Computer Architectures
Session 2

Pipelined implementation of microprocessor

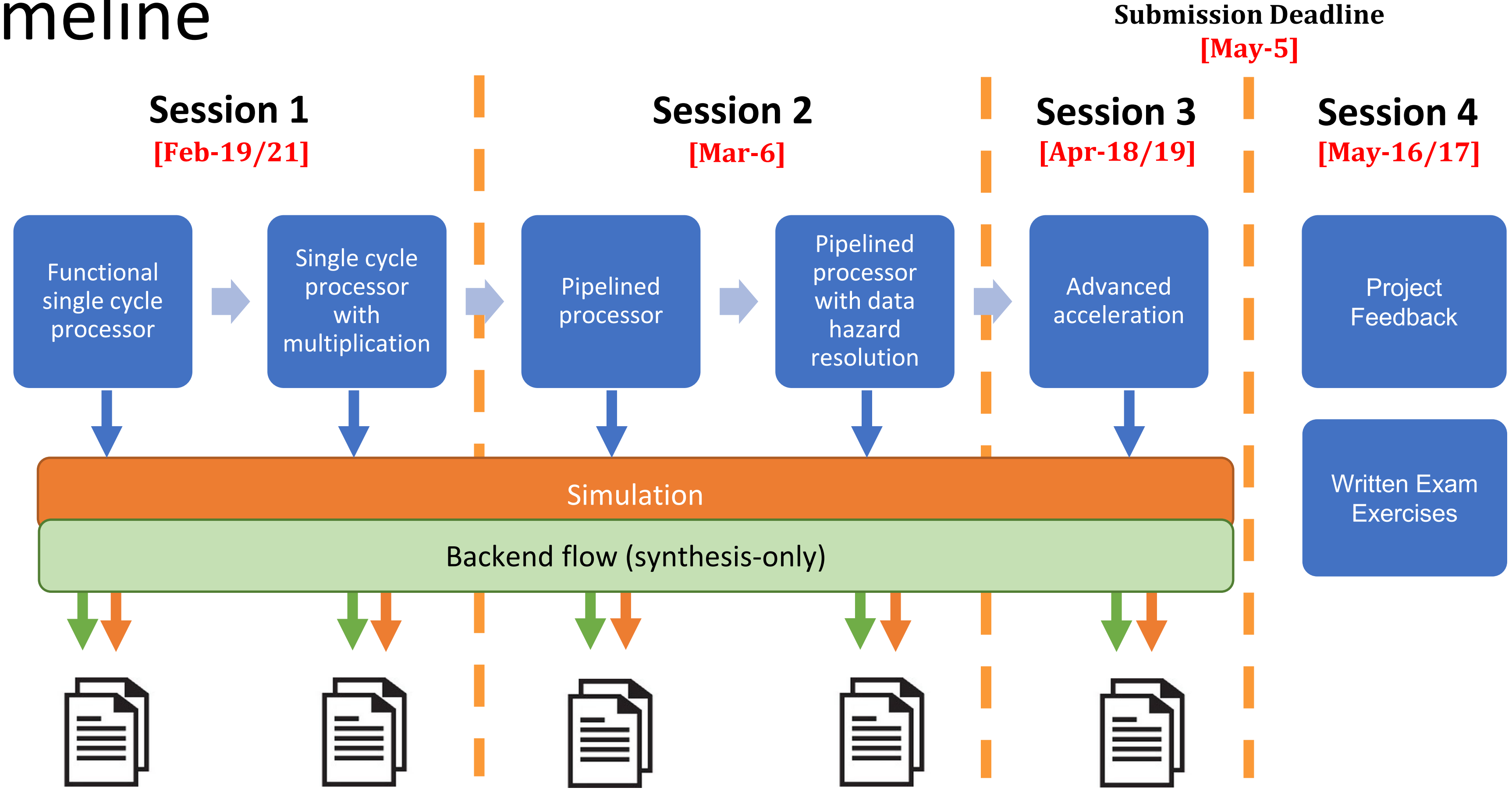
&

Digital backend flow

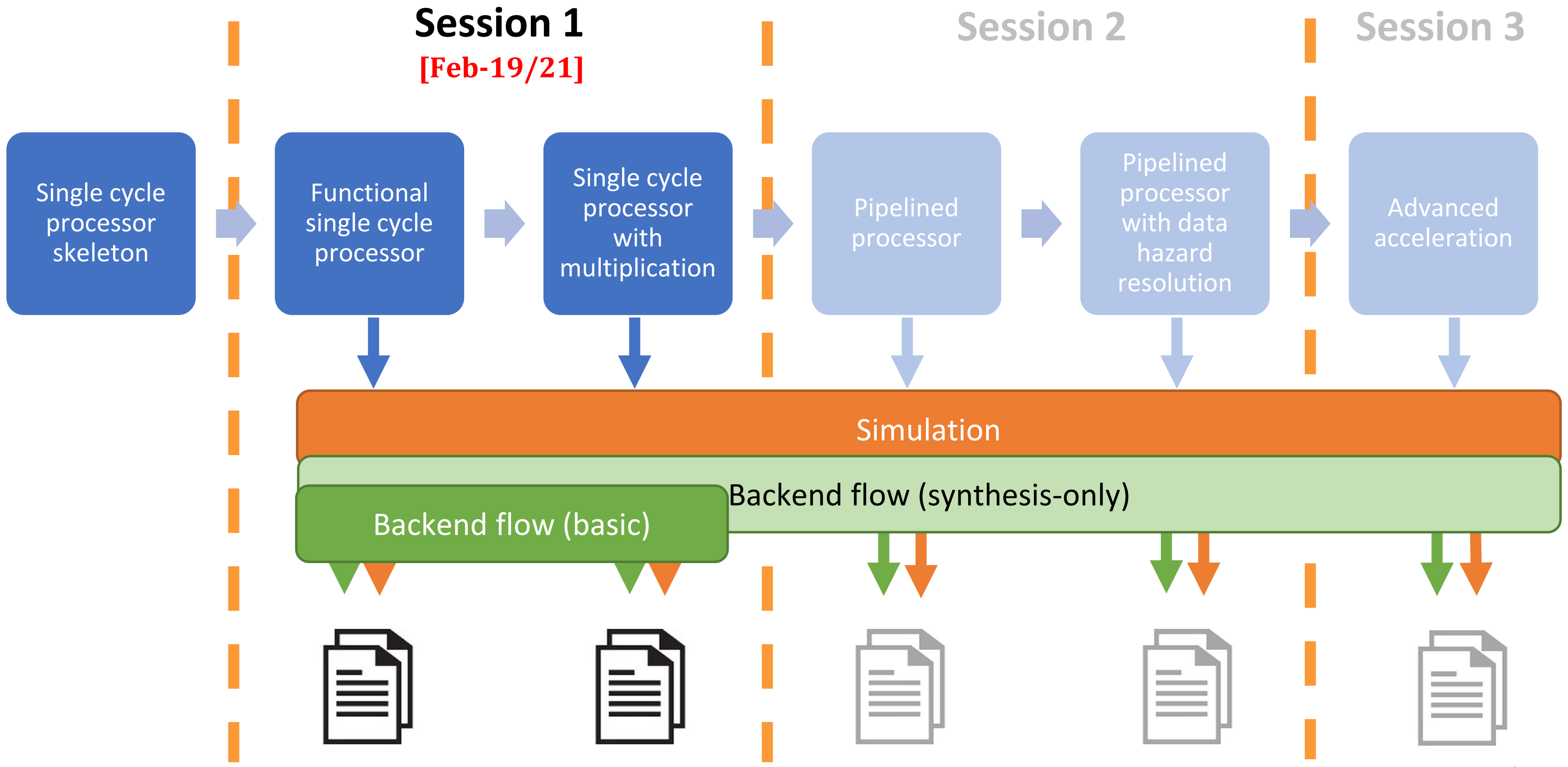
TAs :

