Building a Simple CPU

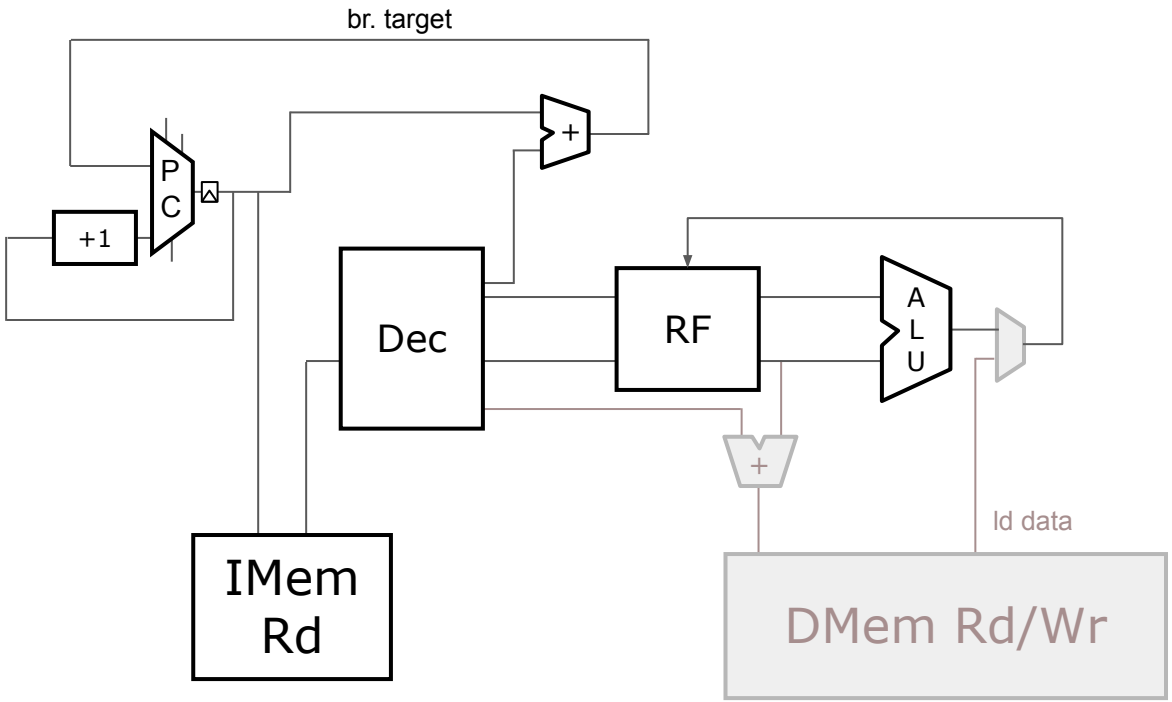
VSDOpen2020

Steve Hoover
Founder, Redwood EDA
Oct. 8, 2020

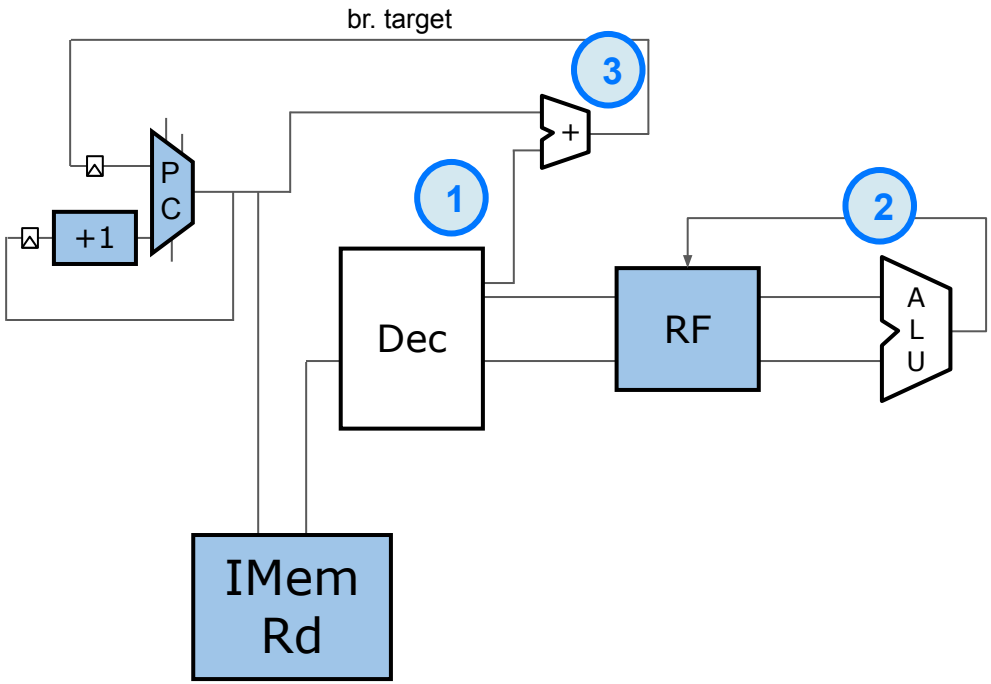
Agenda

- RISC-V
- Makerchip.com
- Combinational logic in TL-Verilog
- Building Your Own Simple RISC-V-Subset Core

Simple CPU Block Diagram



Implementation Plan



Lab: Setup

Github repo can be ignored.
Starting-point code should already be loaded.

- Visit: github.com/stevehoover/VSDOpen2020_TLV_RISC-V_Tutorial
- Open starting-point code.
- Click “Save as new project” (upper right), and bookmark the new URL.
- In the remaining labs, you will:
  - complete “`TBD`” logic expressions
  - fix errors in LOG (it will report “`Simulation FAILED!!!`” until the final lab is completed successfully)
  - examine DIAGRAM, VIZ, and WAVEFORM to ensure correct operation and debug

BTW, the logic inside the yellow "|for_viz_only" box in the DIAGRAM provides the VIZ functionality. It is not part of the hardware model and can be ignored.

Verilog/TL-Verilog Operators Reference

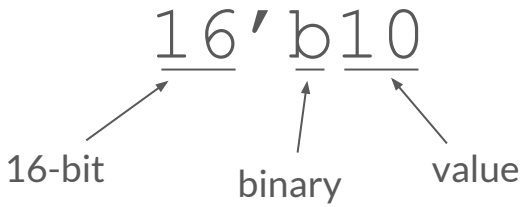
Try other operators until you feel comfortable.

Operation	Boolean	Bitwise
NOT	!	~
AND	&&	&
OR	||	|
XOR	^	^

Use ( ) to group sub-expressions.

