Building a RISC-V Core

RISC-V Based MYTH Workshop
MYTH - Microprocessor for You in Thirty Hours

Steve Hoover
Founder, Redwood EDA
July 31, 2020

Day 3-5 Agenda

Day 3: Digital logic with TL-Verilog in Makerchip IDE

Day 4: Coding a RISC-V CPU subset

Day 5: Pipelining and completing your CPUtest

Agenda

Day 3: Digital logic with TL-Verilog in Makerchip IDE

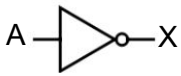
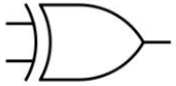
- Logic gates
- Makerchip platform
- Combinational logic
- Sequential logic
- Pipelined logic
- State

Logistics

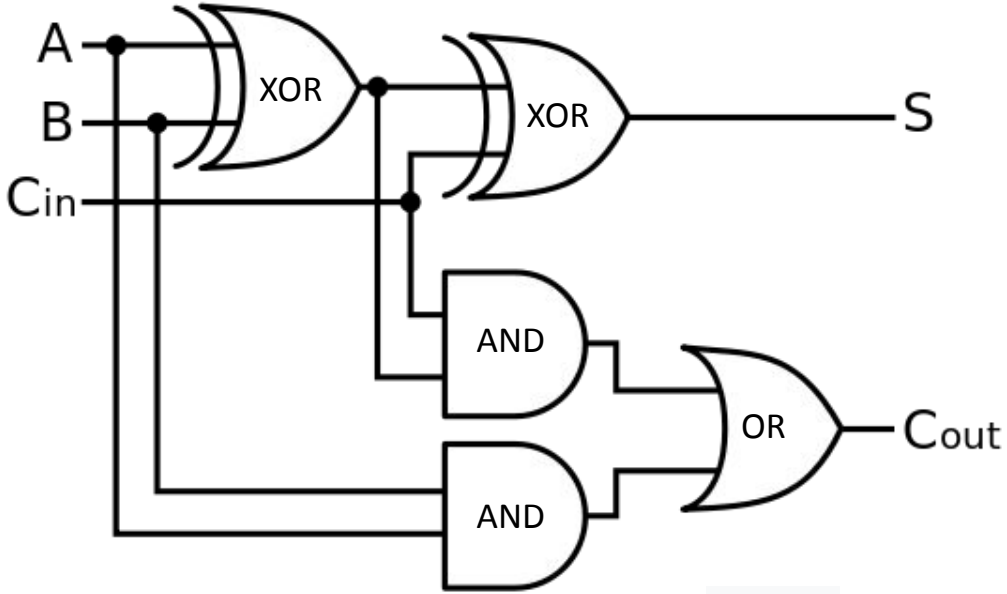
Live updates, lab help, links, etc.:

https://github.com/stevehoover/RISC-V_MYTH_Workshop

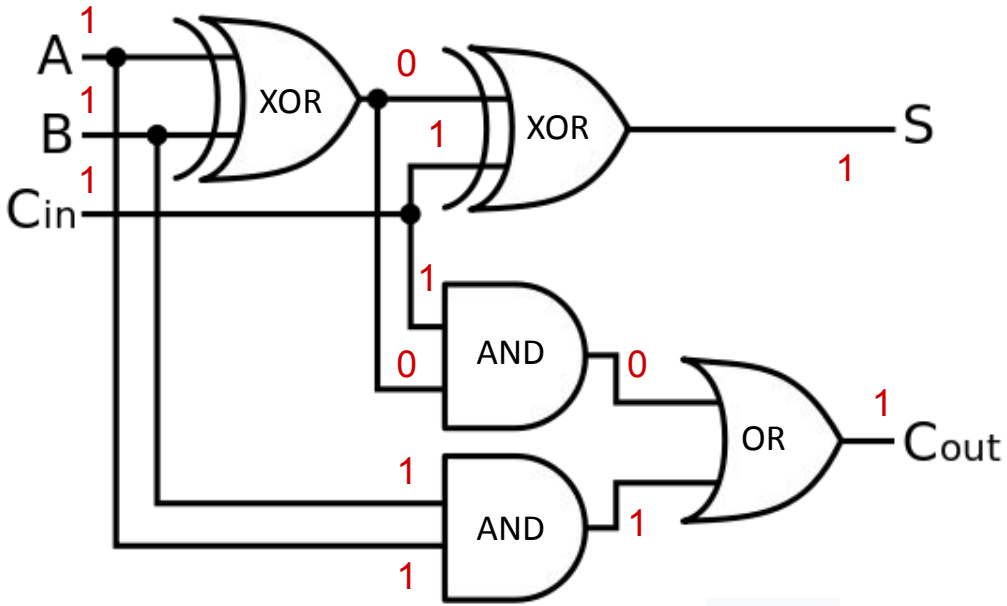
Logic Gates

Name	NOT	AND	OR	XOR	NAND	NOR	XNOR
Symbol							
Truth Table	A X 0 **1** 1 **0**	A B X 0 0 **0** 0 1 **0** 1 0 **0** 1 1 **1**	A B X 0 0 **0** 0 1 **1** 1 0 **1** 1 1 **1**	A B X 0 0 **0** 0 1 **1** 1 0 **1** 1 1 **0**	A B X 0 0 **1** 0 1 **1** 1 0 **1** 1 1 **0**	A B X 0 0 **1** 0 1 **0** 1 0 **0** 1 1 **0**	A B X 0 0 **1** 0 1 **0** 1 0 **0** 1 1 **1**

Combinational Circuit

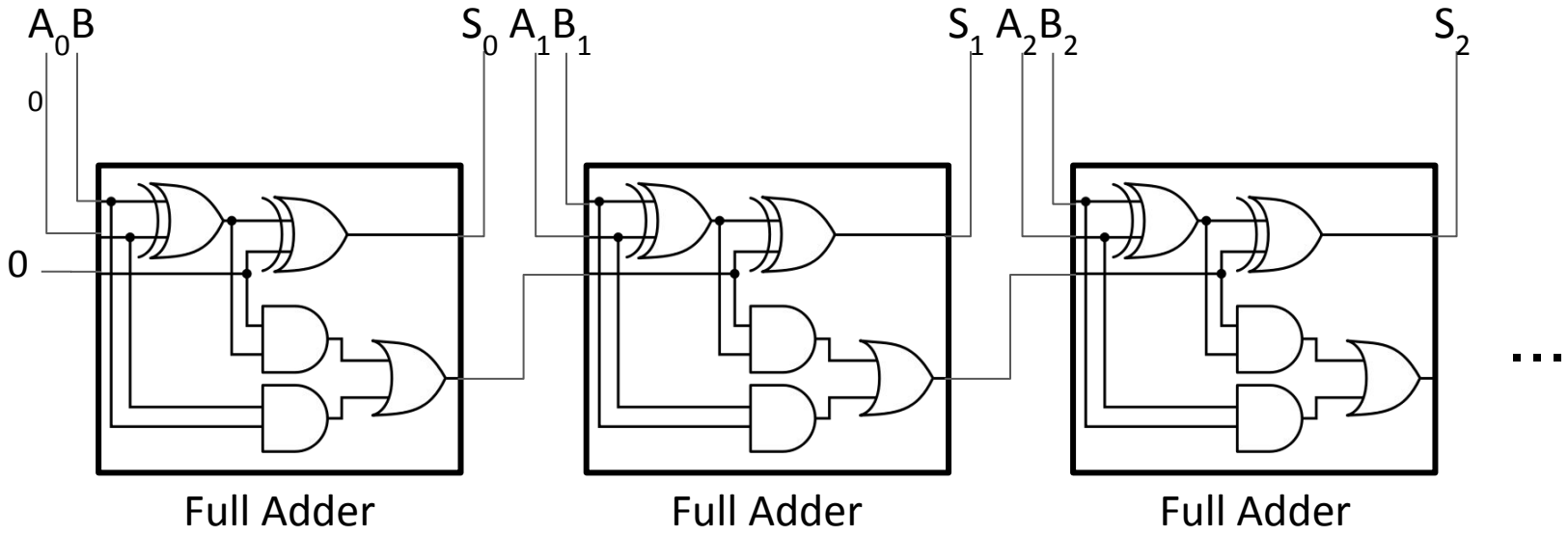


Combinational Circuit



Adder

$$S = A + B$$

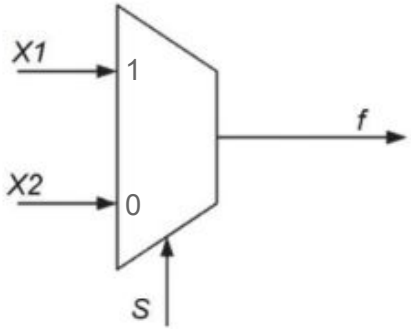


Adder

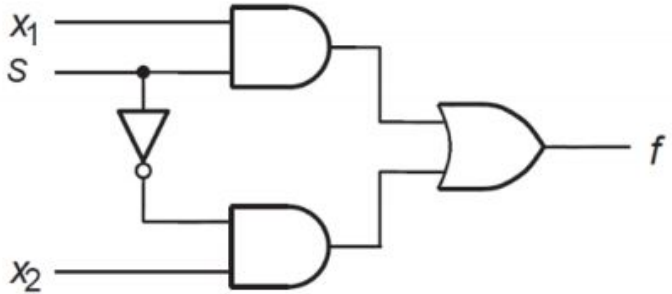
Boolean Operators

Op	Bool Arith	Bool Calc	Verilog	Gate
NOT	$\overline{A}$	¬A	~A (or !A)	
AND	A•B	A∧B	A&B (or &&)	
OR	A+B	A∨B	A|B (or ||)	
XOR	A⊕B	A⊕B	A ^ B	
NAND	$\overline{A•B}$	¬(A∧B)	!(A & B)	
NOR	$\overline{A+B}$	¬(A∨B)	!(A | B)	
XNOR	$\overline{A⊕B}$	¬(A⊕B)	!(A ^ B)	

Multiplexer (MUX)



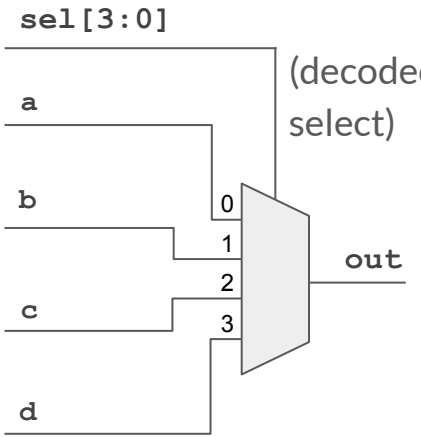
(boolean select)

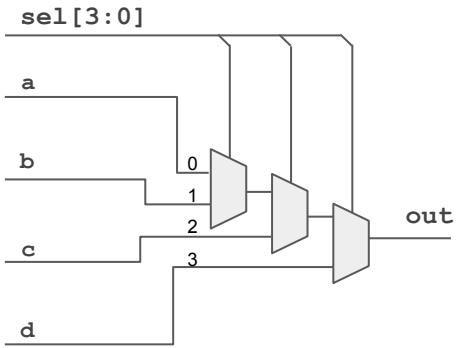


Verilog:

```
assign f = s ? X1 : X2;
```

Chaining Ternary Operator





Verilog:

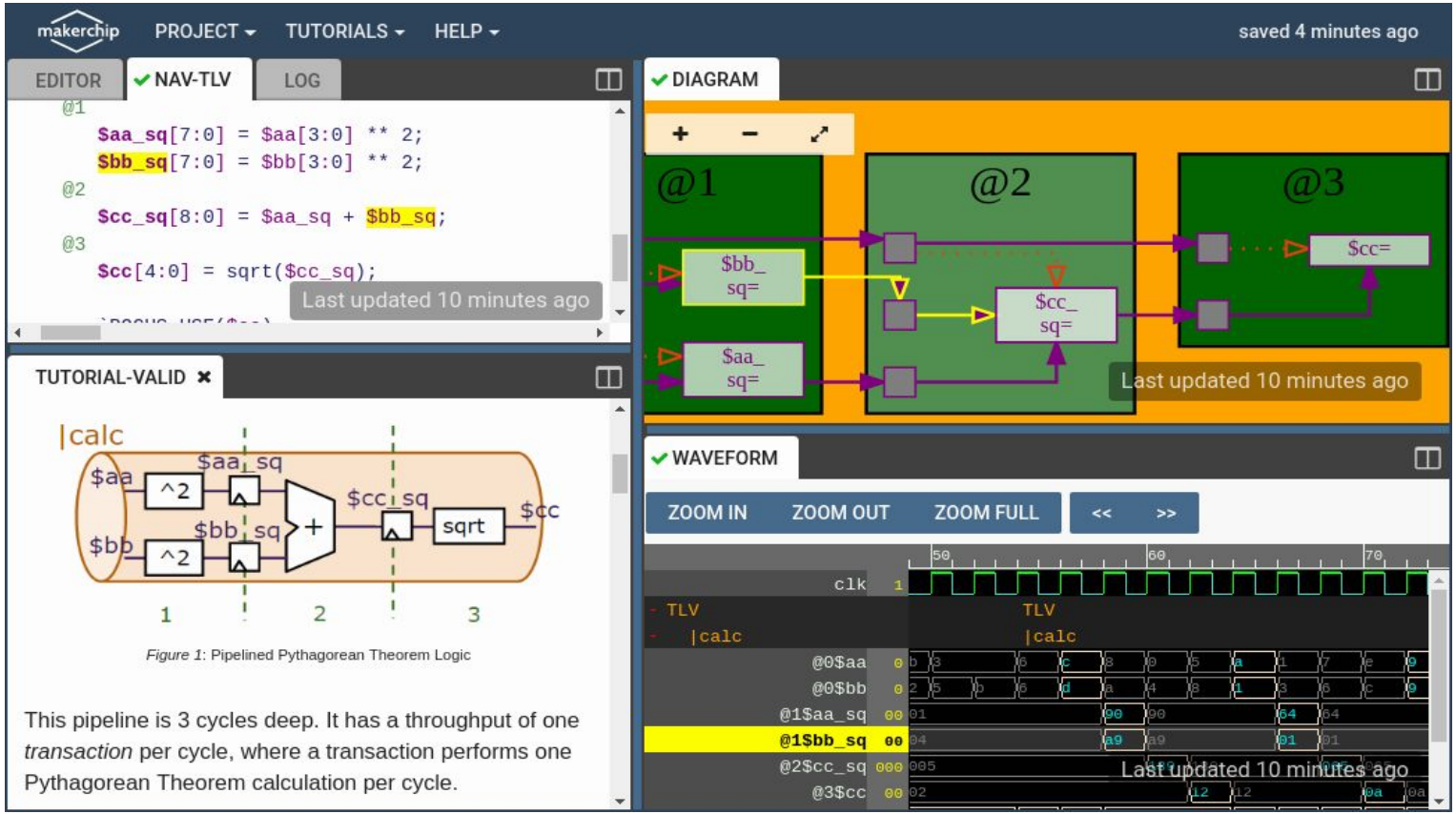
```
assign f =
   sel[3]
      ? d
      : (sel[2]
            ? c
            : (sel[1]
                  ? b
                  : a
               )
         );
```

Equivalently:

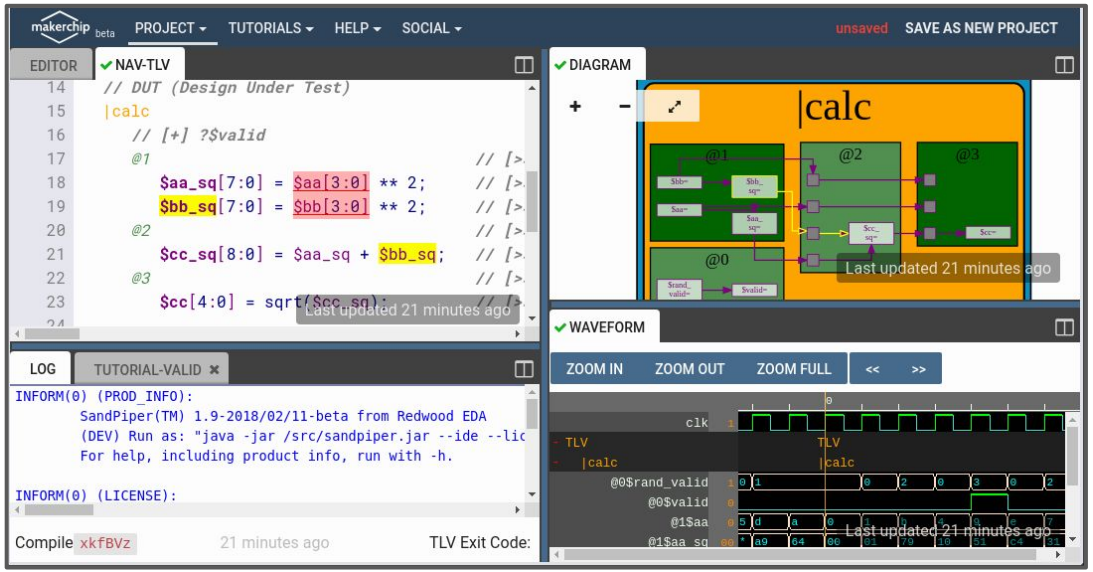
```
assign f =
   sel[3]
      ? d :
   sel[2]
      ? c :
   sel[1]
      ? b :
   //default
        a;
```

(So, highest priority first.)

Makerchip


makerchip
PROJECT
TUTORIALS
HELP
saved 4 minutes ago
EDITOR
NAV-TLV
LOG
@1
$aa_sq[7:0] = $aa[3:0] ** 2;
$bb_sq[7:0] = $bb[3:0] ** 2;
@2
$cc_sq[8:0] = $aa_sq + $bb_sq;
@3
$cc[4:0] = sqrt($cc_sq);
Last updated 10 minutes ago
DIAGRAM
@1
@2
@3
$bb_ sq=
$aa_ sq=
$cc_ sq=
$cc=
Last updated 10 minutes ago
TUTORIAL-VALID
|calc
$aa
$bb
^2
^2
$aa_sq
$bb_sq
+
$cc_sq
sqrt
$cc
1
2
3
Figure 1: Pipelined Pythagorean Theorem Logic
This pipeline is 3 cycles deep. It has a throughput of one transaction per cycle, where a transaction performs one Pythagorean Theorem calculation per cycle.
WAVEFORM
ZOOM IN
ZOOM OUT
ZOOM FULL
<<
>>
clk
TLV
|calc
@0$aa
@0$bb
@1$aa_sq
@1$bb_sq
@2$cc_sq
@3$cc
Last updated 10 minutes ago

Lab: Makerchip Platform



1. On desktop machine, in modern web browser (not IE), go to: makerchip.com
2. Click “IDE”.

Reproduce this screenshot:

1. Open “Tutorials” “Validity Tutorial”.
2. In tutorial, click

Load Pythagorean Example

3. Split panes and move tabs.
4. Zoom/pan in Diagram w/ mouse wheel and drag.
5. Zoom Waveform w/ “Zoom In” button.
6. Click `$bb_sq` to highlight.

Lab: Combinational Logic

A) Inverter

1. Open “Examples” (under “Tutorials”).
2. Load “Default Template”.
3. Make an inverter.
   On line 16, in place of:

```
//...
```

   type:

```
$out = ! $in1;
```

   (Preserve 3-space indentation, no tabs)

4. Compile (“E” menu) & Explore

Note:

There was no need to declare `$out` and `$in1` (unlike Verilog).

There was no need to assign `$in1`. Random stimulus is provided, and a warning is produced.

B) Other logic

Make a 2-input gate.
(Boolean operators: (`&&`, `||`, `^`))

Lab: Vectors

`$out[4:0]` creates a “vector” of 5 bits.

Arithmetic operators operate on vectors as binary numbers.

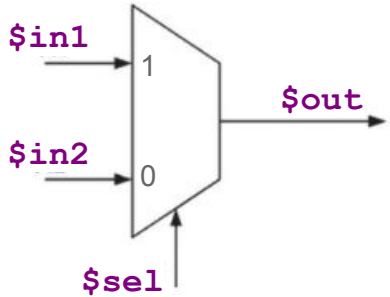
1. Try:

```
$out[4:0] = $in1[3:0] + $in2[3:0];
```

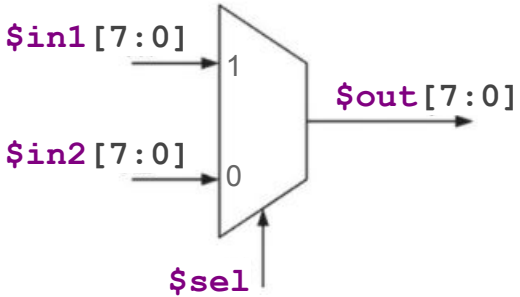
2. View Waveform.

Lab: Mux

`$out = $sel ? $in1 : $in2` creates a multiplexer.



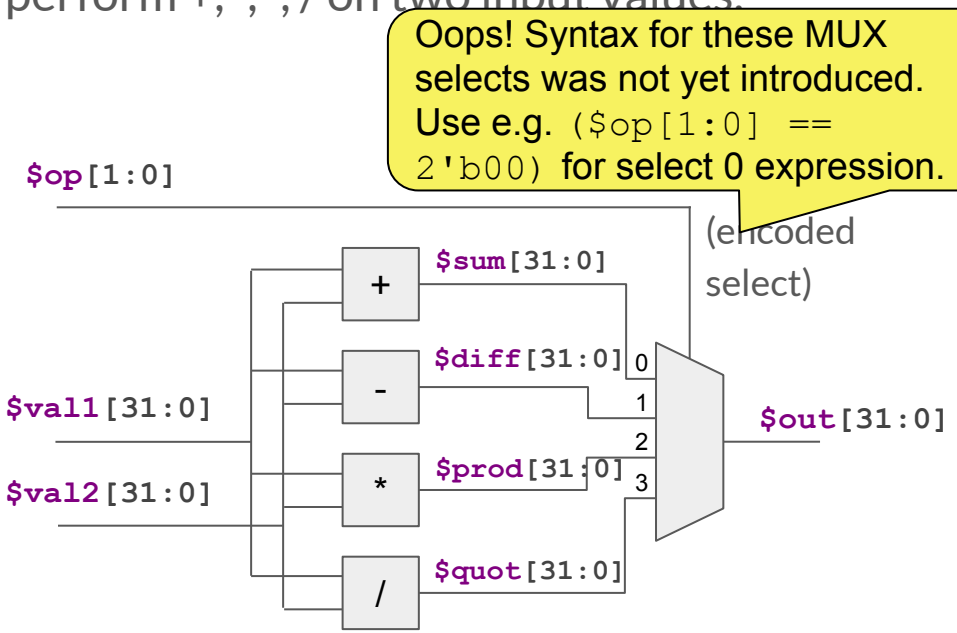
Modify this multiplexer to operate on vectors.



Note that bit ranges can generally be assumed on the right-hand side, but with no assignments to these signals, they must be explicit.

Lab: Combinational Calculator

This circuit implements a calculator that can perform +, -, *, / on two input values.



1. Implement this.
2. Use:

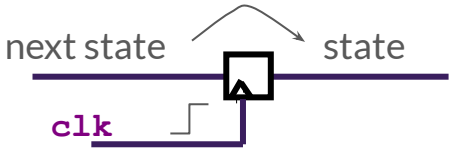
```
$val1[31:0] = $rand1[3:0];
$val2[31:0] = $rand2[3:0];
```

   for inputs to keep values small.
3. We'll return to this, so "Save as new project", bookmark, and open a new Makerchip IDE in a new tab.

Sequential Logic

Sequential logic is sequenced by a clock signal.

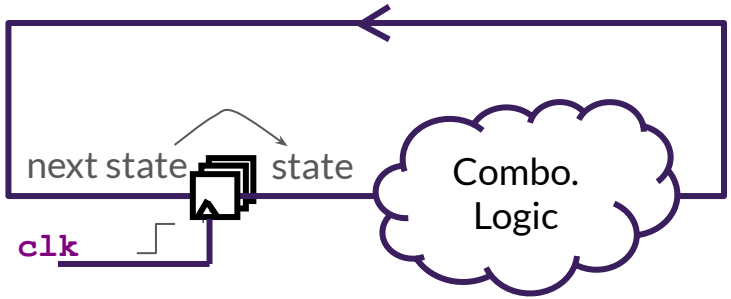
A D-flip-flop transitions next state to current state on a rising clock edge.