Jun Yin (jun.yin@kuleuven.be)
Yuanyang Guo (yuanyang.guo@imec.be)
Xiaoling Yi (xiaoling.yi@kuleuven.be)
Yunzhu Chen (yunzhu.chen@imec.be)

Timeline


Submission Deadline
[May-5]
Session 1
[Feb-19/21]
Session 2
[Mar-6]
Session 3
[Apr-18/19]
Session 4
[May-16/17]
Functional single cycle processor
Single cycle processor with multiplication
Pipelined processor
Pipelined processor with data hazard resolution
Advanced acceleration
Project Feedback
Written Exam Exercises
Simulation
Backend flow (synthesis-only)

Last session recap


Session 1
[Feb-19/21]
Session 2
Session 3
Single cycle processor skeleton
Functional single cycle processor
Single cycle processor with multiplication
Pipelined processor
Pipelined processor with data hazard resolution
Advanced acceleration
Simulation
Backend flow (synthesis-only)
Backend flow (basic)

Last session recap

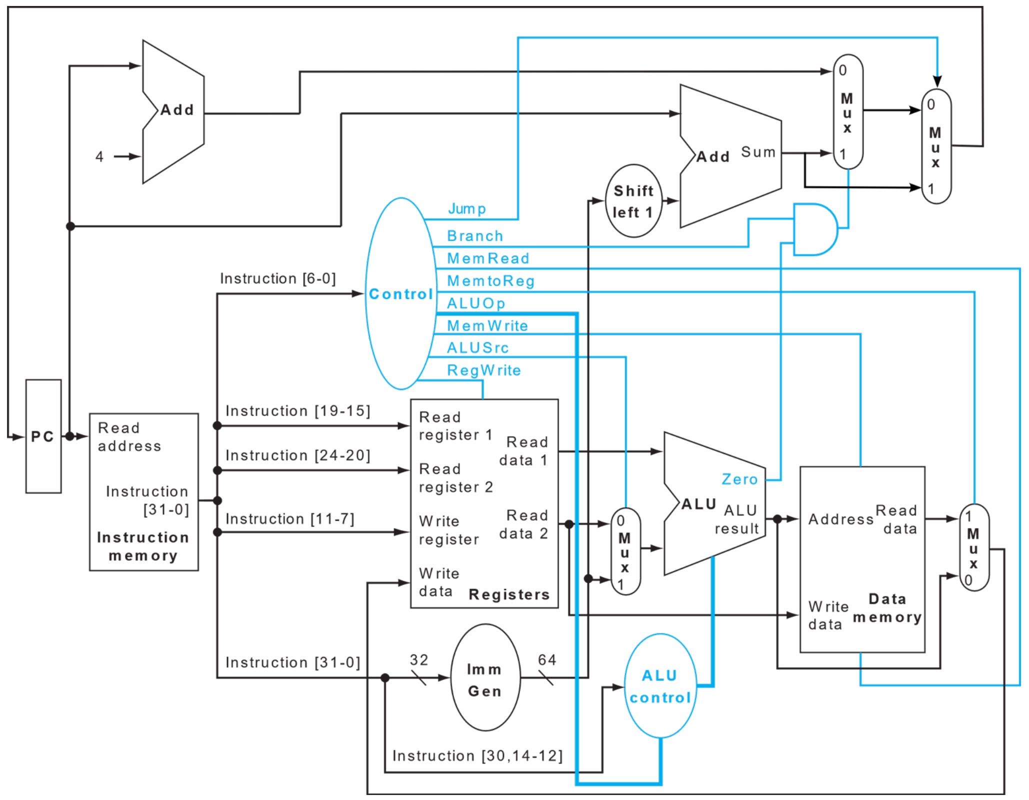
Single Cycle Processor

✓ Simple_program

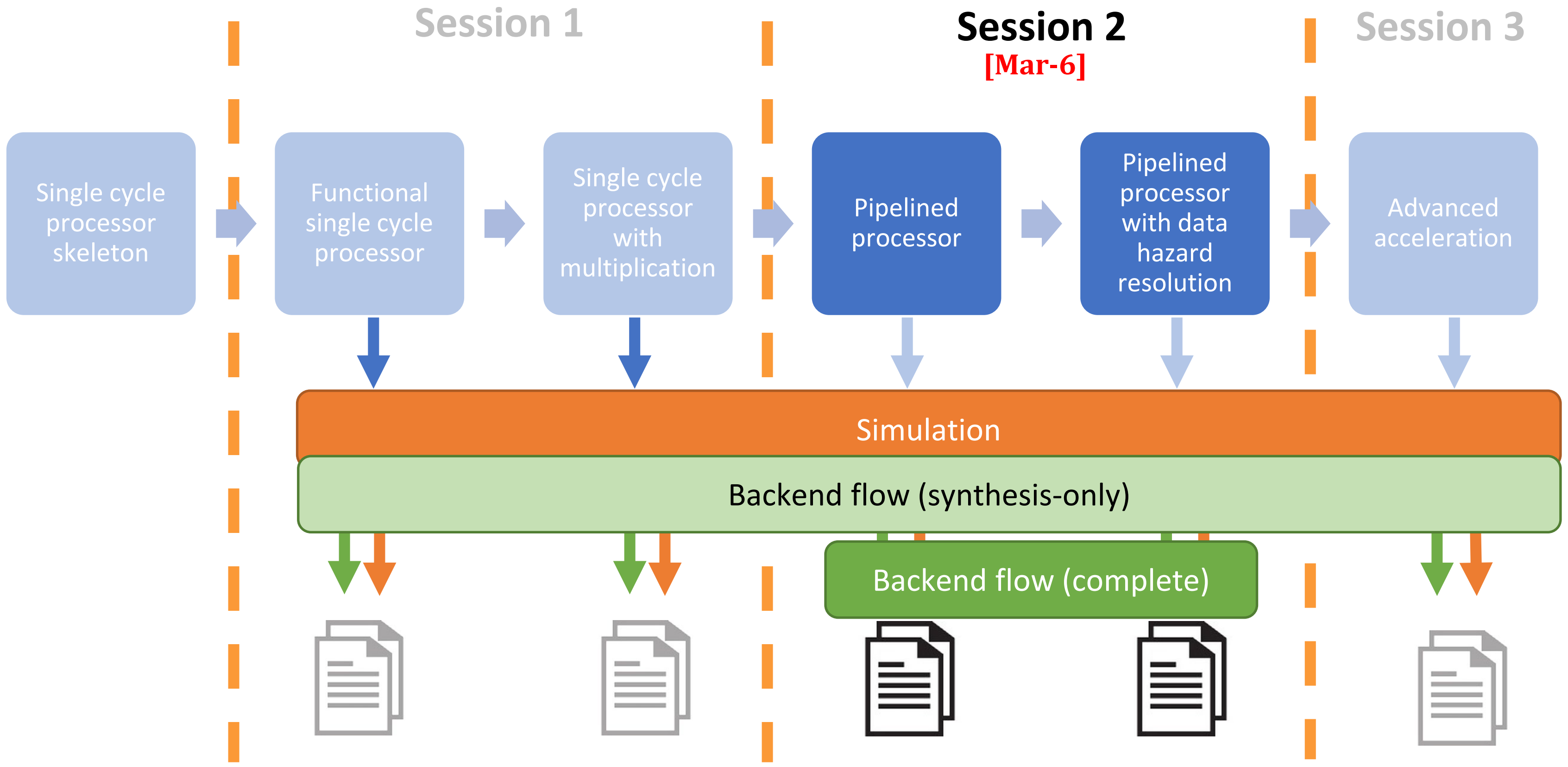
✓ Mult1

✓ Mult2

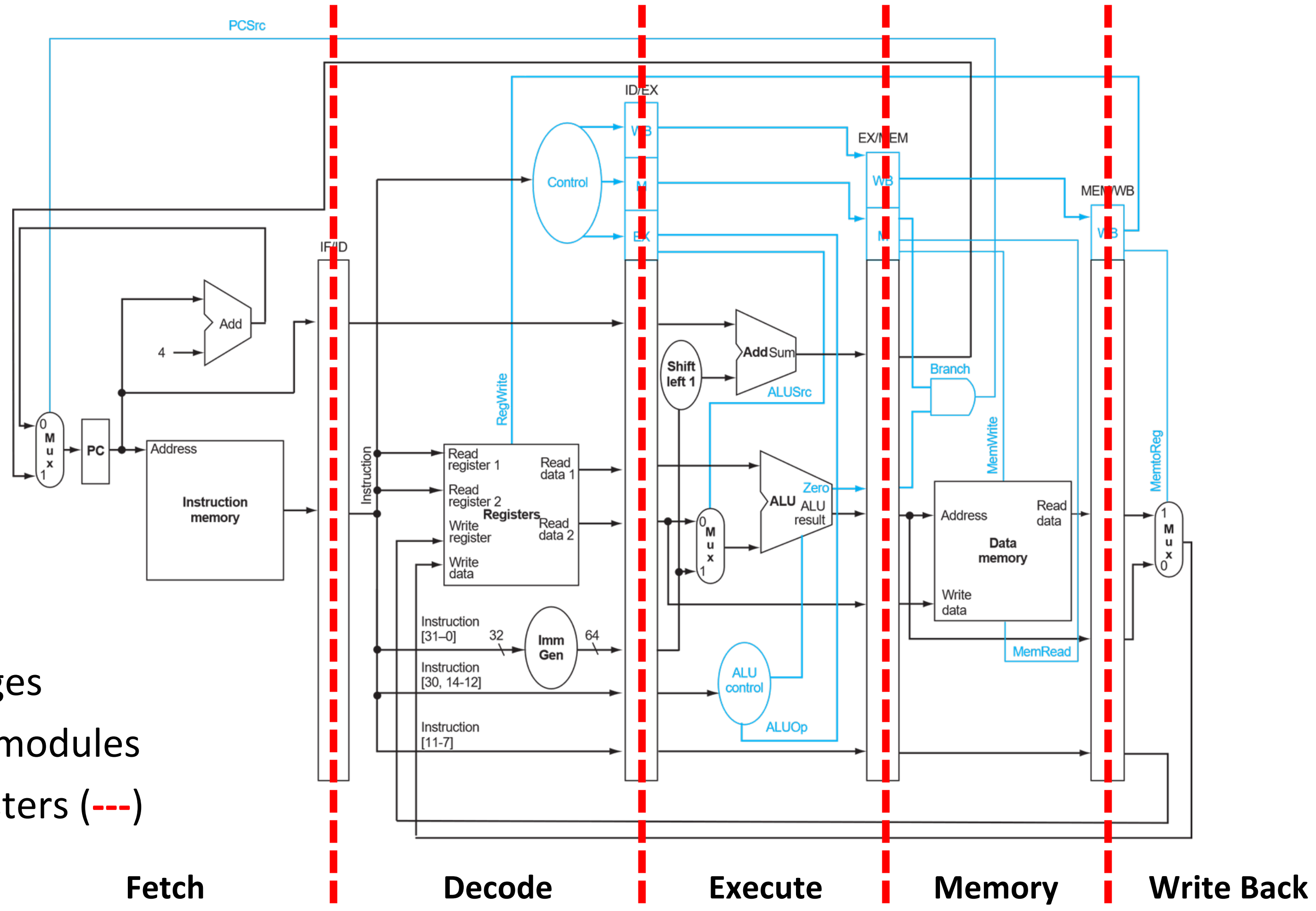
Prerequisite for this session!



Today's session: Pipelined processor

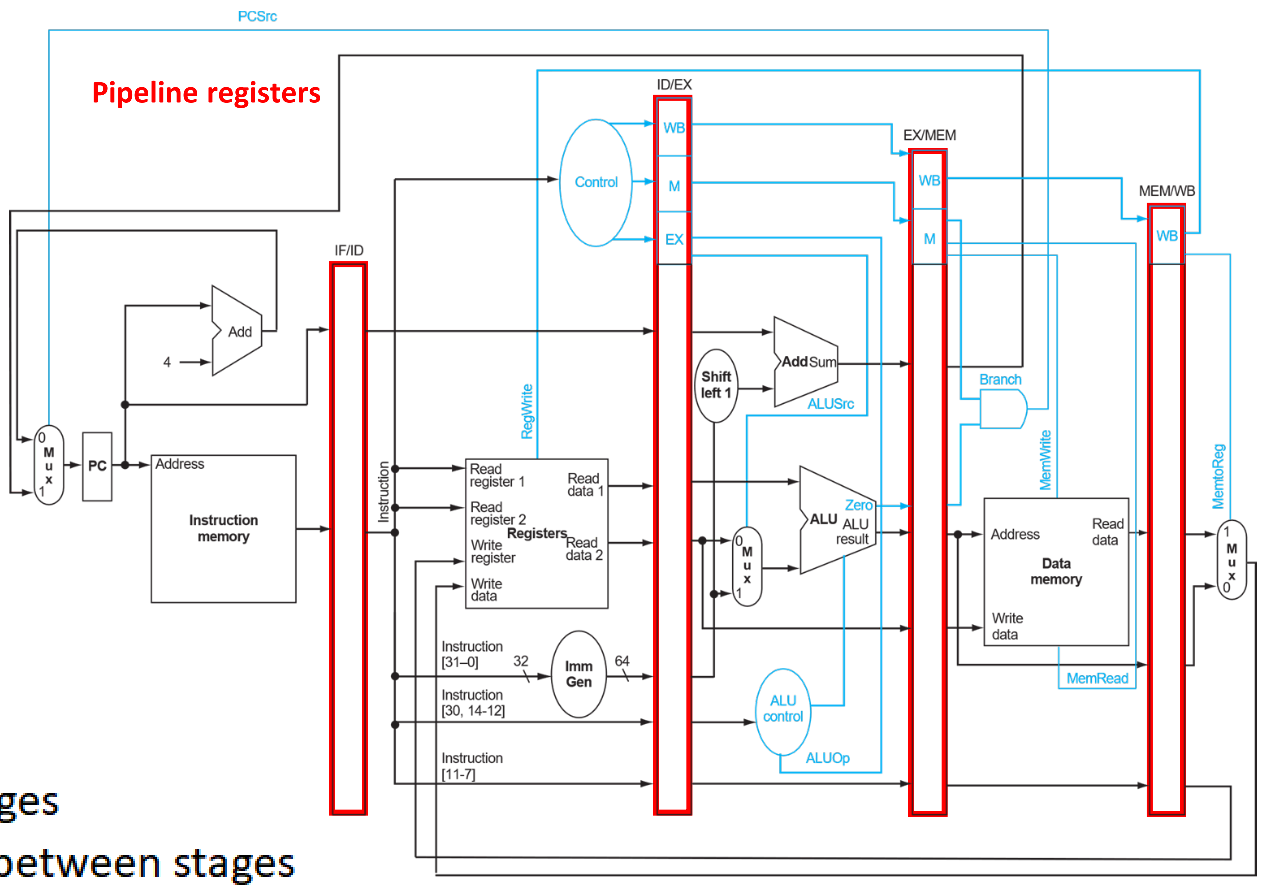

Session 1
Session 2
[Mar-6]
Session 3
Single cycle processor skeleton
Functional single cycle processor
Single cycle processor with multiplication
Pipelined processor
Pipelined processor with data hazard resolution
Advanced acceleration
Simulation
Backend flow (synthesis-only)
Backend flow (complete)