Operation	Operator
Arithmetic	+, -, *, /
Bit Shift	<<, >>
Compare	==, !=
Bit extract	*sig*[*max-bit* : *min-bit*]
Concatenate	{*sig1, sig2*}
Replicate	{*count*{*sig*}}

Constants



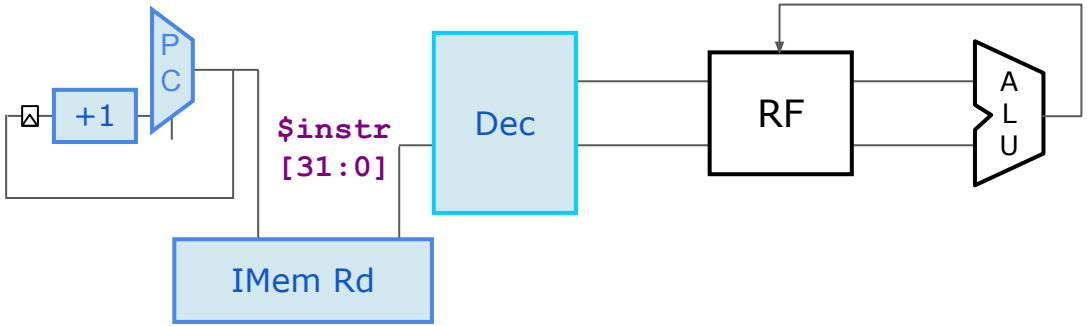
For example:

```
$out[4:0] = $in1 + 5'b01001;
```

Binary (base 2)	Decimal (base 10)	Hexadecimal (base 16)
0: 0000	0: 00	0: 00
1: 0001	1: 01	1: 01
2: 0010	...	...
3: 0011	9: 09	9: 09
4: 0100	10: 10	10: 0A
5: 0101	...	...
6: 0110		15: 0F
...		16: 10
		...

Decode

Be sure to clide the button with the videos to load the starting code.



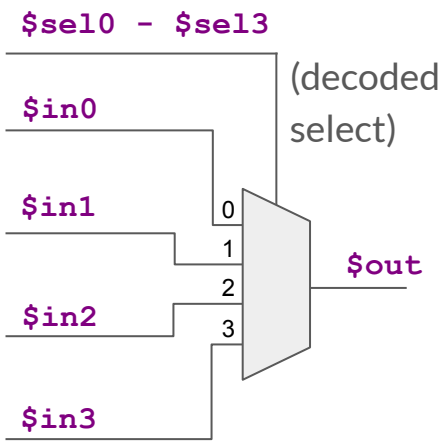
Lab: Instruction Types Decode

In RV32I, instr[6:2] determine instruction type: I, R, S, B, J, U.
For our RISC-V subset, instr[6:5] determines instruction type.

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

```
$is_i_instr = $instr[6:5] == 2'b00;
$is_r_instr = TBD;
$is_b_instr = TBD;
```

Multiplexers



TL-Verilog:

```
$out =
   $sel0 ? $in0 :
   $sel1 ? $in1 :
   $sel2 ? $in2 :
           $in3 ;
```

(highest priority first.)

Lab: Instruction Immediate Decode

Form `$imm[31:0]` based on instruction type. (R-type has no immediate.)

31 30 … 20	19 … 12	11	10 … 5	4 … 1	0	
— inst[31] —			inst[30:25]	inst[24:21]	inst[20]	I-immediate
— inst[31] —			inst[30:25]	inst[11:8]	inst[7]	S-immediate
— inst[31] —		inst[7]	inst[30:25]	inst[11:8]	0	B-immediate
inst[31]	inst[30:20]	inst[19:12]	— 0 —			U-immediate
— inst[31] —		inst[19:12]	inst[20]	inst[30:25]	inst[24:21]	0 J-immediate

```
$imm[31:0] = $is_i_instr ? { {21{$instr[31]}}, $instr[30:20] } :  // I-type
             //TBD                                                // B-type
                                32'b0;
```

Lab: Instruction Field Decode

Extract other instruction fields: `$rs2, $rs1, $funct3, $rd, $opcode`
(funct7 is not needed)

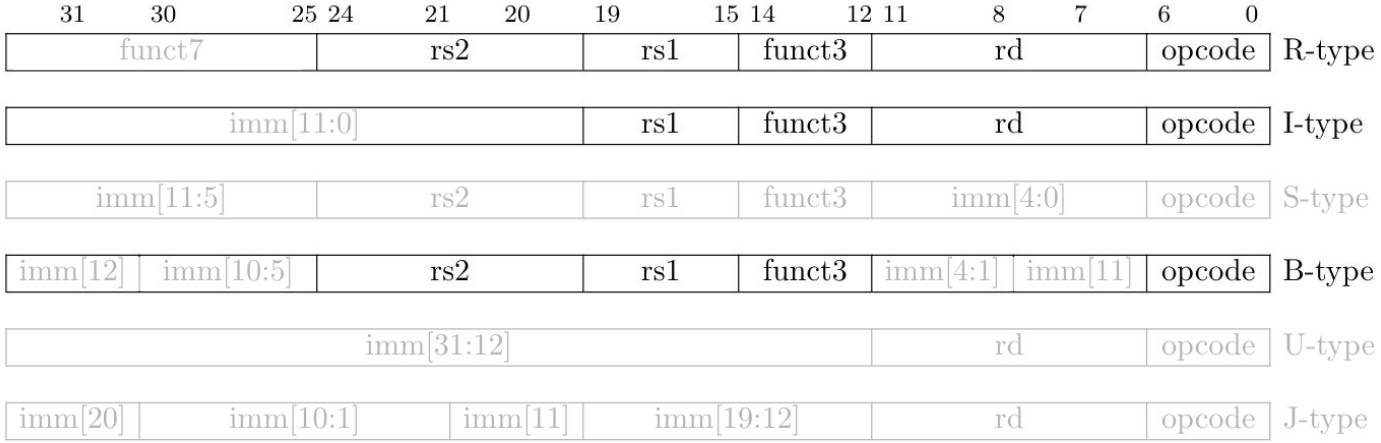


Figure 2.3: RISC-V base instruction formats showing immediate variants.

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

Lab: Register Validity Decode

Determine when register fields are valid: `$rs2_valid, $rs1_valid, $rd_valid` based on `$is_X_instr`.

