Building a RISC-V Core

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

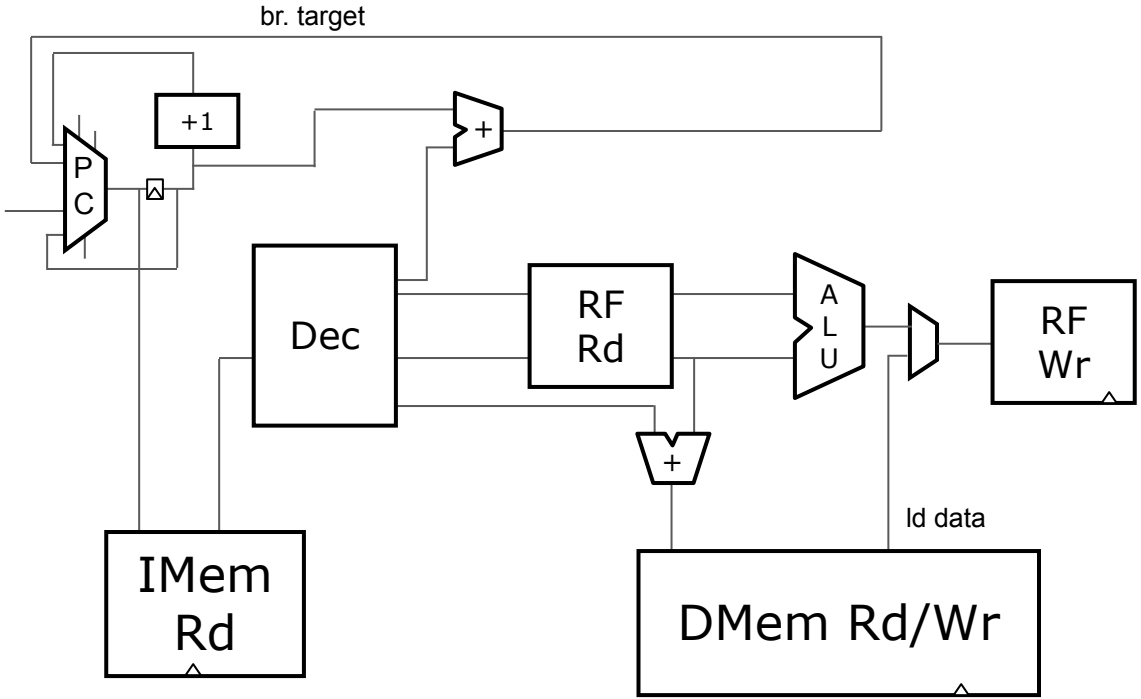
Steve Hoover
Founder, Redwood EDA
July 31, 2020

Day 4 & 5: RISC-V

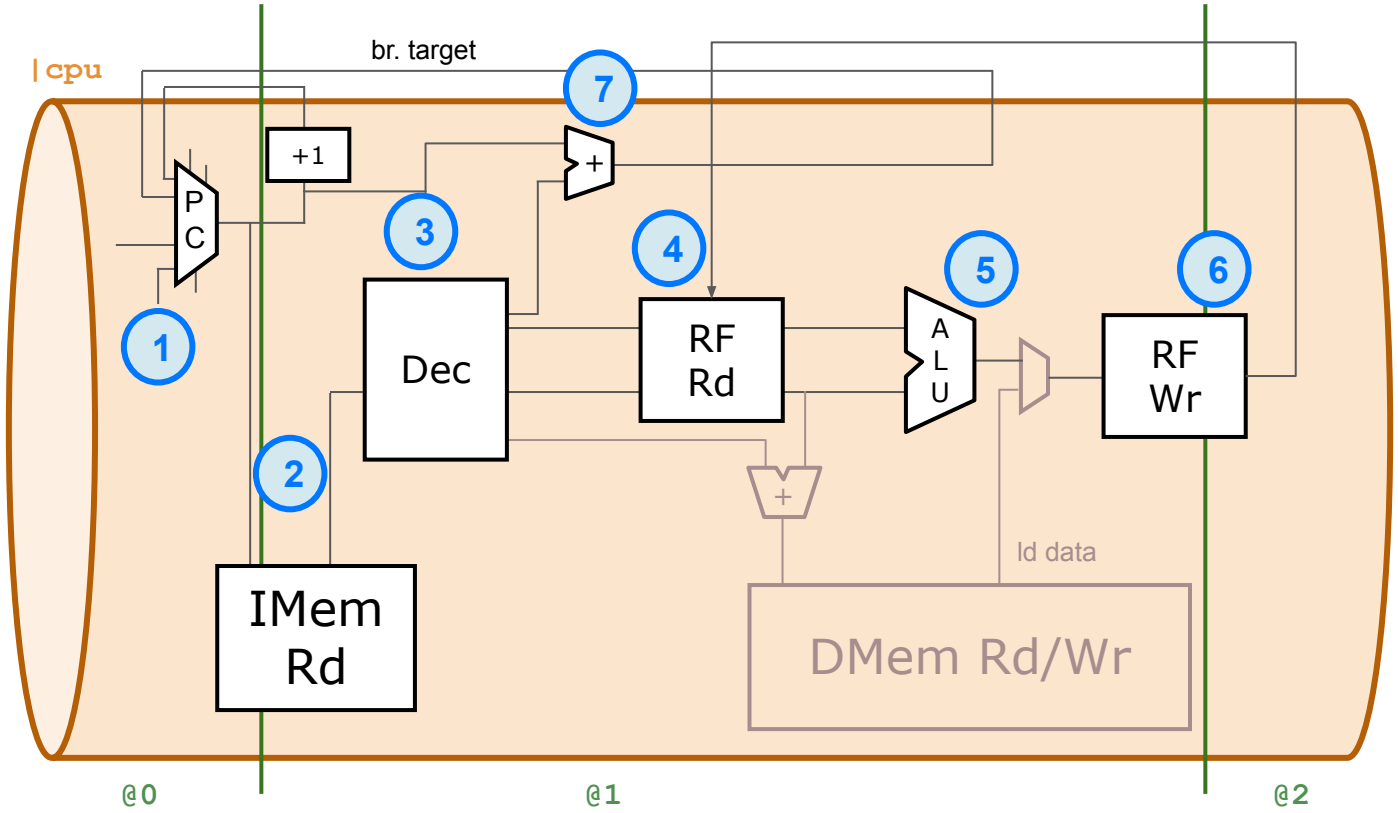
Agenda:

- 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”.

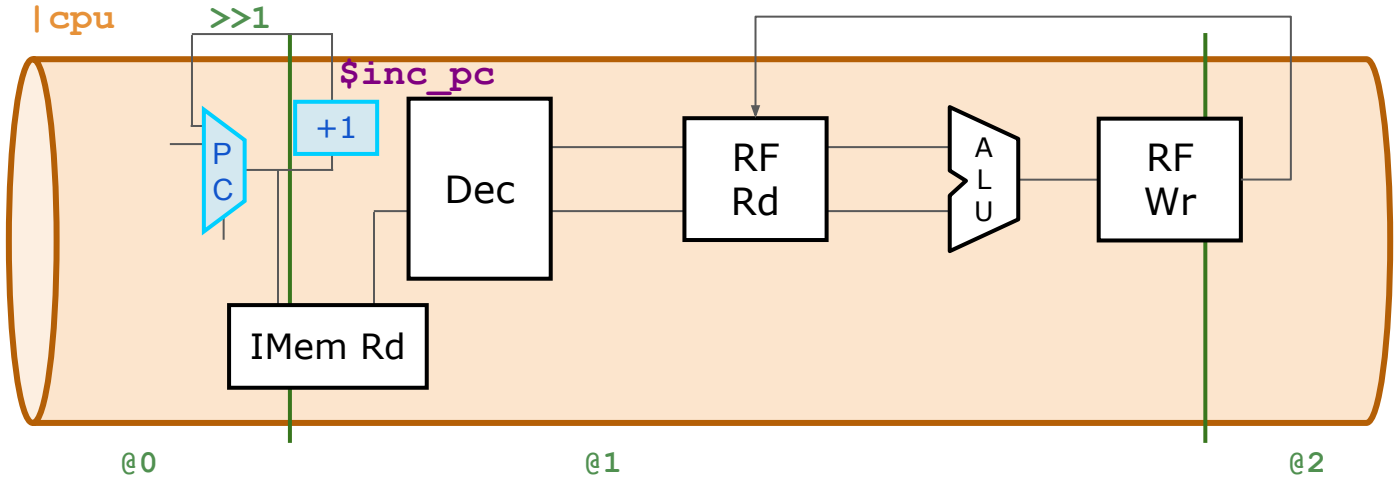
Review code containing (or including):

- 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

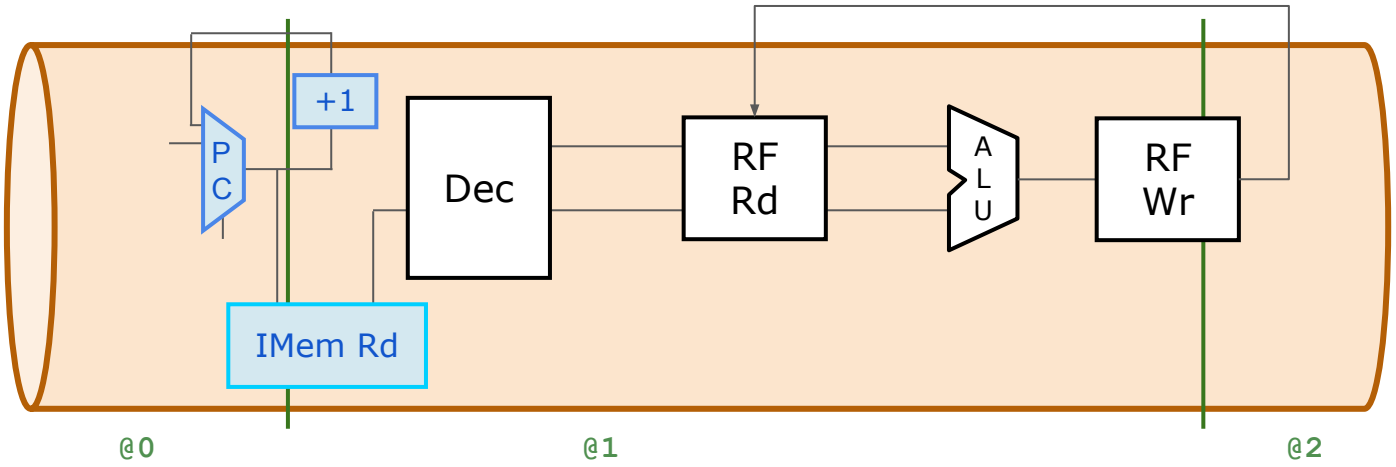
Reset `$pc[31:0]` to 0 if previous instruction was a "reset instruction" (`>>1$reset`), and increment by 1 instruction (`32'd4` bytes) thereafter. (We'll add branch support later.)



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

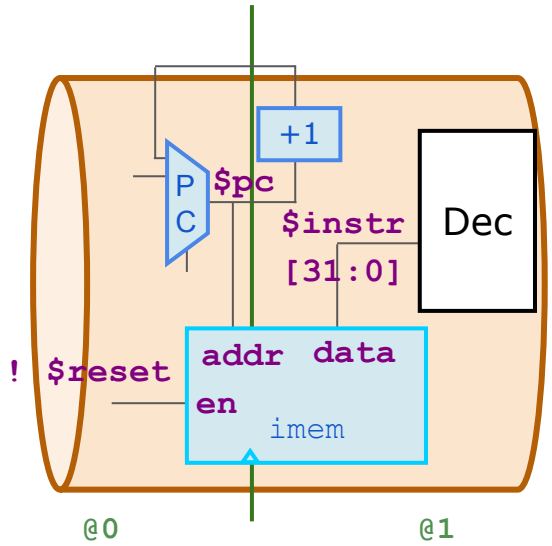
Lab: Fetch (part 1)

Add the instruction memory containing the program (provided).