5-stage Pipelining



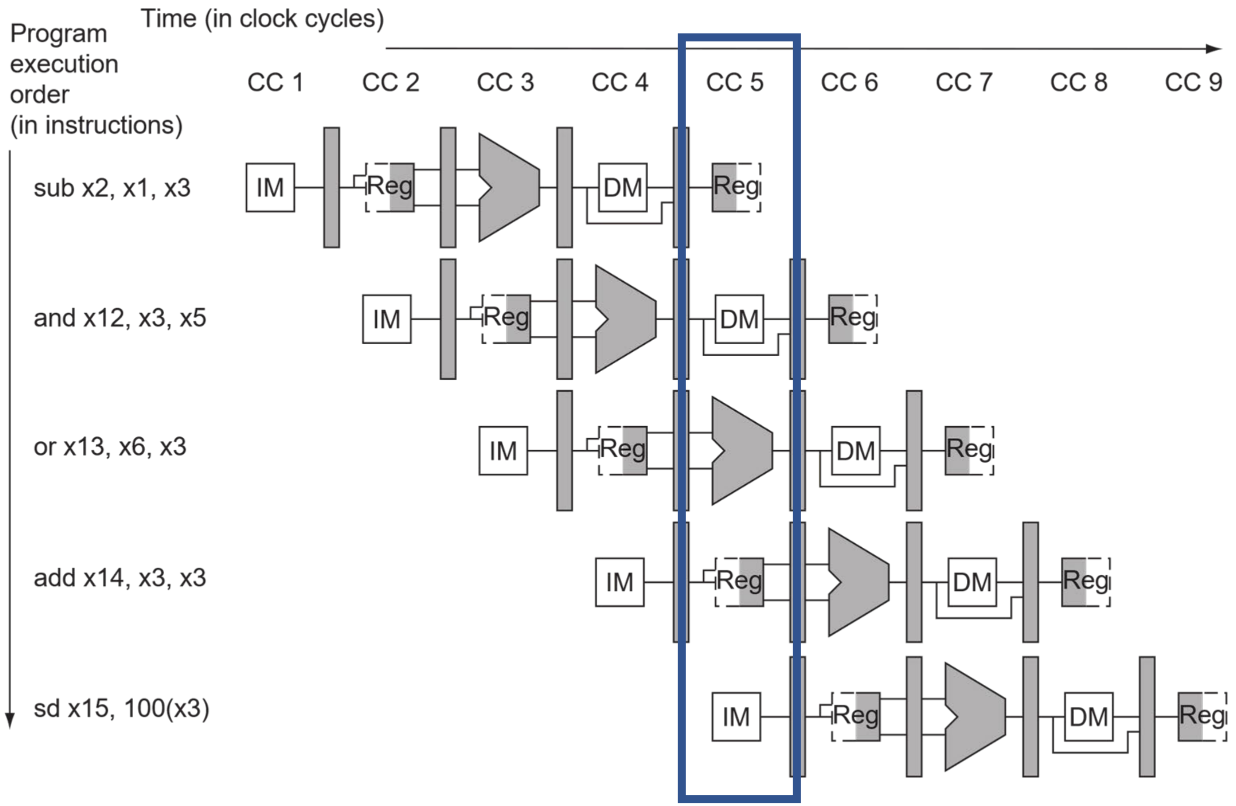
- Define pipeline stages
- Allocate processor modules
- Insert pipeline registers (---)

5-stage Pipelining



- Pipeline registers
  - Separate the stages
  - Transfer signals between stages

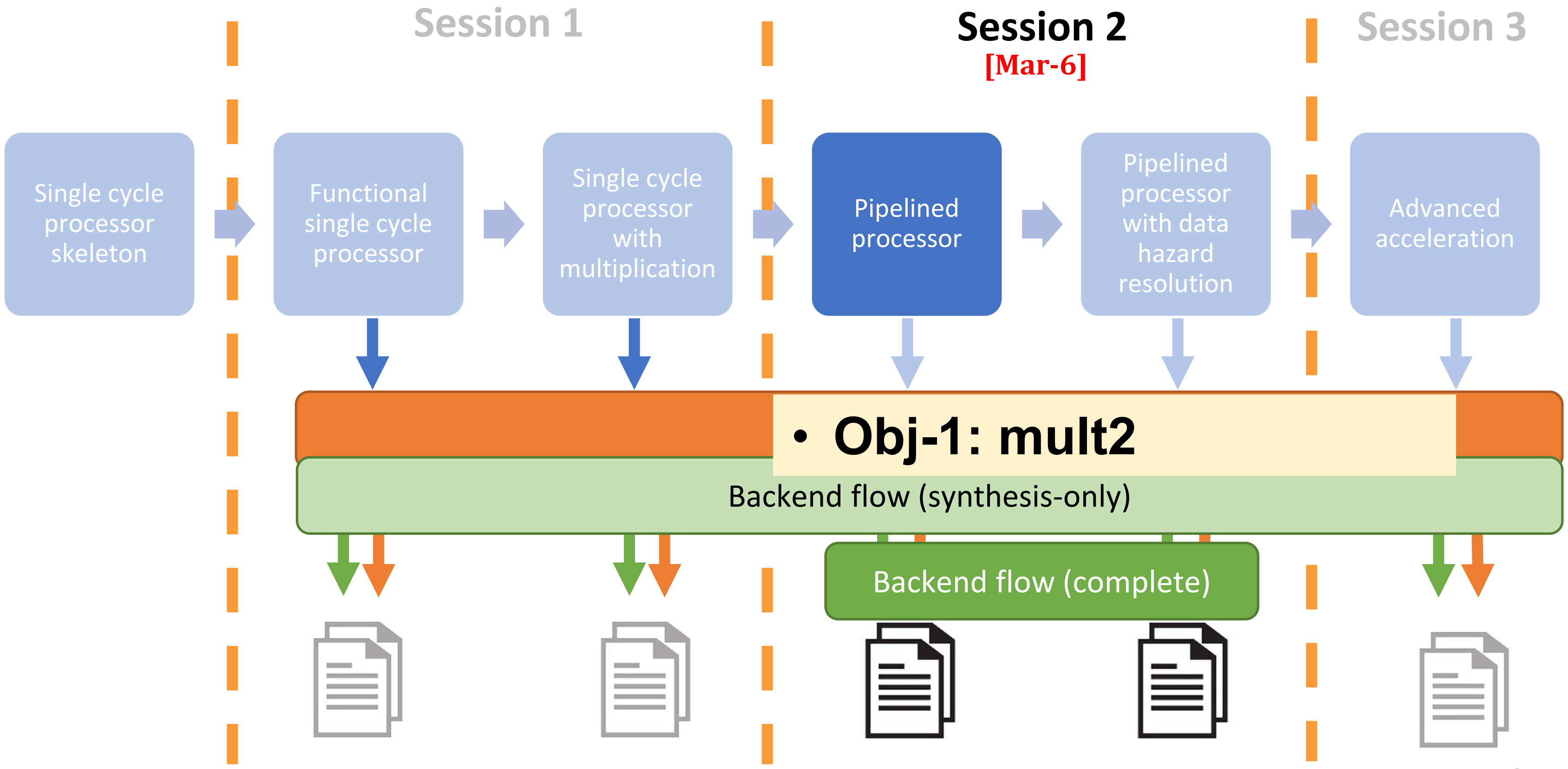
5-stage Pipelining



At clock cycle 5:

- Inst 1 -> write back
- Inst 2 -> data
- Inst 3 -> execute
- Inst 4 -> decode
- Inst 5 -> fetch