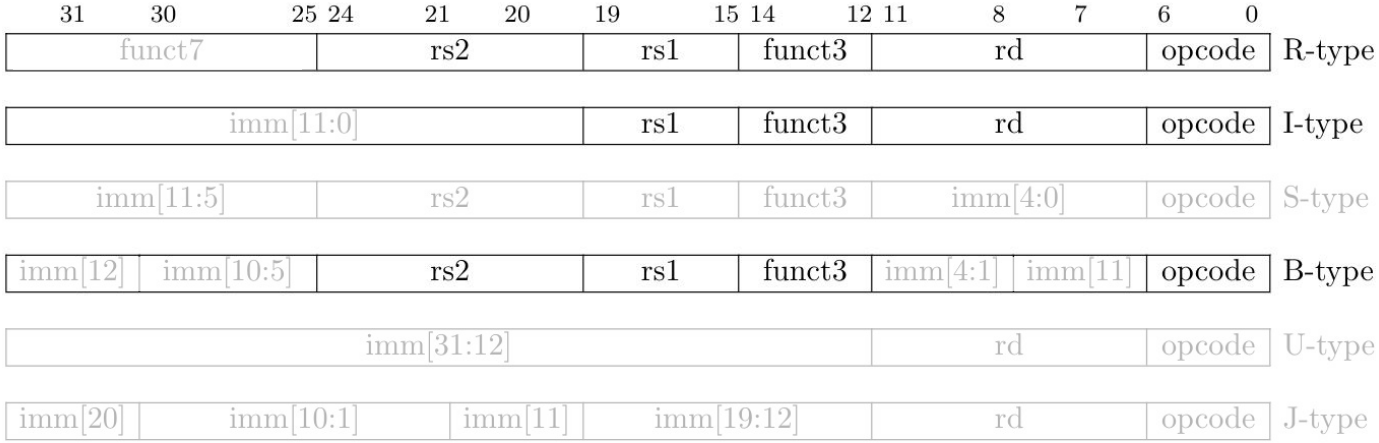
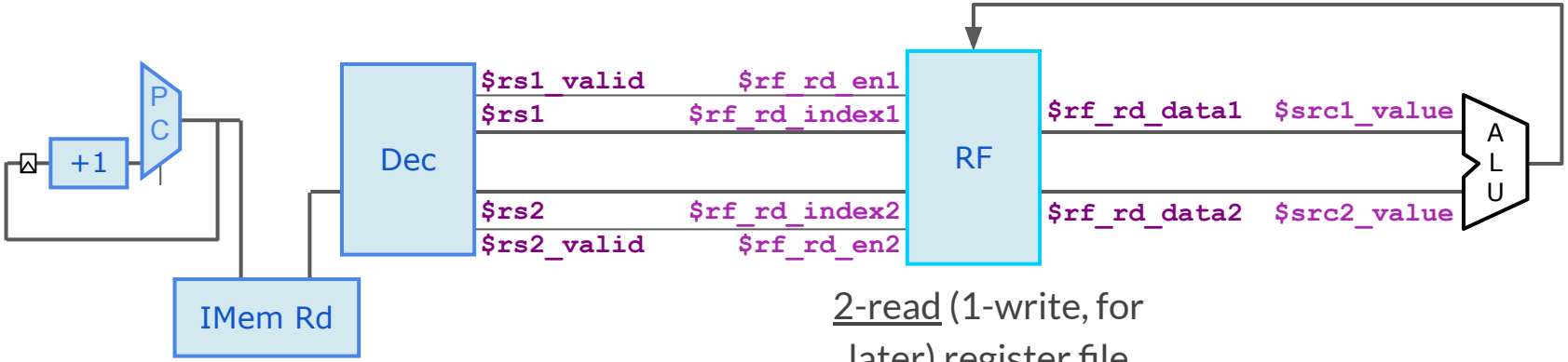


Figure 2.3: RISC-V base instruction formats showing immediate variants.

```
$rs1_valid = $is_r_instr || $is_i_instr || $is_b_instr;
...
```

Register File Read Hookup



2-read (1-write, for later) register file

Lab: 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

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

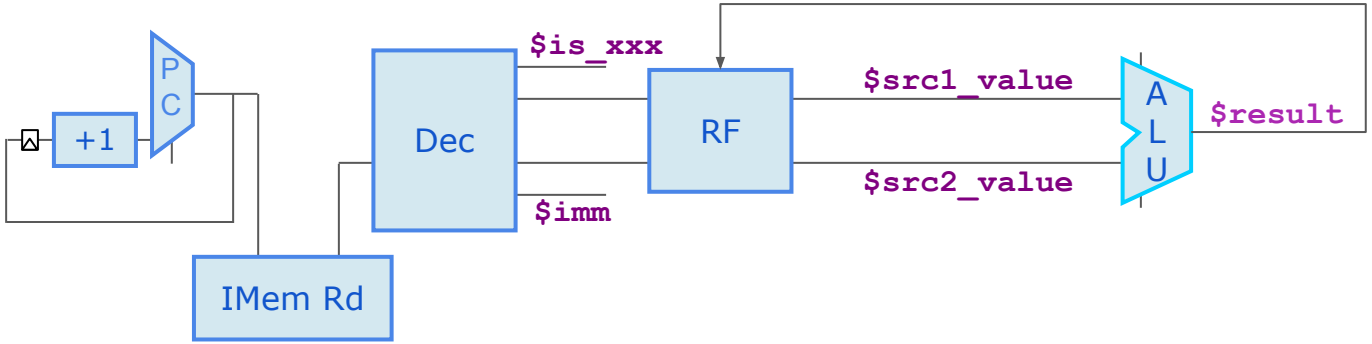
Complete circled instructions.

```
$dec_bits[9:0] = {$funct3, $opcode};
$is_blt = $dec_bits ==
            10'b1001100011;
$is_addi = TBD;
$is_add  = TBD;
```

Check VIZ and debug decode logic as necessary.

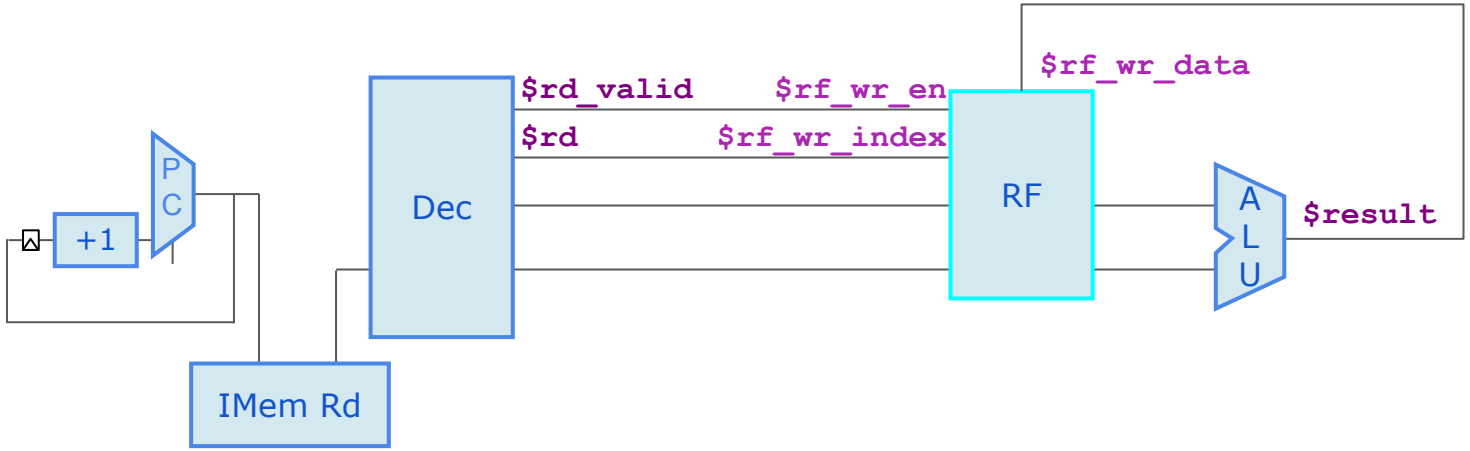
Lab: ALU

Assign the ALU `$result` for ADD and ADDI. (BLT has no result; default to 0.)

