
addi x8, x0, 0
addi x9, x0, 0
addi x10, x0, 1

# Looping over operands
M1:ld x17, 0(x8)
ld x18, 8(x8)
addi x23, x0, 64
addi x19, x0, 1
addi x20, x0, 0
add x22, x0, x17
```

- x16: Total size of data memory
- x8: Data Memory Pointer
- x23: Iteration over the number of bits
- x20: Accumulated result of each multiplication
- x9: Final result

```
# Multiplication
LOOP_0: and x21, x18, x19
beq x21, x0, SHIFTING_0
add x20, x20, x22
SHIFTING_0: sll x22, x22, x10
sll x19, x19, x10
addi x23, x23, -1
beq x23, x0, M2
jal LOOP_0
```

```
# Summing results
M2: add x9, x9, x20
    addi x16, x16, -16
    addi x8, x8, 16
    beq x16, x0, FINISH
    jal M1
FINISH:
```

Obj-3: mult2

- Single cycle processor with multiplication support
 - Modify your `control_unit.v`, `alu_control.v` and `alu.v` to support the *mul* instruction
- Run **MULT2** with your processor

Functional
single cycle
processor



Single cycle
processor
with
multiplication

• MULT2

```
# load operands
ld x8, 0(x0)
ld x9, 8(x0)
ld x10, 16(x0)
ld x11, 24(x0)
ld x12, 32(x0)
ld x13, 40(x0)
ld x14, 48(x0)
ld x15, 56(x0)
ld x16, 64(x0)
ld x17, 72(x0)

# multiplication
mul x18, x8, x9
mul x19, x10, x11
mul x20, x12, x13
mul x21, x14, x15
mul x22, x16, x17

#sums
addi x23, x0, 0
nop
nop
nop
add x23, x23, x18
nop
nop
nop
add x23, x23, x19
nop
nop
nop
add x23, x23, x20
nop
nop
nop
add x23, x23, x21
nop
nop
nop
add x23, x23, x22
nop
nop
nop
nop
```

Obj-3: hints

Single cycle