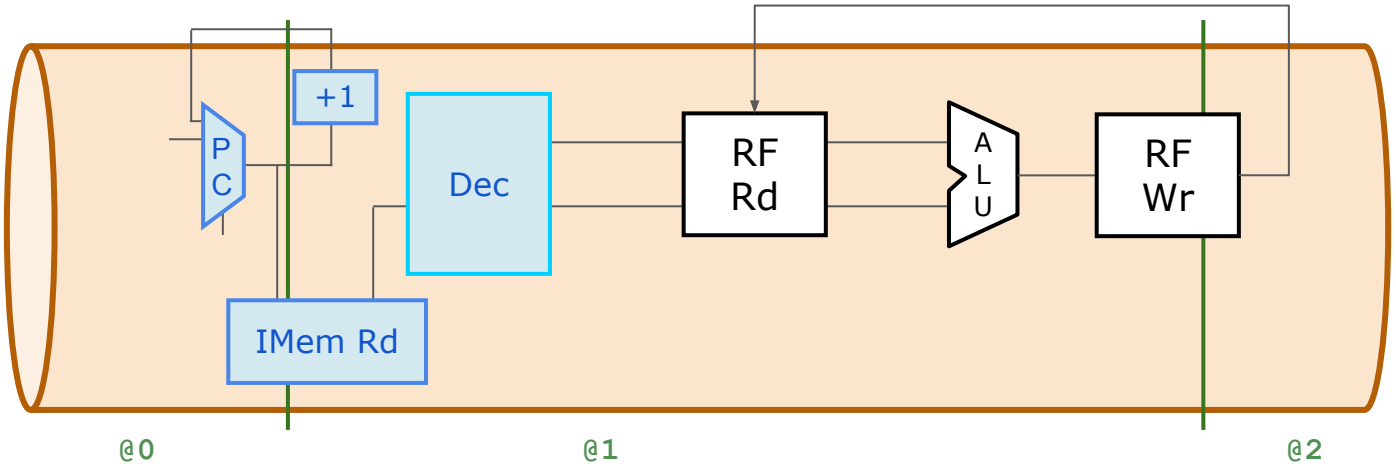
1. Uncomment `//m4+imem(@1)`, and `//m4+cpu_viz(@4)` compile, and observe log errors.

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

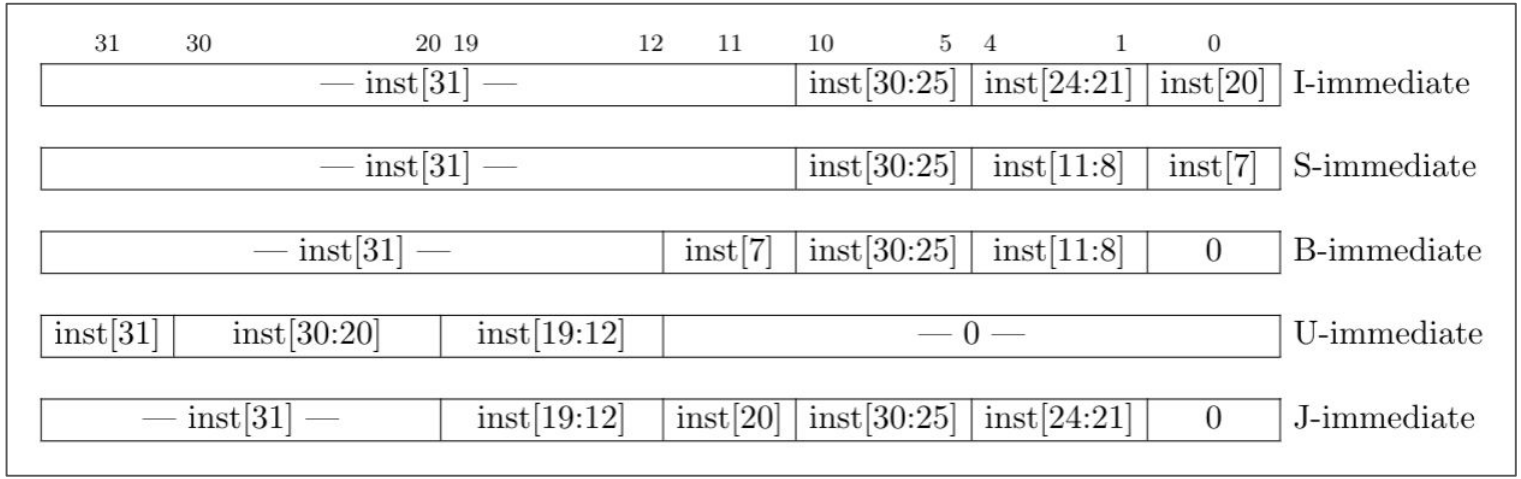
instr[4:2] instr[6:5]	000	001	010	011	100	101	110	111
00	I	I	-	-	I	U	I	-
01	S	S	-	R	R	U	R	-
10	R4	R4	R4	R4	R	-	-	-
11	B	I	-	J	I (unused)	-	-	-

```
$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`

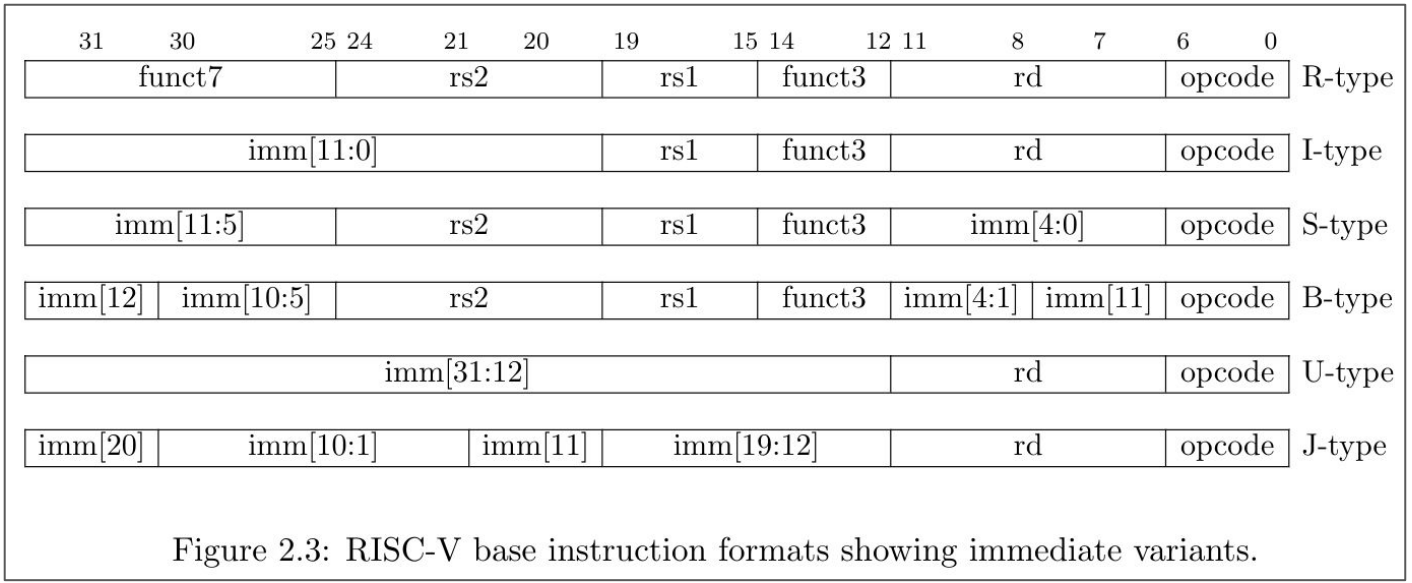


Figure 2.3: RISC-V base instruction formats showing immediate variants.

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

Check behavior in simulation.

Lab: RISC-V Instruction Field Decode

Let's use when conditions

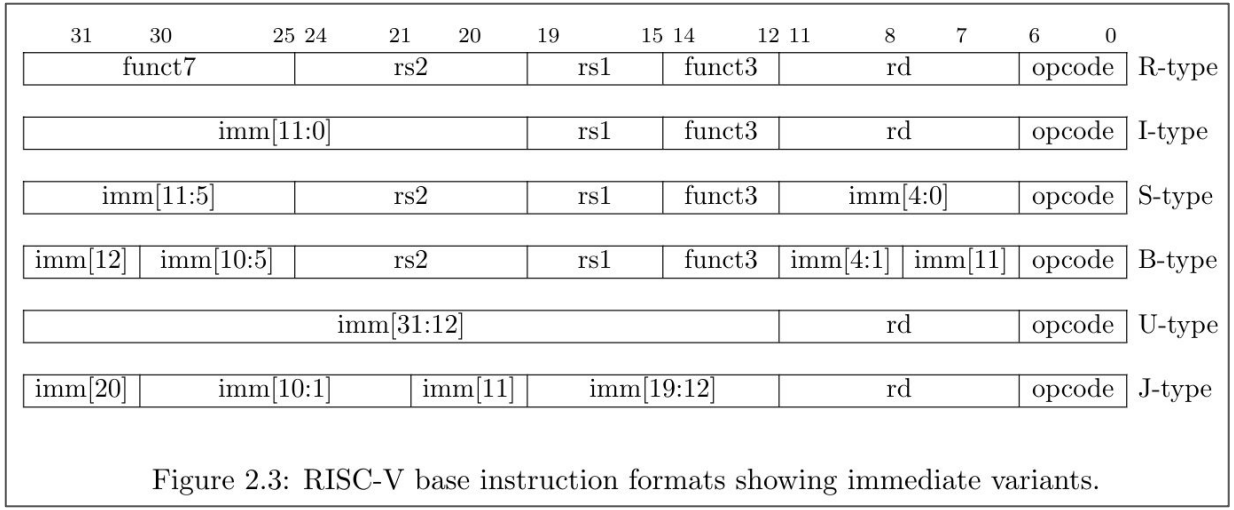


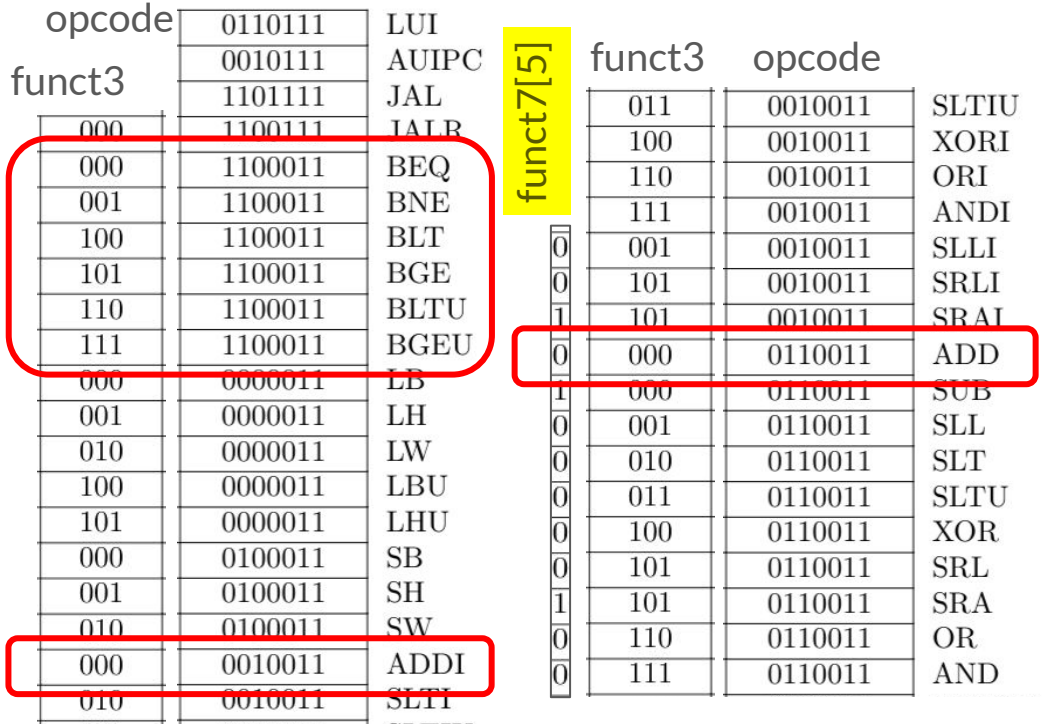
Figure 2.3: RISC-V base instruction formats showing immediate variants.