The circuit is constructed to enter a known state in response to a reset signal.

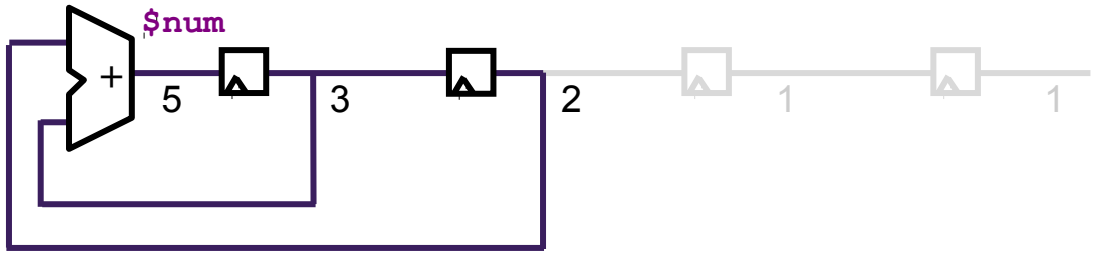
Sequential Logic

The whole circuit can be viewed as a big state machine.



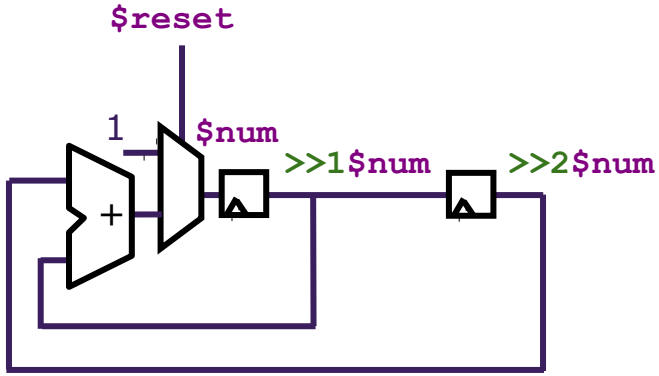
Sequential Logic - Fibonacci Series

Next value is sum of previous two: 1, 1, 2, 3, 5, 8, 13, ...



Fibonacci Series - Reset

Next value is sum of previous two: 1, 1, 2, 3, 5, 8, 13, ...

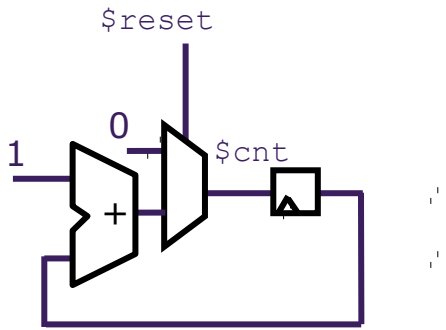



```
$num[31:0] = $reset ? 1 : (>>1$num + >>2$num);
```

(makerchip.com/sandbox/0/0wjhLP)

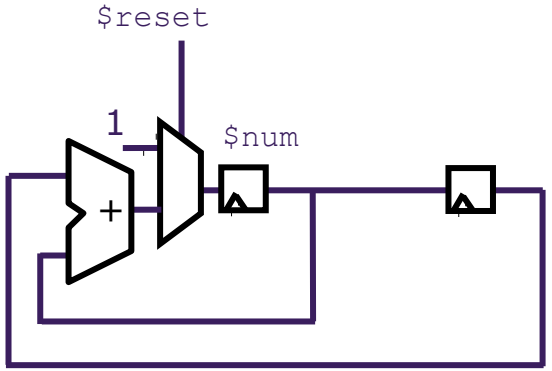
Lab: Counter

1. Design a free-running counter:



2. Include this code in your saved calculator sandbox for later (and confirm that it auto-saves).

Reference Example: Fibonacci Sequence (1, 1, 2, 3, 5, 8, ...)

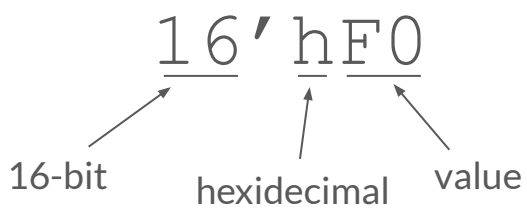



```
\TLV
   $num[31:0] = $reset ? 1 : (>>1$num + >>2$num);
```

3-space indentation
(no tabs)

(makerchip.com/sandbox/0/0wjhLP)

Values in Verilog



`'0`	All 0s (width based on context).
`'X`	All DONT-CARE bits.
`16'd5`	16-bit decimal 5.
`5'b00XX1`	5-bit value with DONT-CARE bits.
`1`	32-bit (signed) 1.

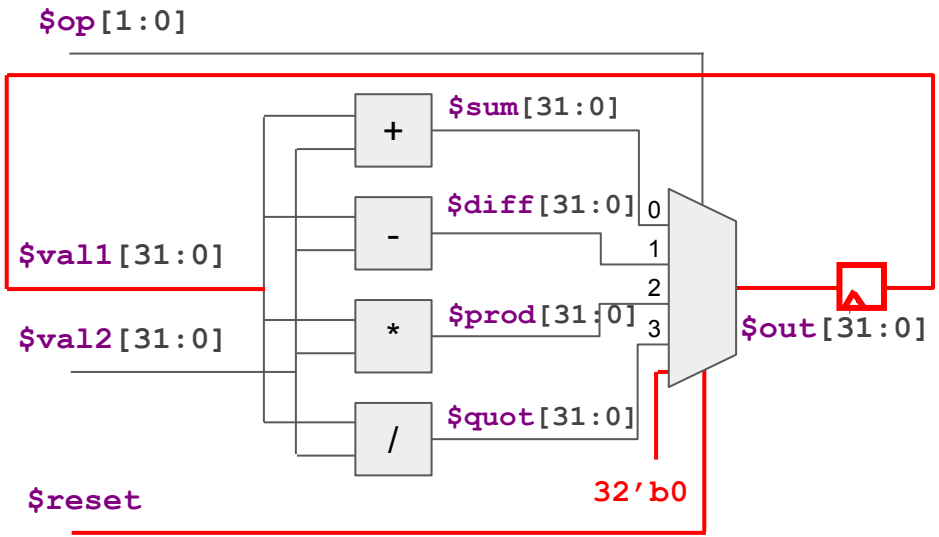
Our simulator configuration:

- will zero-extend or truncate when widths are mismatched (without warning)
- uses 2-state simulation (no X's)

Lab: Sequential Calculator

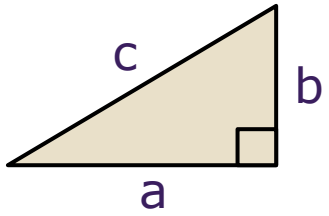
A real calculator remembers the last result, and uses it for the next calculation.

1. Return to the calculator.
2. Update the calculator to perform a new calculation each cycle where `$val1[31:0]` = the result of the previous calculation.
3. Reset $out to zero.
4. Copy code and save outside of Makerchip (just to be safe).

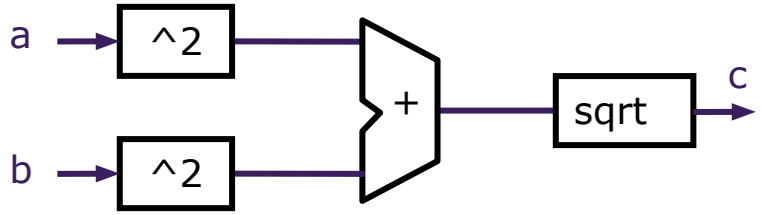


A Simple Pipeline

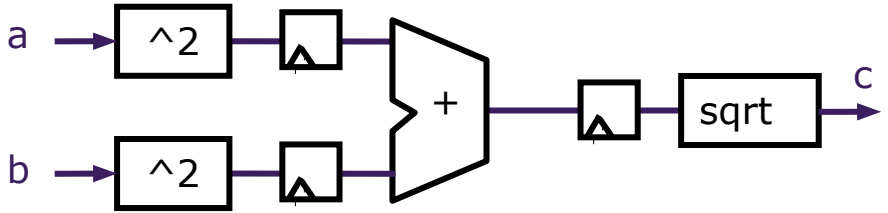
Let's compute Pythagoras's Theorem in hardware.



c = sqrt(a^2 + b^2)

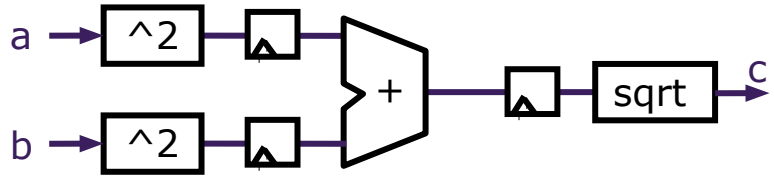


We distribute the calculation over three cycles.

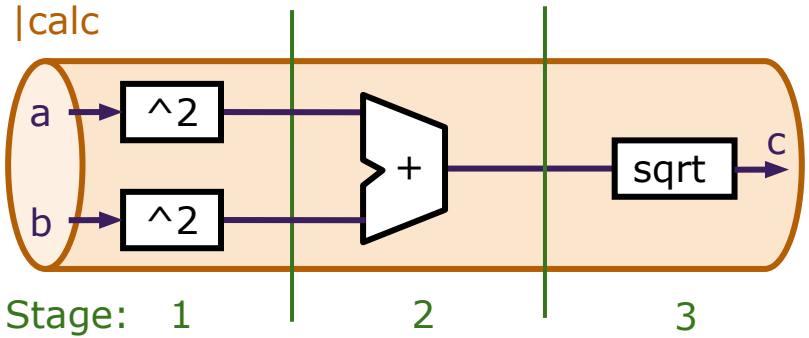


A Simple Pipeline - Timing-Abstract

RTL:

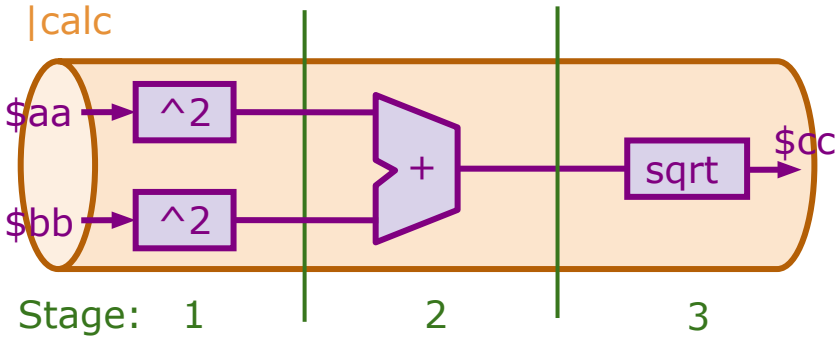


Timing-abstract:



➔ Flip-flops and staged signals are implied from context.

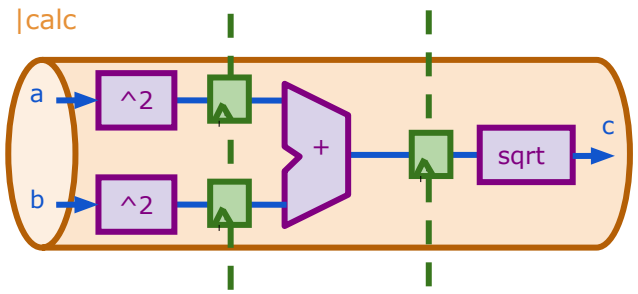
A Simple Pipeline - TL-Verilog



TL-Verilog

```
|calc
   @1
      $aa_sq[31:0] = $aa * $aa;
      $bb_sq[31:0] = $bb * $bb;
   @2
      $cc_sq[31:0] = $aa_sq + $bb_sq;
   @3
      $cc[31:0] = sqrt($cc_sq);
```

SystemVerilog vs. TL-Verilog



System Verilog

```
// Calc Pipeline
logic [31:0] a_C1;
logic [31:0] b_C1;
logic [31:0] a_sq_C1,
             a_sq_C2;
logic [31:0] b_sq_C1,
             b_sq_C2;
logic [31:0] c_sq_C2,
             c_sq_C3;
logic [31:0] c_C3;
always_ff @(posedge clk) a_sq_C2 <= a_sq_C1;
always_ff @(posedge clk) b_sq_C2 <= b_sq_C1;
always_ff @(posedge clk) c_sq_C3 <= c_sq_C2;
// Stage 1
assign a_sq_C1 = a_C1 * a_C1;
assign b_sq_C1 = b_C1 * b_C1;
// Stage 2
assign c_sq_C2 = a_sq_C2 + b_sq_C2;
// Stage 3
assign c_C3 = sqrt(c_sq_C3);
```

~3.5x

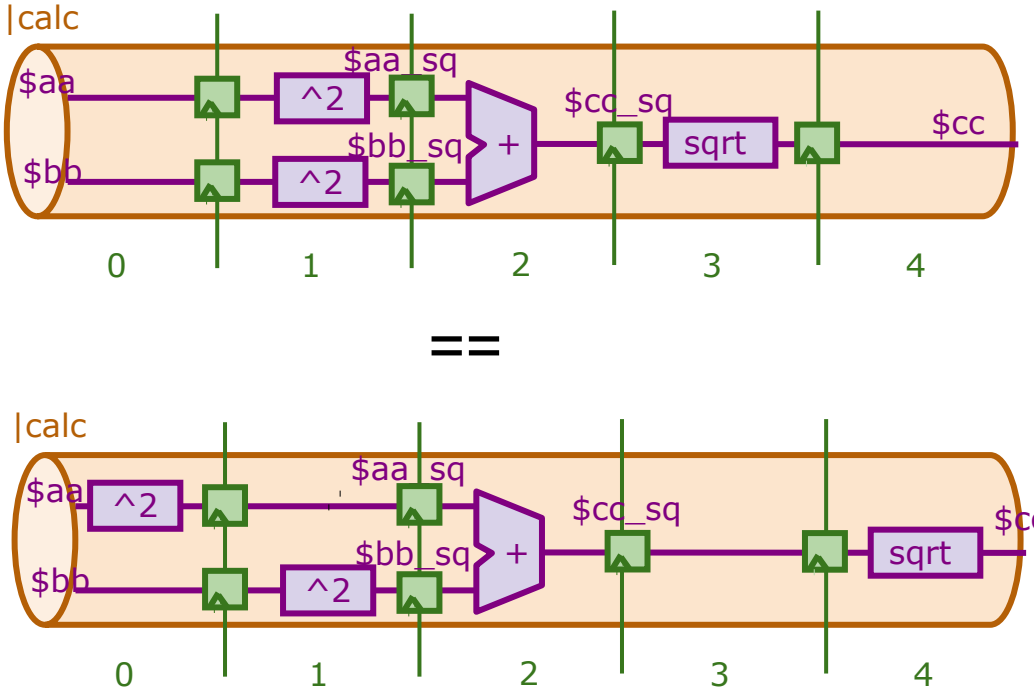
TL-Verilog

```
|calc
   @1
      $aa_sq[31:0] = $aa * $aa;
      $bb_sq[31:0] = $bb * $bb;
   @2
      $cc_sq[31:0] = $aa_sq + $bb_sq;
   @3
      $cc[31:0] = sqrt($cc_sq);
```

Retiming -- Easy and Safe

```
|calc
   @1
      $aa_sq[31:0] = $aa * $aa;
      $bb_sq[31:0] = $bb * $bb;
   @2
      $cc_sq[31:0] = $aa_sq + $bb_sq;
   @3
      $cc[31:0] = sqrt($cc_sq);
```

```
|calc
   @0
      $aa_sq[31:0] = $aa * $aa;
   @1
      $bb_sq[31:0] = $bb * $bb;
   @2
      $cc_sq[31:0] = $aa_sq + $bb_sq;
   @4
      $cc[31:0] = sqrt($cc_sq);
```



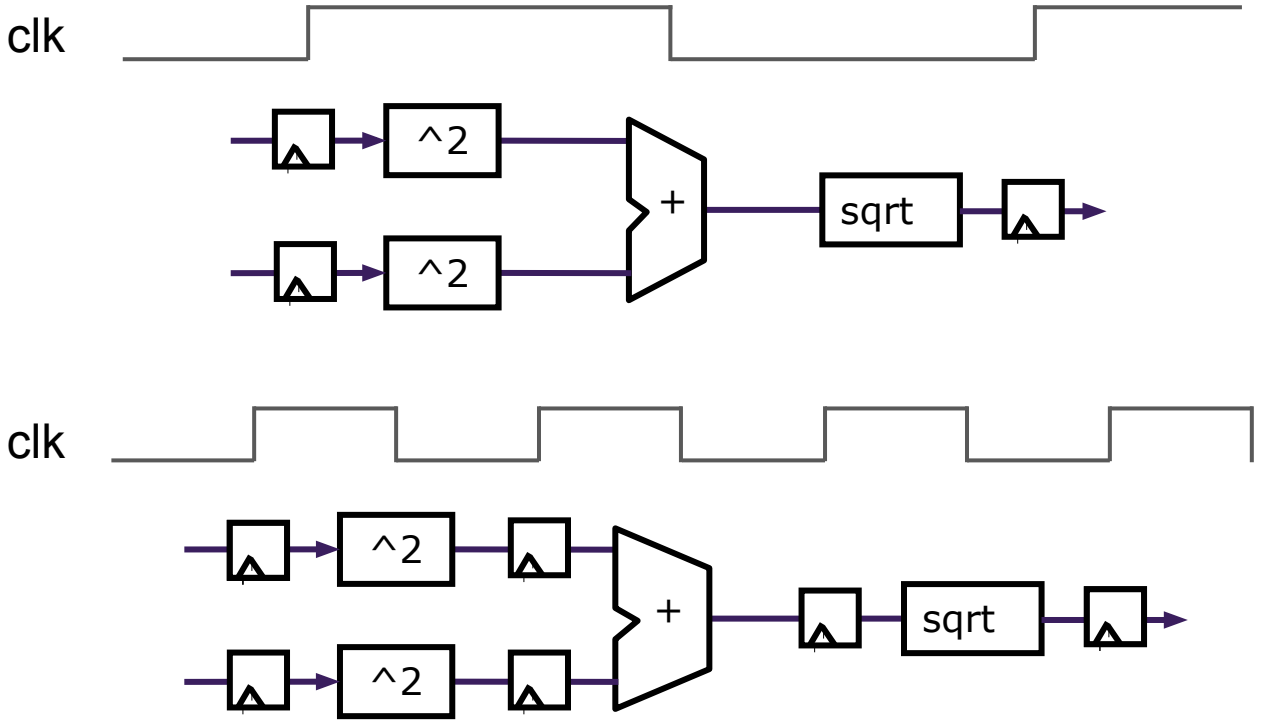
Staging is a physical attribute. No impact to behavior.

Retiming in SystemVerilog

```
// Calc Pipeline
logic [31:0] a_C1;
logic [31:0] b_C1;
logic [31:0] a_sq_C0,
             a_sq_C1,
             a_sq_C2;
logic [31:0] b_sq_C1,
             b_sq_C2;
logic [31:0] c_sq_C2,
             c_sq_C3,
             c_sq_C4;
logic [31:0] c_C3;
always_ff @(posedge clk) a_sq_C2 <= a_sq_C1;
always_ff @(posedge clk) b_sq_C2 <= b_sq_C1;
always_ff @(posedge clk) c_sq_C3 <= c_sq_C2;
always_ff @(posedge clk) c_sq_C4 <= c_sq_C3;
// Stage 1
assign a_sq_C1 = a_C1 * a_C1;
assign b_sq_C1 = b_C1 * b_C1;
// Stage 2
assign c_sq_C2 = a_sq_C2 + b_sq_C2;
// Stage 3
assign c_C3 = sqrt(c_sq_C3);
```

VERY BUG-PRONE!

High Frequency

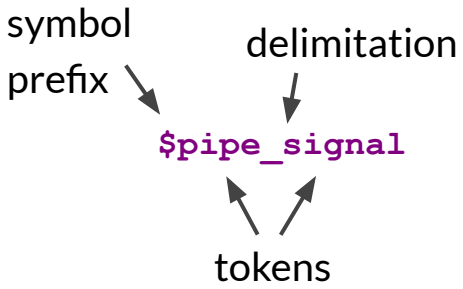


Makerchip - a level deeper

(Exploration of pipelined logic within Makerchip.)

Identifiers and Types

Type of an identifier determined by symbol prefix and case/delimitation style. E.g.:



First token must start with two alpha chars. These determine delimitation style

- `$lower_case`: pipe signal
- `$CamelCase`: state signal (technically, this is "Pascal case")
- `$UPPER_CASE`: keyword signal

Numbers end tokens (after alphas)

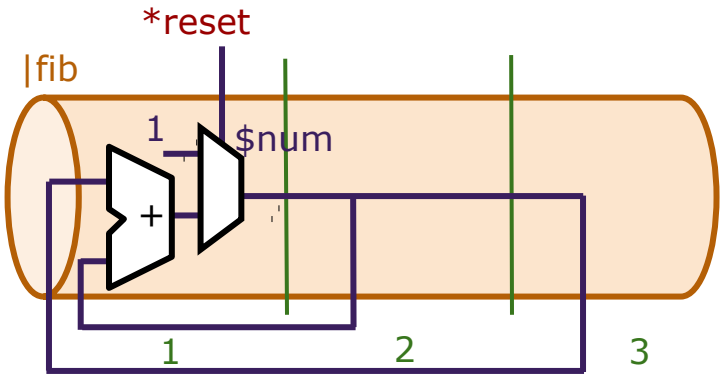
- `$base64_value`: good
- `$bad_name_5`: bad

Numeric identifiers

- `>>1`: ahead by 1

Fibonacci Series in a Pipeline

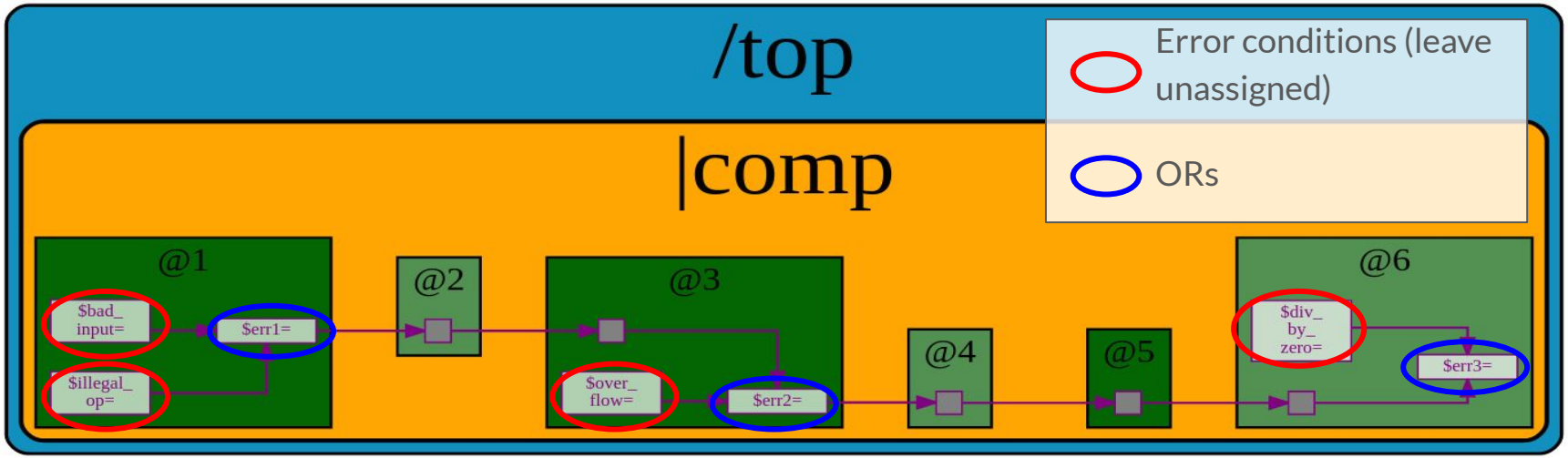
Next value is sum of previous two: 1, 1, 2, 3, 5, 8, 13, ...




```
|fib
   @1
      $num[31:0] = *reset ? 1 : (>>1$num + >>2$num);
```

Lab: Pipeline

See if you can produce this:



which ORs together (||) various error conditions that can occur within a computation pipeline.