Today's session: Pipelined processor


Session 1
Session 2
[Mar-6]
Session 3
Single cycle processor skeleton
Functional single cycle processor
Single cycle processor with multiplication
Pipelined processor
Pipelined processor with data hazard resolution
Advanced acceleration
Backend flow (synthesis-only)
Backend flow (complete)

- **Obj-1: mult2**

Obj-1: mult2

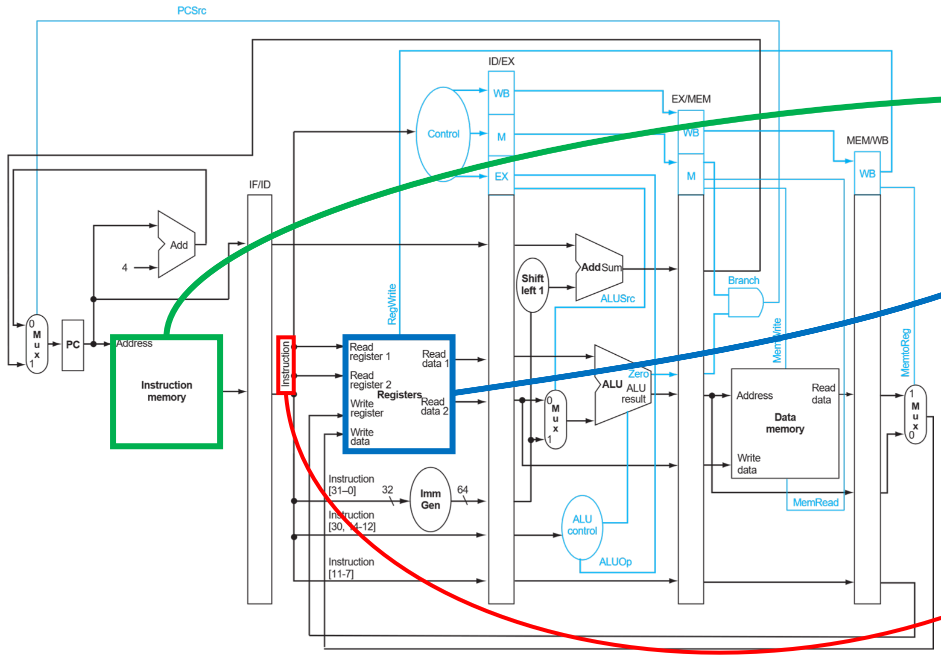
- MULT2

Pipelined processor

- Pipelined processor
  - Introduce the pipeline registers to **cpu.v** to support pipelined execution
  - Run **MULT2**
  - Run backend flow, track the timing and area change along the steps.
    - Synthesis -> Floorplan -> Signoff
  - Complete the report

```
# 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
```

Implementation Method



Stage	IF	ID	EXE	MEM	WB
Data Path resources	instruction_ memory	register_file			

Pipe. Reg.	IF/ID	ID/EXE	EXE/MEM	MEM/WB
Signals	instruction			

You can find this helper table in TASKS TO BE DONE of ***session_guide.pdf***

EXAMPLE

Implementation Method

Stage	IF	ID	EXE	MEM	WB
Data Path resources	Instruction_memory	register_file			

Pipe. Reg.	IF/ID	ID/EXE	EXE/MEM	MEM/WB
Signals	Instruction_IF_ID			Instruction_MEM_WB

You can find this helper table in TASKS TO BE DONE of ***session_guide.pdf***

EXAMPLE

Coding Example (cpu.v)

```
// IF STAGE
wire [       31:0] instruction, instruction_IF_ID, instruction_MEM_WB;

sram_BW32 #(
   .ADDR_W(9 ),
   .DATA_W(32)
) instruction_memory(
   .clk      (clk          ),
   .addr     (current_pc   ),
   .wen      (1'b0         ),
   .ren      (1'b1         ),
   .wdata    (32'b0        ),
   .rdata    (instruction  ), // Wiring component
   .addr_ext (addr_ext     ), // to appropriate stage
   .wen_ext  (wen_ext      ),
   .ren_ext  (ren_ext      ),
   .wdata_ext(wdata_ext    ),
   .rdata_ext(rdata_ext    )
);
```

_ext wires are only for the testbench (do not change).

```
// IF_ID Pipeline register for instruction signal
reg_arstn_en#(
   .DATA_W(32) // width of the forwarded signal
)signal_pipe_IF_ID(
   .clk     (clk              ),
   .arst_n  (arst_n           ),
   .din     (instruction      ),
   .en      (enable           ),
   .dout   (instruction_IF_ID)
)
```

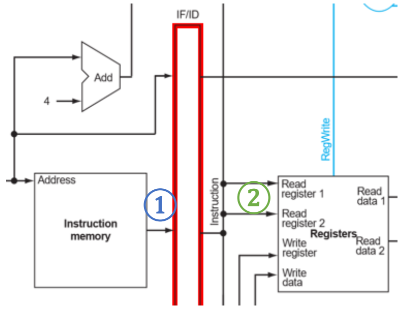
The IF_ID Pipeline Register.

```
// ID STAGE
register_file #(
   .DATA_W(64)
) register_file(
   .clk      (clk                    ),
   .arst_n   (arst_n                 ),
   .reg_write(reg_write              ),
   .raddr_1  (instruction_IF_ID[19:15]), // Wiring component
   .raddr_2  (instruction_IF_ID[24:20]), // to appropriate stage
   .waddr    (instruction_MEM_WB[11:7] ),// Reg_write addr
   .wdata    (regfile_wdata          ),
   .rdata_1  (regfile_rdata_1        ),
   .rdata_2  (regfile_rdata_2        )
);
```

Tips:

- Be careful with each register's DATA_WIDTH.
- Always define the signal before usage in coding.

Implementation Method



Stage	IF	ID
Data Path resources	Instruction_memory	register_file

① ②

Pipe. Reg.	IF/ID
Signals	Instruction_IF_ID

① ②

Sample Code Snippet

```
// IF STAGE
wire [       31:0] instruction, instruction_IF_ID, instruction_MEM_WB;

sram_BW32 #(
    .ADDR_W(9 ),
    .DATA_W(32)
) instruction_memory(
    .clk      (clk          ),
    .addr     (current_pc   ),
    .wen      (1'b0         ),
    .ren      (1'b1         ),
    .wdata    (32'b0        ),
    .rdata    (instruction  ), // Wiring component
    .addr_ext (addr_ext     ), // to appropriate stage
    .wen_ext  (wen_ext      ),
    .ren_ext  (ren_ext      ),
    .wdata_ext(wdata_ext    ),
    .rdata_ext(rdata_ext    )
);
```

_ext wires are only for the cpu_tb.v initialization.

```
// IF_ID Pipeline register for instruction signal
reg_arstn_en#(
    .DATA_W(32) // width of the forwarded signal
)signal_pipe_IF_ID(
    .clk     (clk            ),
    .arst_n  (arst_n         ),
    .din     (instruction    ),
    .en      (enable         ),
    .dout   (instruction_IF_ID)
)
```

The IF_ID Pipeline Register.

```
// ID STAGE
register_file #(
    .DATA_W(64)
) register_file(
    .clk      (clk                     ),
    .arst_n   (arst_n                  ),
    .reg_write(reg_write               ),
    .raddr_1  (instruction_IF_ID[19:15]), // Wiring component
    .raddr_2  (instruction_IF_ID[24:20]), // to appropriate stage
    .waddr    (instruction_MEM_WB[11:7] ),// Reg_write addr
    .wdata    (regfile_wdata           ),
    .rdata_1  (regfile_rdata_1         ),
    .rdata_2  (regfile_rdata_2         )
);
```

Tips:

- Be careful with each register's DATA_WIDTH.
- Always put the signal definition before usage.