```
$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

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

Complete circled instructions.

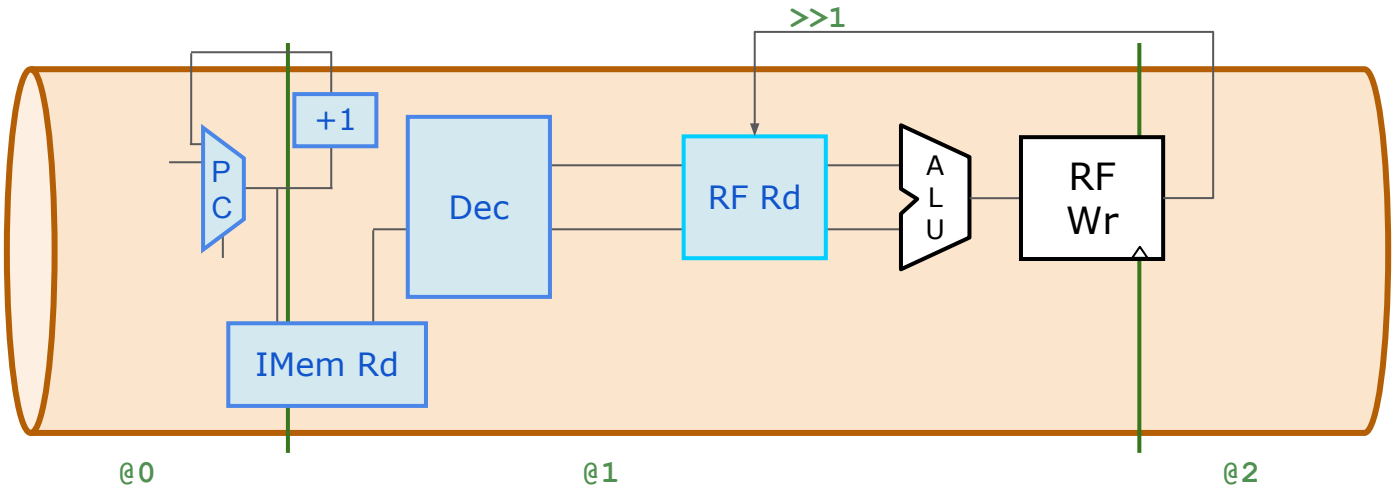
```
$dec_bits[10:0] =
   {$funct7[5],$funct3,$opcode};
$is_beq = $dec_bits ==?
          11'bx_000_1100011;

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

(no comma)

Check behavior in simulation and confirm save.

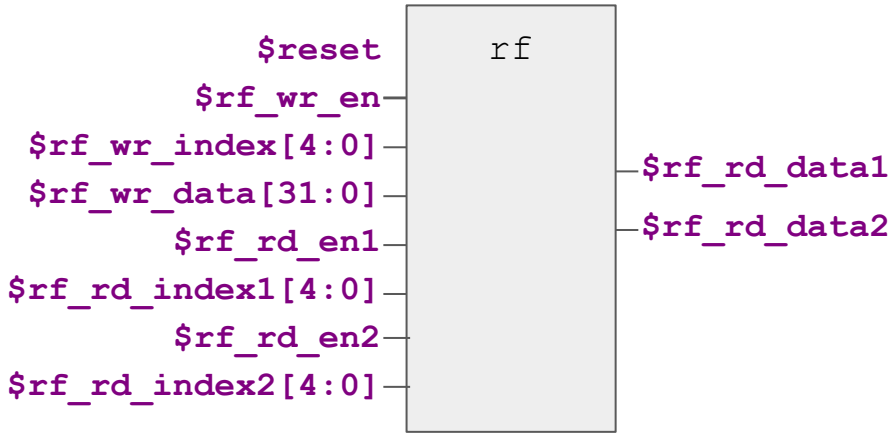
Register File Read



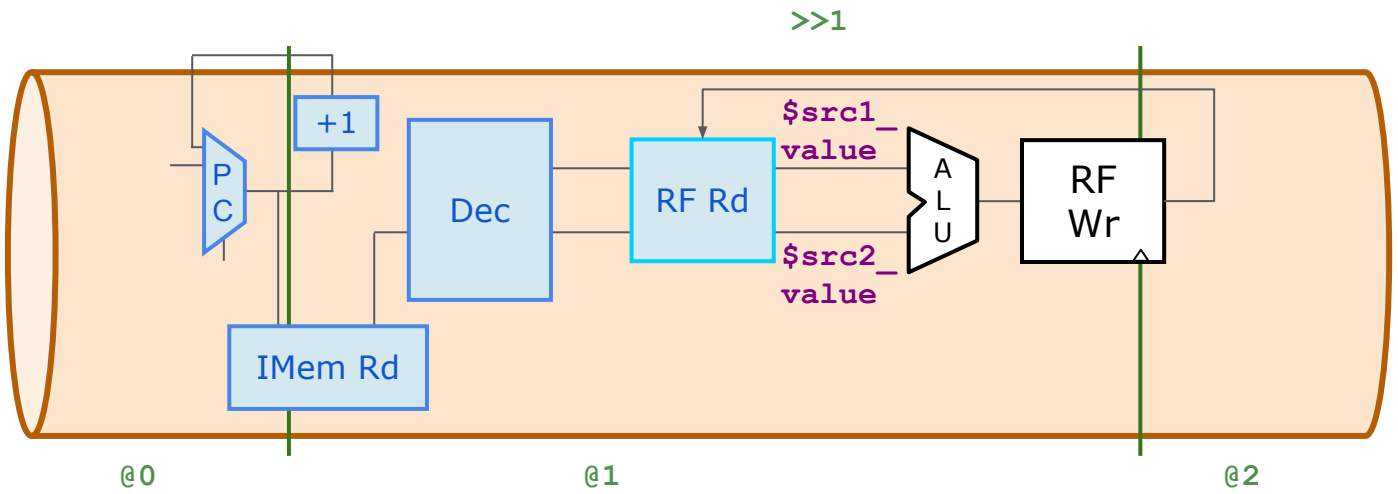
Lab: Register File Read

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:



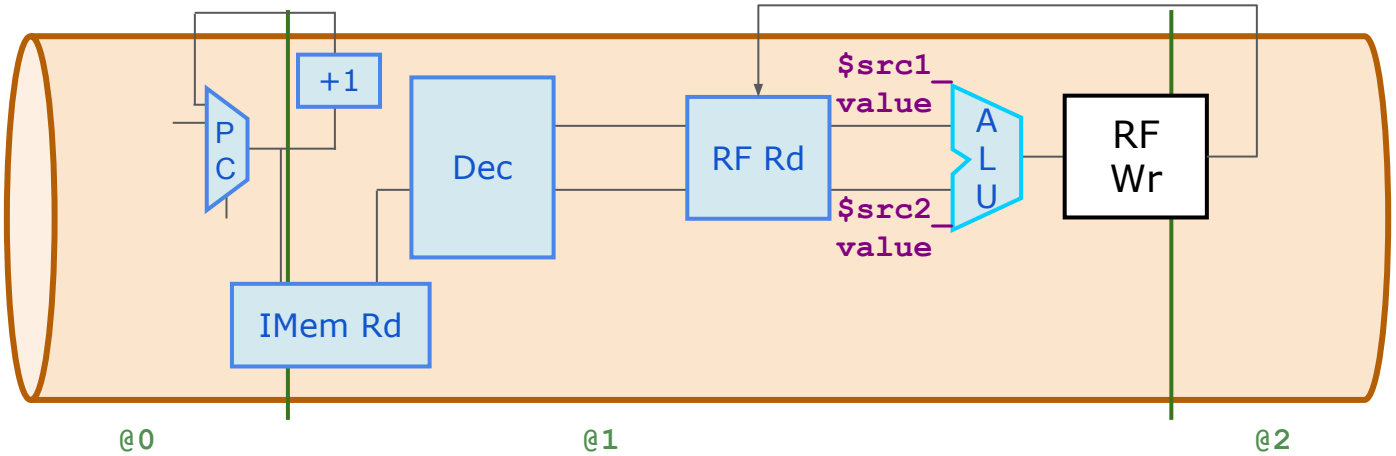
Lab: Register File Read (part 2)