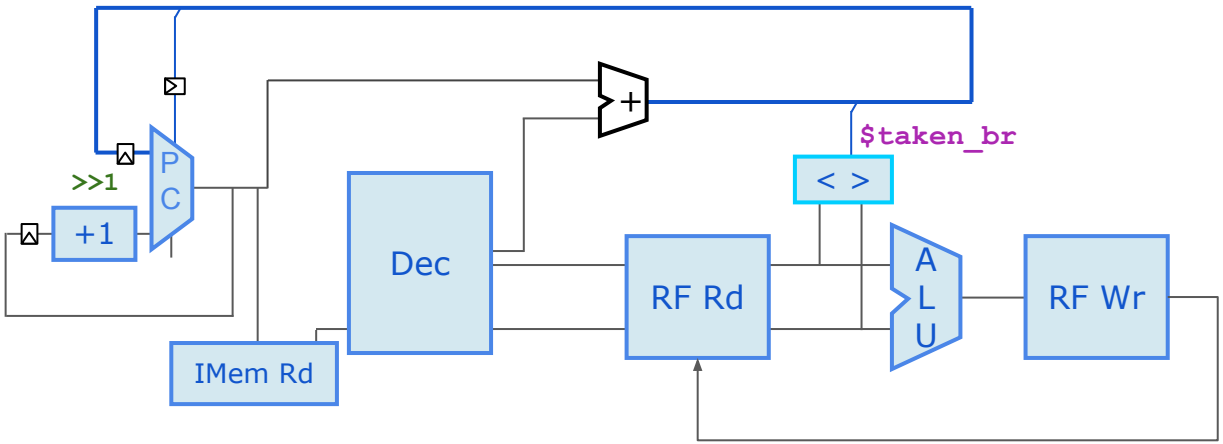



```
$result[31:0] =
    $is_addi ? $src1_value + $imm :
    //TBD
                32'b0;
```

Register File Write Hookup

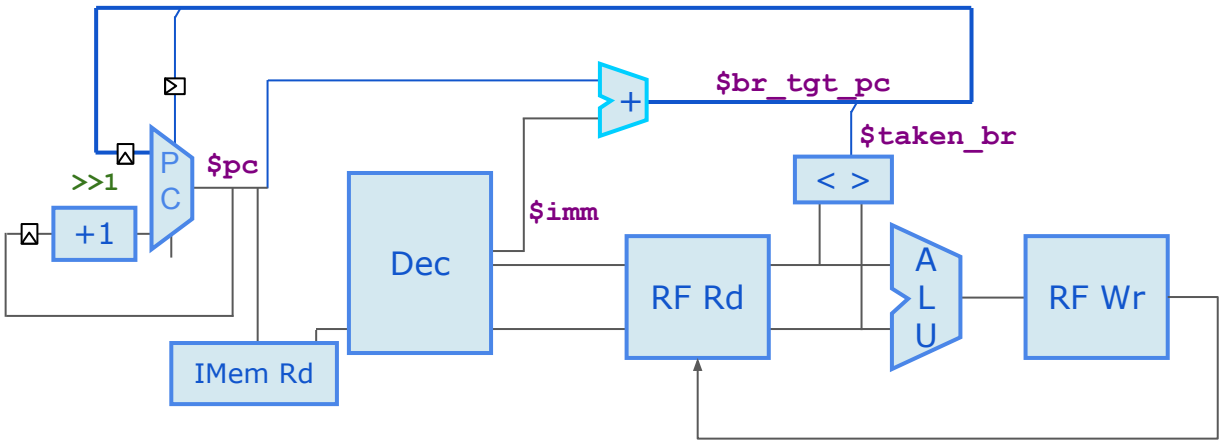


Lab: Branch Condition



1. Assert `$taken_br` for BLT instructions (`$is_blt`) with `$src1_value < $src2_value`.
2. Confirm save.

Lab: Branch Target



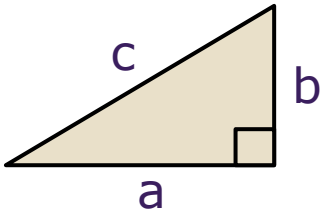
1. Compute `$br_tgt_pc` (PC + imm)
2. Check behavior in simulation. Program should now branch properly, and should sum values {1..9}. If all is working, it will report "Simulation Passed" in LOG!
3. Debug as needed, and save outside of Makerchip.

YOU DID IT!!!!!

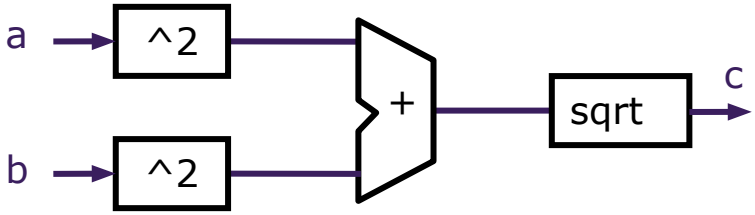
You built a mini-CPU

A Simple Pipeline

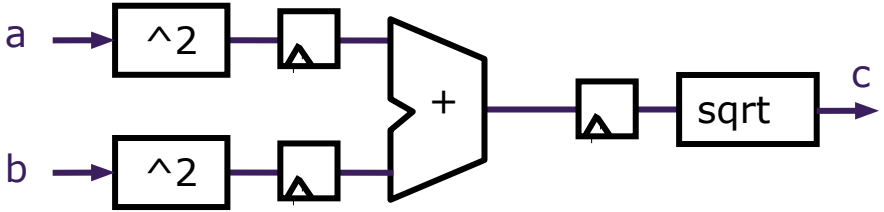
Let's compute Pythagoras's Theorem in hardware.



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

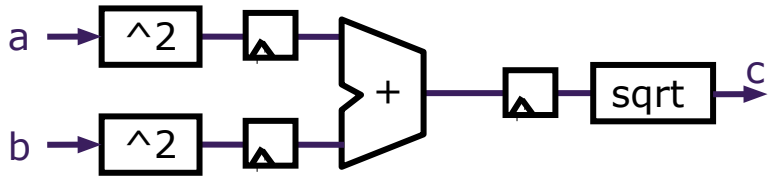


We distribute the calculation over three cycles.