Refer to the 5-stage pipeline to implement step by step.

EXAMPLE

Implementation Method

Module instantiation

1. The Template Definition

```
// reg_arstn_en.v

module reg_arstn_en#(
parameter integer DATA_W     = 20,
parameter integer PRESET_VAL = 0
   )(
        input                  clk,
        input                  arst_n,
        input                  en,
        input   [ DATA_W-1:0]  din,
        output  [ DATA_W-1:0]  dout
);
```

2. The Instance Declaration

```
// cpu.v

reg_arstn_en#(
   .DATA_W(32)
)signal_pipe_IF_ID(
   .clk     (clk            ),
   .arst_n  (arst_n         ),
   .din     (instruction    ),
   .en      (enable         ),
   .dout    (instruction_IF_ID)
)
```

- ***Template Name***
- ***Parameterization***
- ***Instance Name***

Port Name | ***Signal Connection***

If you need a customized module, you should also follow this approach.

EXAMPLE

Implementation Method

- *What should the final table be like?*

Stage	IF	ID	EXE	MEM	WB
Data Path resources	**IMEM** **PC**	**Control Unit** **Register File** …	**ALU** …	…	…

Pipe. Reg.	IF/ID	ID/EXE	EXE/MEM	MEM/WB
Signals	**Instruction_IF_ID** **Updated_pc_IF_ID**	**ControlSignals_ID_EX** **RegFile_data_1_ID_EX** …	…	…

These should be the wire names (instances) you choose in your design.

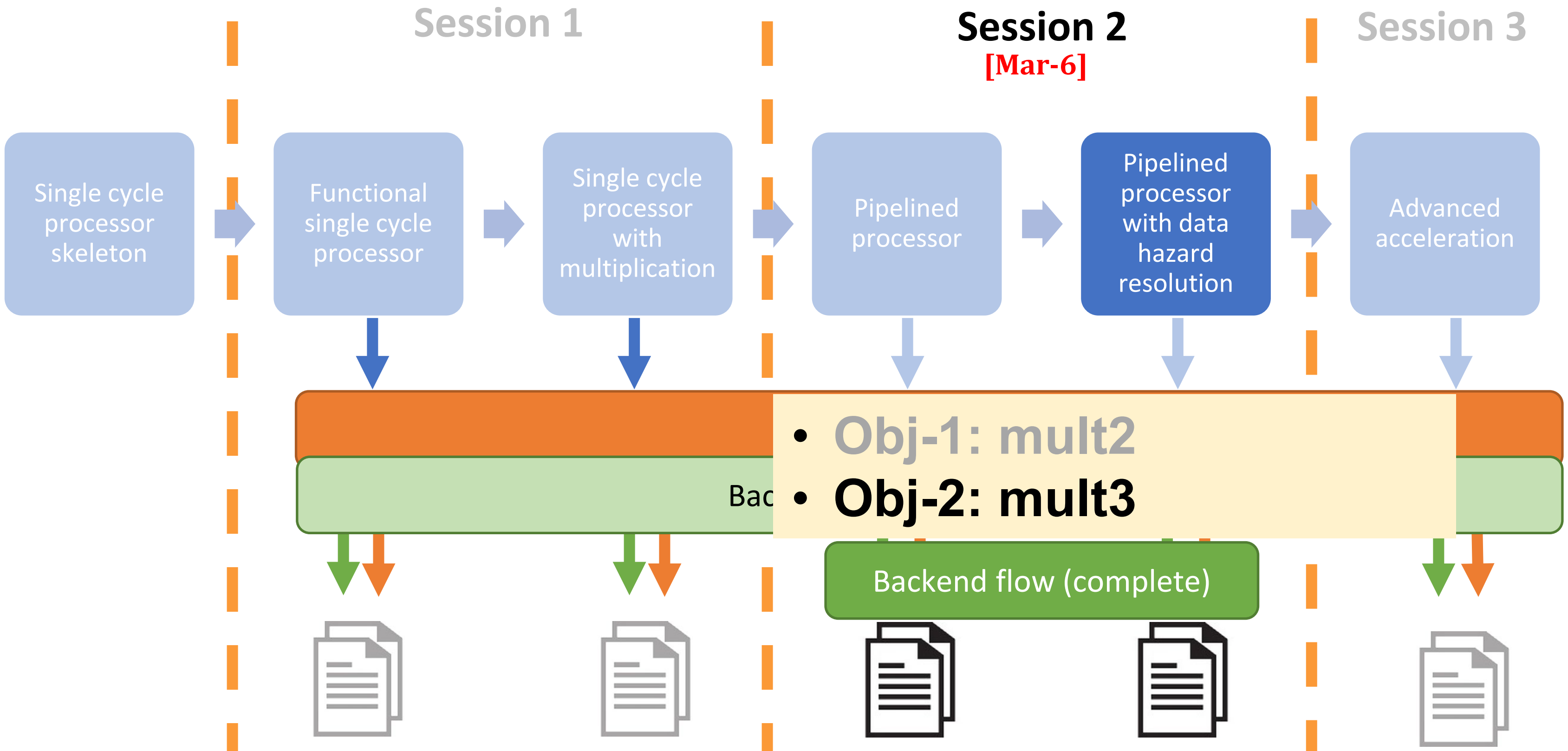
- *Use it as a checklist to save the debugging time!*

EXAMPLE

Linting your code.

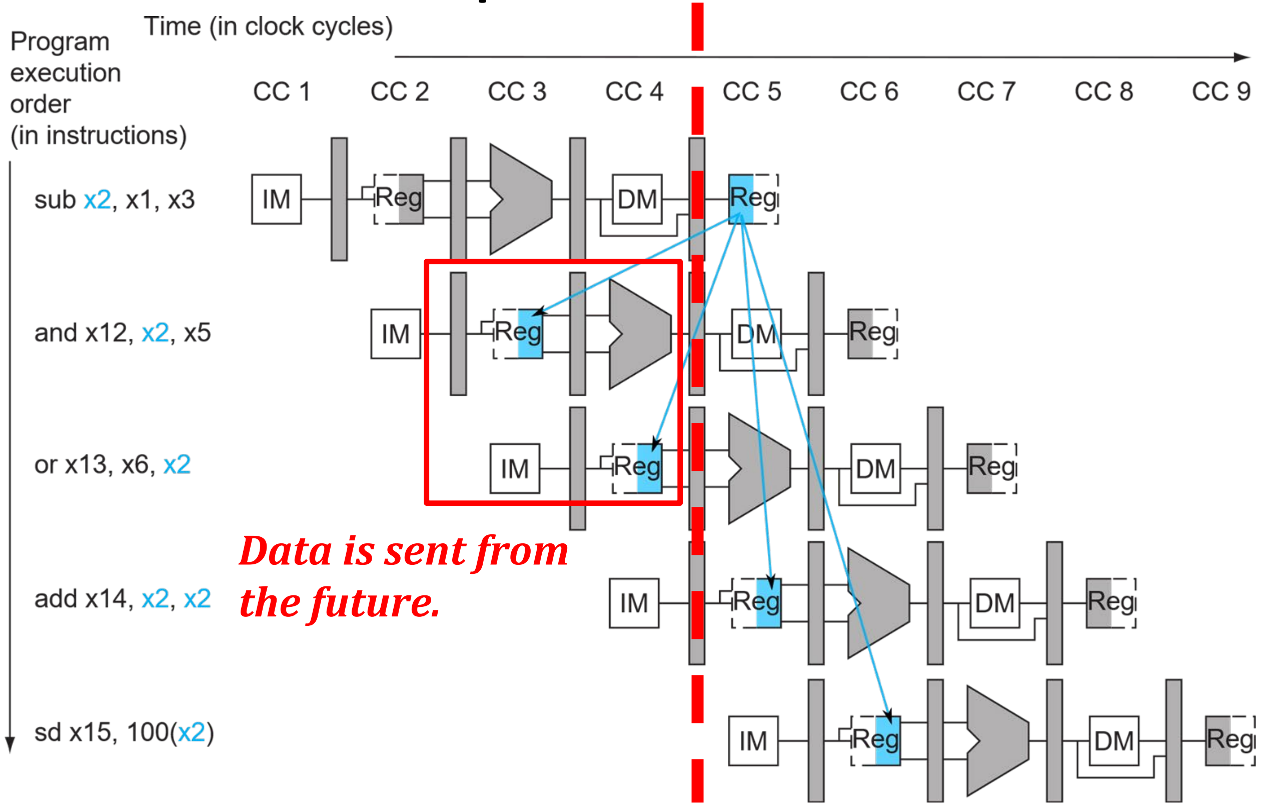
```
// IF STAGE BEGIN
...
// IF STAGE END

// IF_ID REG BEGIN
...
// IF_ID REG END

// ID STAGE BEGIN
...
// ID STAGE END

// ID_EX REG BEGIN
...
// ID_EX REG END

// EX STAGE BEGIN
...
// EX STAGE END

// EX_MEM REG BEGIN
...
// EX_MEM REG END

// MEM STAGE BEGIN
...
// MEM STAGE END

// MEM_WB REG BEGIN
...
// MEM_WB REG END

// WB STAGE BEGIN
...
// WB STAGE END
```