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

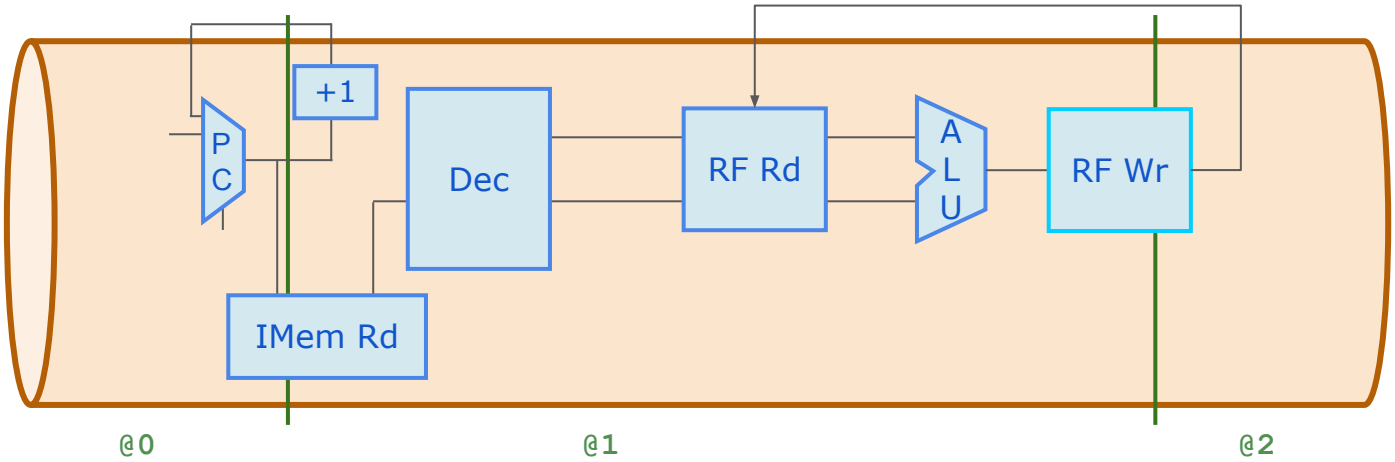
Lab: ALU

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




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

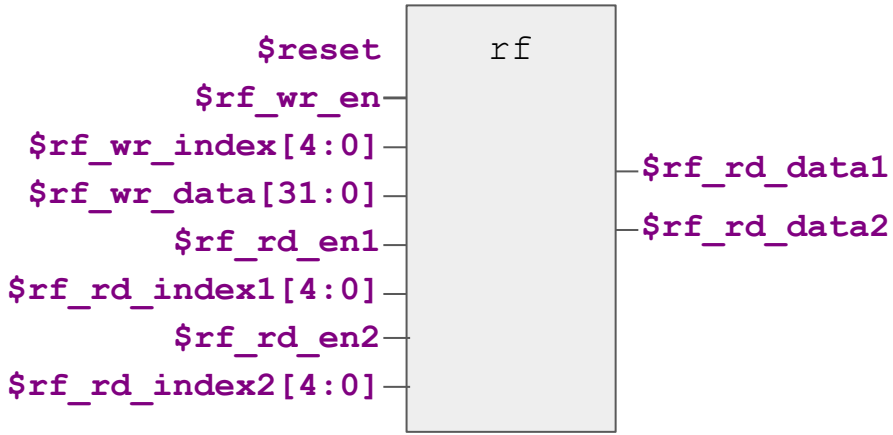
Register File Write



Lab: Register File Write

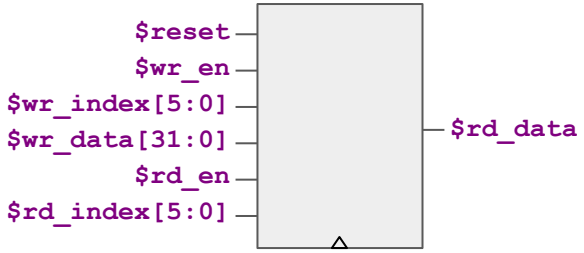
1. Provide proper input assignments to enable RF write (`wr`) of `$result` to `$rd` (dest reg) when `$rd_valid` for a valid instruction.
2. Debug in simulation. Should be writing and reading registers.
3. But wait, in RISC-V, x0 is "always-zero". Writes should be ignored. Add logic to disable write if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

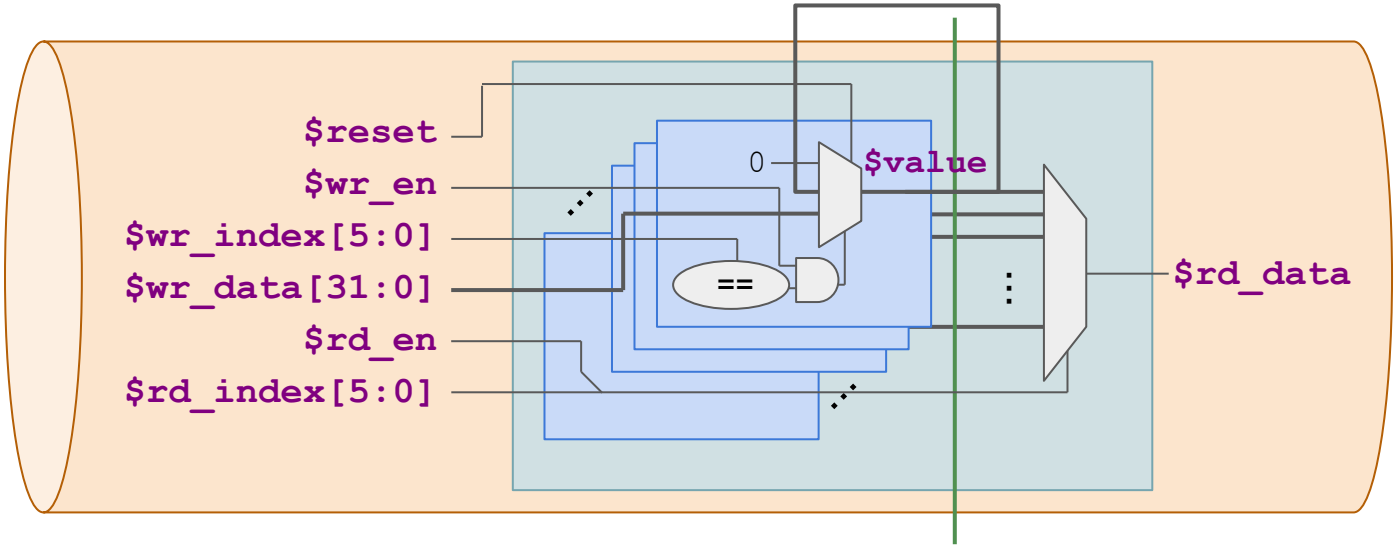