processor
skeleton



Functional
single cycle
processor

- RISC-V **MUL** operation needs more control bits than provided in `alu_control.v`
- Refer to the RISC-V ISA and start from the module IO definition.

```
4 module alu_control(  
5+     input wire func7_5,  
6     //  
7     input wire [2:0] func3,  
8     input wire [1:0] alu_op,  
9     output reg [3:0] alu_control  
);
```

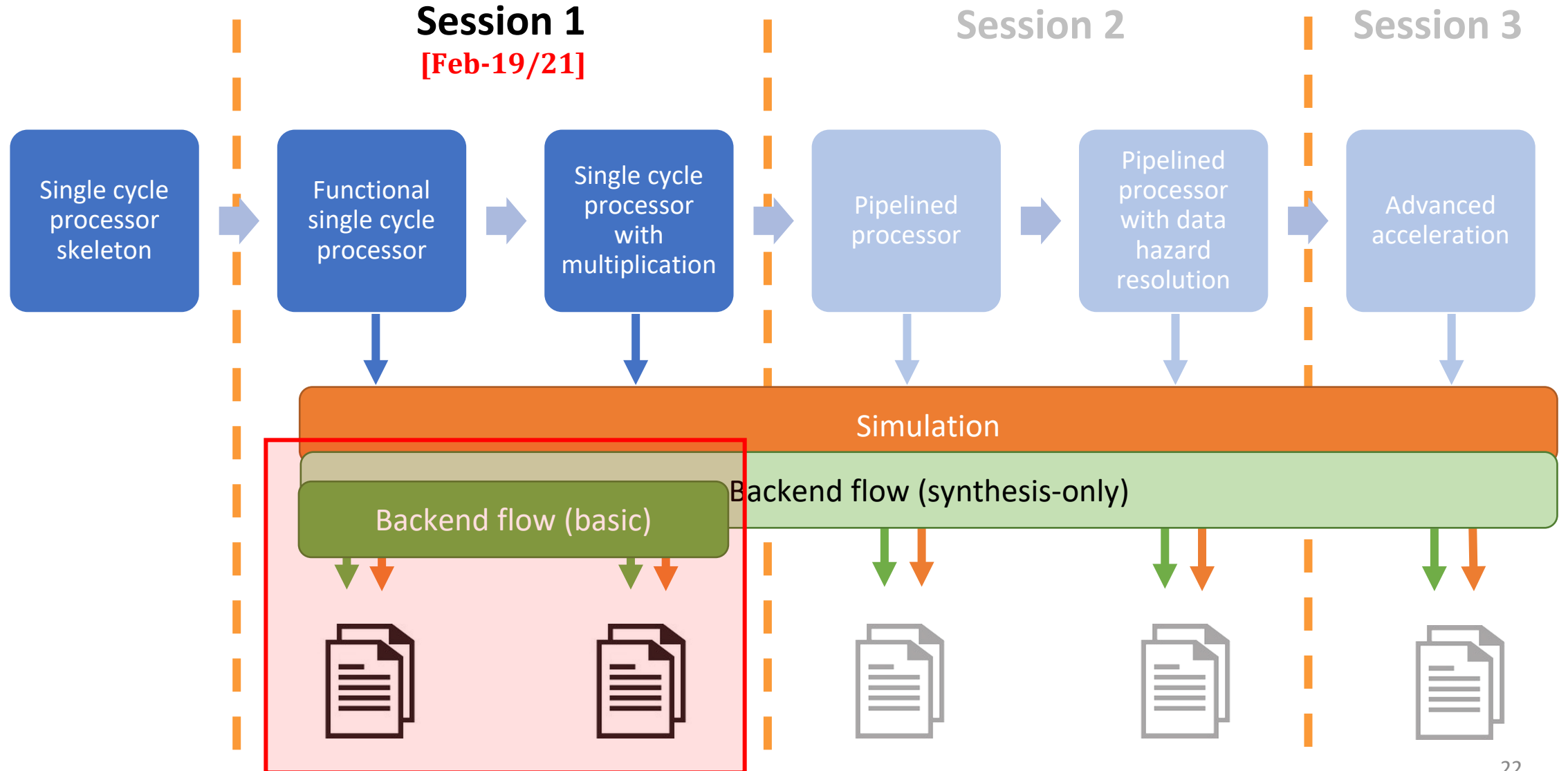
Module Definition

```
alu_control alu_ctrl(  
    .func7_5      (instruction[30]),  
    .func3        (instruction[14:12]),  
    .alu_op       (alu_op),  
    .alu_control  (alu_control)  
);
```

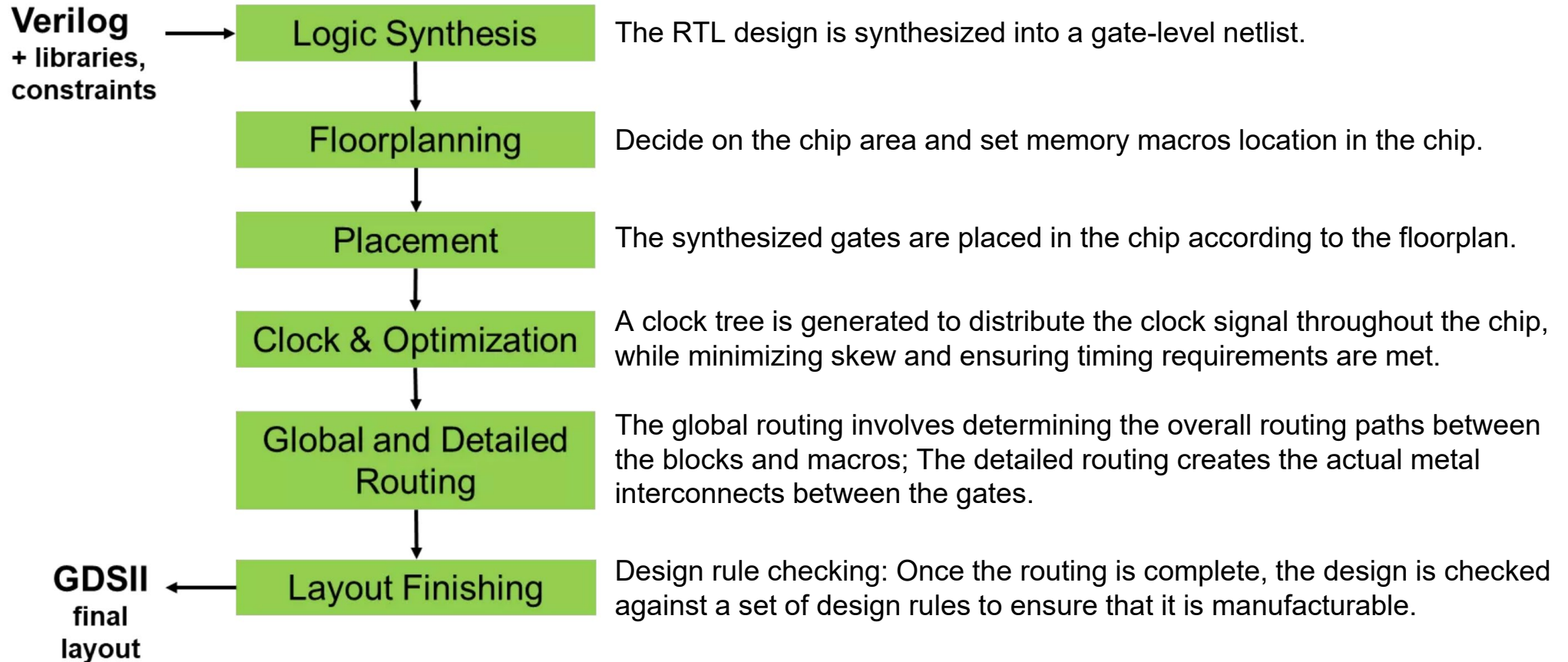
Module Instance