Today's session: Pipelined processor w/ hazard


Session 1
Session 2
[Mar-6]
Session 3
Single cycle processor skeleton
Functional single cycle processor
Single cycle processor with multiplication
Pipelined processor
Pipelined processor with data hazard resolution
Advanced acceleration
Bac
Backend flow (complete)

- Obj-1: mult2
- **Obj-2: mult3**

Data Hazard Example



Patterson Book. FIGURE 4.50

Data Hazard Resolution

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
```

x23 dependency

Pipeline clean up

Software: Insert 'nop' to delay the pipeline.

MULT3

```
# 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
add x23, x23, x18
add x23, x23, x19
add x23, x23, x20
add x23, x23, x21
add x23, x23, x22
```

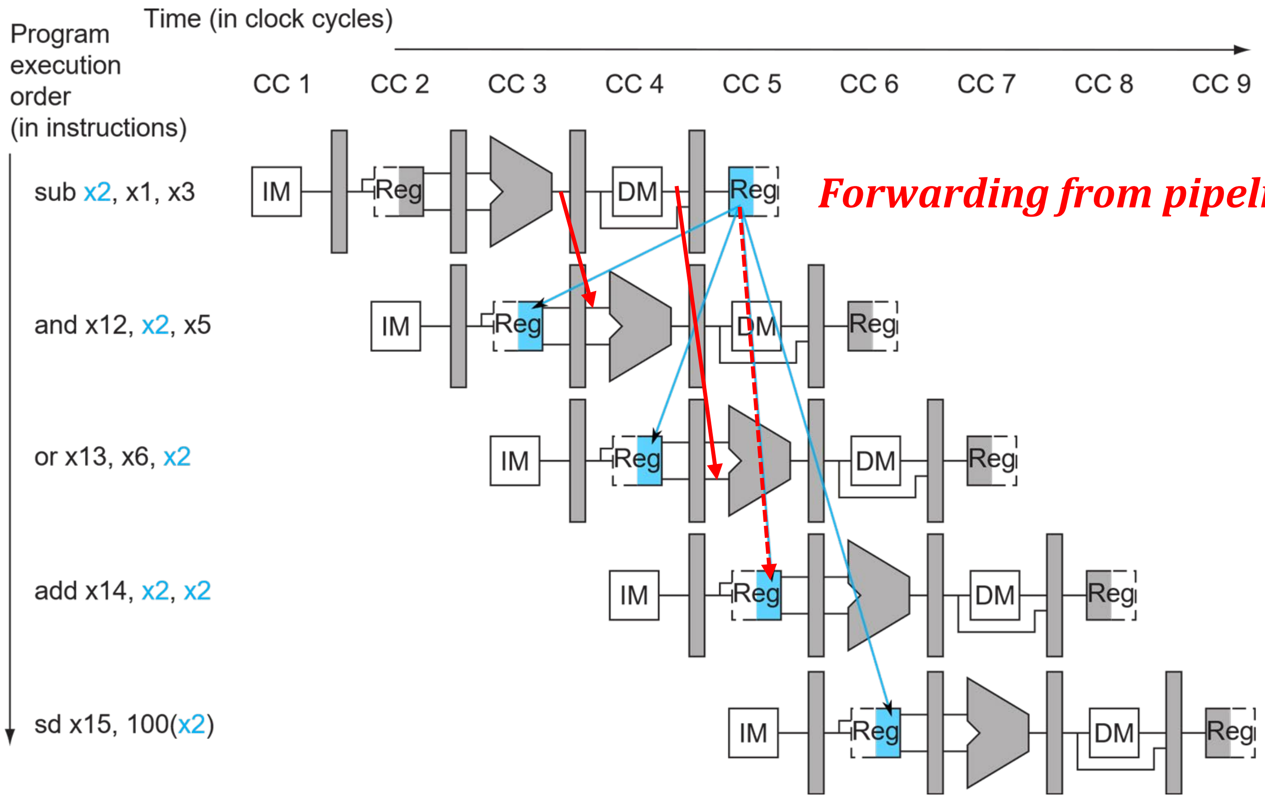
```
nop
nop
nop
nop
```

Pipeline clean up
is still required!

Data Hazard resolved!

Hardware: forwarding mechanism required.

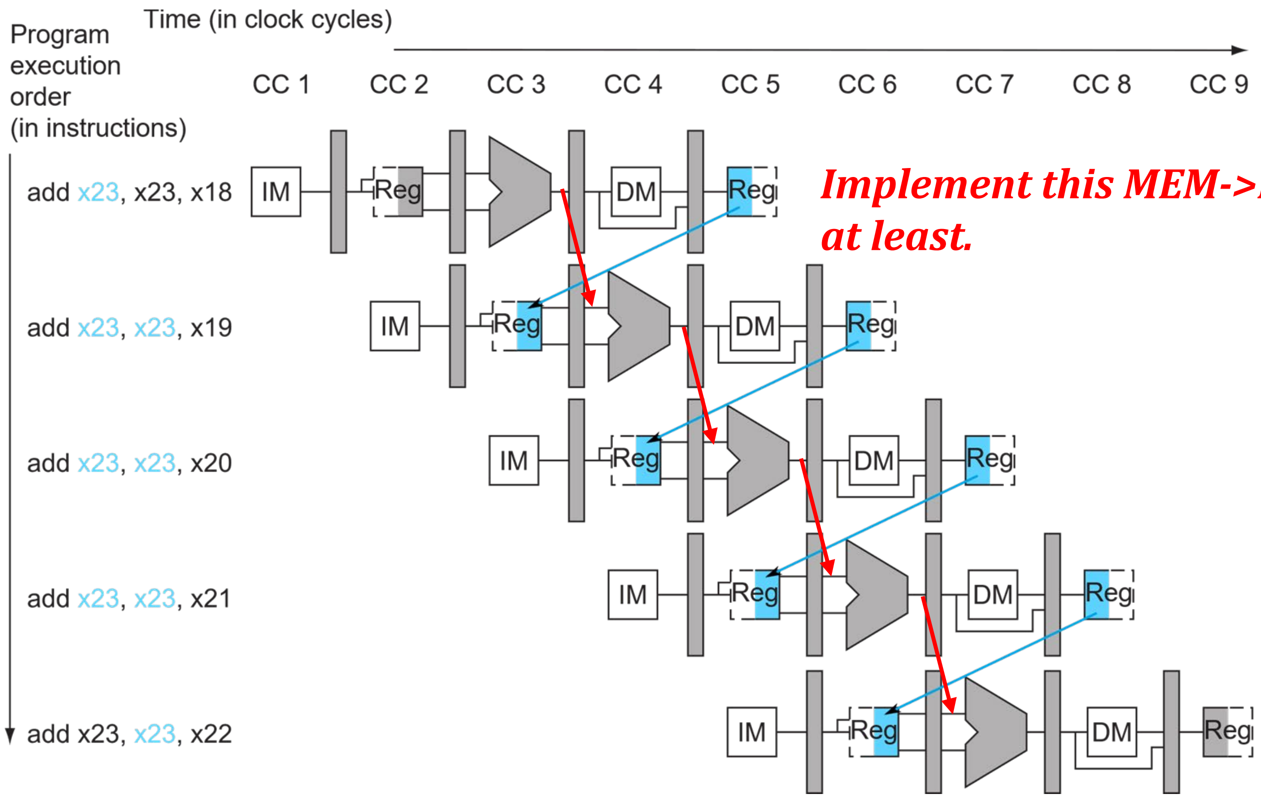
Data Hazard Resolution: Hardware



Forwarding from pipeline "shortcuts"

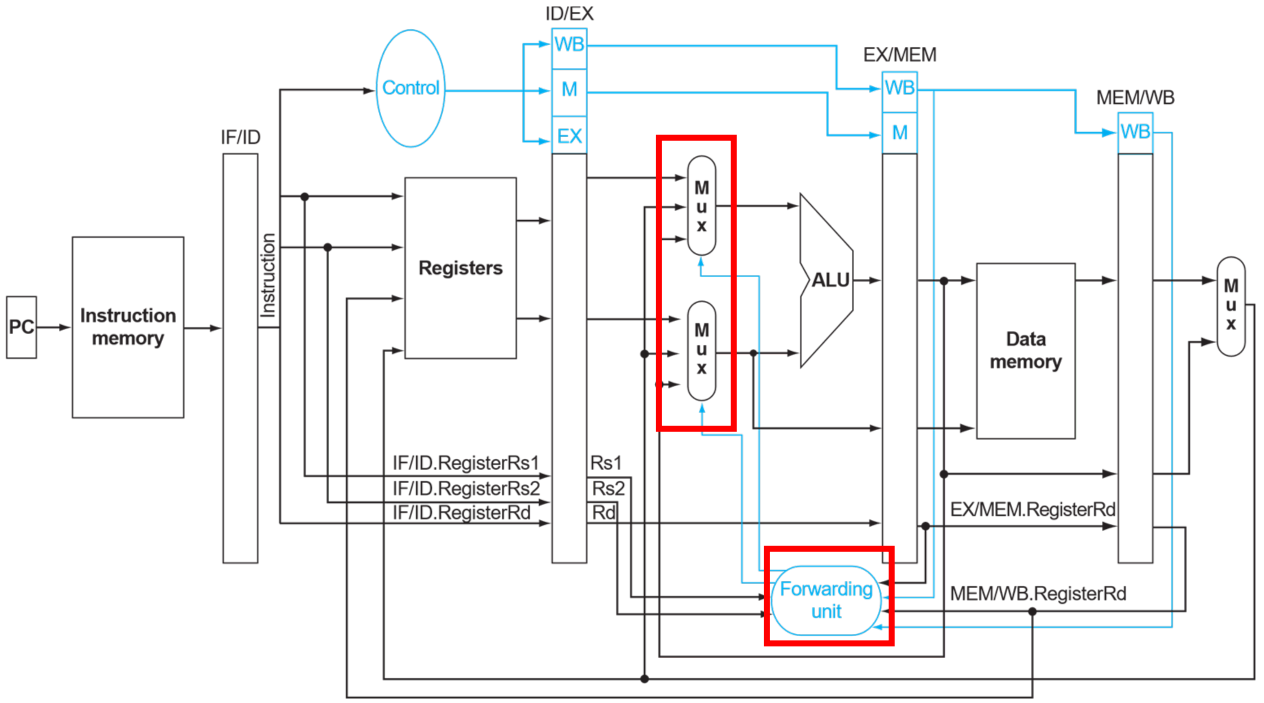
Patterson Book. FIGURE 4.51

Data Hazard Resolution – MULT3



Implement this MEM->EX forwarding at least.

Implementation: Forwarding unit