2-read, 1-write register file:



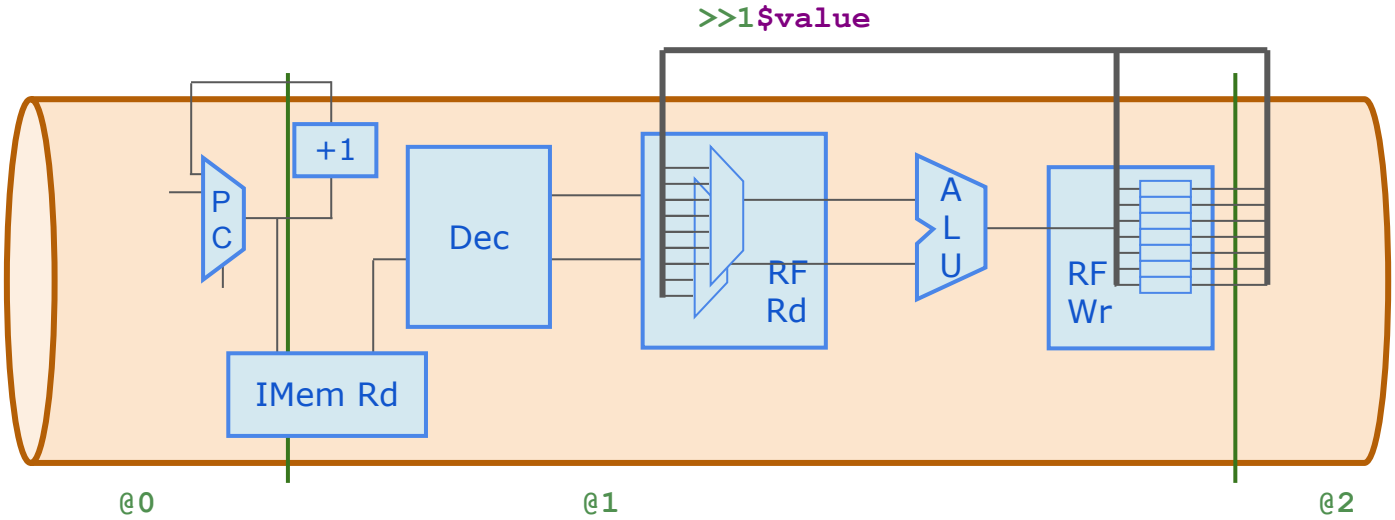
Arrays - What's inside?

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

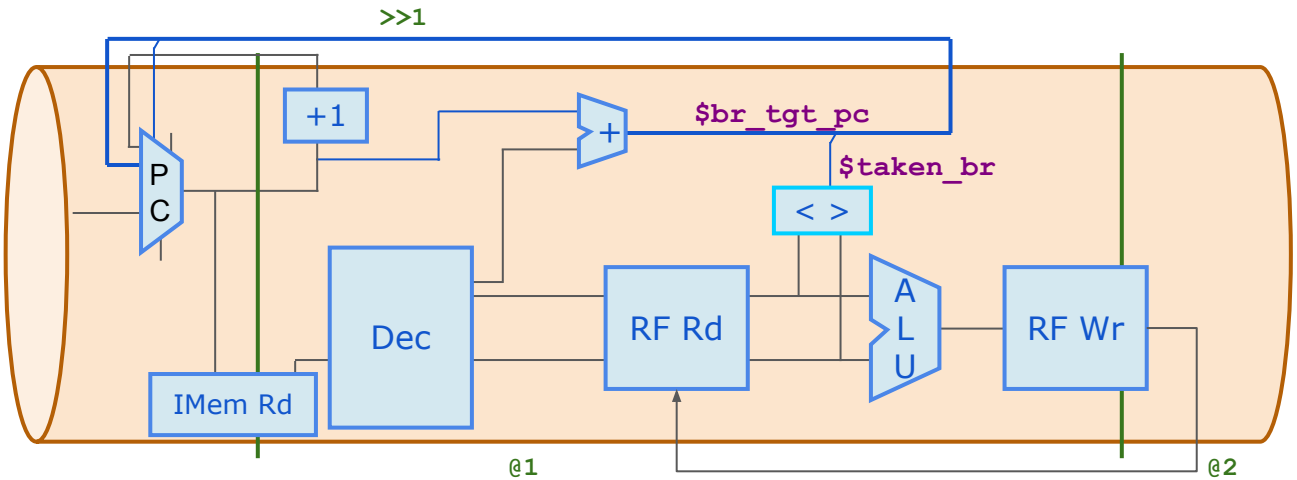




Register File - Detailed



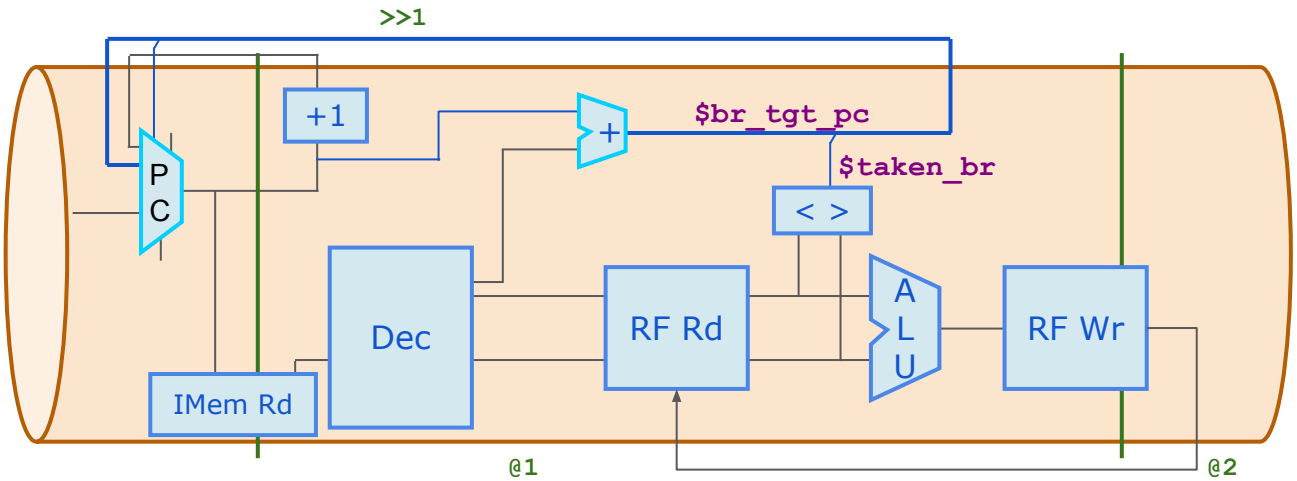
Lab: Branches



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 **`$br_tgt_pc`** when the previous instruction was **`$taken_br`**.
3. Check behavior in simulation. Program should now sum values {1..9}!
4. Debug as needed, and save outside of Makerchip.

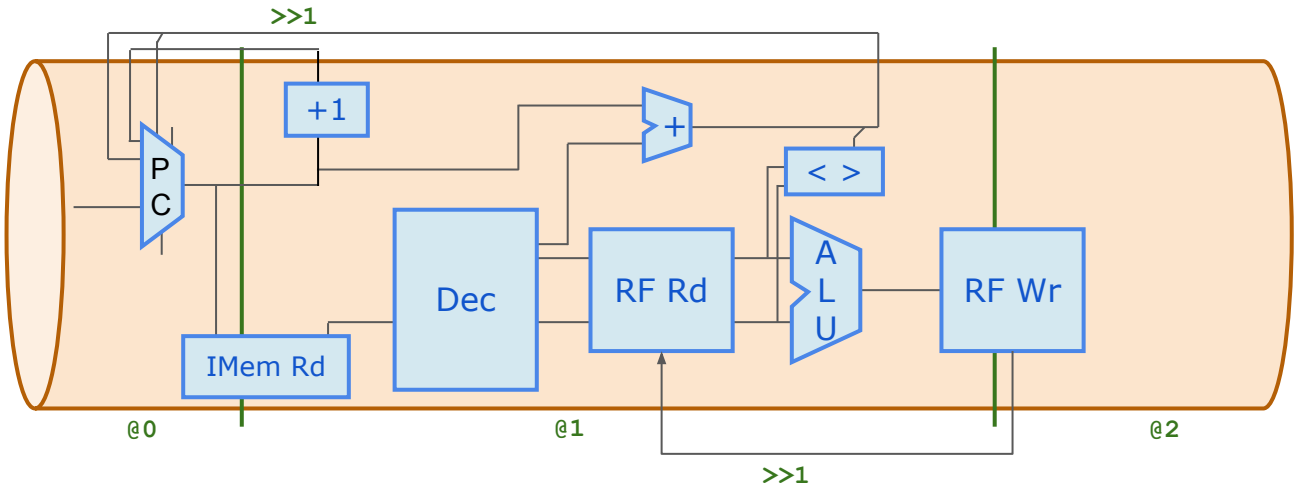
Lab: Testbench

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

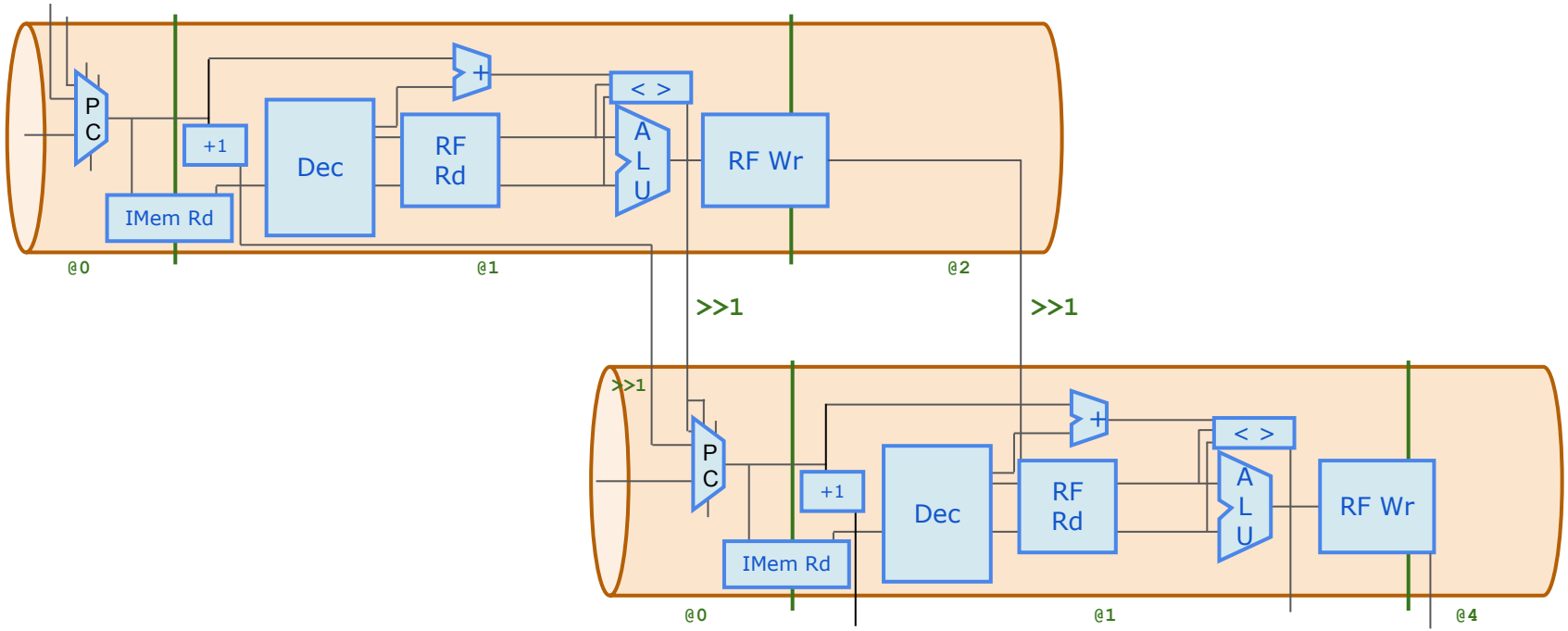