- Do not forget to sync with the module instance (in `cpu.v`) after modification.

Today's session: Single cycle processor



Digital Backend Flow

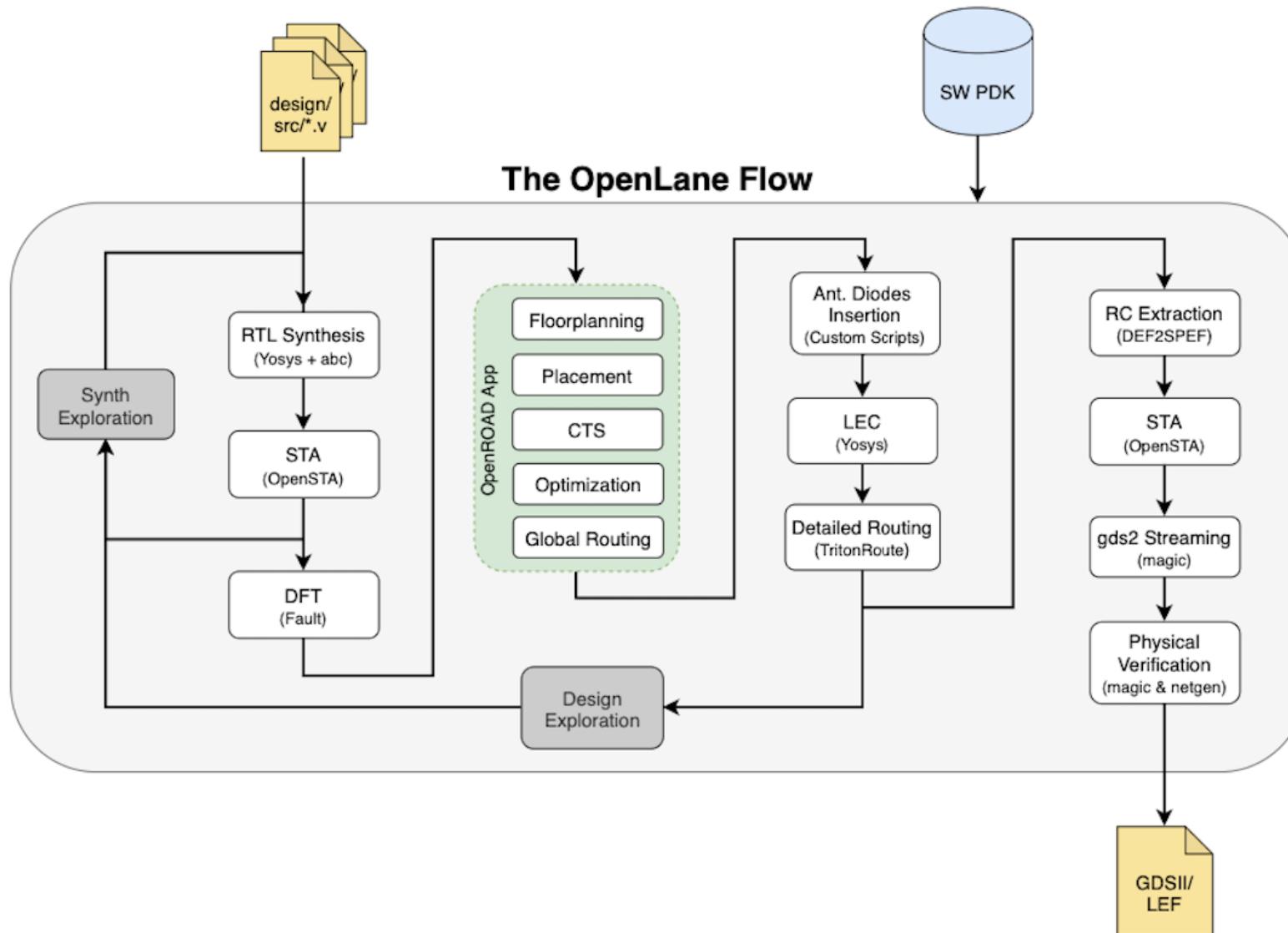


Backend Flow in this exercise

Three key components:

1. Flow tool: OpenLane
2. PDK library: SKY130
3. Memory macros: OpenRAM

1. Automated Backend Flow: OpenLane



OpenLane is an **automated RTL to GDSII flow** based on several components.

2. Open-source PDK: SKY130



PDK: Process Design Kit - information about the process technology

PDK includes

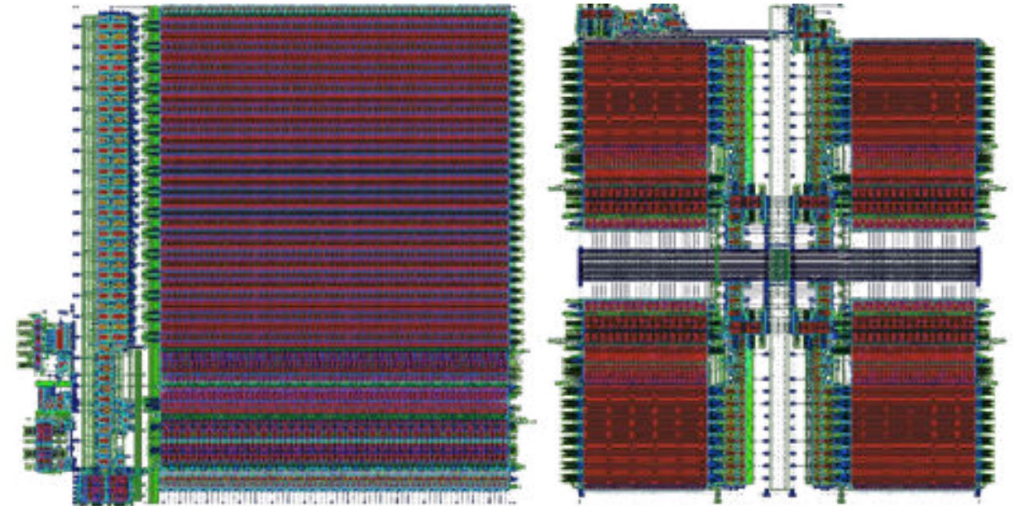
- Process documentation
- Device models
- Layout design rules
- Design libraries: Pre-designed circuit blocks, such as standard cells, memories, and IO cells
- ...