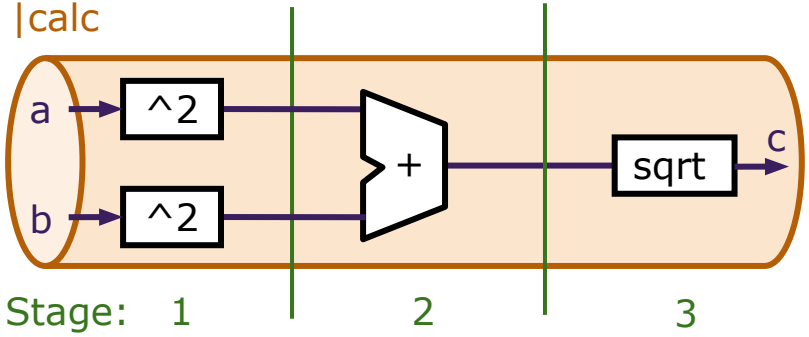


A Simple Pipeline - Timing-Abstract

RTL:

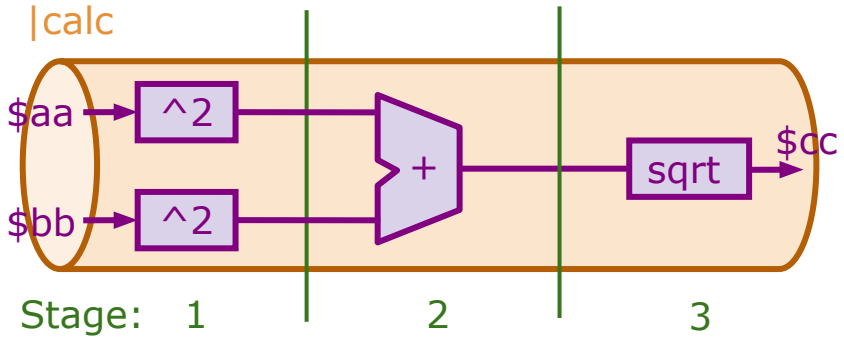


Timing-abstract:



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

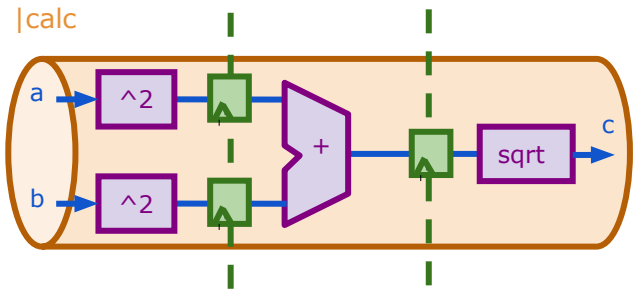
A Simple Pipeline - TL-Verilog



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

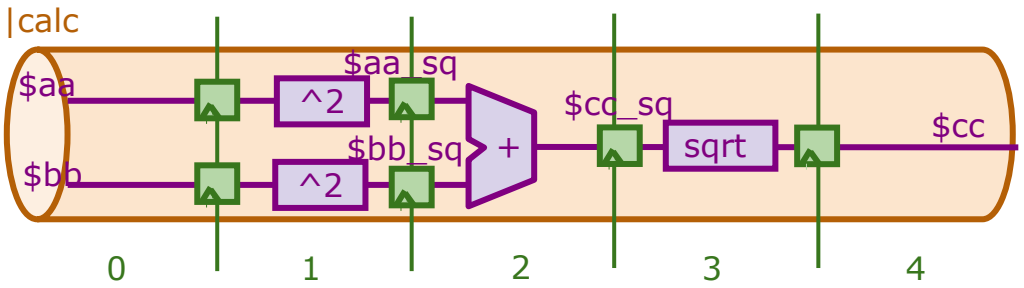
TL-Verilog

```
|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);
```

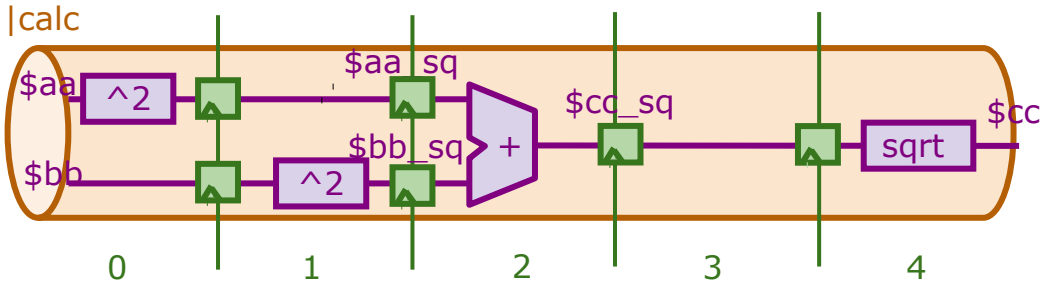
Retiming -- Easy and Safe

```
|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);
```



==



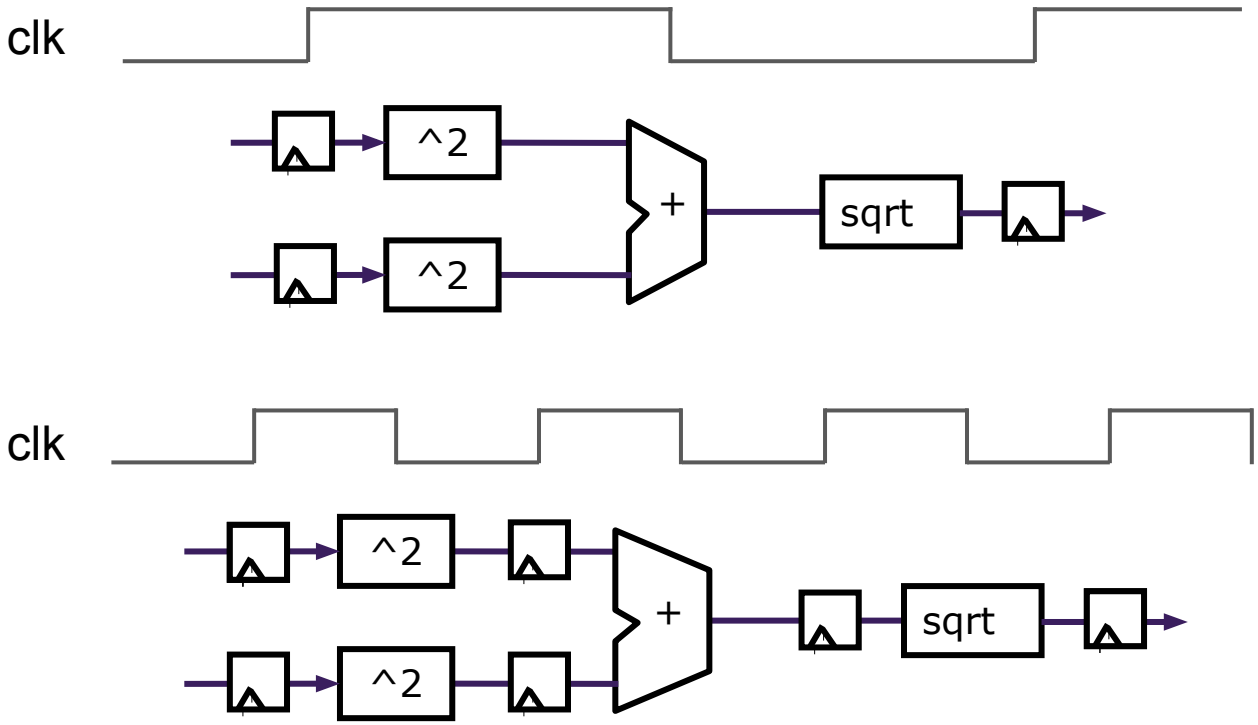
Staging is a physical attribute. No impact to behavior.