```
*passed = |cpu/xreg[10]>>5$value == (1+2+3+4+5+6+7+8+9);
```

Check log for passed message.

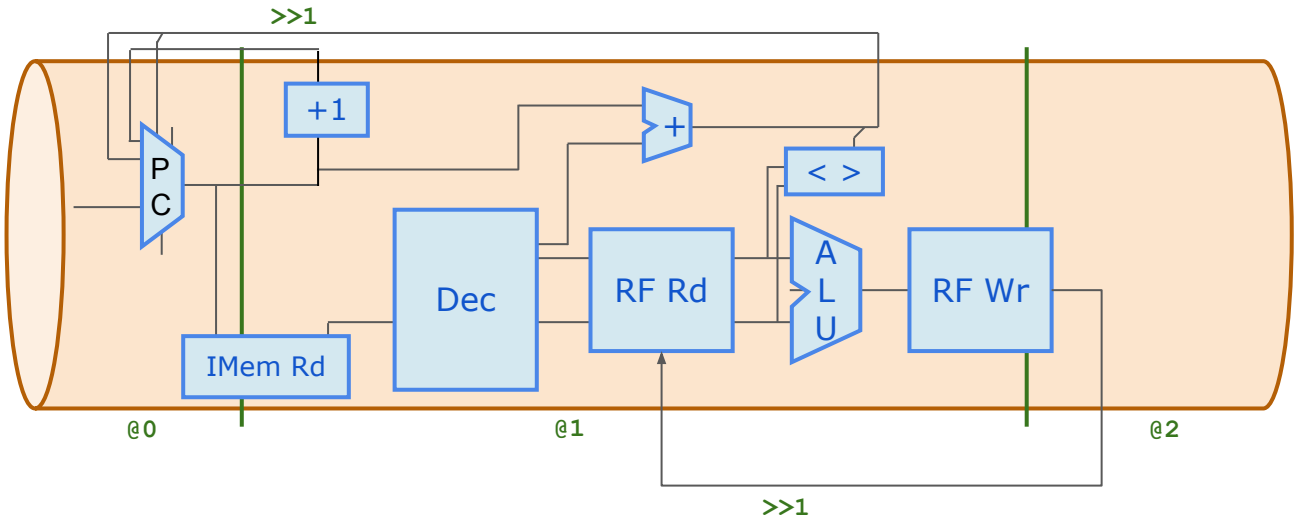
Pipelining Your RISC-V



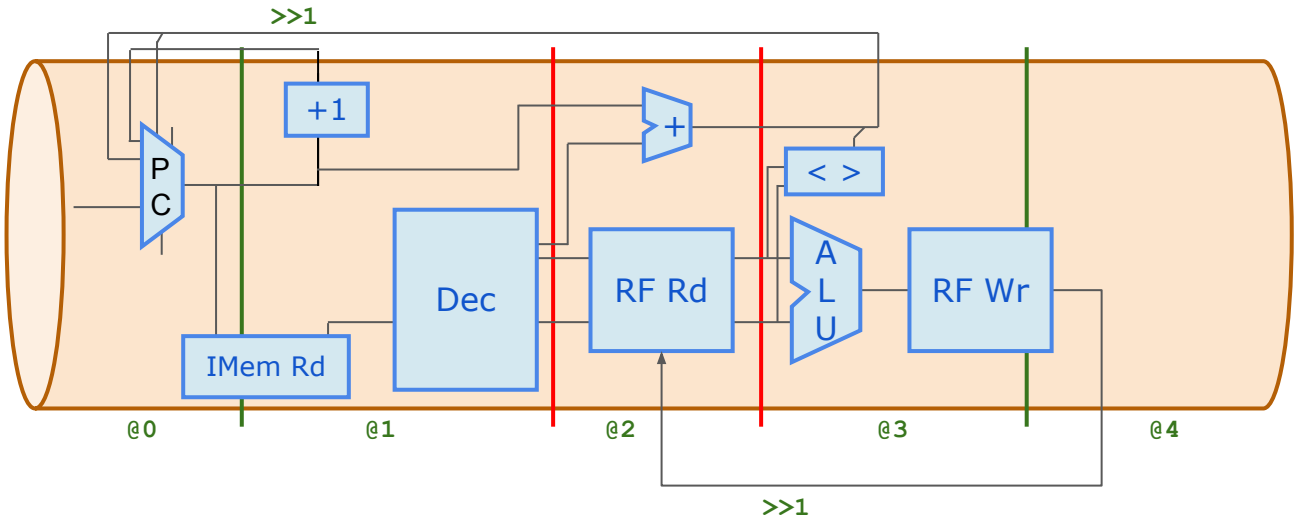
Waterfall Logic Diagram



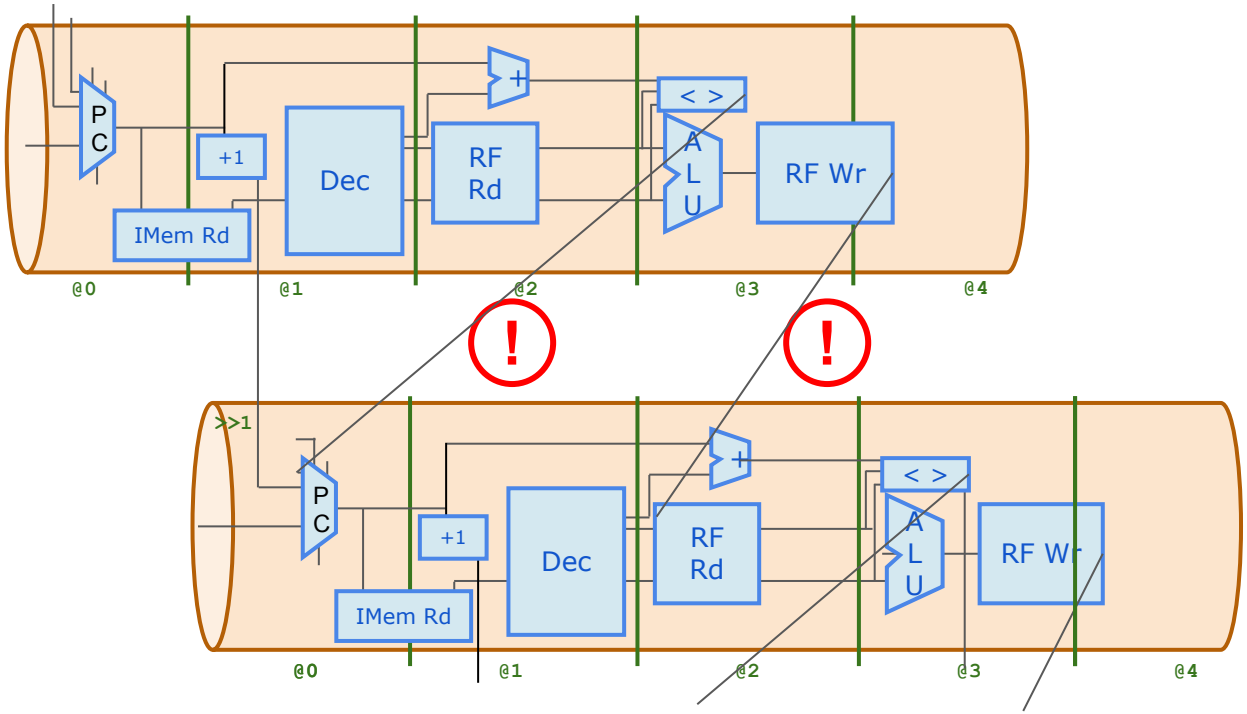
Pipelining Your RISC-V



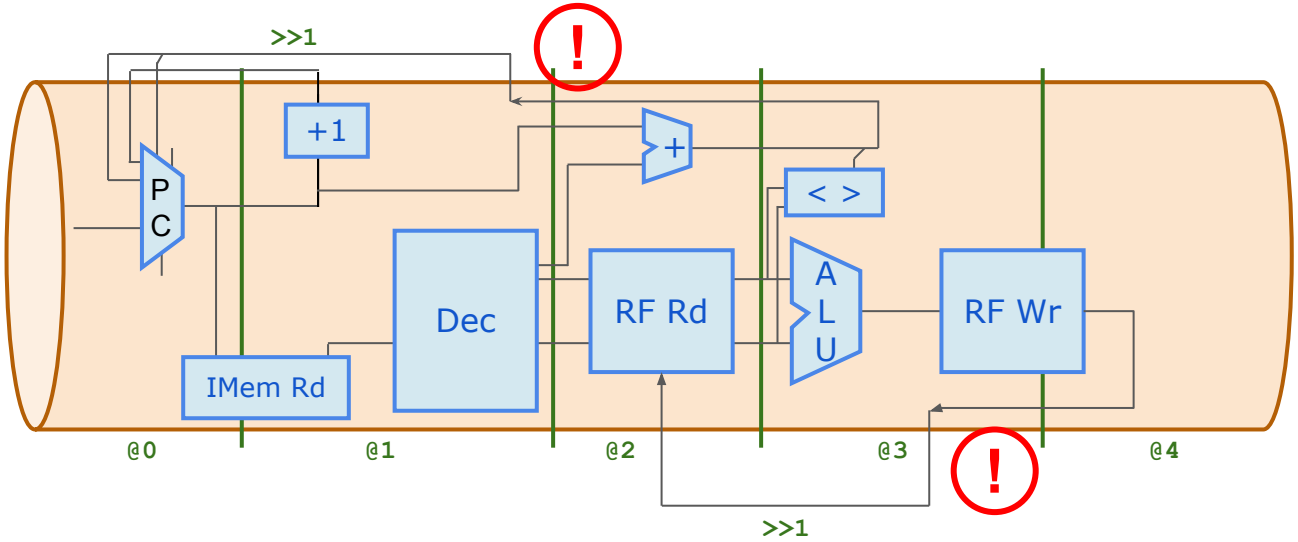
Pipelining Your RISC-V



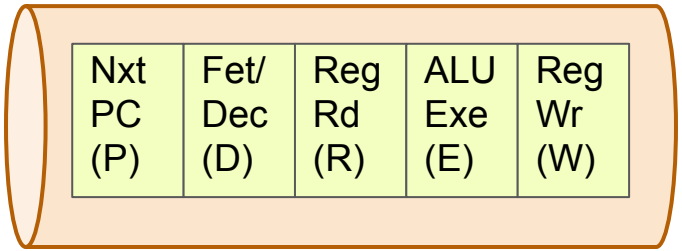
Waterfall Logic Diagram

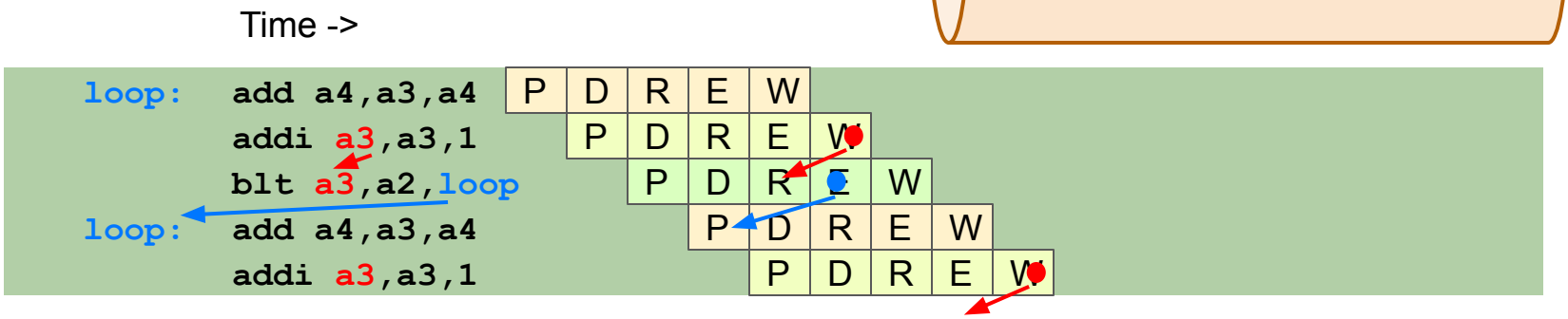


Pipelining Your RISC-V

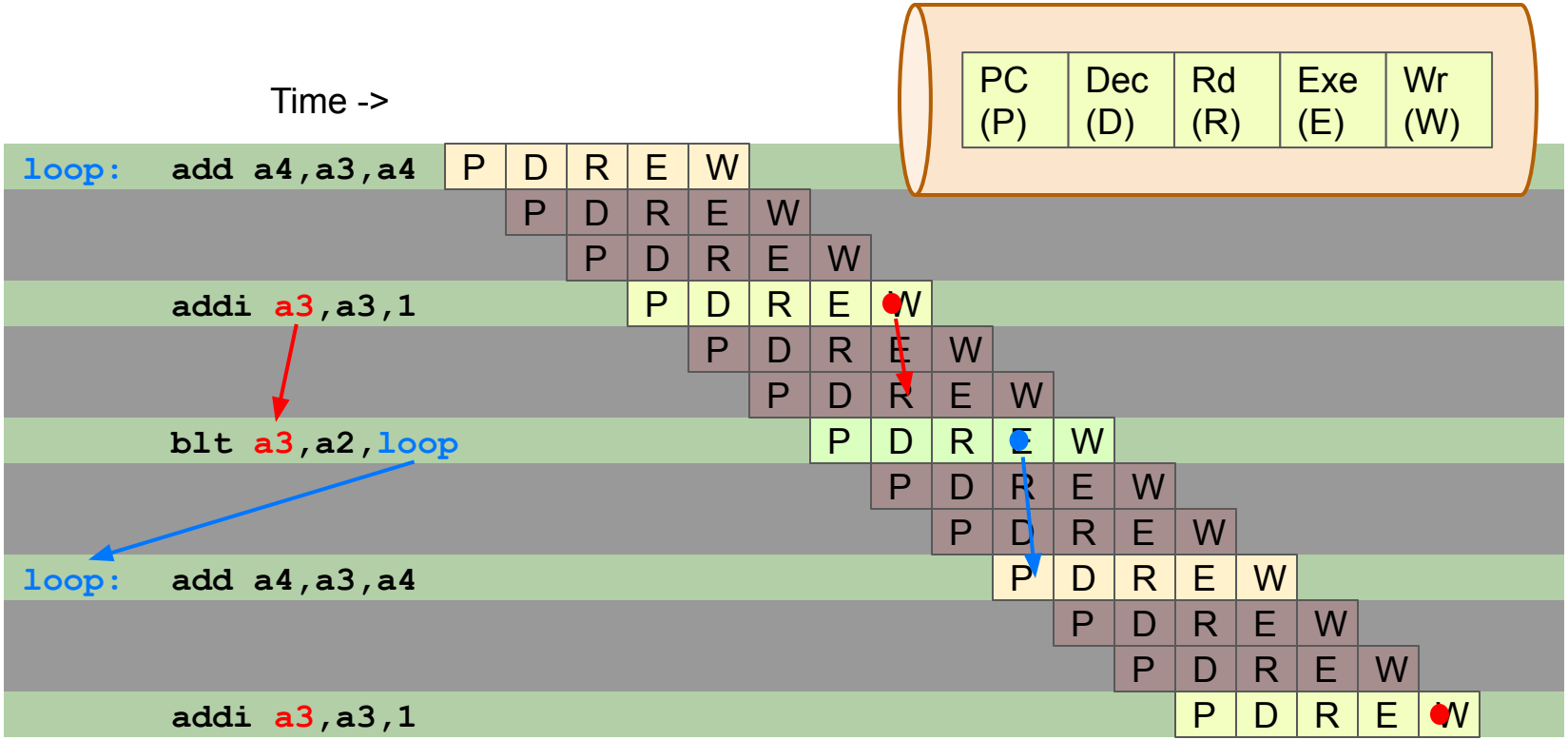


RISC-V Waterfall Diagram & Hazards

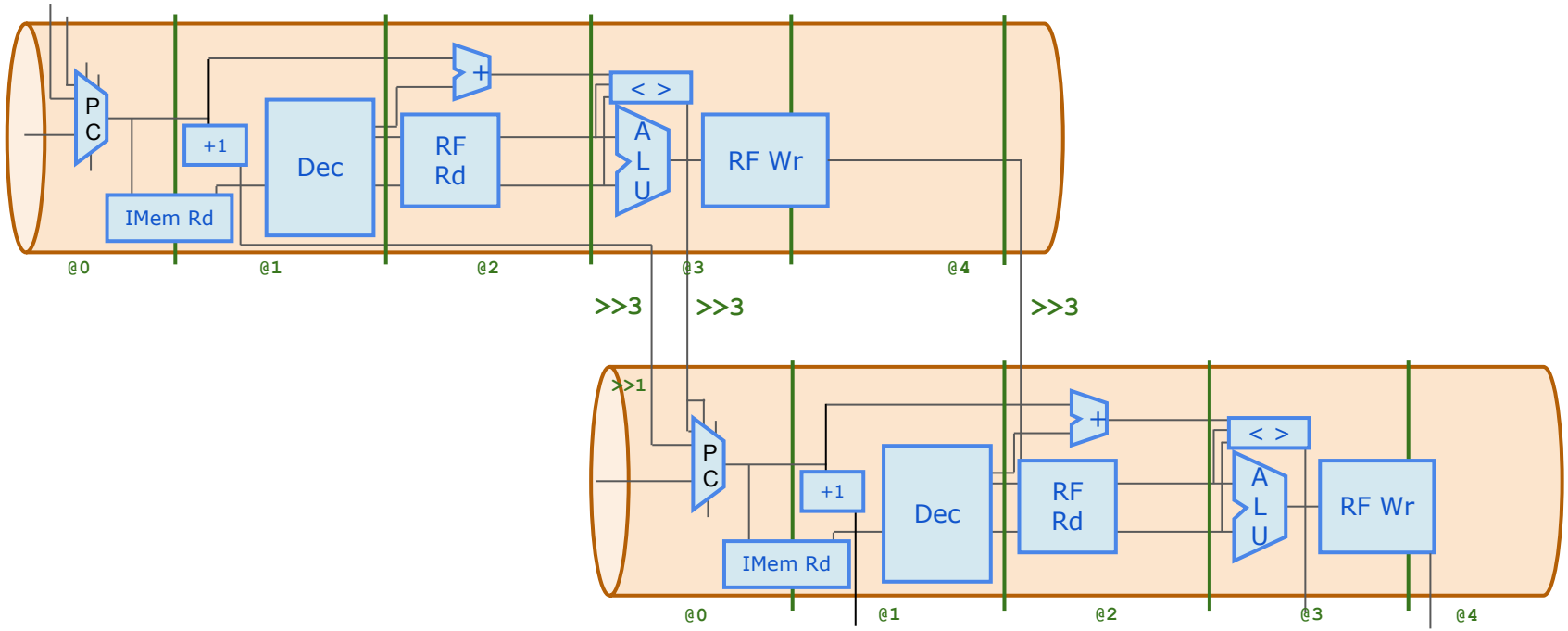




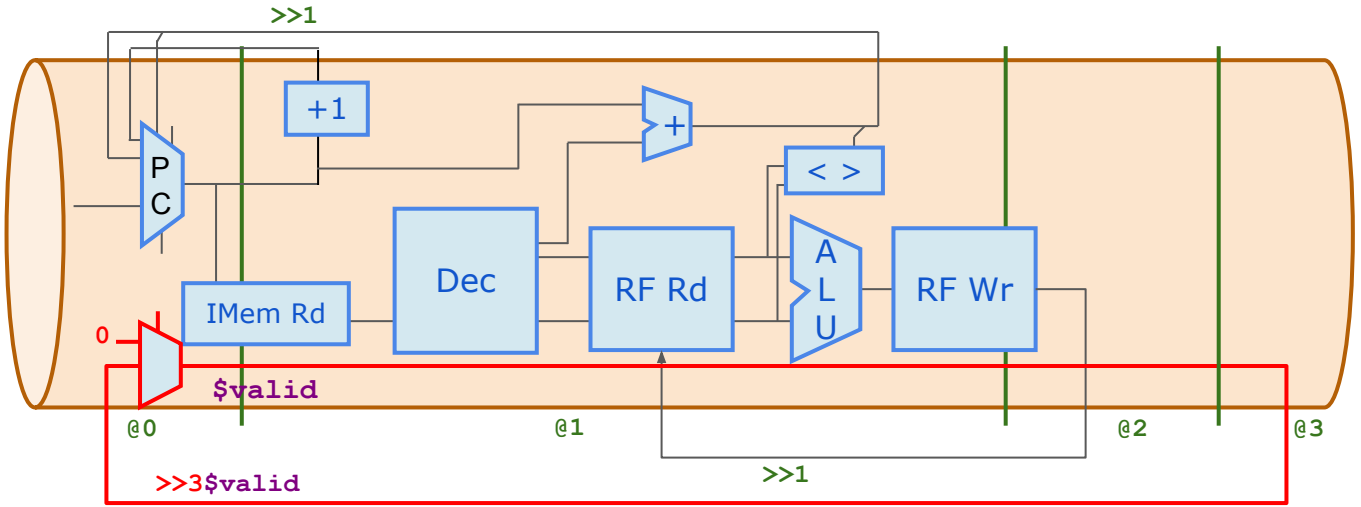
RISC-V Waterfall Diagram & Hazards