Open in Makerchip

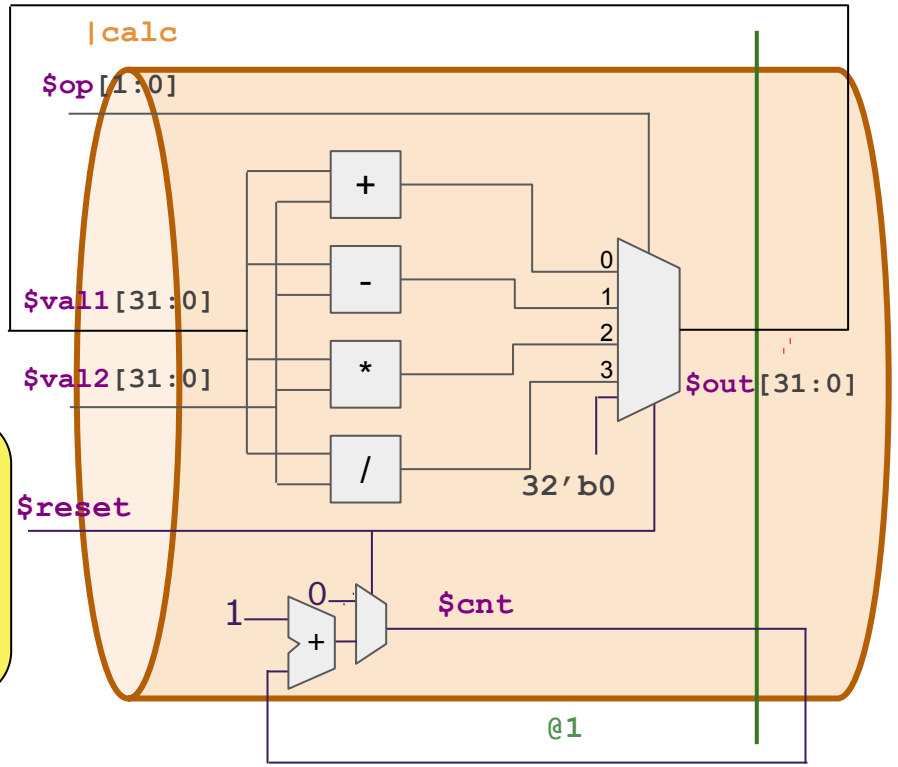
(makerchip.com/sandbox/0/0xGhJP)

Lab: Counter and Calculator in Pipeline

1. Put calculator and counter in stage `@1` of a `|calc` pipeline.
2. Check log, diagram, and waveform.
3. Confirm save.

At this point, be sure to use the Calculator Starter Code from the github repo. After this exercise, you can comment out the viz macro and use the VIZ tab to debug.

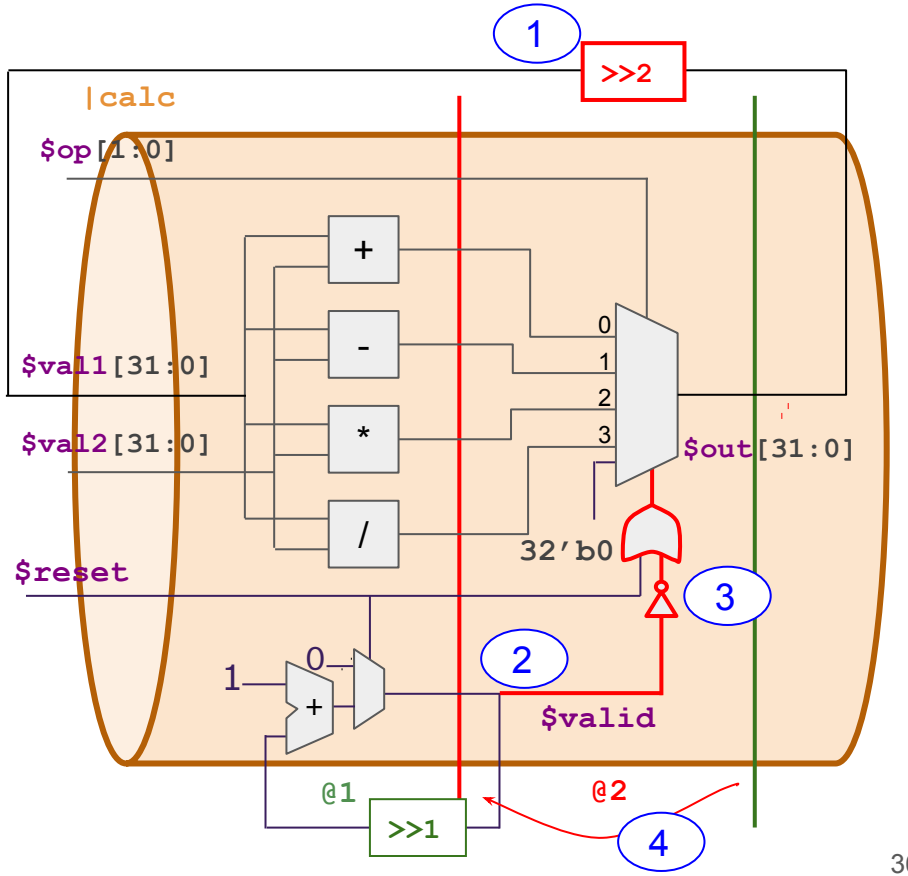
The `$reset = *reset` expression should be moved under the pipeline and pipestage as well.



Lab: 2-Cycle Calculator

At high frequency, we might need to calculate every other cycle.

1. Change alignment of `$out` (to calculate every other cycle).
2. Change counter to single-bit (to indicate every other cycle).
3. Connect `$valid` (to clear alternate outputs).
4. Retime mux to `@2` (to ease timing; no functional change).
5. Verify behavior in waveform.
6. Save.

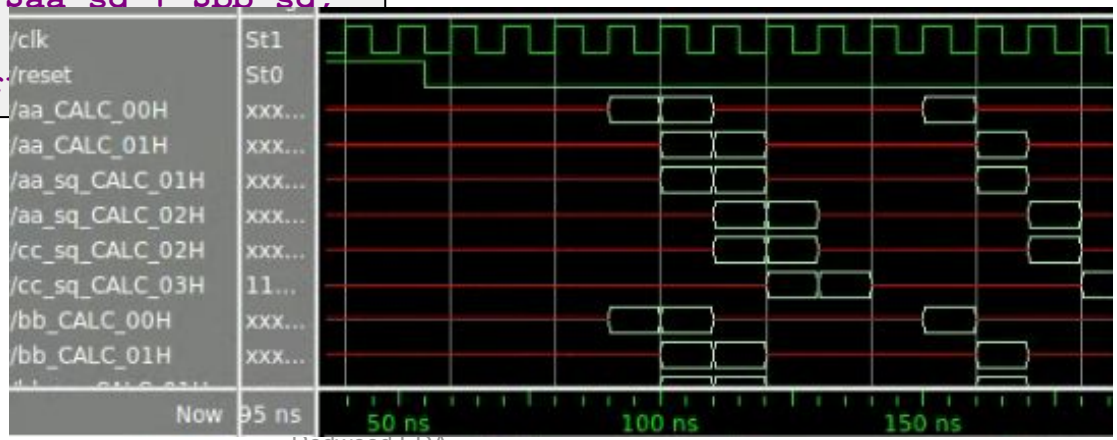


Validity

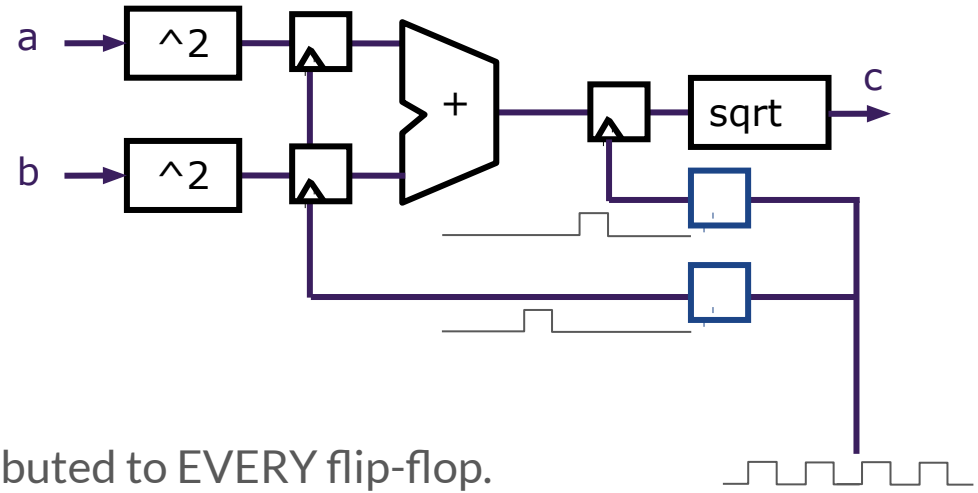
```
|calc
   @1
      $valid = ...;
   ?$valid
      @1
         $aa_sq[31:0] = $aa * $aa;
         $bb_sq[31:0] = $bb * $bb;
      @2
         $cc_sq[31:0] = $aa_sq + $bb_sq;
      @3
         $cc[31:0] = sqr
```

Validity provides:

- Easier debug
- Cleaner design
- Better error checking
- Automated clock gating

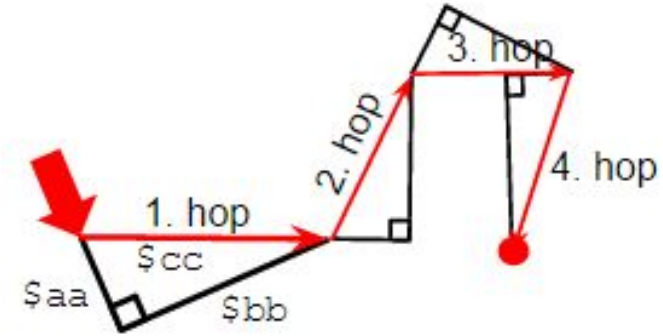


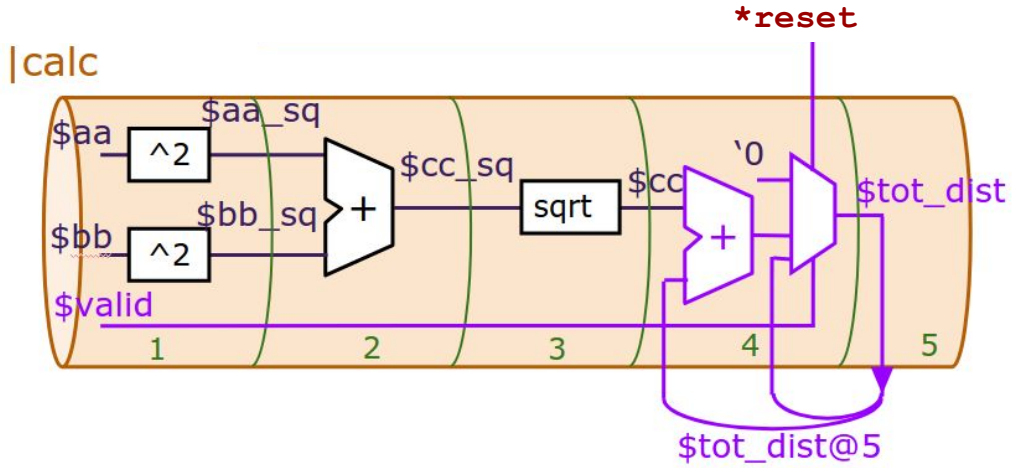
Clock Gating



- Motivation:
  - Clock signals are distributed to EVERY flip-flop.
  - Clocks toggle twice per cycle.
  - This consumes power.
- Clock gating avoids toggling clock signals.
- TL-Verilog can produce fine-grained gating (or enables).

Total Distance (Makerchip walkthrough)





Lab: 2-Cycle Calculator with Validity

1. Use:

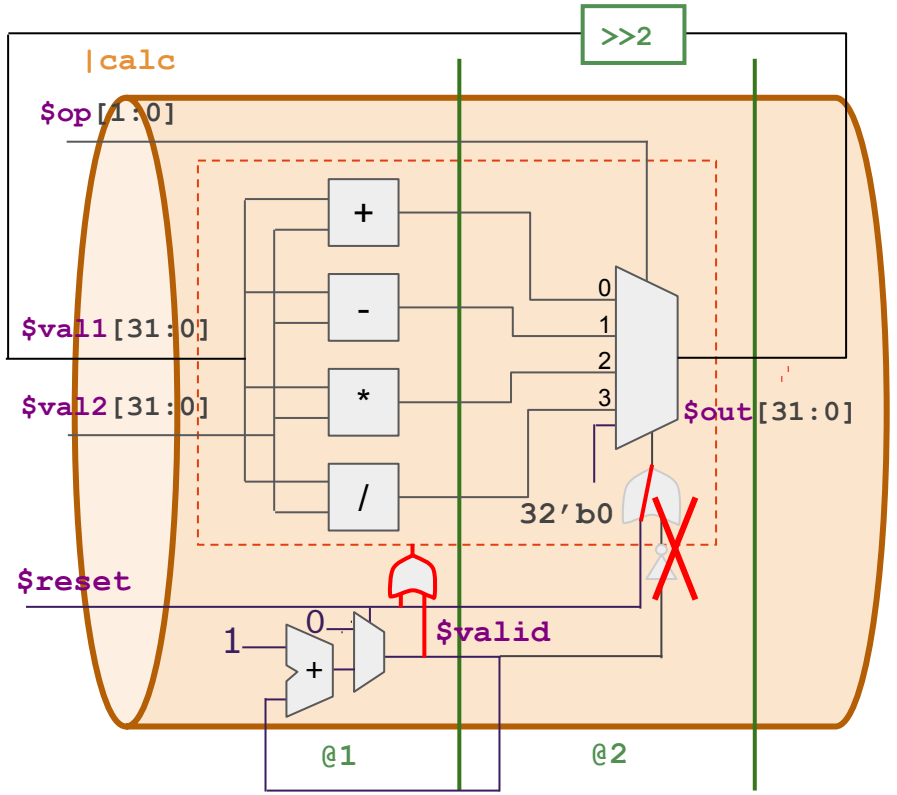
   `$valid_or_reset = $valid || $reset;`

   as a when condition for calculation instead of zeroing `$out`.

   For reference:

```
|calc
   @1
      $valid = ...;
   ?$valid
      @1
         $aa_sq[31:0] = $aa * $aa;
         ...
```

2. Verify behavior in waveform.

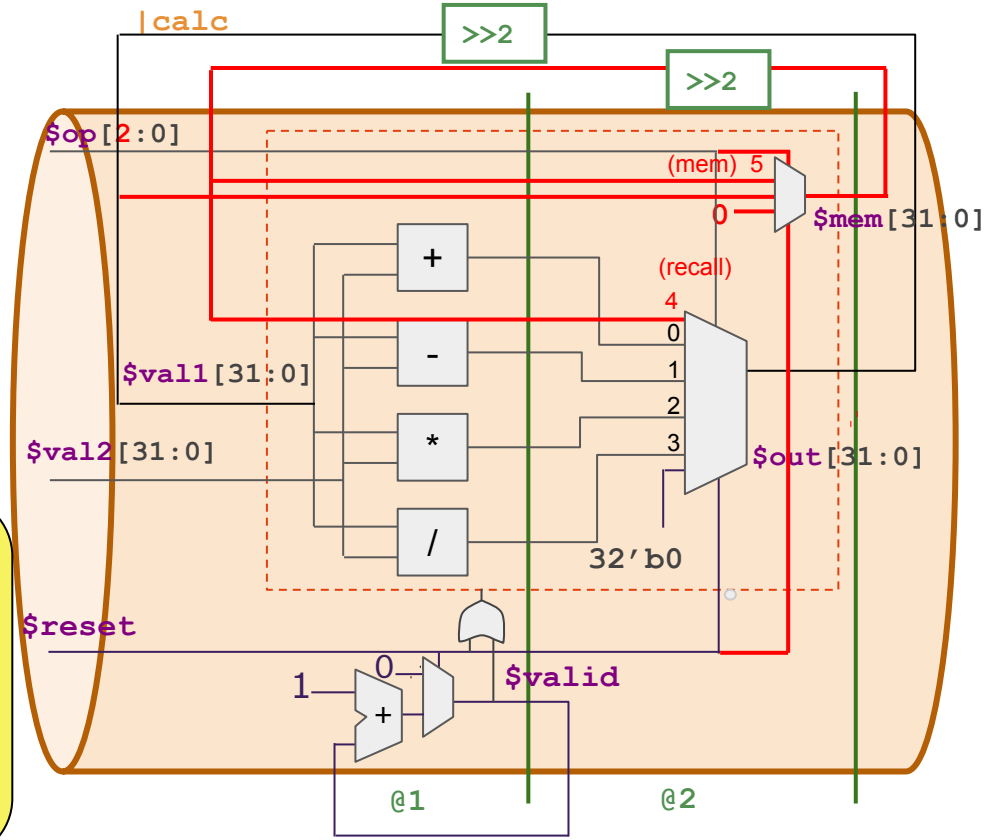


Lab: Calculator with Single-Value Memory

Calculators support “mem” and “recall”, to remember and recall a value.

1. Extend `$op` to 3 bits.
2. Add memory MUX.
3. Select recall value in output MUX.
4. Verify behavior in waveform.

Missing from this logic: it is necessary to provide a default recirculation of $out for $out mux. Previously there was no need for a "retain value" case, but now mem operations must retain $out (and it doesn't hurt to retain for illegal (6, 7) op inputs as well).



Building a RISC-V Core

RISC-V Based MYTH Workshop
MYTH - Microprocessor for You in Thirty Hours

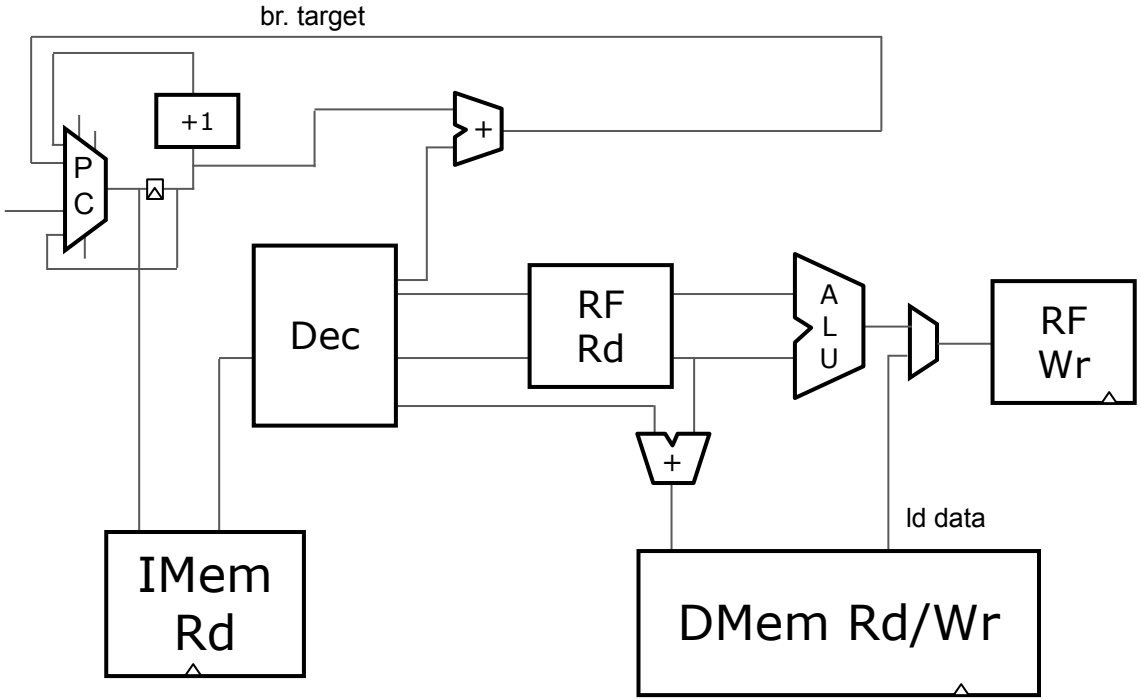
Steve Hoover
Founder, Redwood EDA
July 31, 2020

Day 4 & 5: RISC-V

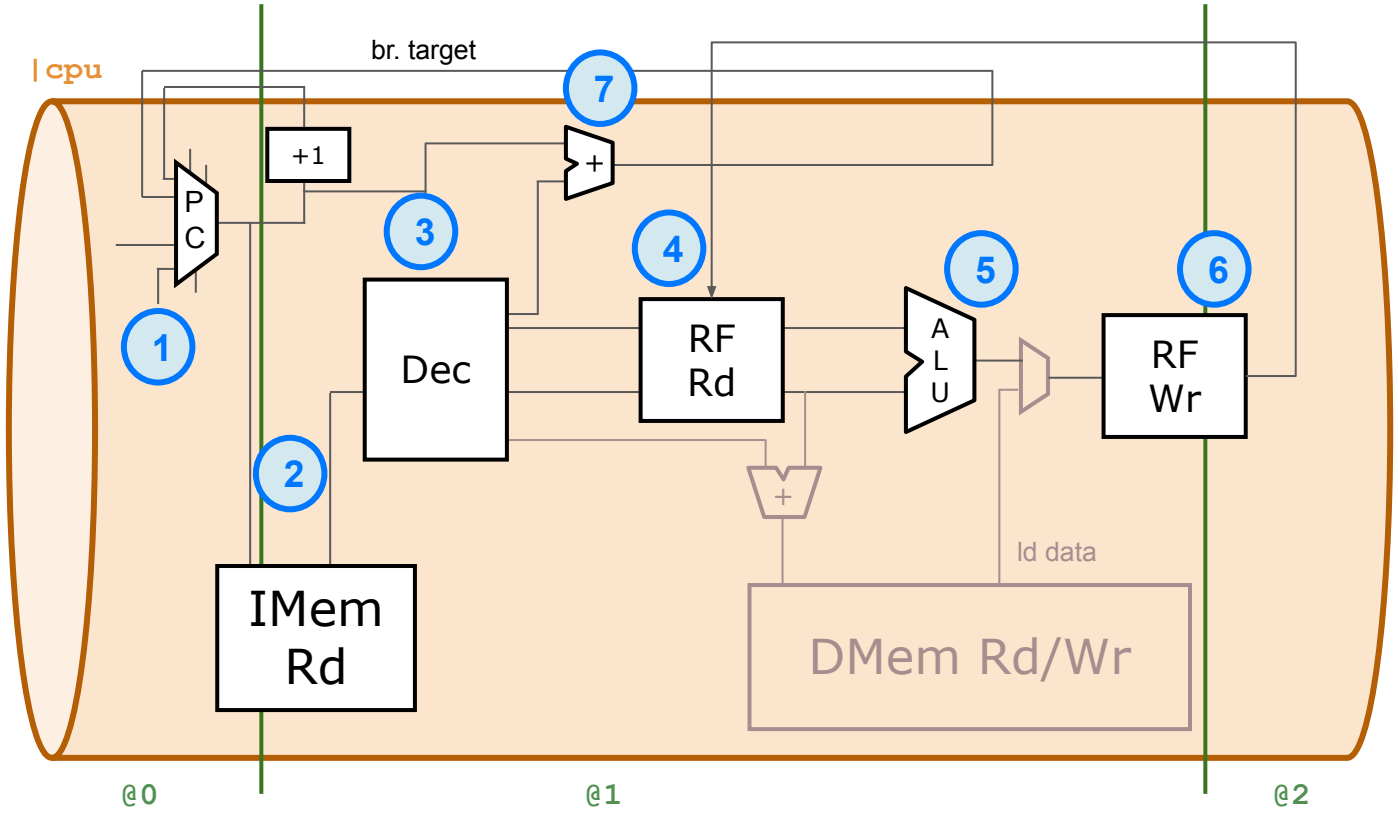
Agenda:

- Simple RISC-V subset
- Pipelined RISC-V subset
- Complete (almost) RISC-V (RV32I)

Example RISC-V Block Diagram



Implementation Plan



Lab: RISC-V Shell Code

Review information at:
`https://github.com/stevehoover/RISC-V_MYTH_Workshop`

Open link for “RISC-V lab starting-point code”.