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);
```

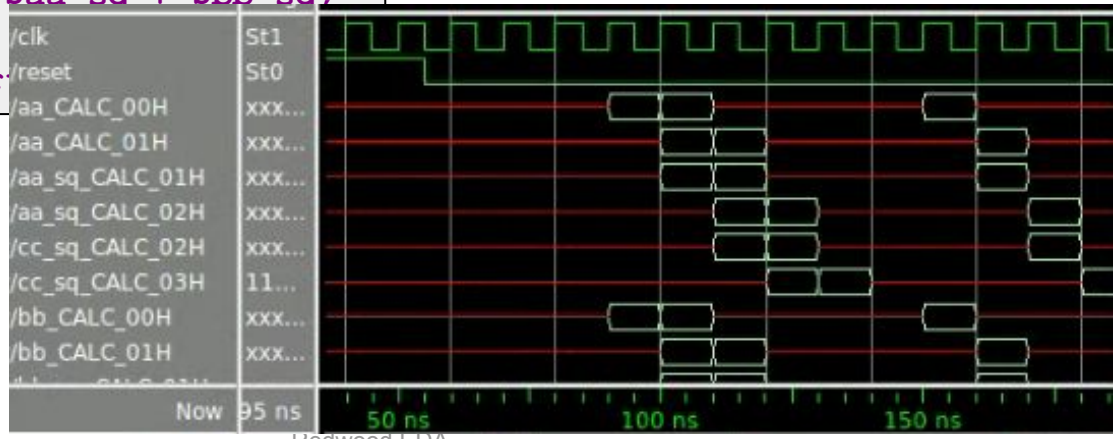
VERY BUG-PRONE!

High Frequency



Validity

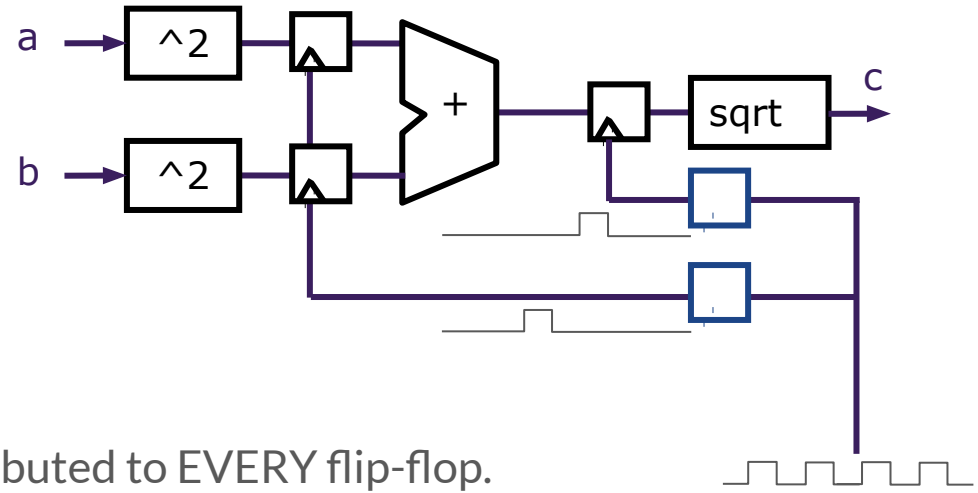
```
|calc
   @1
      $valid = ...;
   ?$valid
      @1
         $aa_sq[31:0] = $aa * $aa;
         $bb_sq[31:0] = $bb * $bb;
      @2
         $cc_sq[31:0] = $aa_sq + $bb_sq;
      @3
         $cc[31:0] = sqr
```



Validity provides:

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

Clock Gating



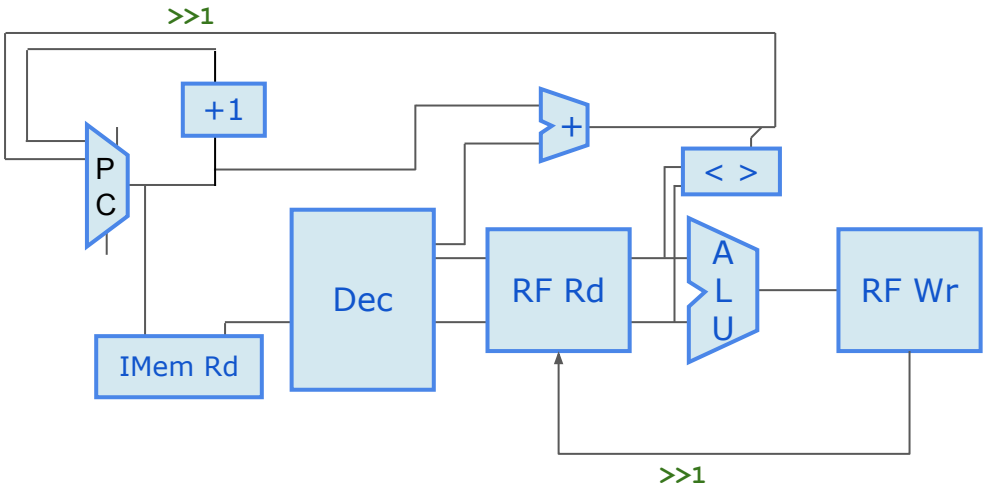
- Motivation:
  - Clock signals are distributed to EVERY flip-flop.
  - Clocks toggle twice per cycle.
  - This consumes power.
- Clock gating avoids toggling clock signals.