Waterfall Logic Diagram

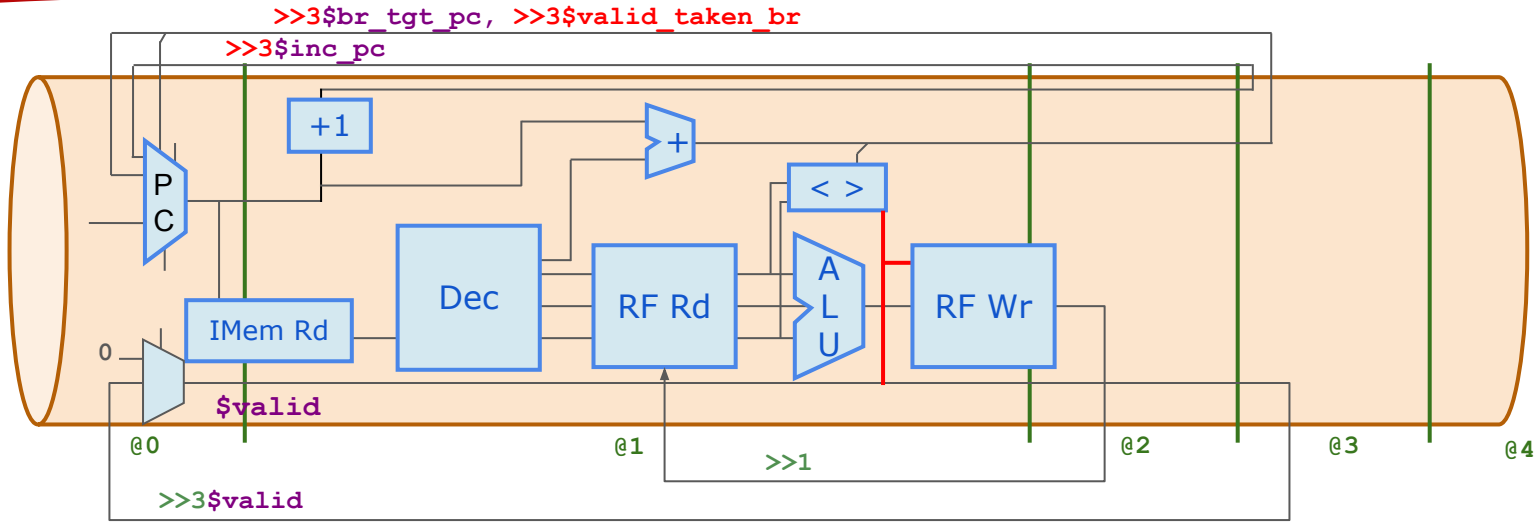


Lab: 3-Cycle $valid



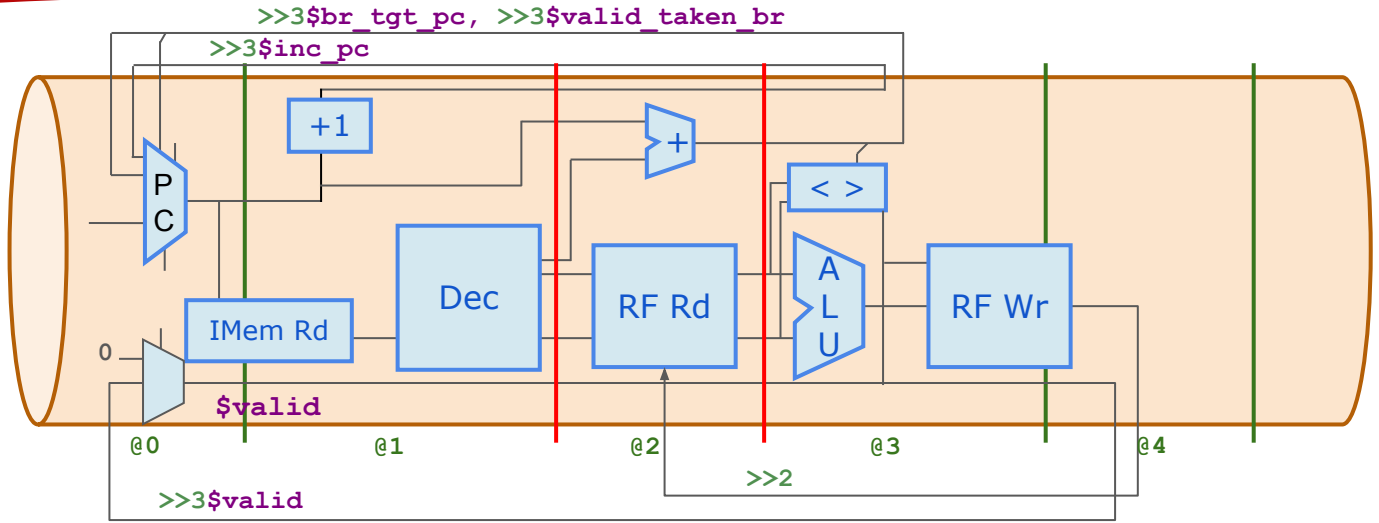
1. Create **$start** to provide first **$valid** pulse (reset last cycle, but not this cycle).
2. Create **$valid**: 0 during **$reset**, 1 for **$start**, **>>3$valid** o/w. Check simulation.

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.

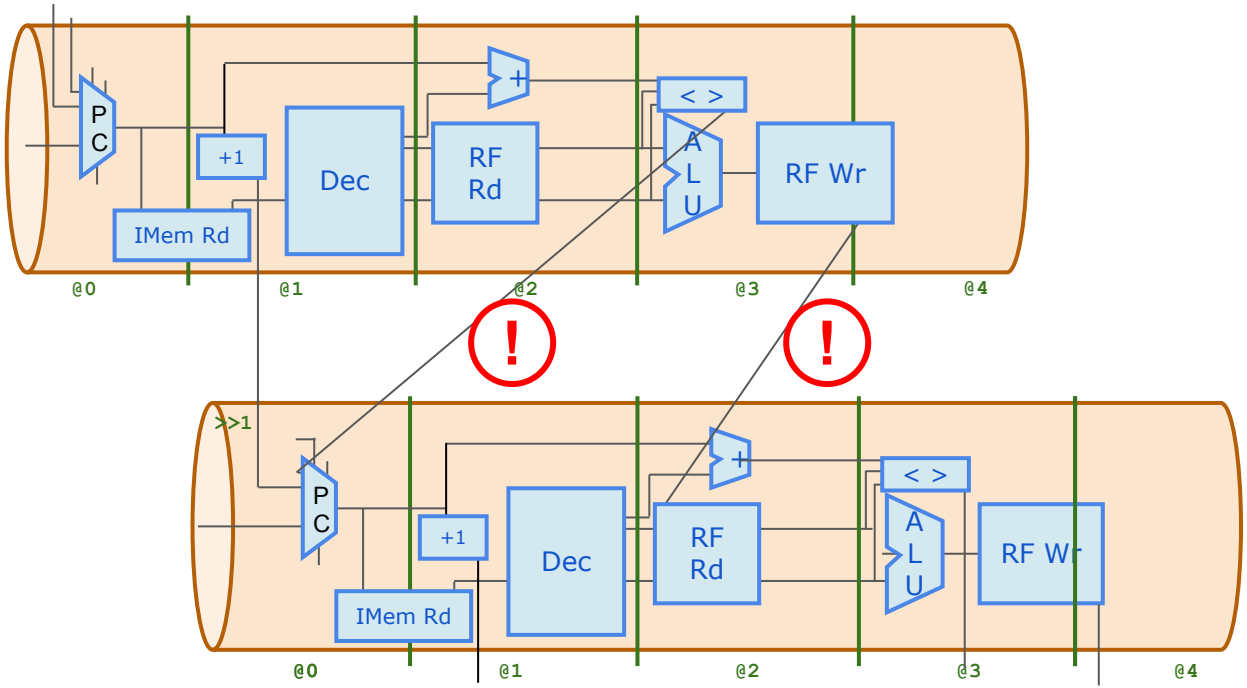
Lab: 3-Cycle RISC-V



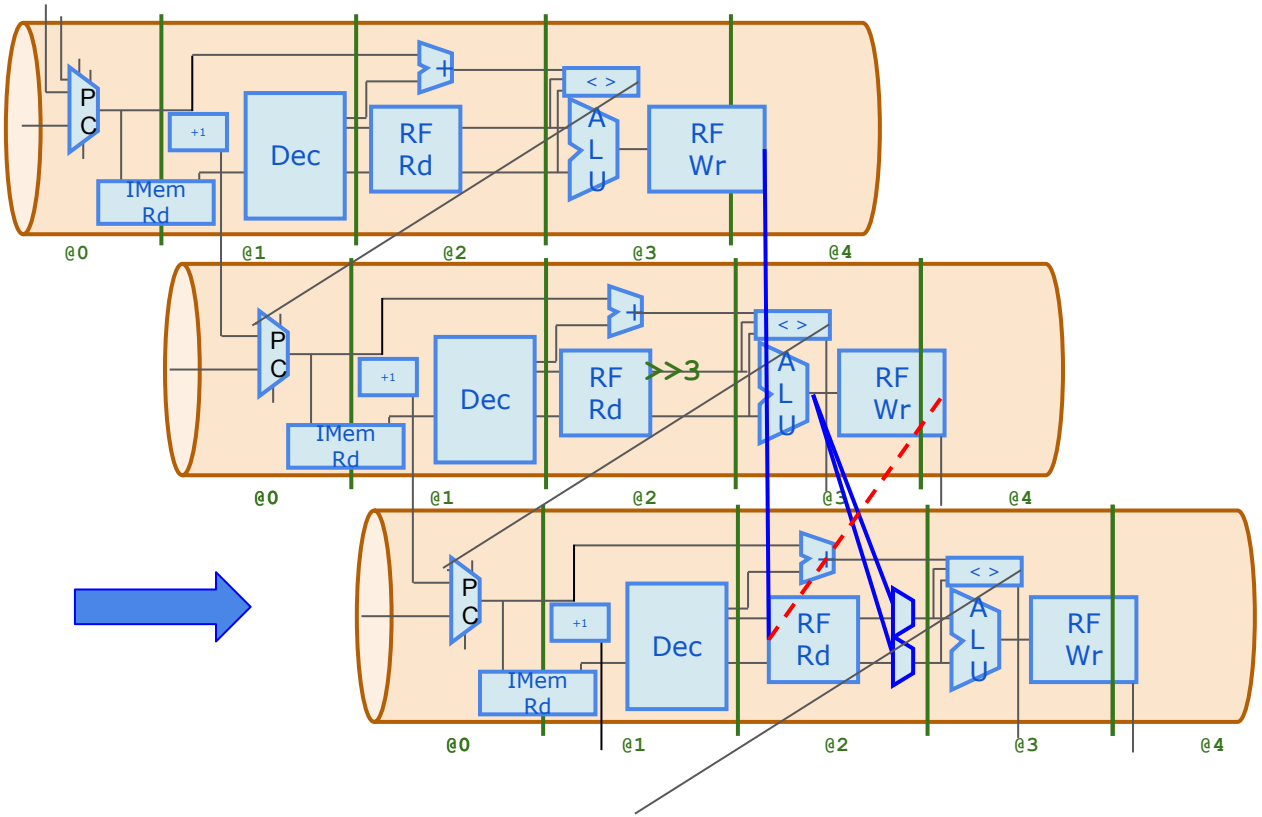
1. Partition logic into pipeline stages as above. Add stages; cut-n-paste code.
2. For RF use `m4+rf(@2, @3)`, implying `>>2`. (Since previous 2 instructions do not update RF, `>>1`, `>>2`, `>>3` are functionally equivalent.)