3. Open-source SRAM Compiler: OpenRAM



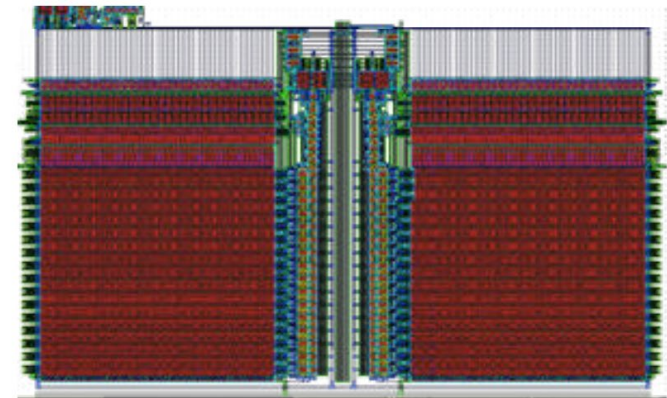
Create SRAM for ASIC design:

- **Layout**
- **Netlists**
- **Timing and power models**
- **Placement and routing models**



2 Kb (1 bank x 32 words x 64 bits)

4 Kb (4 banks x 32 words x 32 bits)



16 Kb (2 banks x 128 words x 64 bits)

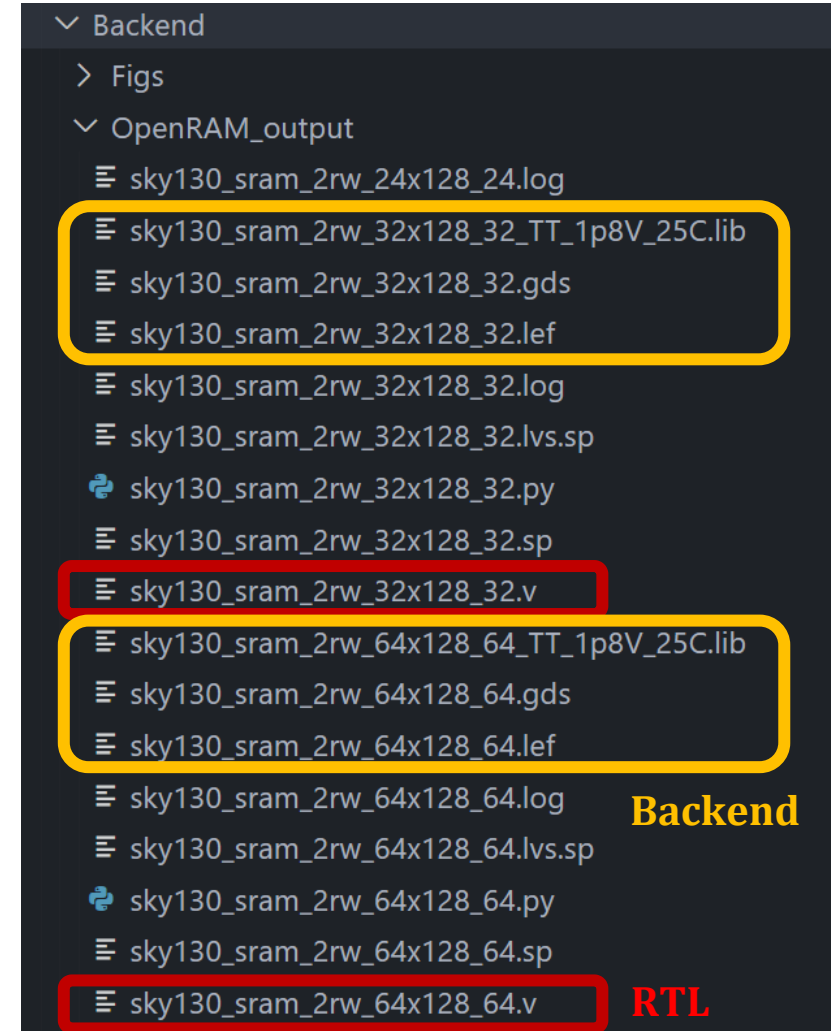
3. Open-source SRAM Compiler: OpenRAM



Create SRAM for ASIC design:

- **Layout**
- **Netlists**
- **Timing and power models**
- **Placement and routing models**

*In the exercise, the memory macros used by instruction memory and data memory are **already generated for you**.*



Environment settings + Backend tools installation

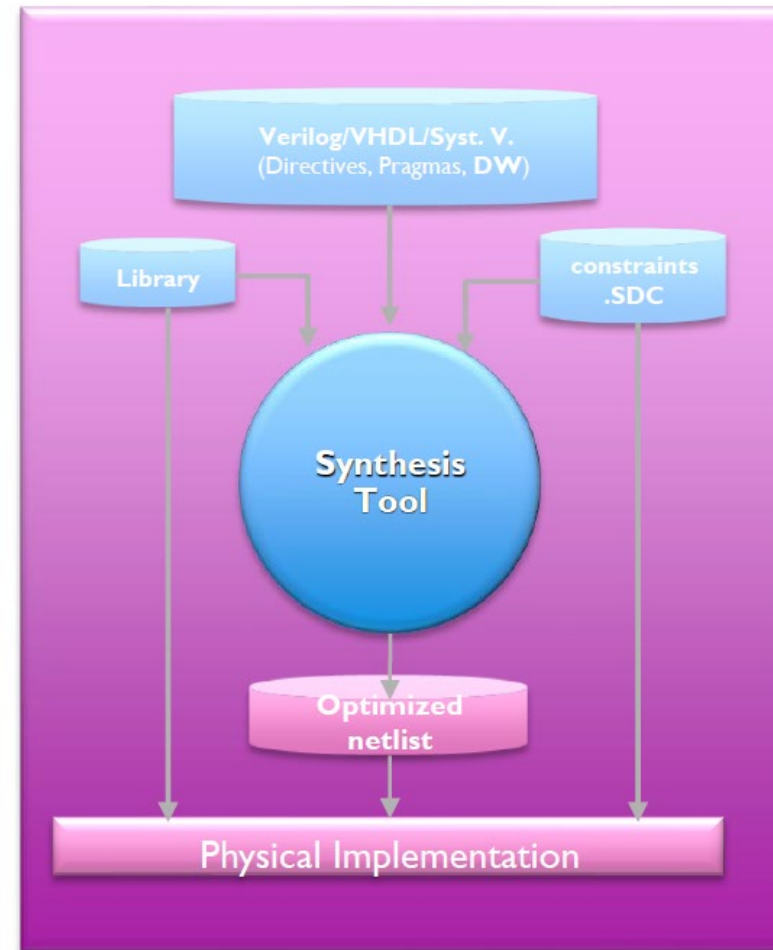