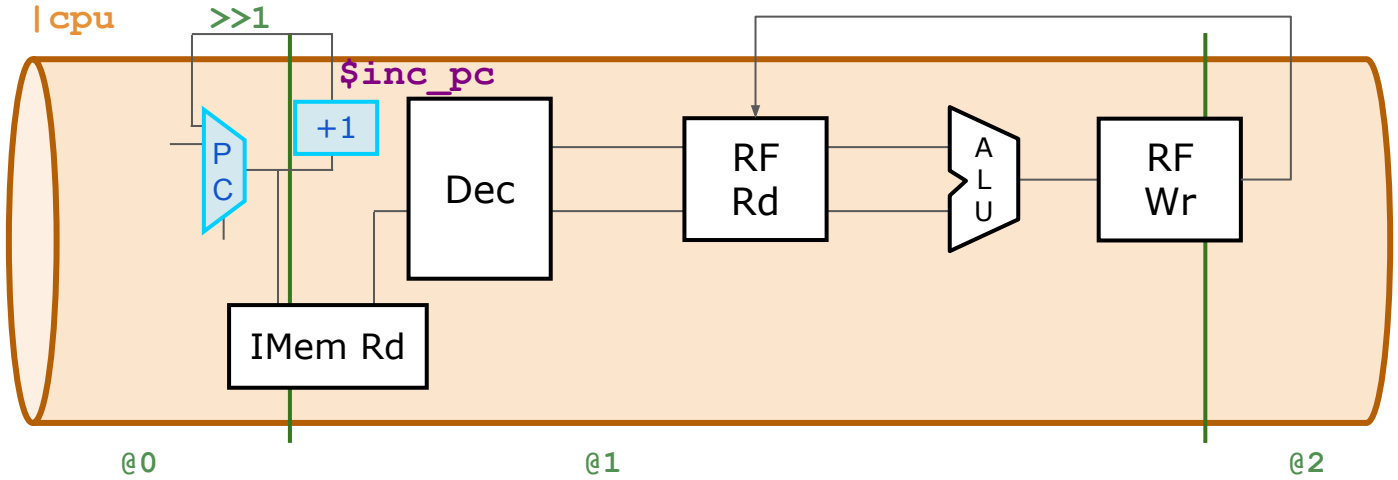
Review code containing (or including):

- A simple RISC-V assembler.
- An instruction memory containing the sum 1..9 test program.
- Commented code for register file and memory.
- Visualization.

Test-driven development: Develop tests first, then functionality.

Lab: Next PC

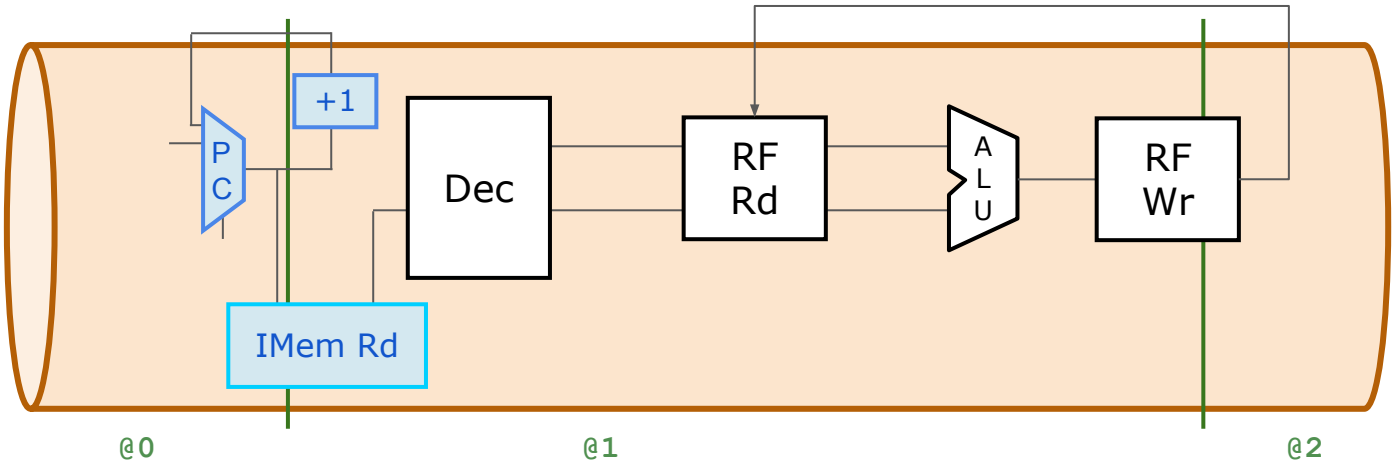
Reset `$pc[31:0]` to 0 if previous instruction was a "reset instruction" (`>>1$reset`), and increment by 1 instruction (`32'd4` bytes) thereafter. (We'll add branch support later.)



Check PC value in simulation and confirm save. PC's after reset should be 0, 4, ...

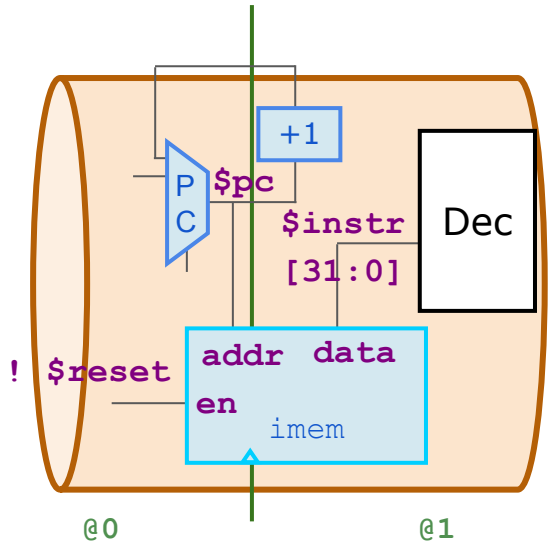
Lab: Fetch (part 1)

Add the instruction memory containing the program (provided).



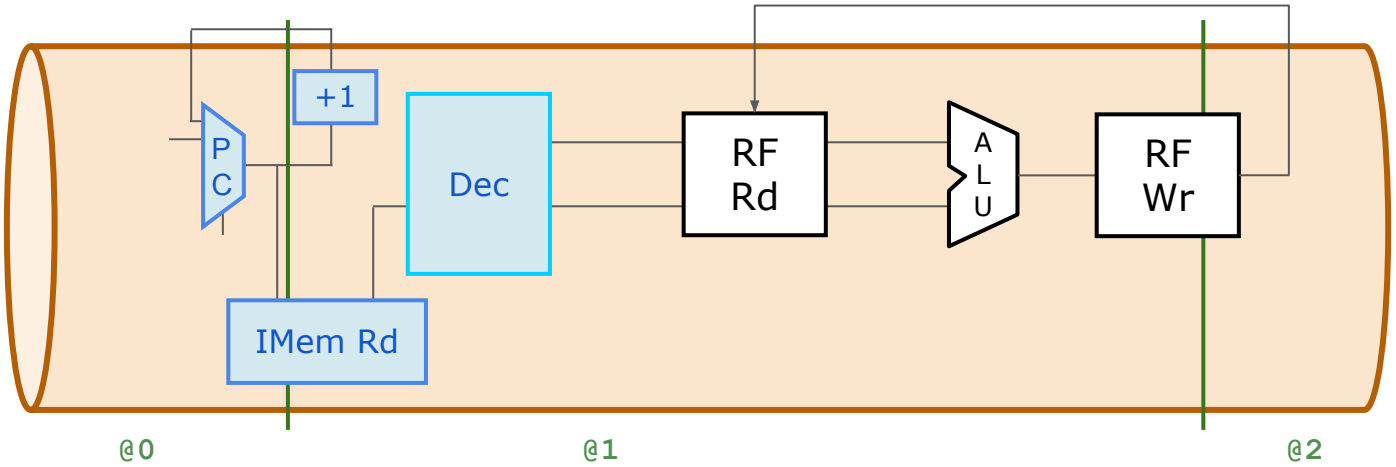
1. Uncomment `//m4+imem(@1)`, and `//m4+cpu_viz(@4)` compile, and observe log errors.

Lab: Fetch (part 2)



1. **`imem`** expects inputs:
   a. In: **`$imem_rd_en`** (read enable)
   b. In: **`$imem_rd_addr`**
      `[M4_IMEM_INDEX_CNT-1:0]`
      and provides output:
   c. Out: **`$imem_rd_data[31:0]`**
2. Connect `imem` interface to read into **`$instr[31:0]`** addressed by **`$pc[M4_IMEM_INDEX_CNT+1:2]`** enabled every cycle after reset.
3. Check **`$instr`** in simulation and confirm save.

Decode



Lab: Instruction Types Decode

instr[6:2] determine instruction type: I, R, S, B, J, U

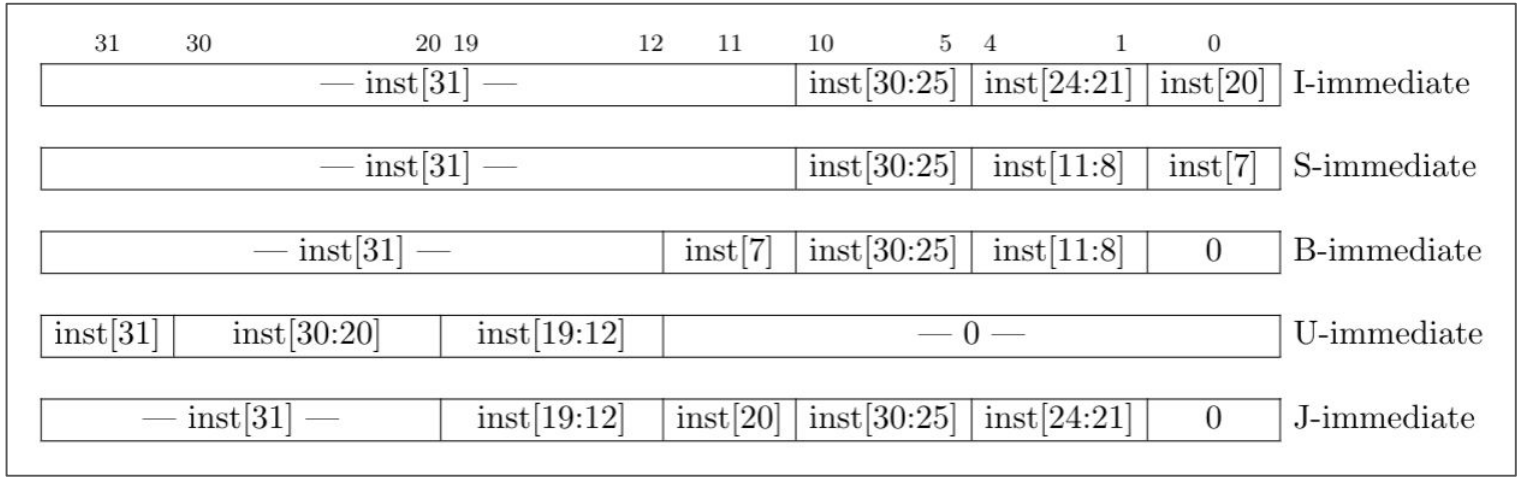
instr[4:2] instr[6:5]	000	001	010	011	100	101	110	111
00	I	I	-	-	I	U	I	-
01	S	S	-	R	R	U	R	-
10	R4	R4	R4	R4	R	-	-	-
11	B	I	-	J	I (unused)	-	-	-

```
$is_i_instr = $instr[6:2] ==? 5'b0000x ||
              $instr[6:2] ==? 5'b001x0 ||
              ...;
...
```

Check behavior in simulation.

Lab: Instruction Immediate Decode

Form `$imm[31:0]` based on instruction type.




```
$imm[31:0] = $is_i_instr ? { {21{$instr[31]}}, $instr[30:20] } :
             $is_s_instr ? {...} :
             ...;
```

Check behavior in simulation.

Lab: Instruction Decode

Extract other instruction fields: `$funct7, $funct3, $rs1, $rs2, $rd, $opcode`

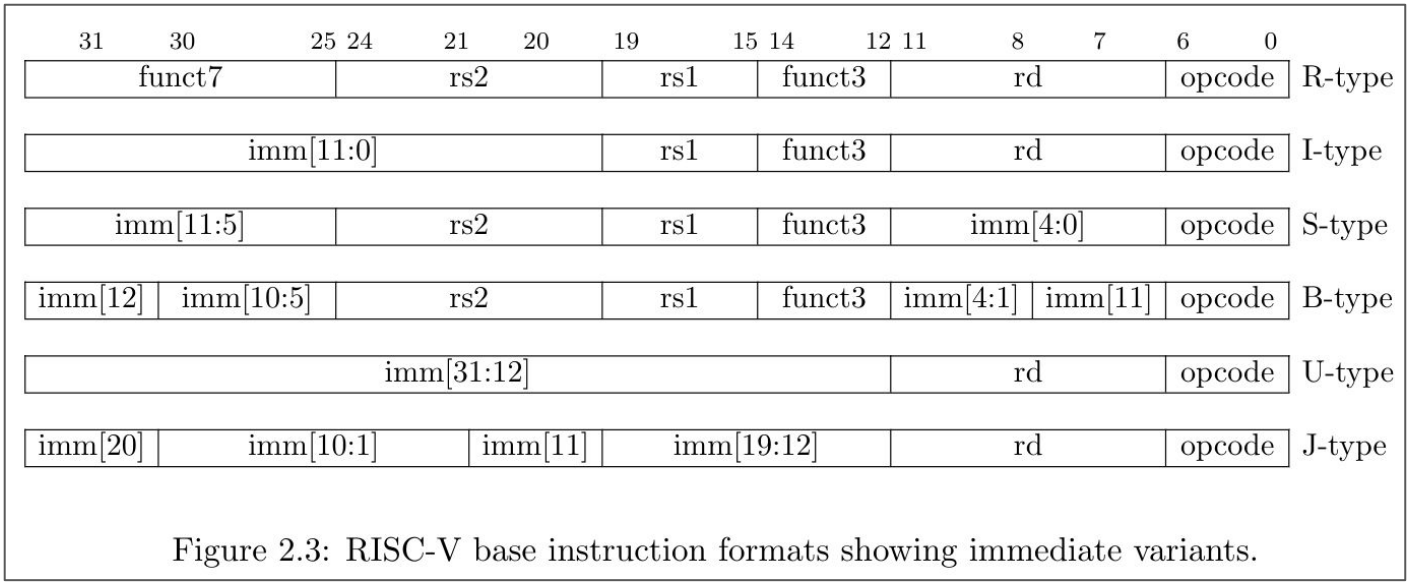


Figure 2.3: RISC-V base instruction formats showing immediate variants.

```
$rs2[4:0] = $instr[24:20];
...
```

Check behavior in simulation.

Lab: RISC-V Instruction Field Decode

Let's use when conditions

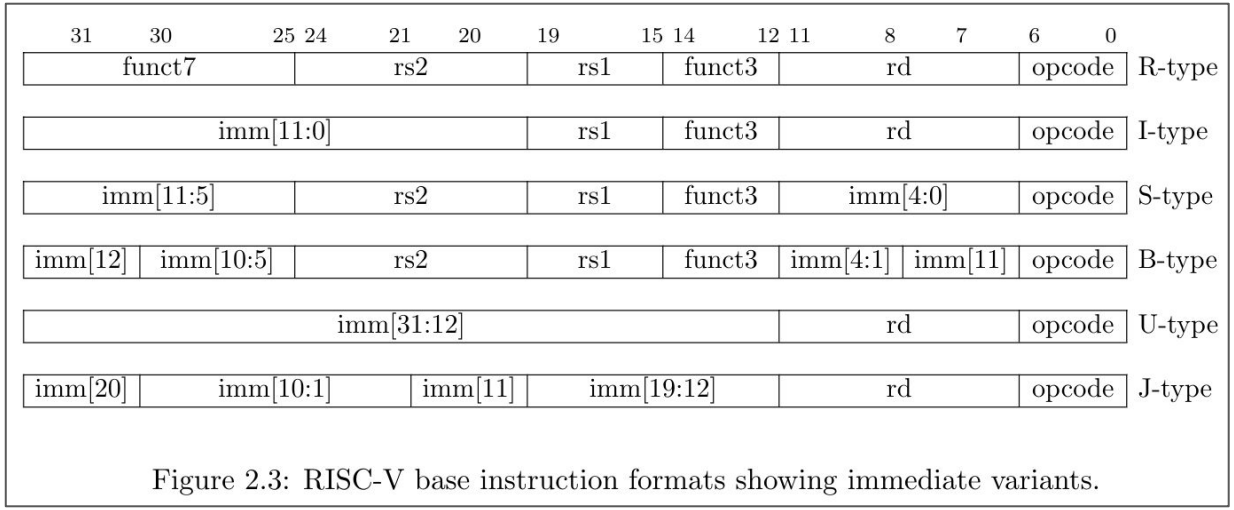


Figure 2.3: RISC-V base instruction formats showing immediate variants.

```
$rs2_valid = $is_r_instr || $is_s_instr || $is_b_instr;
?$rs2_valid
   $rs2[4:0] = $instr[24:20];
...
```

Check behavior in simulation.

Lab: Instruction Decode

`$funct7` is not valid for SLLI/SRLI/SRAI, so we shouldn't be using it for these instructions. In any case, it will not actually cause any harm for this workshop.

RV32I Base Instruction Set (except CSR*, FENCE, ECALL, EBREAK).

funct3	opcode	
	0110111	LUI
	0010111	AUIPC
	1101111	JAL
000	1100111	JALR
000	1100011	BEQ
001	1100011	BNE
100	1100011	BLT
101	1100011	BGE
110	1100011	BLTU
111	1100011	BGEU
000	0000011	LB
001	0000011	LH
010	0000011	LW
100	0000011	LBU
101	0000011	LHU
000	0100011	SB
001	0100011	SH
010	0100011	SW
000	0010011	ADDI
010	0010011	SLTI

funct7[5]	funct3	opcode	
	011	0010011	SLTIU
	100	0010011	XORI
	110	0010011	ORI
	111	0010011	ANDI
0	001	0010011	SLLI
0	101	0010011	SRLI
1	101	0010011	SRAI
0	000	0110011	ADD
1	000	0110011	SUB
0	001	0110011	SLL
0	010	0110011	SLT
0	011	0110011	SLTU
0	100	0110011	XOR
0	101	0110011	SRL
1	101	0110011	SRA
0	110	0110011	OR
0	111	0110011	AND

Complete circled instructions.

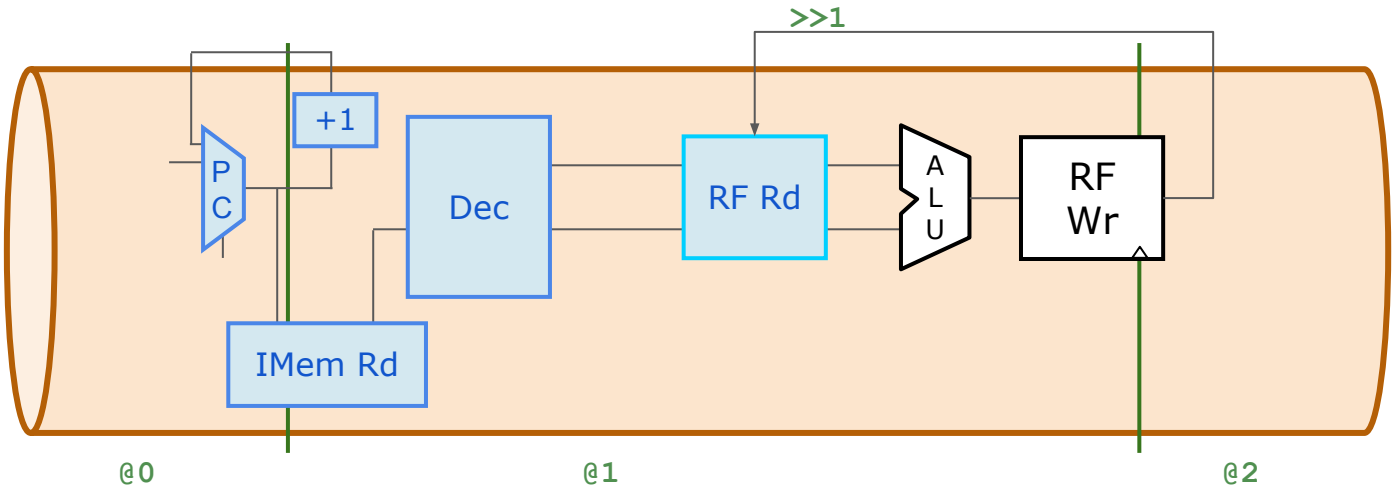
```
$dec_bits[10:0] =
   {$funct7[5],$funct3,$opcode};
$is_beq = $dec_bits ==?
          11'bx_000_1100011;

// Until instrs are implemented,
// quiet down the warnings.
`BOGUS_USE($is_beq $is_bne ...)
```

backtick

(no comma)

Check behavior in simulation and confirm save.

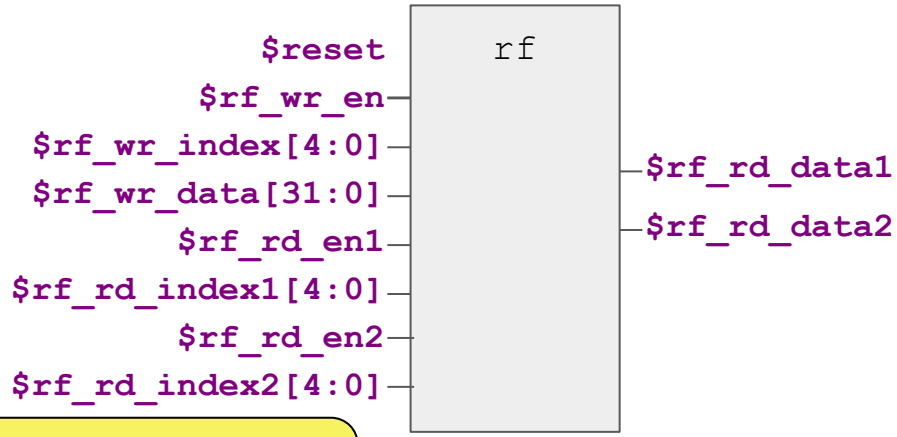
Register File Read



Lab: Register File Read

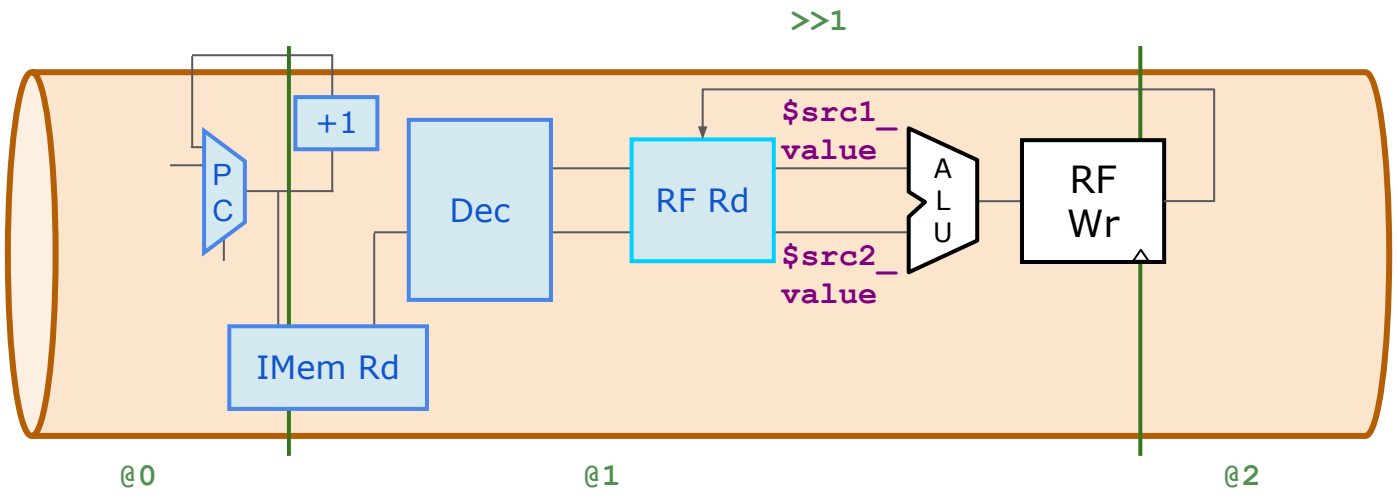
1. Uncomment `//m4+rf(@1, @1)` instantiation. (Default values are provided for inputs.)
2. Provide proper input assignments to enable RF read (**`rd`**) of **`$rs1/2`** when **`$rs1/2_valid`**.
3. Debug in simulation. Note that, on reset, register values are set to their index (e.g. x5 = 32'd5) so you can see something meaningful in simulation.