- TL-Verilog can produce fine-grained gating (or enables).

Pipelining Your RISC-V Core

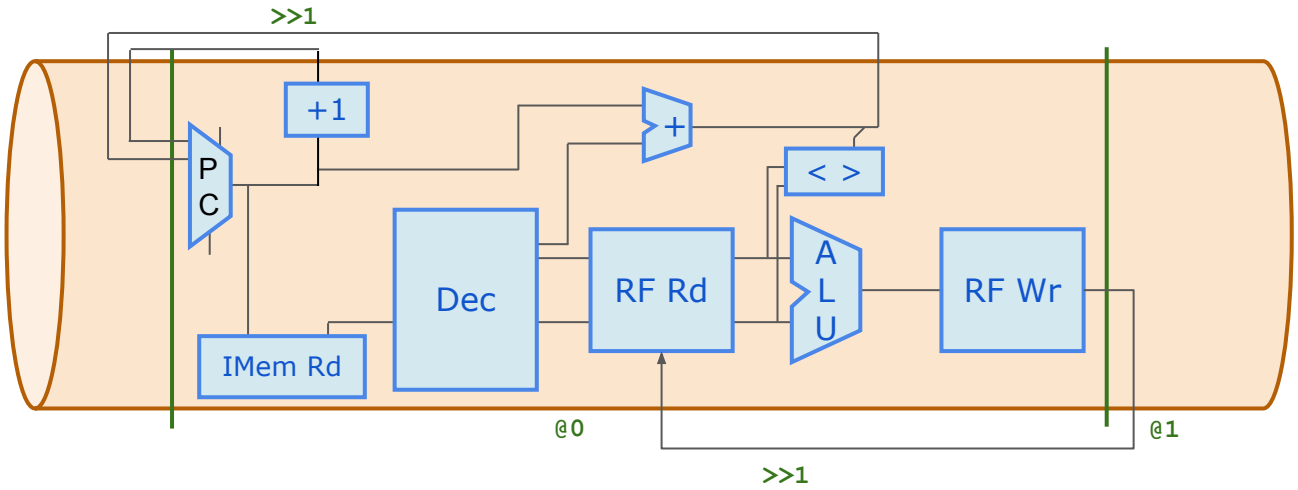
ChipEXPO-2021

Steve Hoover
Founder, Redwood EDA
July 31, 2020

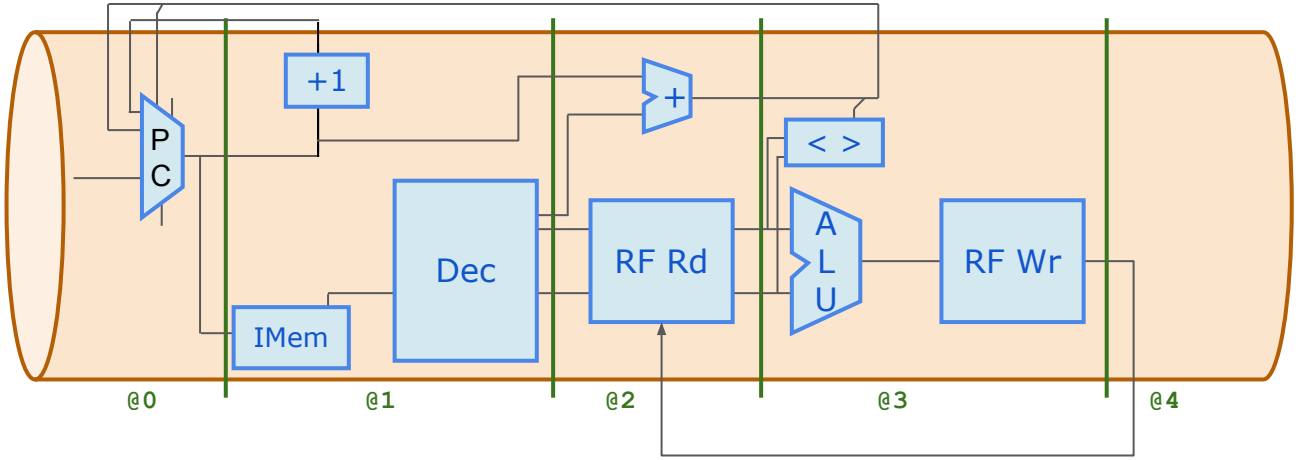
Pipelining Your RISC-V



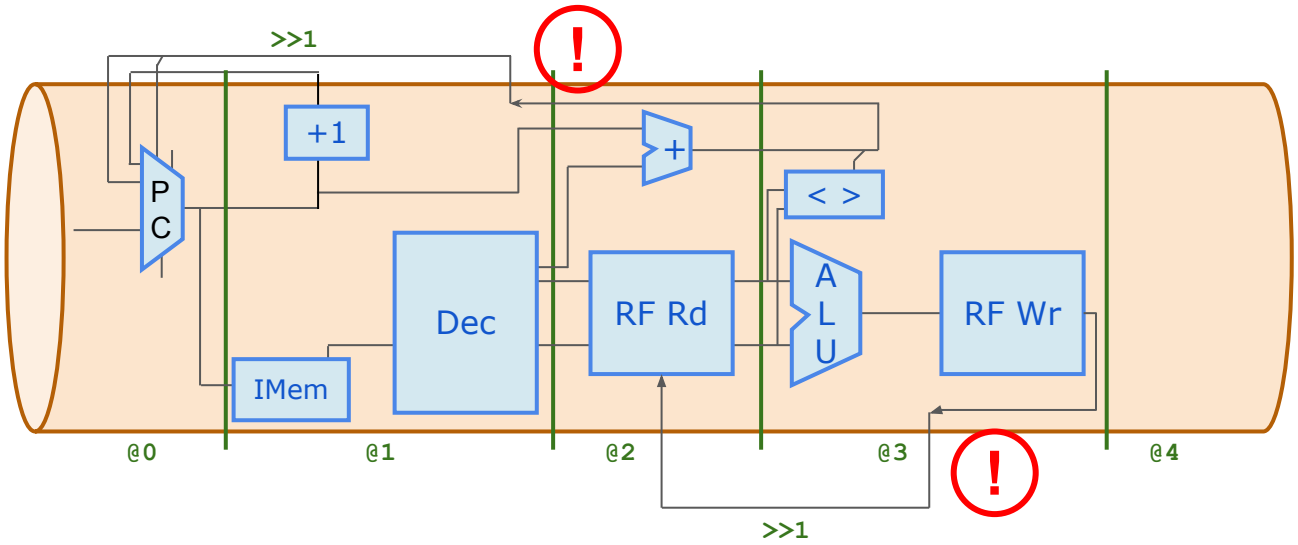
Pipelining Your RISC-V



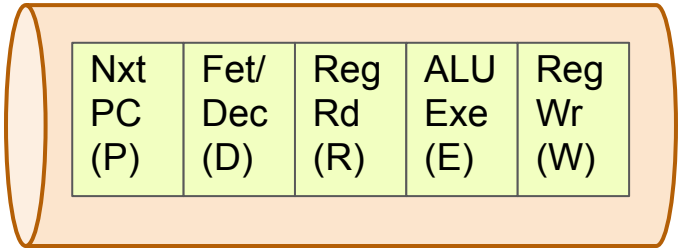
Pipelining Your RISC-V



Pipelining Your RISC-V



RISC-V Waterfall Diagram & Hazards

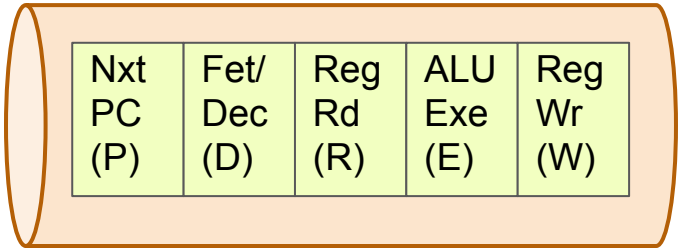


Time ->

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

P	D	R	E	W				
	P	D	R	E	W			
		P	D	R	E	W		
			P	D	R	E	W	
				P	D	R	E	W

Branches

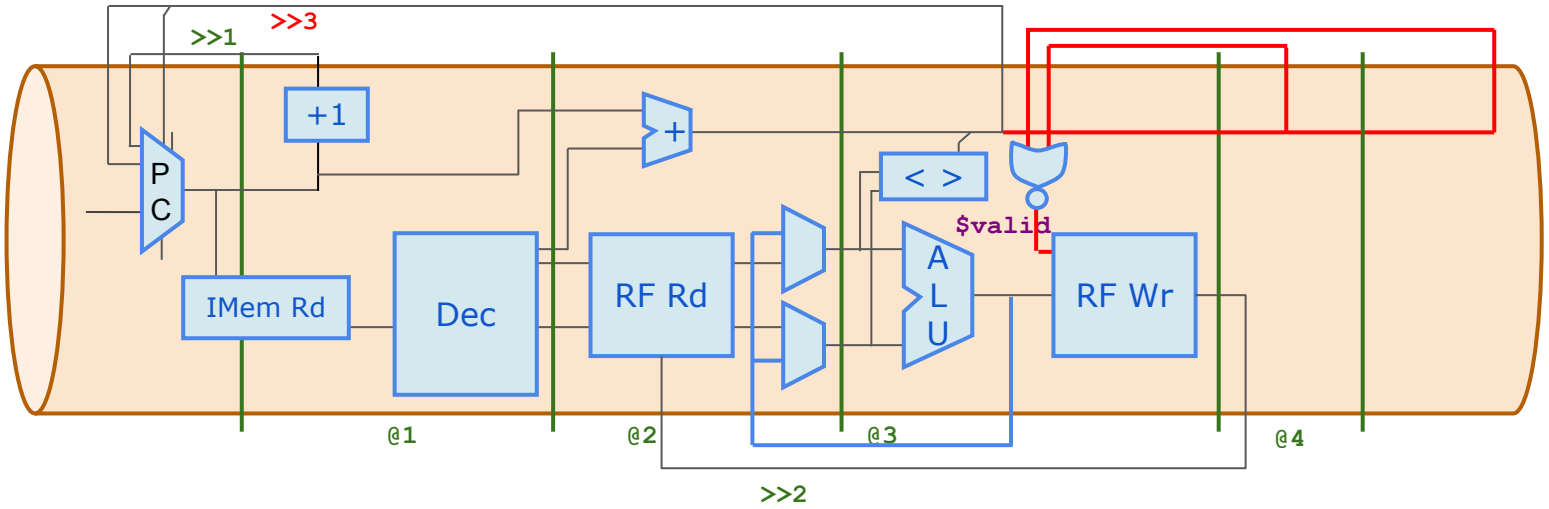


Time ->

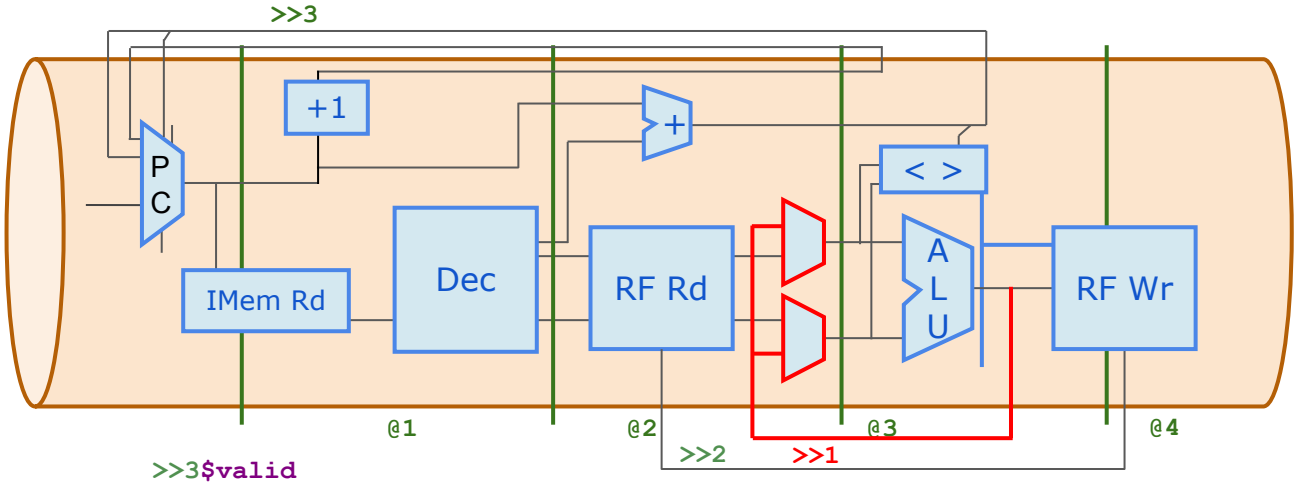
```
loop:   add a4,a3,a4
        addi a3,a3,1
        blt a3,a2,loop
        add a0,a4,zero
        ret
loop:   add a4,a3,a4
        addi a3,a3,1
```

P	D	R	E	W					
	P	D	R	E	W				
		P	D	R	E	W			
			P	D	R	E	W		
				P	D	R	E	W	
					P	D	R	E	W
						P	D	R	E	W
							P	D	R	E	W

Lab: Branches



Lab: Register File Bypass



In the ChipEXPO-2021 repo: https://github.com/stevehoover/ChipEXPO-2021 open the "part 2 starting-point code".

=> Update expressions for `$srcx_value` to select:
previous `$result` **if** it was written to RF (write enable for RF) **and if** previous `$rd` == `$rsx`.

YOU DID IT!!!!!

Skills You Have Acquired

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

Fundamental Skills!

Crazy-Hot Topic!

Latest Technology!