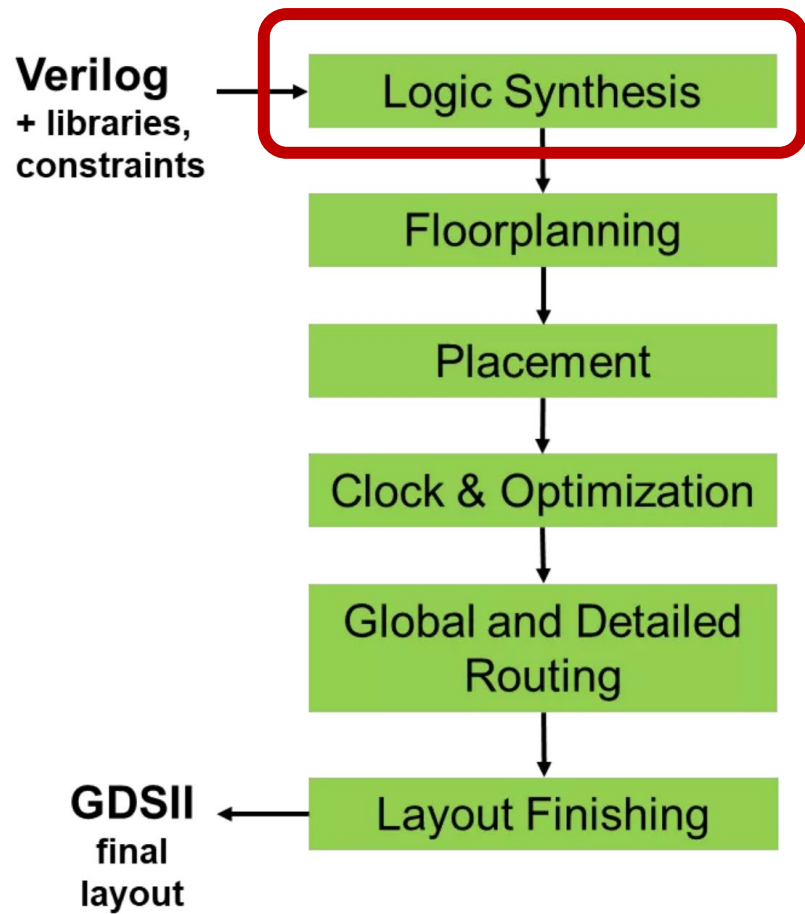
Backend tools installation:

- Enter the folder: Backend/
- In the folder, run the command: `source setup.sh`. This should start installing the backend tools and jupyter by micromamba. (This step takes about 5 min)
- In the terminal, type the command `jupyter-lab`. This should introduce you to the website of jupyter notebook.

PS1: every time you want to use the backend tools and jupyter, please run the command `source setup.sh` to check the integrity of the toolchain.

PS2: if you want to use jupyter-lab at home, see this manual on how to open *jupyter notebook remotely*. (*CA_Documents/Jupyter_VNC_Manual.pdf*)

Synthesis: a critical step in the backend flow



- Pass the **RUN CYCLE-ACCURATE SIMULATION** and prepare your source code in RTL_SOLUTION* folders.
- Follow the **RUN BACKEND FLOW** in **session_guide.pdf** and the instructions in **CA_RISCV.ipynb**
- In **Session 1** we only do the **synthesis stage**.

Today's session: task summary

With **session_guide.pdf**

- Study the **RUN CYCLE-ACCURATE SIMULATION** and **RUN BACKEND FLOW**
- Follow the **TASKS TO BE DONE** and fill in the **report.docx**

Copy-paste your finished **/RTL/*.v** into the SOLUTION folders.

- **Obj-1&2** → **RTL_SOLUTION1_simple_program_and_MULT1**
- **Obj-3** → **RTL_SOLUTION2_multiplication_support_MULT2**

- **Note:**

1. We use universal test patterns for fair grading.
2. **Do not modify cpu_tb.v & sky130_sram_2rw.v**
3. **Do not modify *mem_content.txt**