2-read, 1-write register file:



Your design is growing. You might notice the waveform viewer has hidden the signals. Look for the "+" to open them.

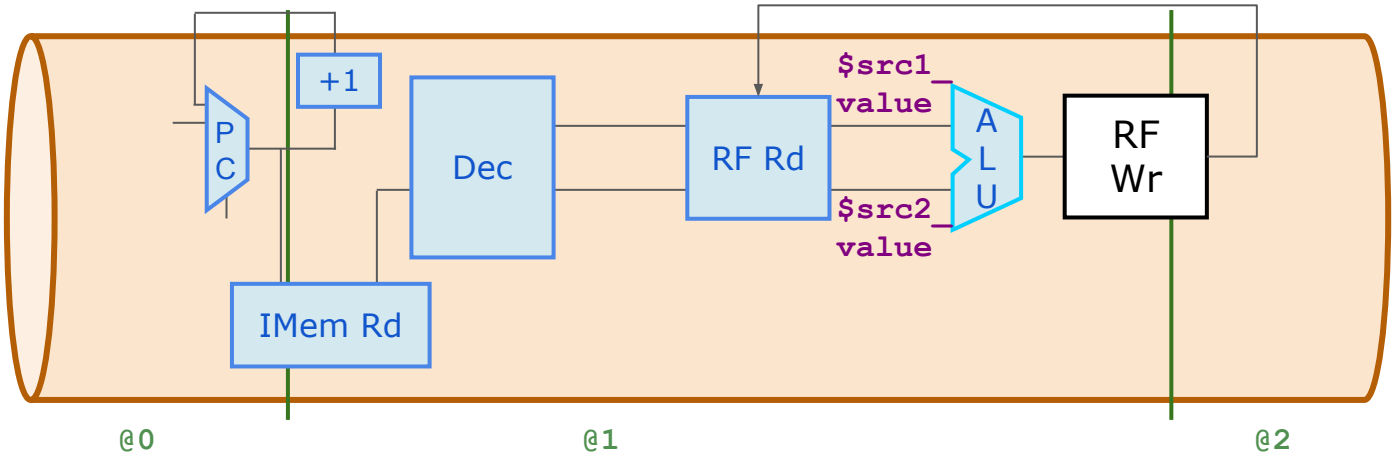
Lab: Register File Read (part 2)



Assign `$src1/2_value[31:0]` to register file outputs.

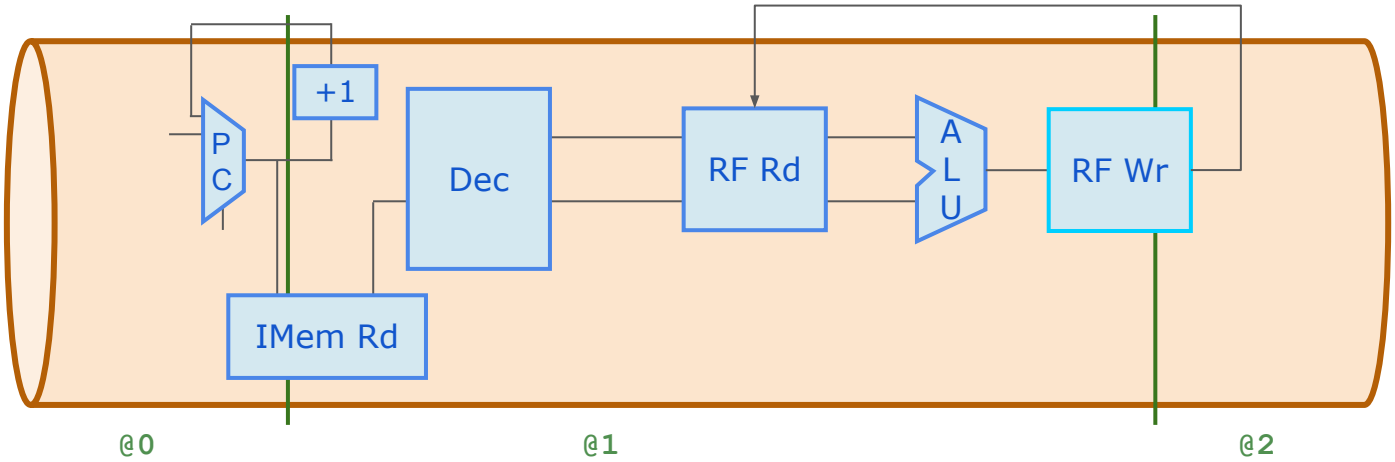
Lab: ALU

Assign the ALU **`$result`** for ADD and ADDI. (You'll fill in others later.)




```
$result[31:0] =
    $is_addi ? $src1_value + $imm :
    ...
               32'bx;
```

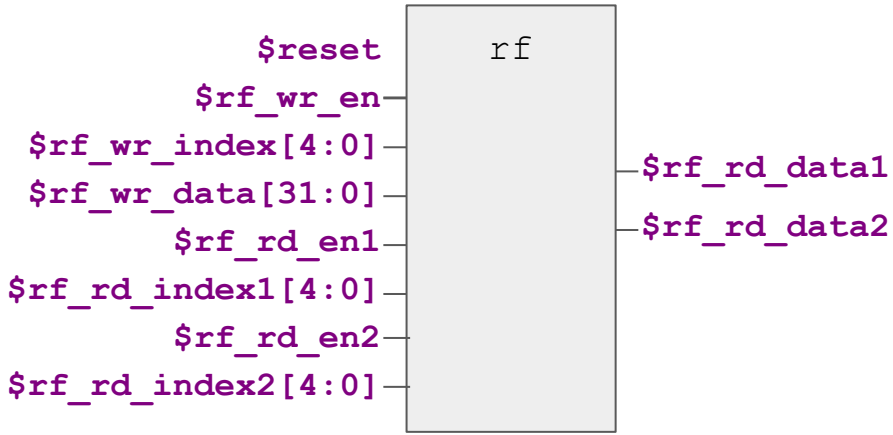
Register File Write



Lab: Register File Write

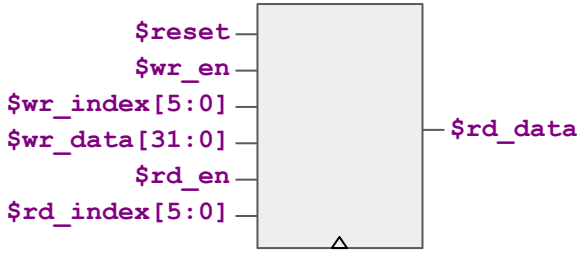
1. Provide proper input assignments to enable RF write (`wr`) of `$result` to `$rd` (dest reg) when `$rd_valid` for a valid instruction.
2. Debug in simulation. Should be writing and reading registers.
3. But wait, in RISC-V, x0 is "always-zero". Writes should be ignored. Add logic to disable write if `$rd` is 0.
4. Save outside of Makerchip.

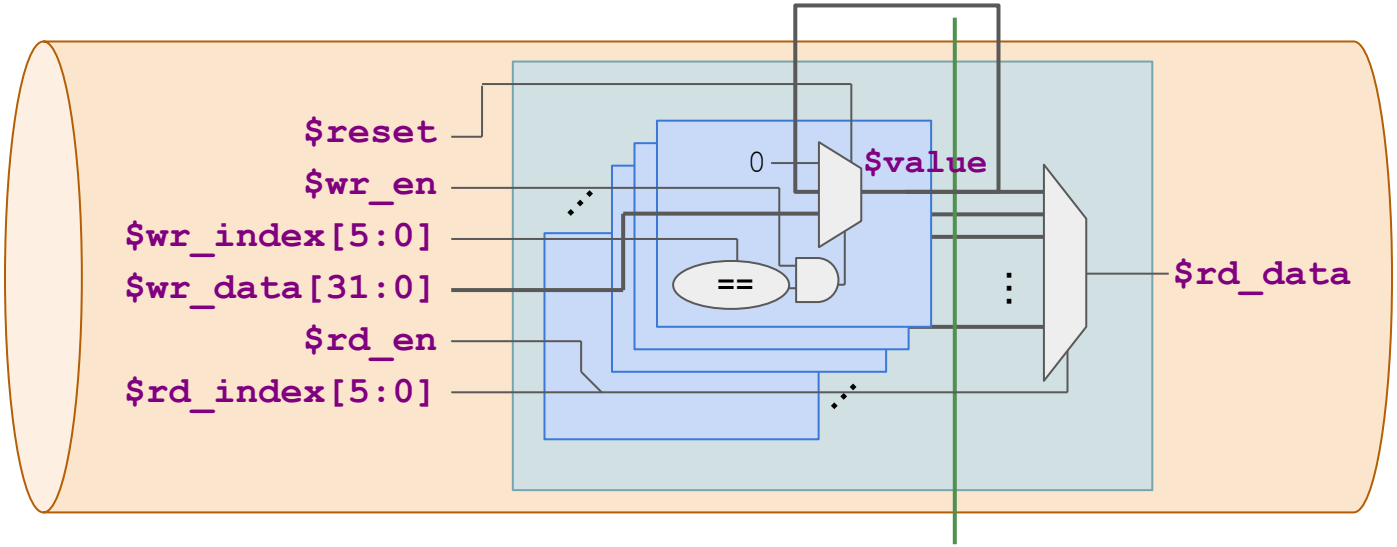
2-read, 1-write register file:



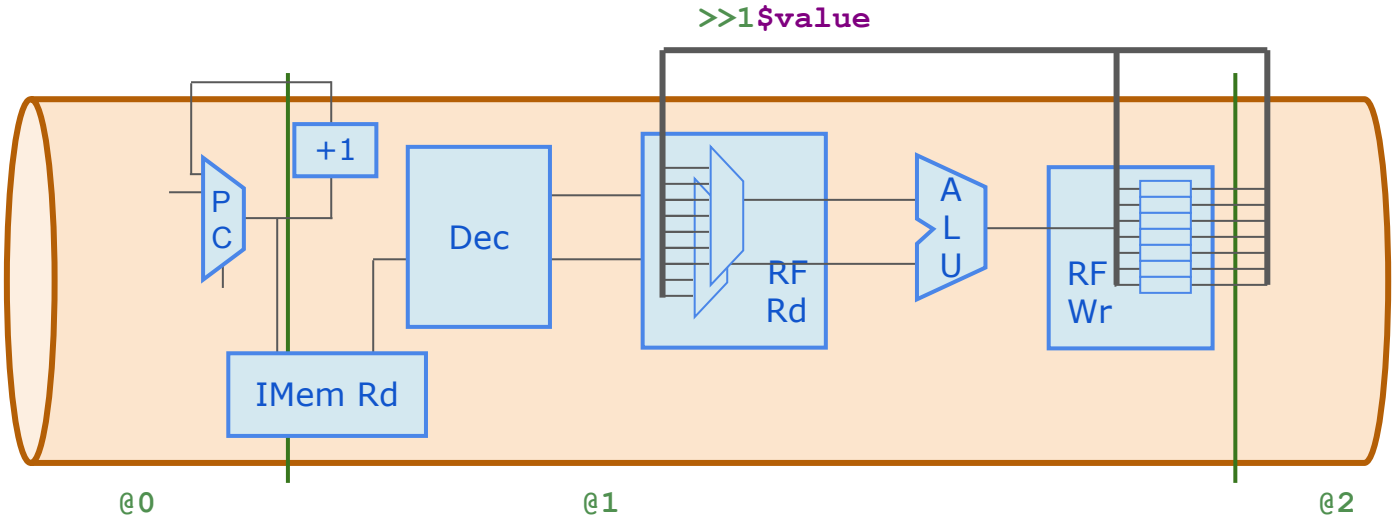
Arrays - What's inside?

A low-level implementation of a 1-read, 1-write, array:

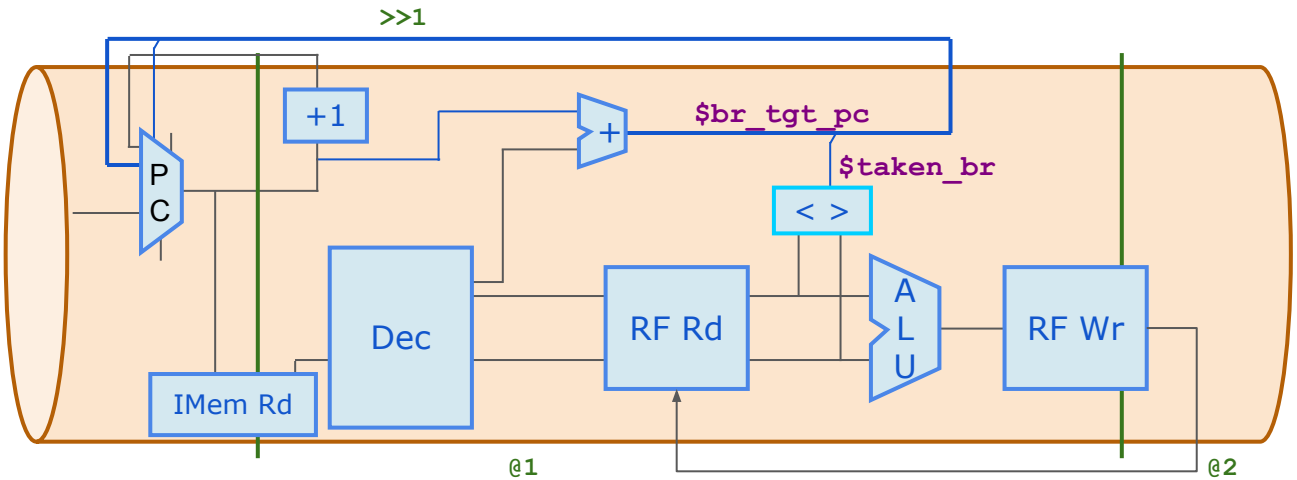




Register File - Detailed



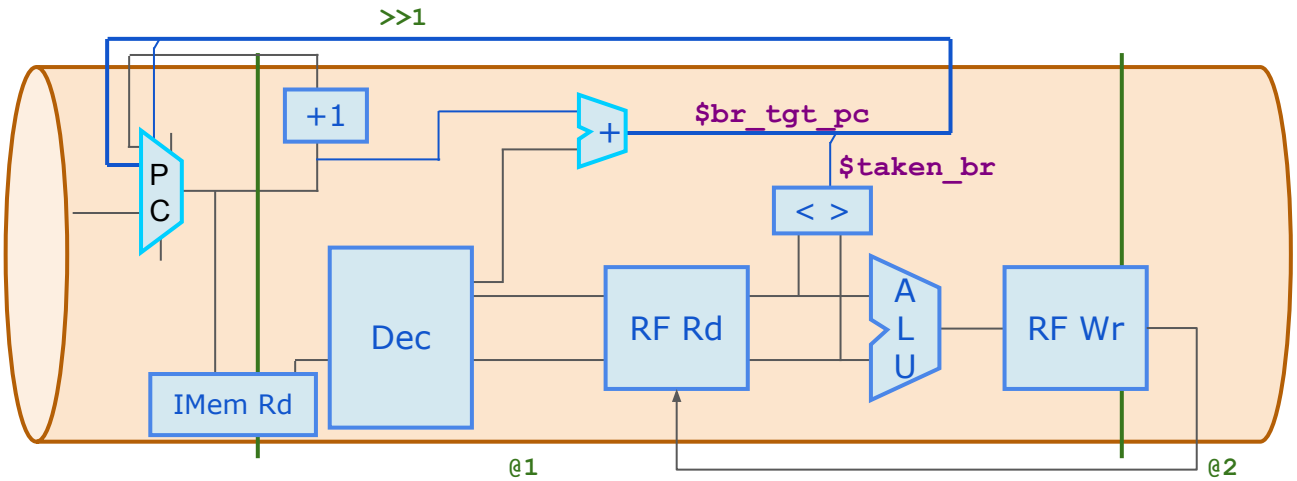
Lab: Branches



1. Determine `$taken_br = ...` as a ternary expression based on `$is_bxx`, defaulting to `1'b0`.
2. Confirm save.

BEQ: ==
BNE: !=
BLT: (x1 < x2) ^ (x1[31] != x2[31])
BGE: (x1 >= x2) ^ (x1[31] != x2[31])
BLTU: <
BGEU: >=

Lab: Branches



1. Compute **`$br_tgt_pc`** (PC + imm)
2. Modify **`$pc`** MUX expression to use the previous **`$br_tgt_pc`** if the previous instruction was **`$taken_br`**.
3. Check behavior in simulation. Program should now sum values {1..9}!
4. Debug as needed, and save outside of Makerchip.

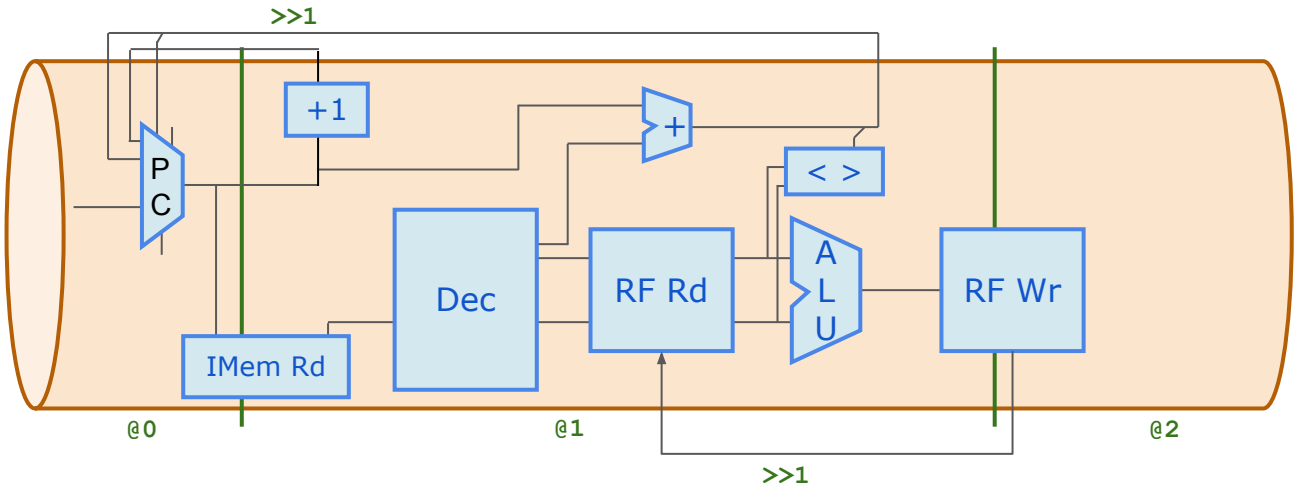
Lab: Testbench

Tell Makerchip when simulation passes by monitoring the value in register x10 (containing the sum) (within `@1`):

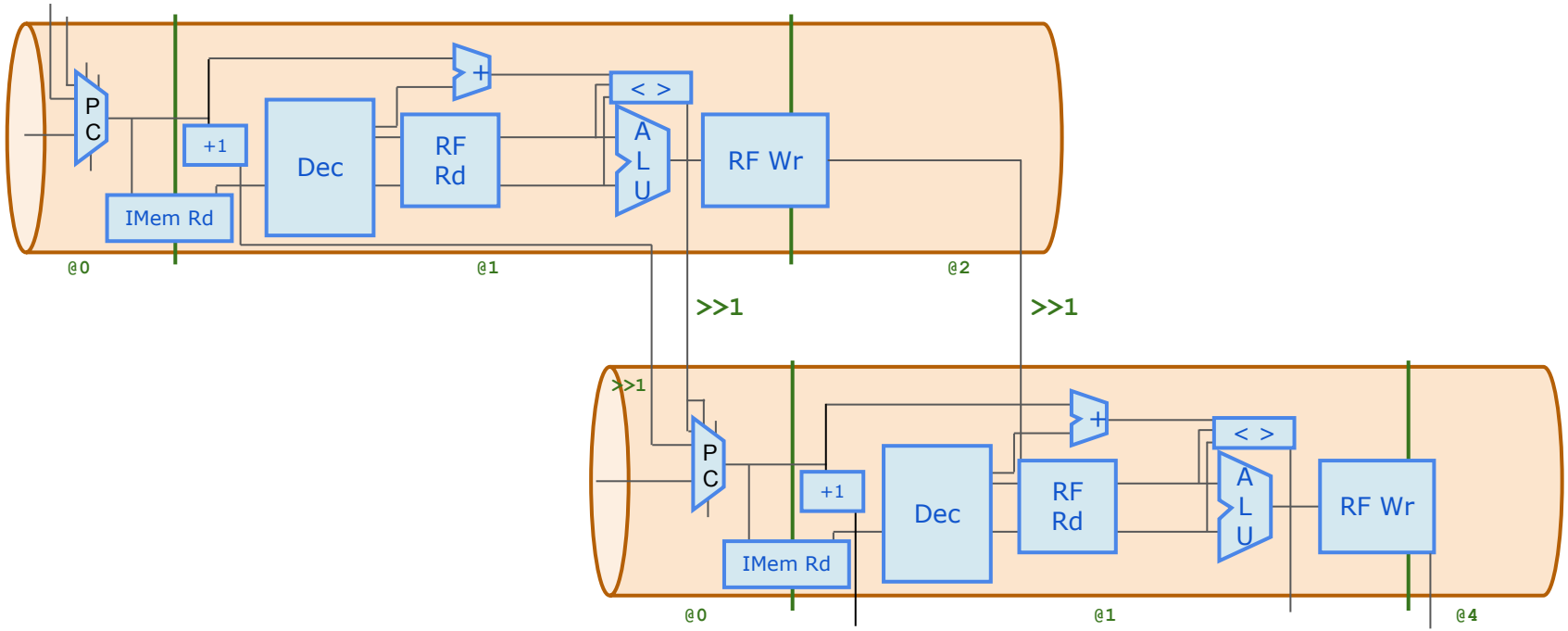
```
*passed = |cpu/xreg[10]>>5$value == (1+2+3+4+5+6+7+8+9);
```

Check log for passed message.