```
// your_fw_fname.v or
an/existing/Verilog/file

module YOUR_FW_UNIT_NAME#(
parameter integer … (if needed)
    )(
        input               …,
        input   [ ?:0]      …,
        output  [ ?:0]      …
);

    YOUR_FW_UNIT_LOGIC;

endmodule
```

Refer to slide: Implementation Method

Obj-2: mult3

- Pipelined processor with data hazard resolution
  - Implement a forwarding unit to resolve the data hazard
  - Run MULT3
  - Run backend flow
    - Synthesize
  - Complete the report

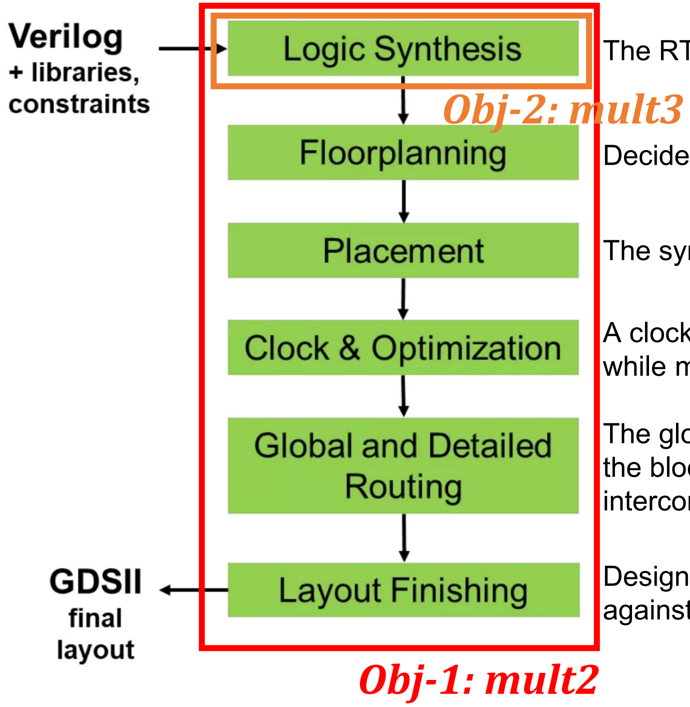
MULT3

```
# 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
add x23, x23, x18
add x23, x23, x19
add x23, x23, x20
add x23, x23, x21
add x23, x23, x22
```

Note:

- Do not forget to update files_verilog.f in /SIM folder, if you add new RTL source file, or change the RTL location.
- Please refer to the mux-2 to customize a mux-3, if needed.
- Do not forget to add/modify all the signal wires in the top cpu.v according to your new modules.

Today's session: Backend flow



The RTL design is synthesized into a gate-level netlist.

Decide on the chip area and set memory macros location in the chip.

The synthesized gates are placed in the chip according to the floorplan.

A clock tree is generated to distribute the clock signal throughout the chip, while minimizing skew and ensuring timing requirements are met.

The global routing involves determining the overall routing paths between the blocks and macros; The detailed routing creates the actual metal interconnects between the gates.

Design rule checking: Once the routing is complete, the design is checked against a set of design rules to ensure that it is manufacturable.

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** → **RTL_SOLUTION3_pipeline_basic_MULT2**
- **Obj-2** → **RTL_SOLUTION4_pipeline_hazard_MULT3**

- **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**