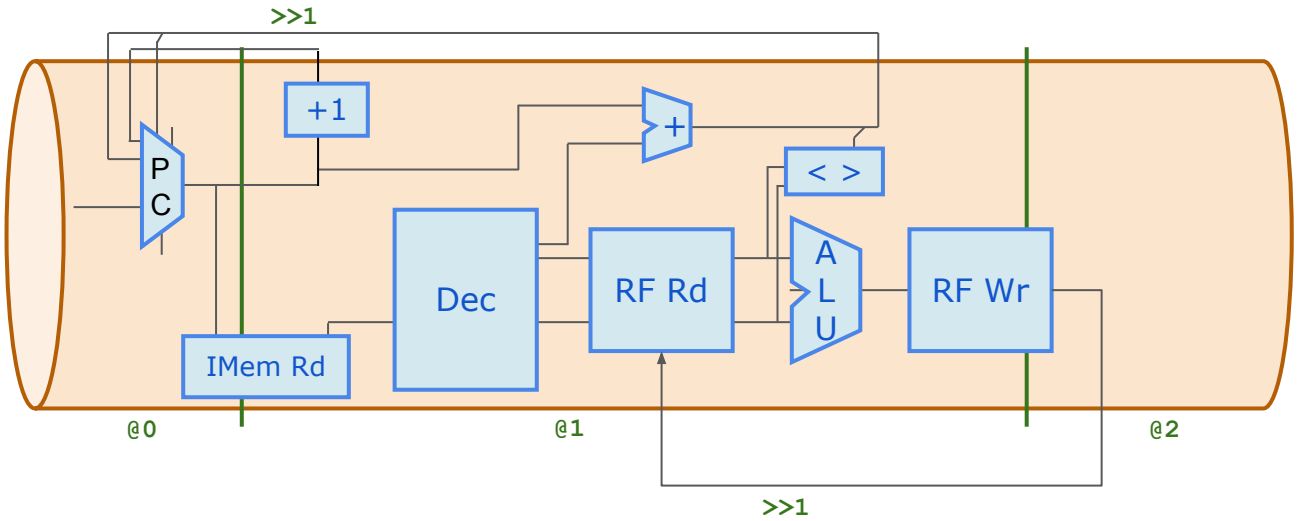
Pipelining Your RISC-V



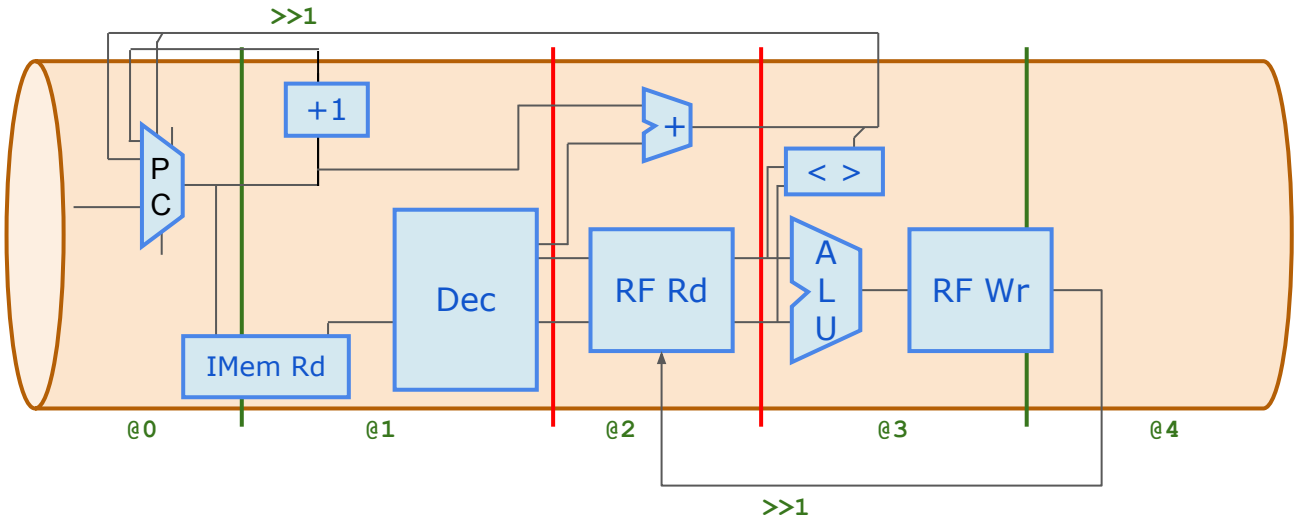
Waterfall Logic Diagram



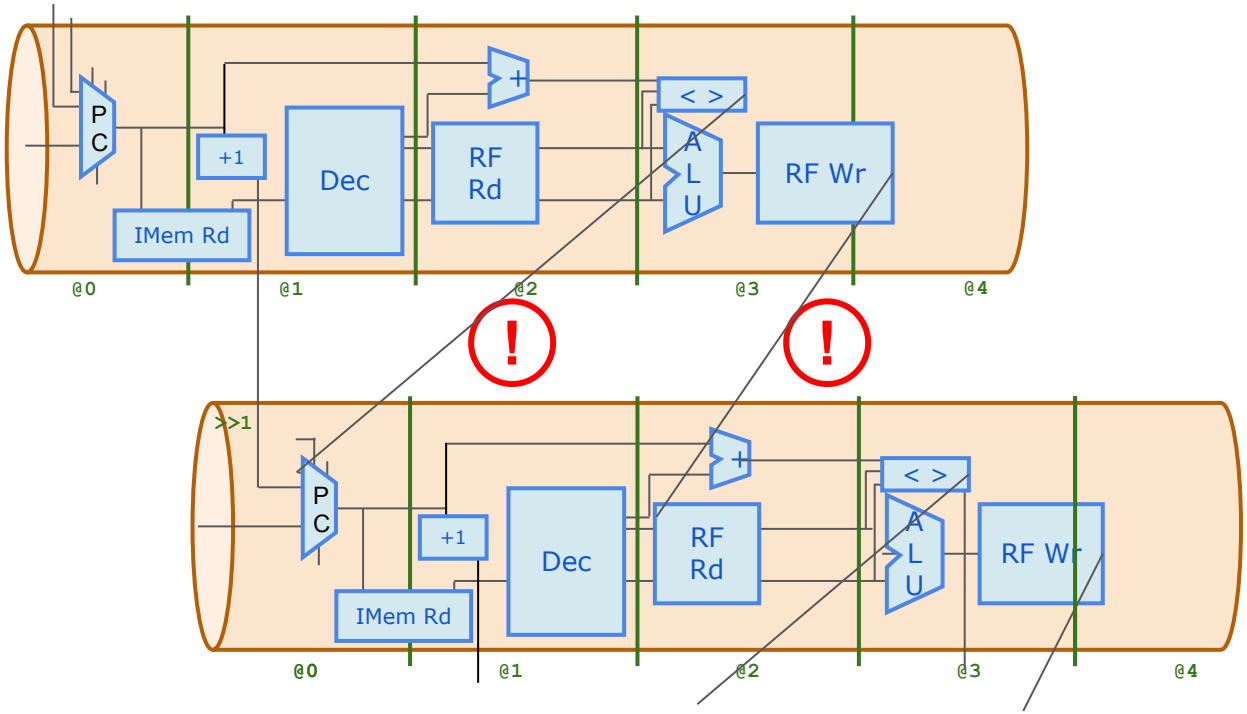
Pipelining Your RISC-V



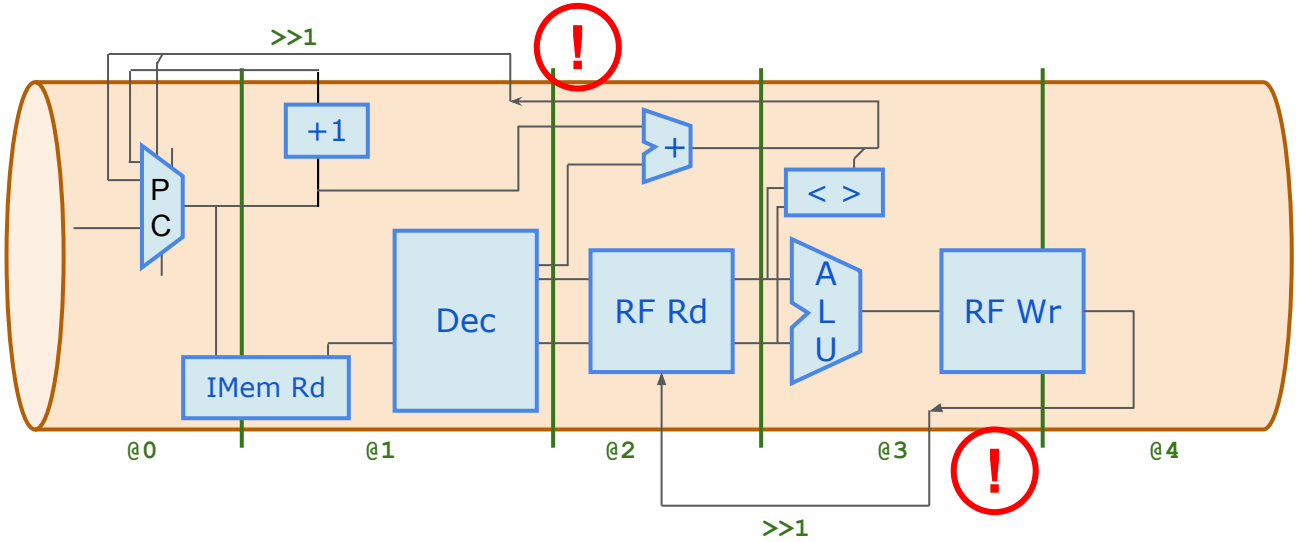
Pipelining Your RISC-V



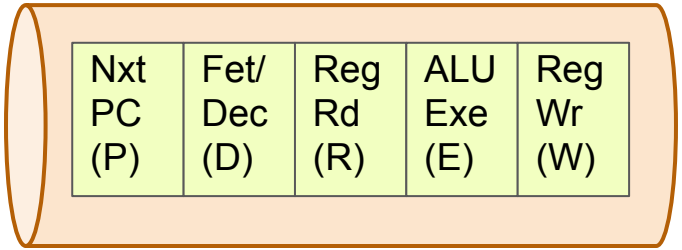
Waterfall Logic Diagram



Pipelining Your RISC-V

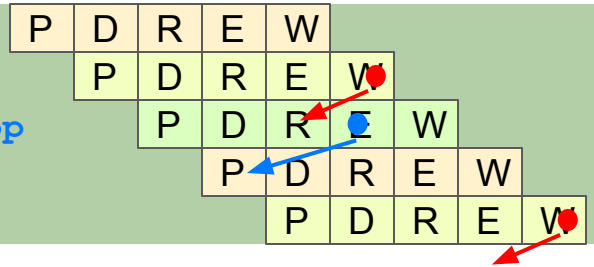


RISC-V Waterfall Diagram & Hazards

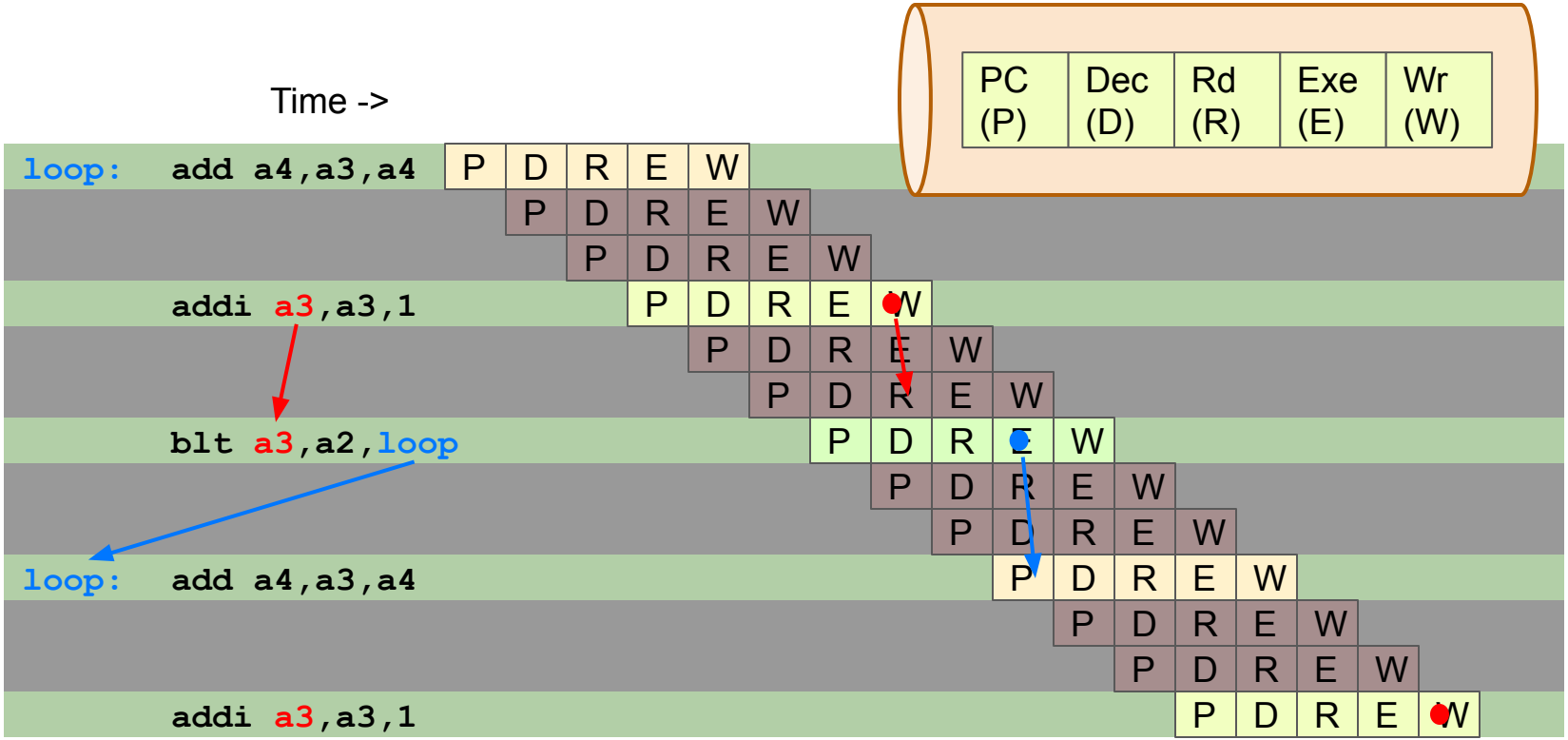


Time ->

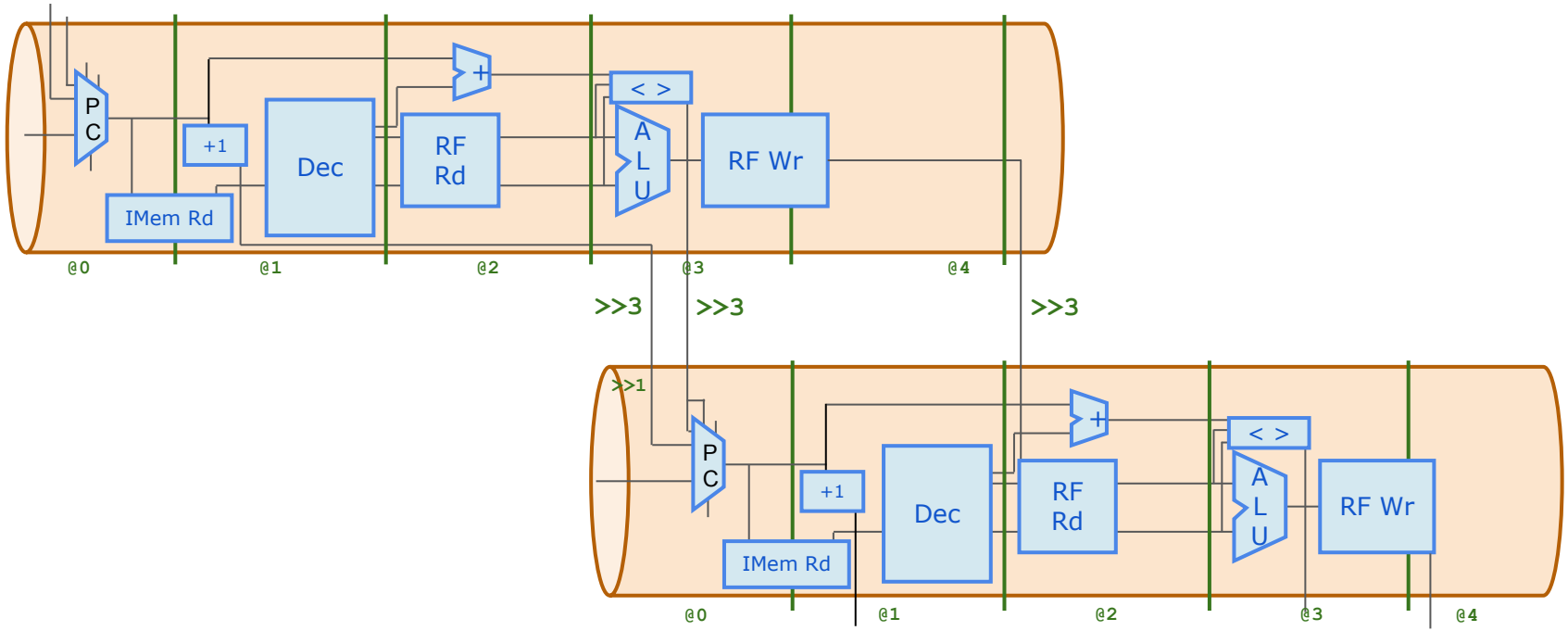
```
loop:   add a4,a3,a4
        addi a3,a3,1
        blt a3,a2,loop
loop:   add a4,a3,a4
        addi a3,a3,1
        ...
```



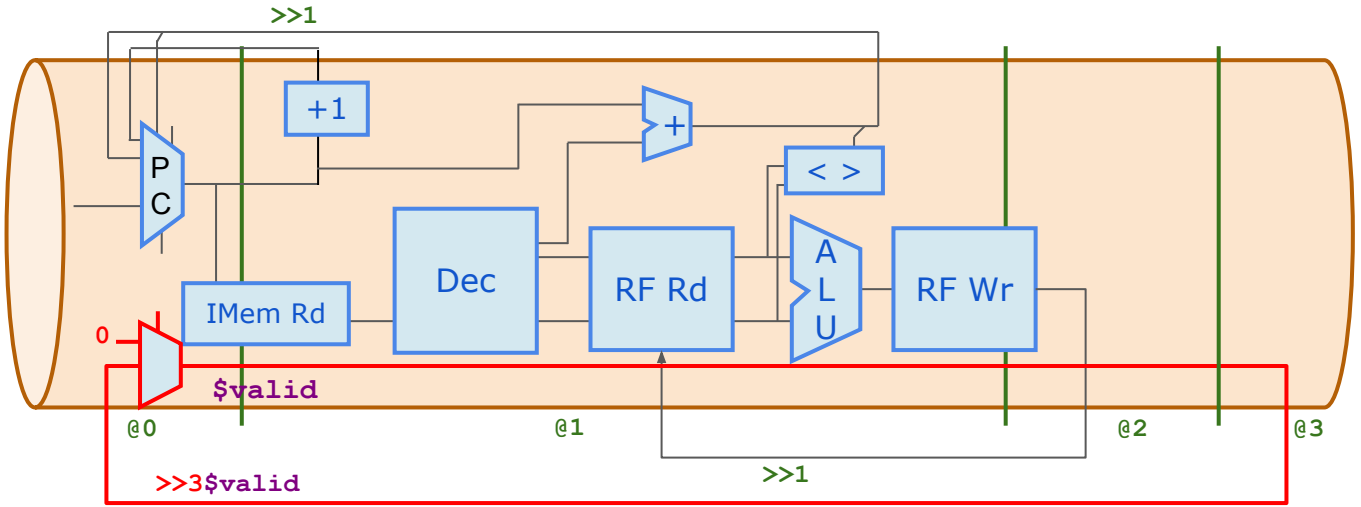
RISC-V Waterfall Diagram & Hazards



Waterfall Logic Diagram

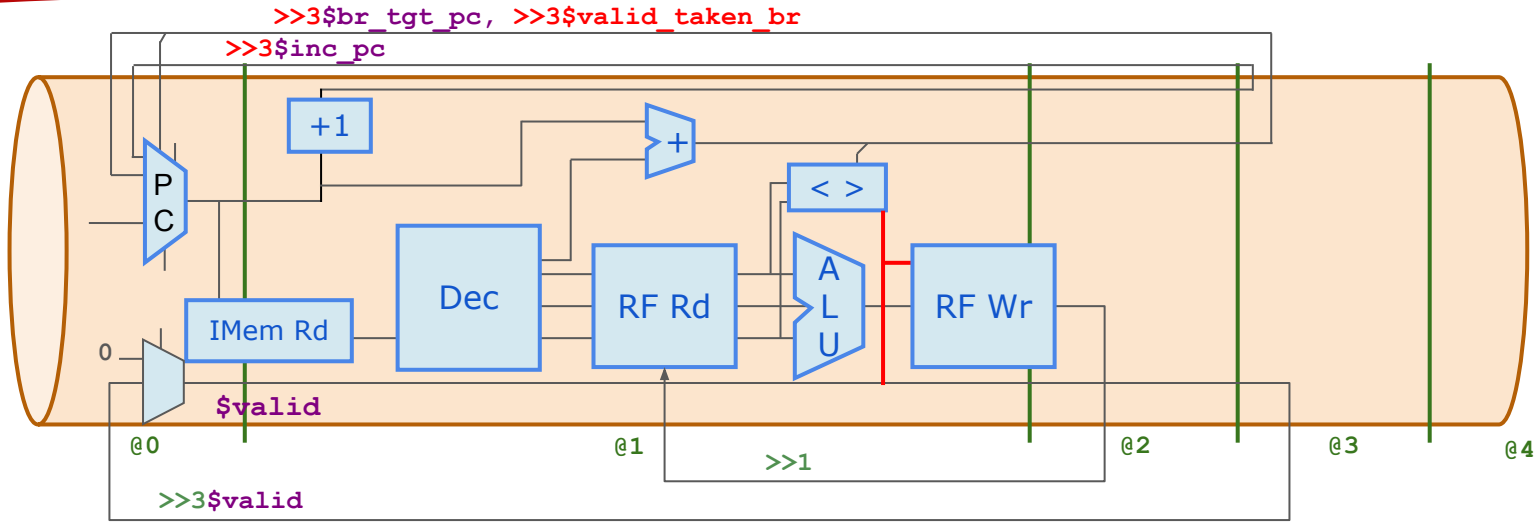


Lab: 3-Cycle $valid



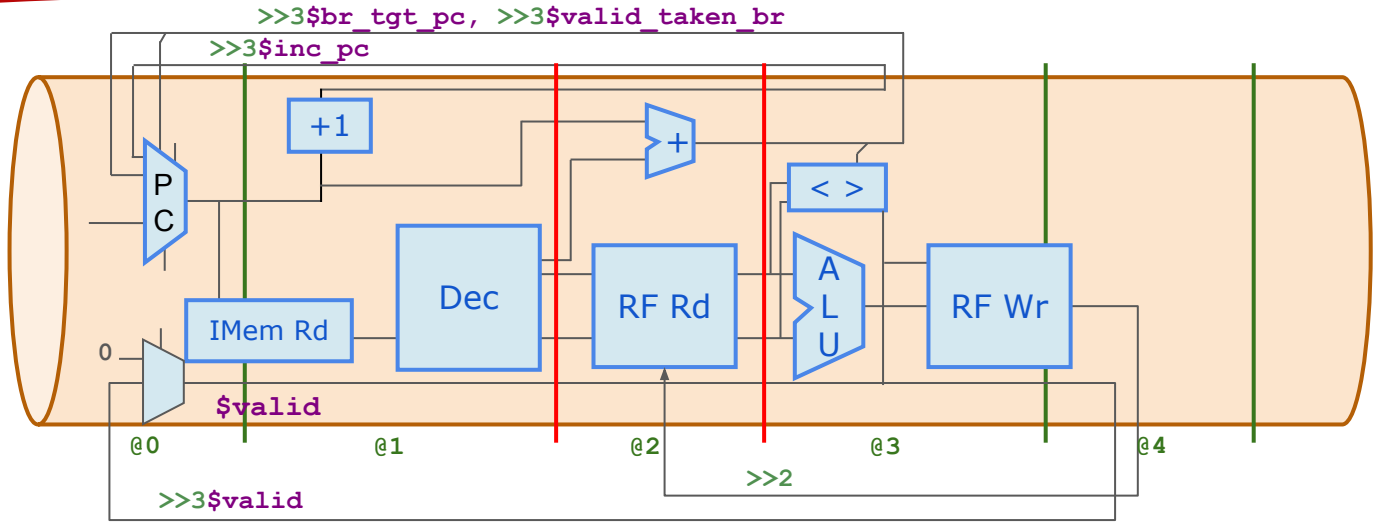
1. Create **$start** to provide first **$valid** pulse (reset last cycle, but not this cycle).
2. Create **$valid**: 0 during **$reset**, 1 for **$start**, **>>3$valid** o/w. Check simulation.

Lab: 3-Cycle RISC-V



1. Avoid writing RF for invalid instructions.
2. Avoid redirecting PC for invalid (branch) instructions.
   Introduce: `$valid_taken_br = $valid && $taken_br;` and use it in PC mux.
3. Update inter-instruction dependency alignments (`>>3`).
4. Debug until passing. Confirm save.

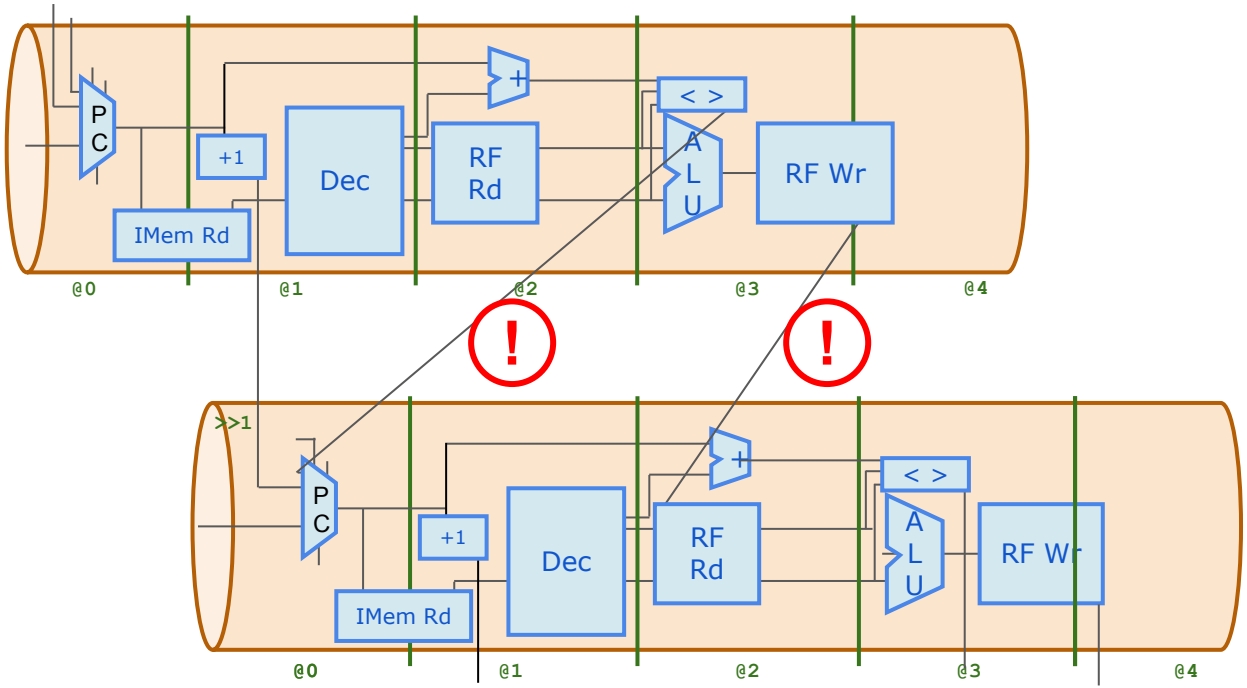
Lab: 3-Cycle RISC-V



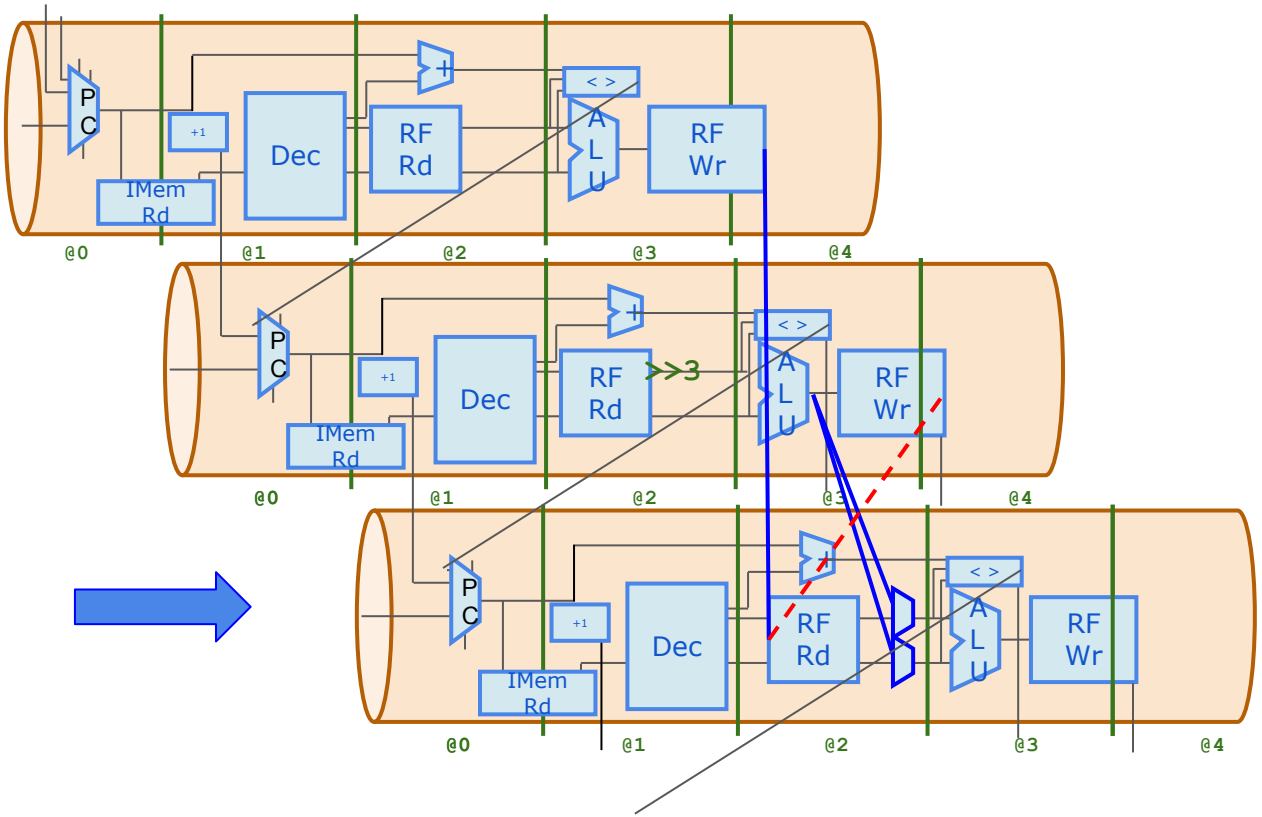
1. Partition logic into pipeline stages as above. Add stages; cut-n-paste code.
2. For RF use `m4+rf(@2, @3)`, implying `>>2`. (Since previous 2 instructions do not update RF, `>>1`, `>>2`, `>>3` are functionally equivalent.)
3. Debug as needed.

(You just changed most of your RTL code and added many lines!)

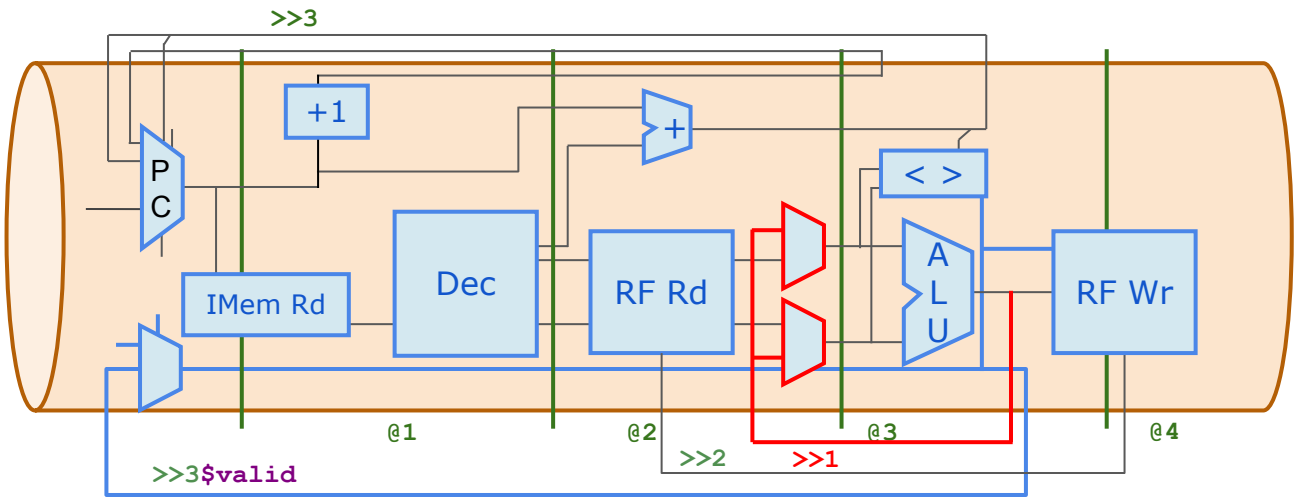
~1 Instruction Per Cycle (~1 IPC)



Register File Bypass

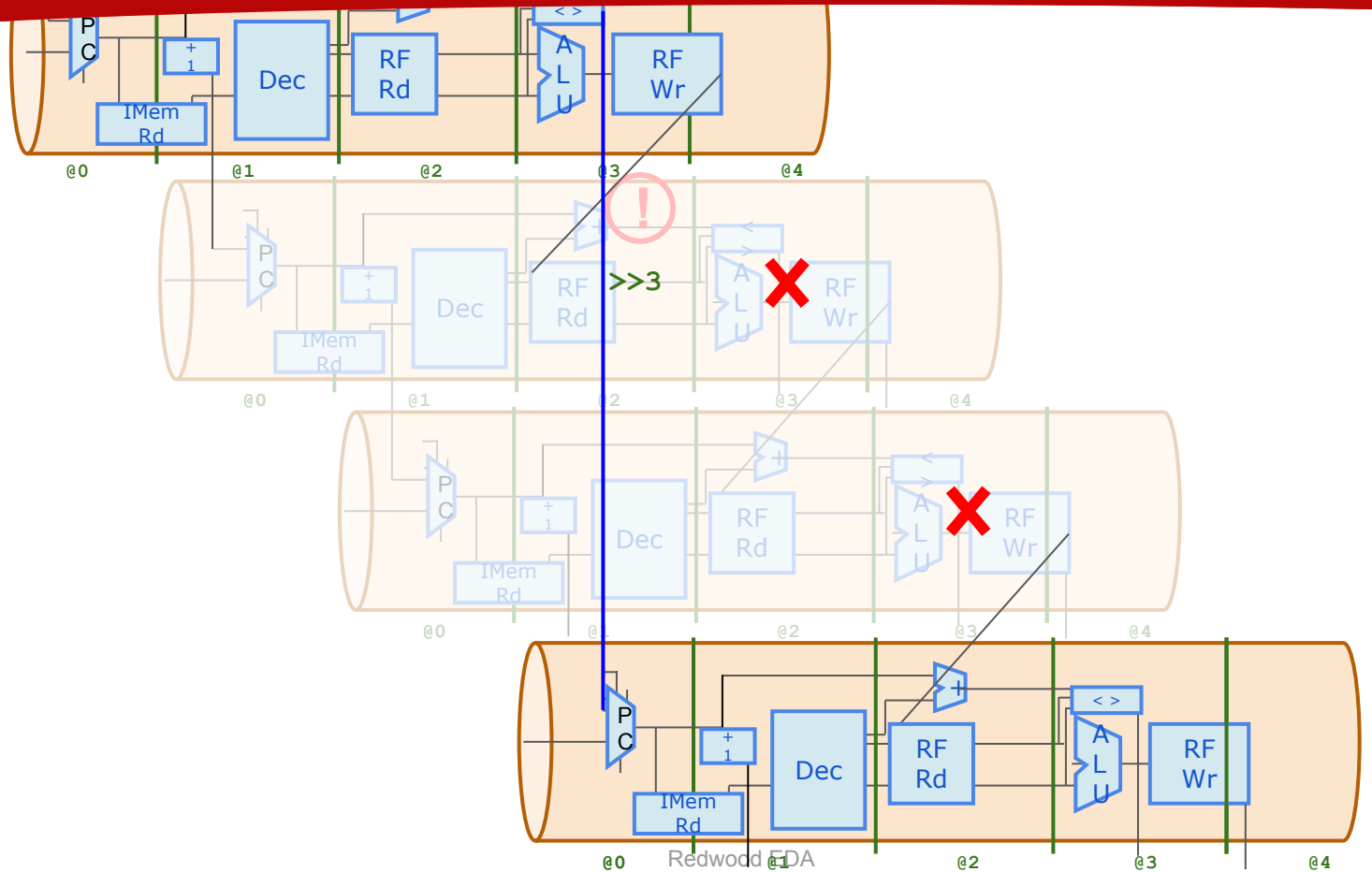


Lab: Register File Bypass

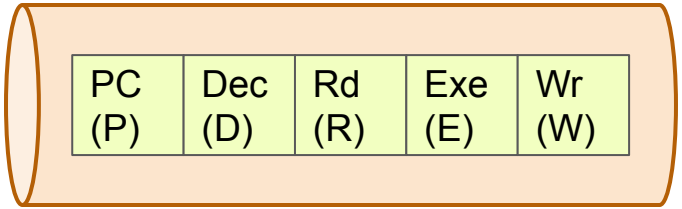


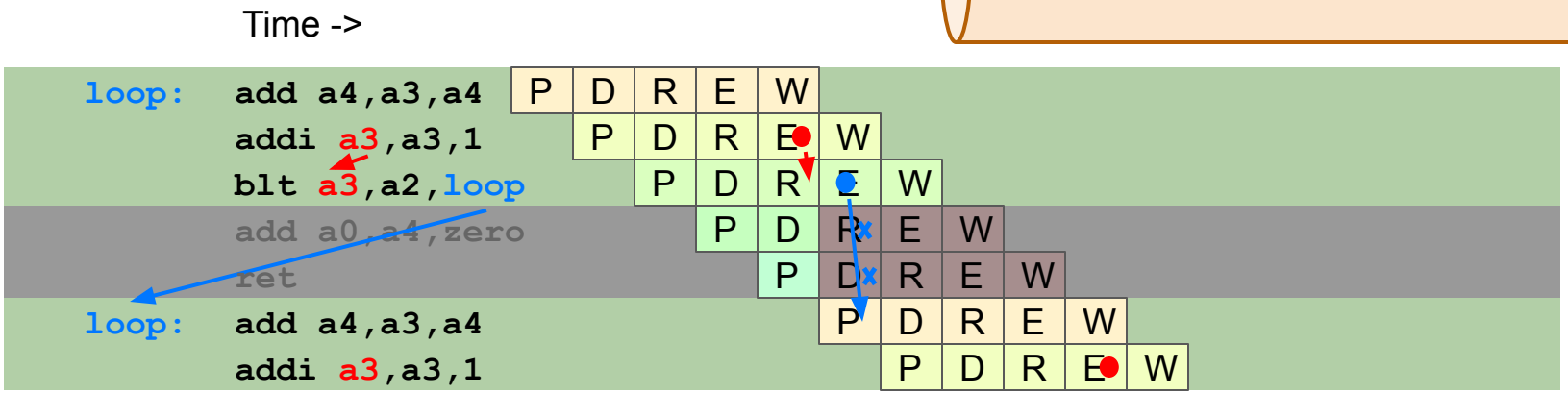
1. RF read uses RF as written 2 instructions ago (already correct).
2. Update expressions for `$srcX_value` to select previous `$result` if it was written to RF (write enable for RF) and if previous `$rd` == `$rsX`.
3. (Should have no effect yet)

Branches

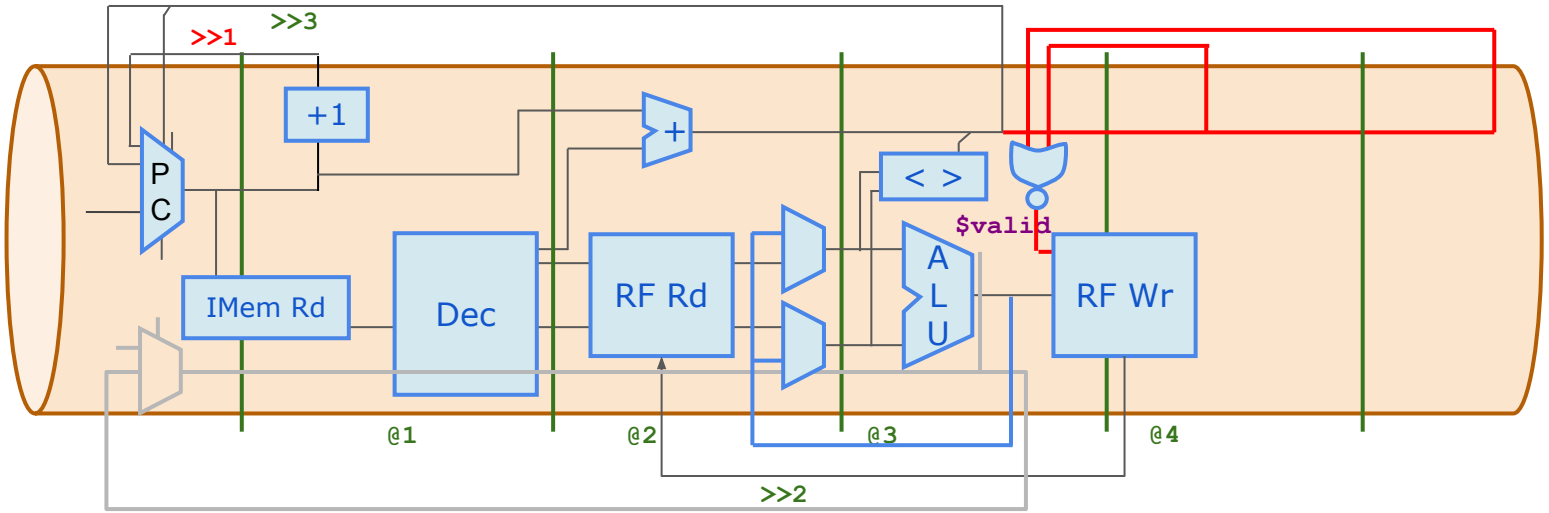


Branches


PC
(P)
Dec
(D)
Rd
(R)
Exe
(E)
Wr
(W)


Time ->
loop: add a4,a3,a4
addi a3,a3,1
blt a3,a2,loop
add a0,a4,zero
ret
loop: add a4,a3,a4
addi a3,a3,1
P D R E W

Lab: Branches



1. Replace `@1 $valid` assignment with `@3 $valid` assignment based on the non-existence of a valid `$taken_br`'s in previous two instrutions.
2. Increment PC every cycle (not every 3 cycles)
3. (PC redirect for branches is already 3-cycle. No change.)
4. Debug. Save outside of Makerchip.

Lab: Complete Instruction Decode

RV32I Base Instruction Set (except FENCE, ECALL, EBREAK):

funct3	opcode	
	0110111	LUI
	0010111	AUIPC
	1101111	JAL
000	1100111	JALR
000	1100011	BEQ
001	1100011	BNE
100	1100011	BLT
101	1100011	BGE
110	1100011	BLTU
111	1100011	BGEU
000	0000011	LB
001	0000011	LH
010	0000011	LW
100	0000011	LBU
101	0000011	LHU
000	0100011	SB
001	0100011	SH
010	0100011	SW
000	0010011	ADDI
010	0010011	SLTI

funct7[5]	funct3	opcode	
	011	0010011	SLTIU
	100	0010011	XORI
	110	0010011	ORI
	111	0010011	ANDI
0	001	0010011	SLLI
0	101	0010011	SRLI
1	101	0010011	SRAI
0	000	0110011	ADD
1	000	0110011	SUB
0	001	0110011	SLL
0	010	0110011	SLT
0	011	0110011	SLTU
0	100	0110011	XOR
0	101	0110011	SRL
1	101	0110011	SRA
0	110	0110011	OR
0	111	0110011	AND

1. Complete remaining instrs, except loads (L*).

```
$dec_bits[10:0] =
   {$funct7[5],$funct3,$opcode};
$is_beq = $dec_bits ==?
          11'bx_000_1100011;

// Until instrs are implemented,
// quiet down the warnings.
`BOGUS_USE($is_beq $is_bne ...)
```

2. We'll treat all loads the same, so generate `$is_load` based on opcode only.
3. Confirm save.

Lab: Complete ALU

Assign **`$result`** for other instrs

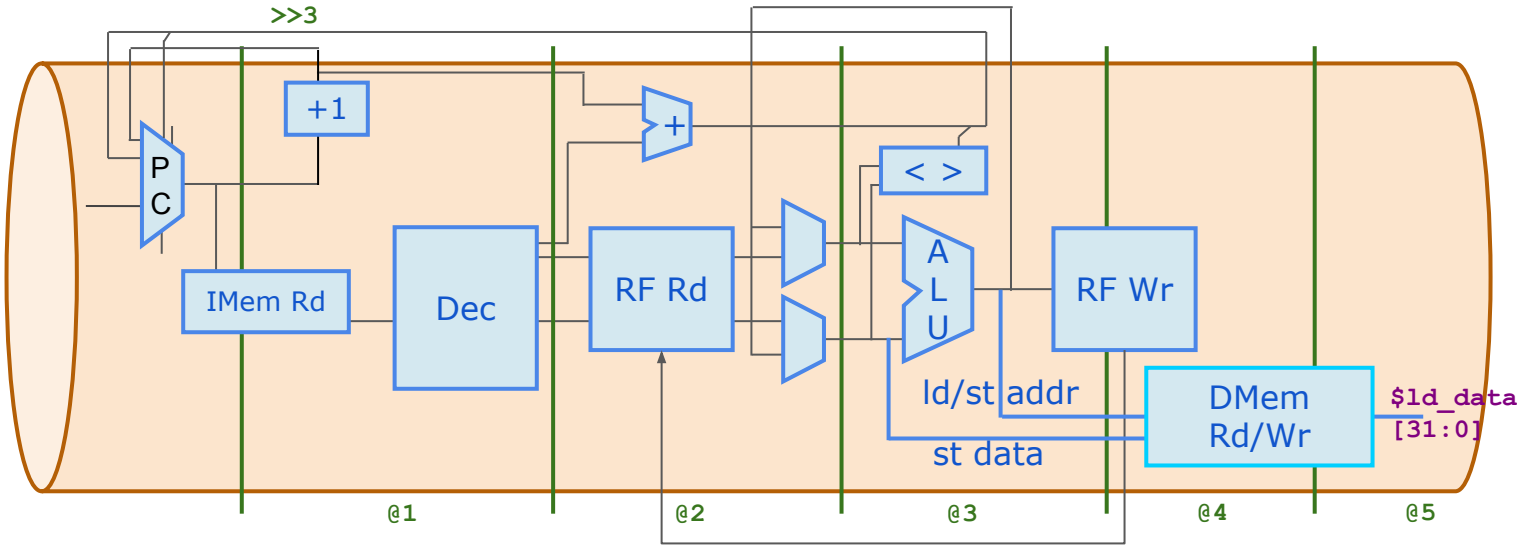
```
ANDI   $src1_value & $imm;
ORI    $src1_value | $imm;
XORI   $src1_value ^ $imm;
ADDI   $src1_value + $imm;
SLLI   $src1_value << $imm[5:0];
SRLI   $src1_value >> $imm[5:0];
AND    $src1_value & $src2_value;
OR     $src1_value | $src2_value;
XOR    $src1_value ^ $src2_value;
```

```
ADD    $src1_value + $src2_value;
SUB    $src1_value - $src2_value;
SLL    $src1_value << $src2_value[4:0];
SRL    $src1_value >> $src2_value[4:0];
SLTU   $src1_value < $src2_value;
SLTIU  $src1_value < $imm;
LUI    {$imm[31:12], 12'b0};
AUIPC  $pc + $imm;
JAL    $pc + 4;
JALR   $pc + 4;
```

Need intermediate result signals for these.

```
SRAI   { {32{$src1_value[31]}}, $src1_value} >> $imm[4:0];
SLT    ($src1_value[31] == $src2_value[31]) ? $sltu_rslt : {31'b0,$src1_value[31]};
SLTI   ($src1_value[31] == $imm[31]) ? $sltiu_rslt : {31'b0,$src1_value[31]};
SRA    { {32{$src1_value[31]}}, $src1_value} >> $src2_value[4:0];
```

Loads/Stores



`LOAD (LW,LH,LB,LHU,LBU)`

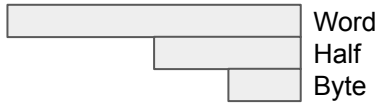
`LOAD rd, imm(rs1)`

rd <= DMem[addr]

`STORE (SW,SH,SB)`

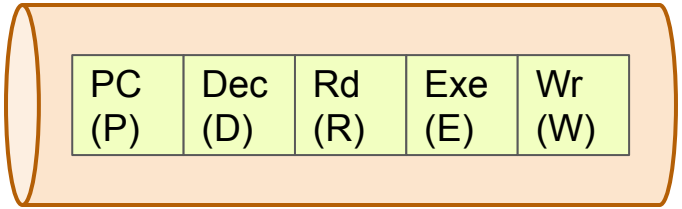
`STORE rs2, imm(rs1)`

DMem[addr] <= rs2

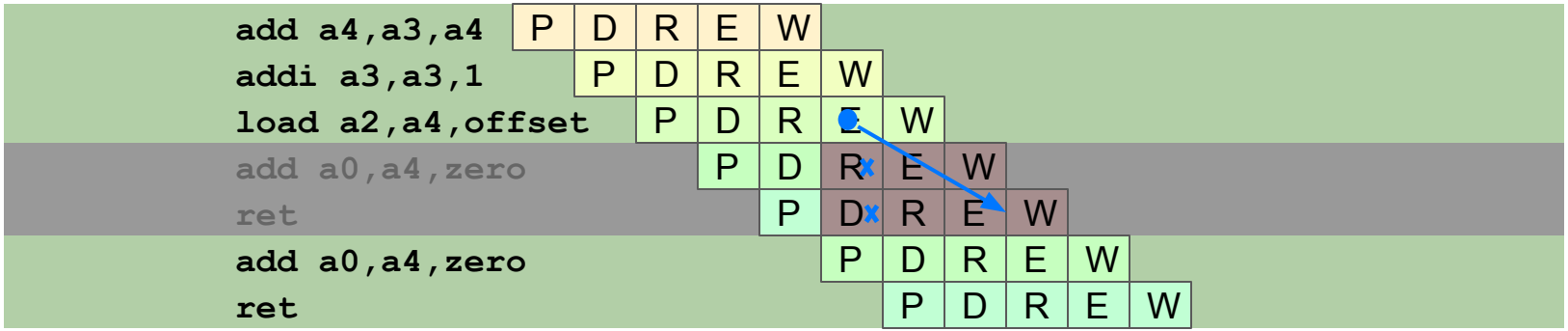


where, addr <= rs1 + imm (like addi)

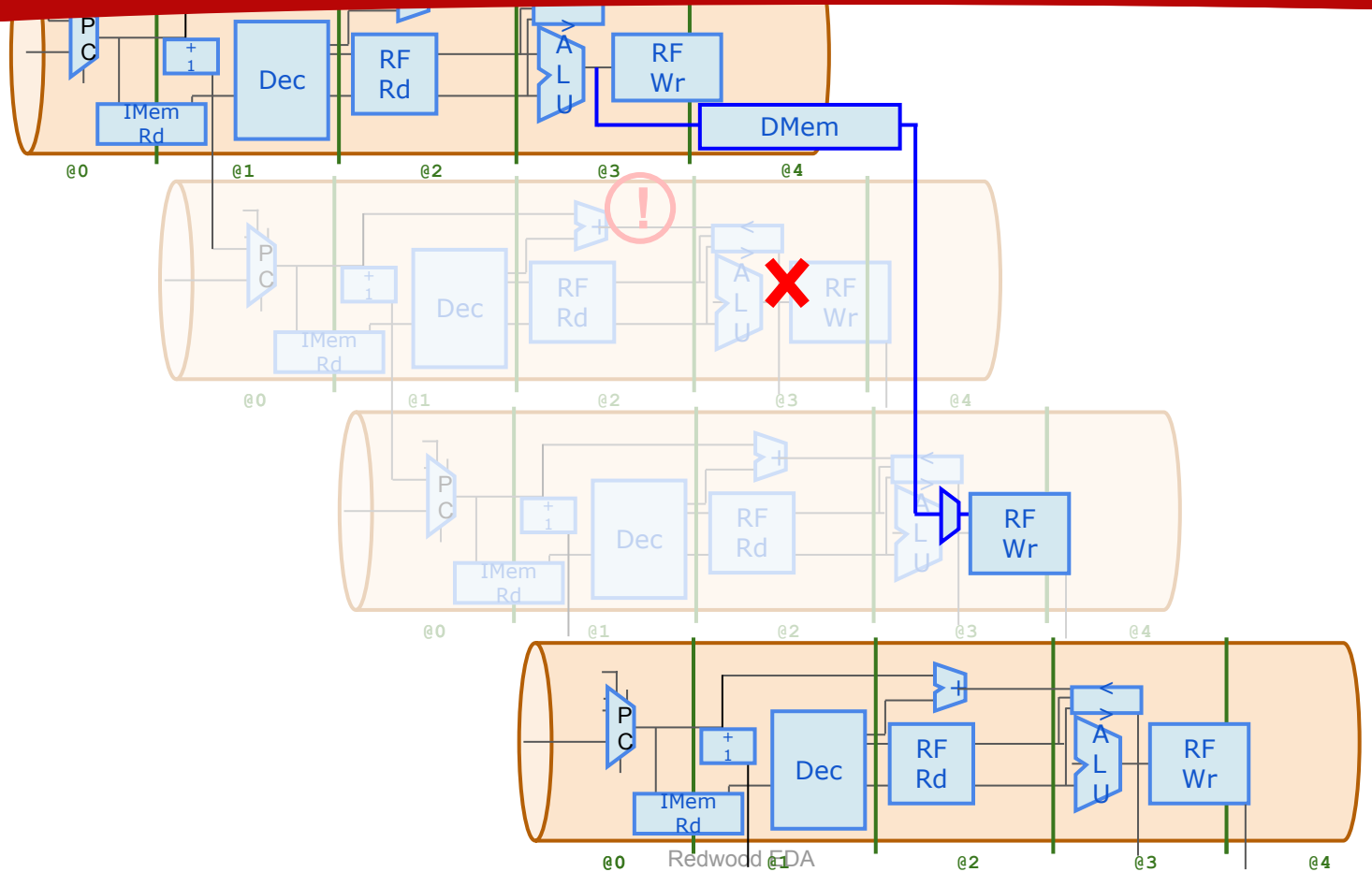
Loads


PC (P)
Dec (D)
Rd (R)
Exe (E)
Wr (W)

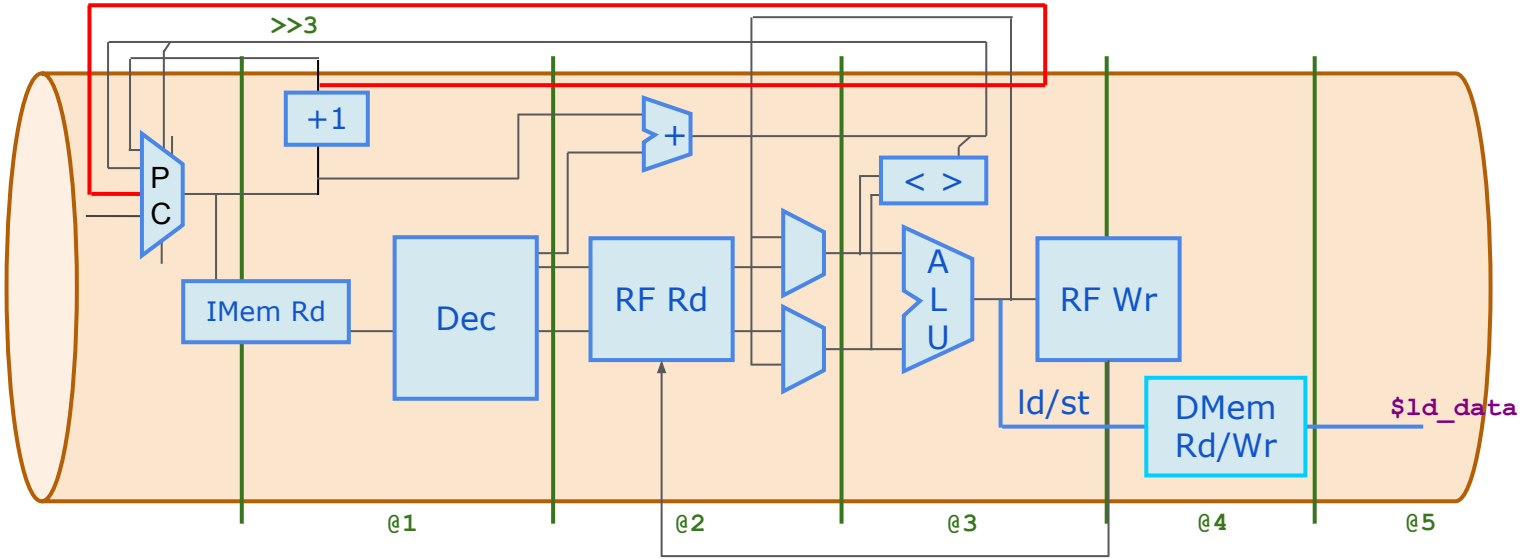
Time ->


add a4,a3,a4
addi a3,a3,1
load a2,a4,offset
add a0,a4,zero
ret
add a0,a4,zero
ret
P D R E W

Loads

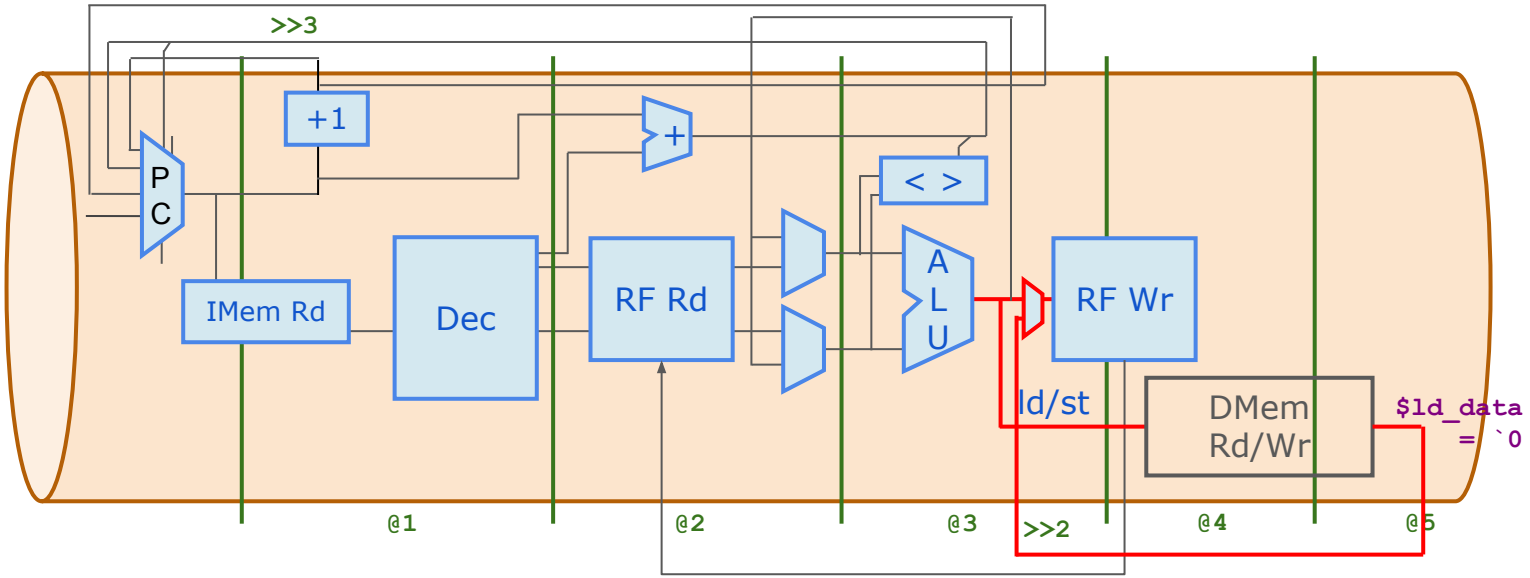


Lab: Redirect Loads



1. Clear **`$valid`** in the “shadow” of a load (like branch).
2. Select **`$inc_pc`** from 3 instructions ago for load redirect.
3. Debug. Confirm save.

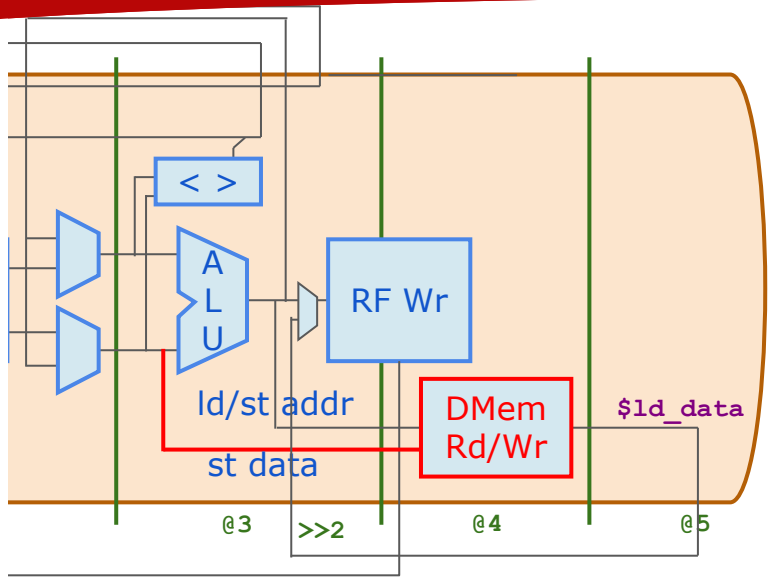
Lab: Load Data



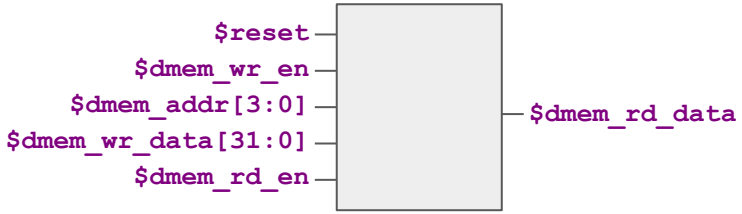
1. For loads/stores (`$is_load`/`$is_s_instr`), compute same result as for `addi`.
2. Add the RF-wr-data MUX to select `>>2$ld_data` and `>>2$rd` as RF inputs for `!$valid` instructions.
3. Enable write of `$ld_data` 2 instructions after valid `$load`.
4. Confirm save.

Also provide the right register index for the write of `$ld_data`.

Lab: Load Data



`dmem`: mini 1-R/W memory
(16-entries, 32-bits wide)



In videos, the final input of the memory (`$dmem_rd_index[5:0]`) should not exist.

1. Uncomment `//m4+dmem(@4)`.
2. Connect interface signals above using address bits [5:2] to perform load and store (when valid).

Lab: Load/Store in Program

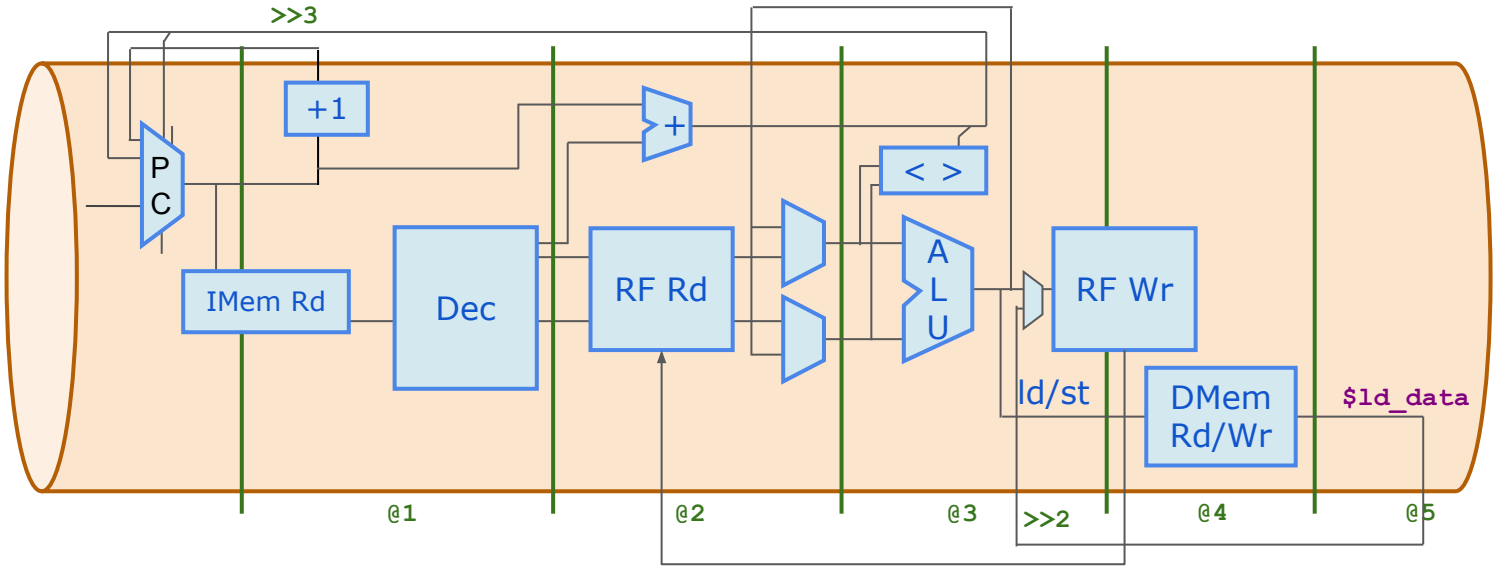
Modify the test program to store the final result value to address 4, then load it into x15.

```
m4_asm(SW, r0, r10, 100)
m4_asm(LW, r15, r0, 100)
```

Update passing condition to look in `xreg[15]`.

Debug. Does the loop properly fall-through and execute store/load.

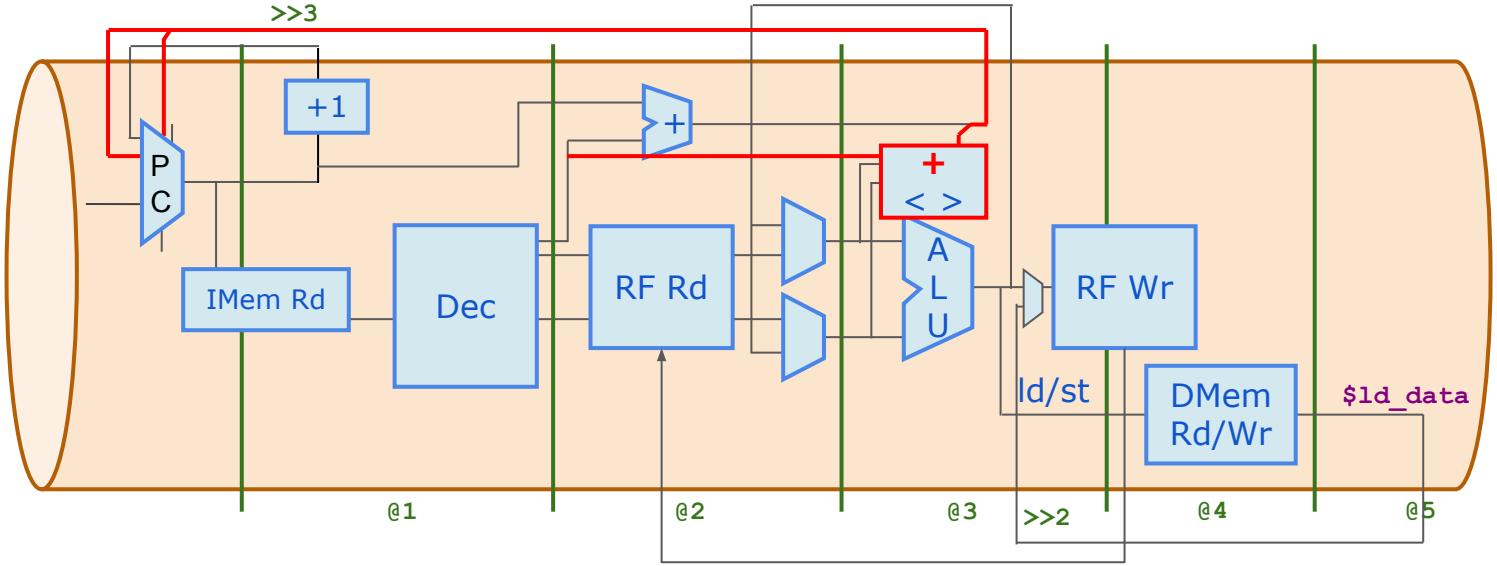
Jumps



JAL: Jump to PC + IMM (aka unconditional branch, so **`$br_tgt_pc`**)

JALR: Jumps to SRC1 + IMM.

Lab: Jumps

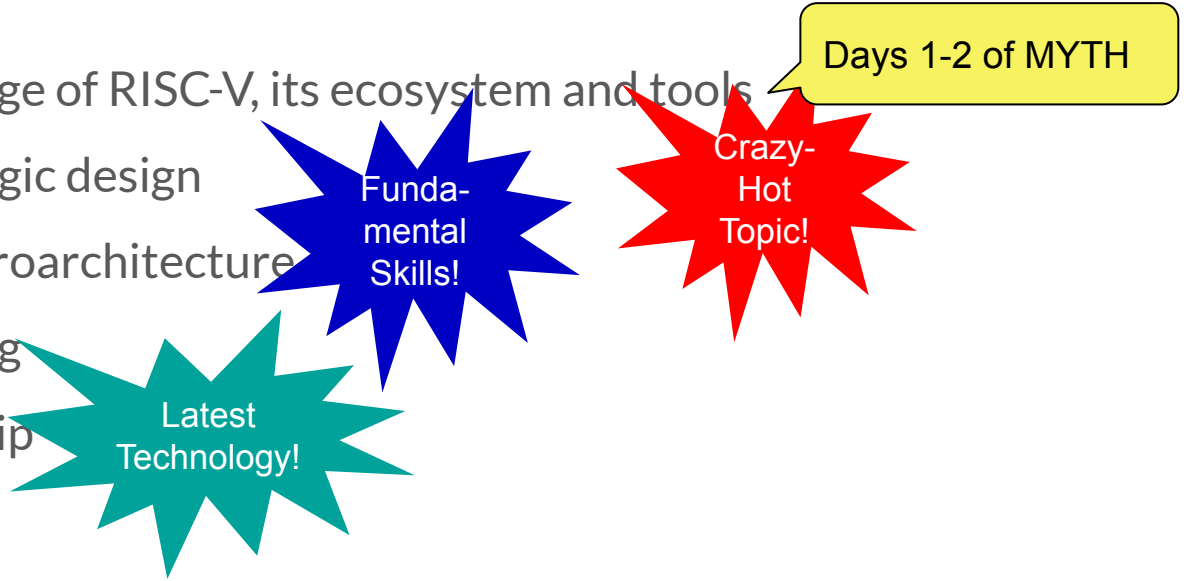


1. Define **`$is_jump`** (JAL or JALR), and, like **`$taken_br`**, create invalid cycles.
2. Compute **`$jalr_tgt_pc`** (SRC1 + IMM).
3. Select correct **`$pc`** for JAL (**`>>3$br_tgt_pc`**) and JALR (**`>>3$jalr_tgt_pc`**).
4. Save.

YOU DID IT!!!!

Skills You Have Acquired

- Knowledge of RISC-V, its ecosystem and tools
- Digital logic design
- CPU microarchitecture
- TL-Verilog
- Makerchip



You are not just learning the industry… you are leading it!!!

Final Steps

- Submit your work for certification
- Brag about your accomplishments

MYTH Workshop only