3. Debug as needed.

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

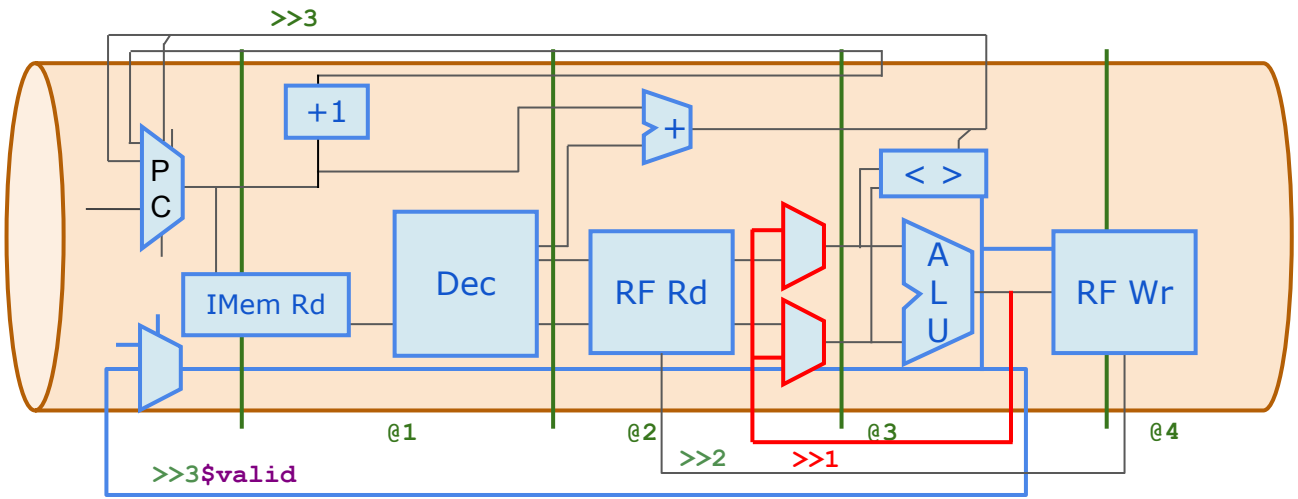
~1 Instruction Per Cycle (~1 IPC)



Register File Bypass

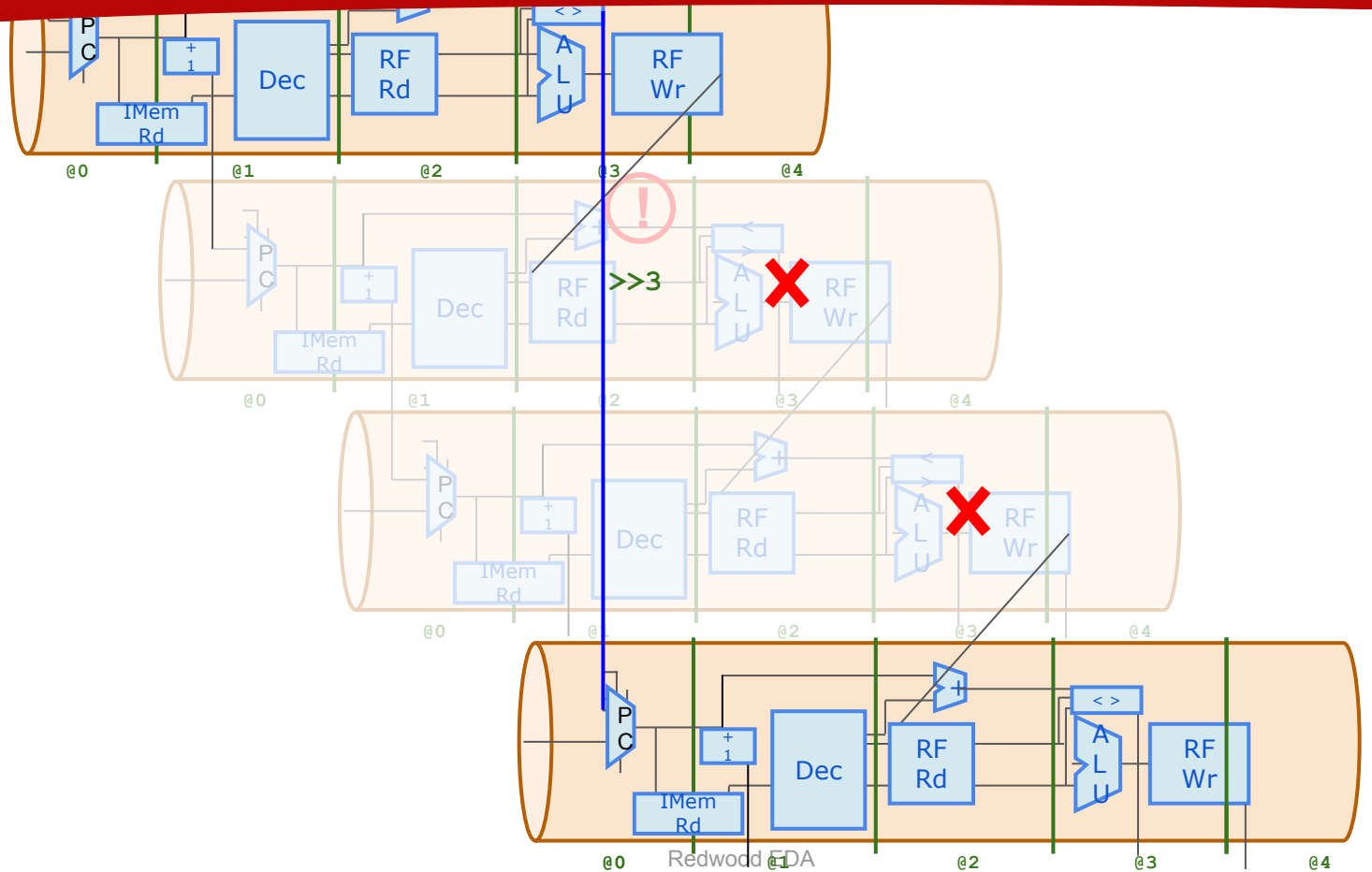


Lab: Register File Bypass

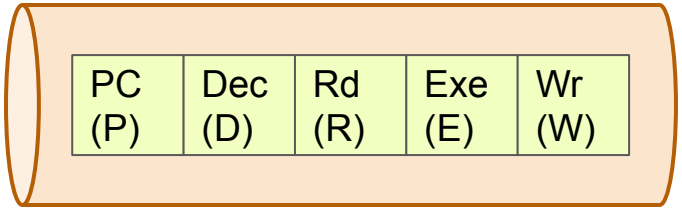


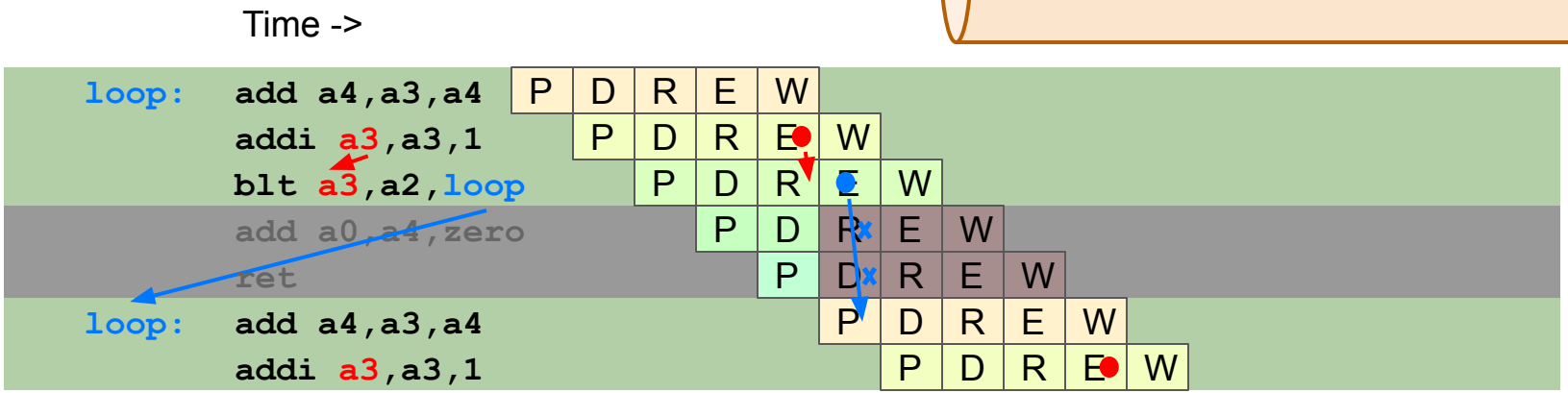
1. RF read uses RF as written 2 instructions ago (already correct).
2. Update expressions for `$srcX_value` to select previous `$result` if it was written to RF (write enable for RF) and if previous `$rd` == `$rsX`.
3. (Should have no effect yet)

Branches

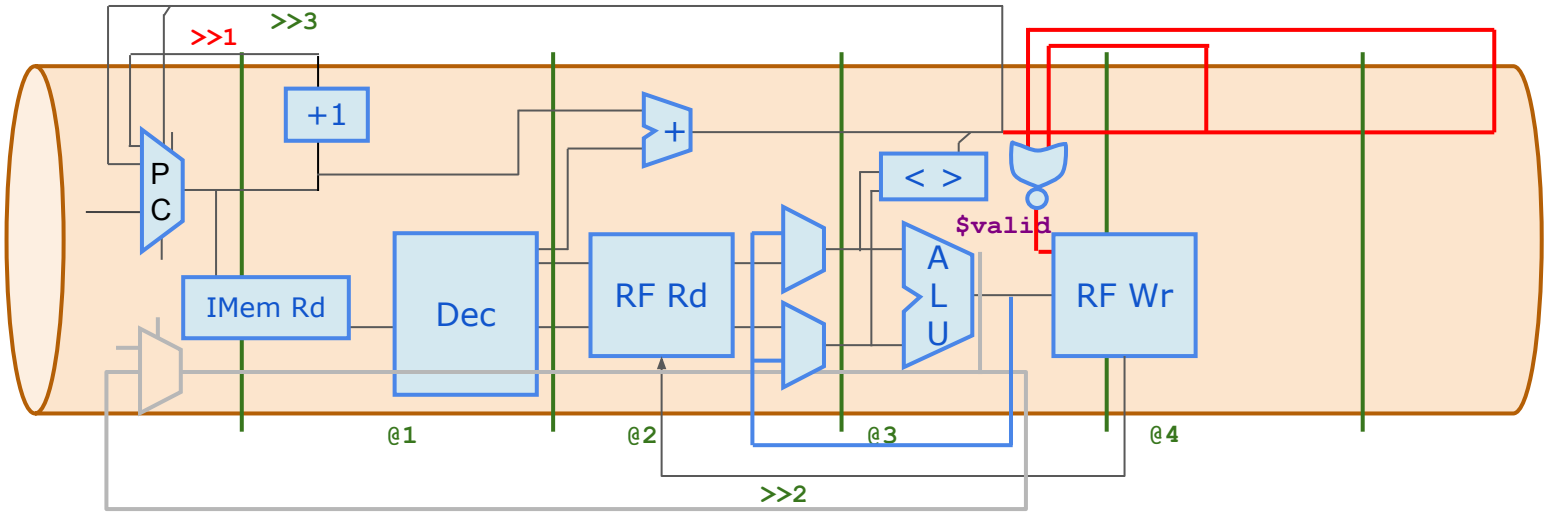


Branches


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

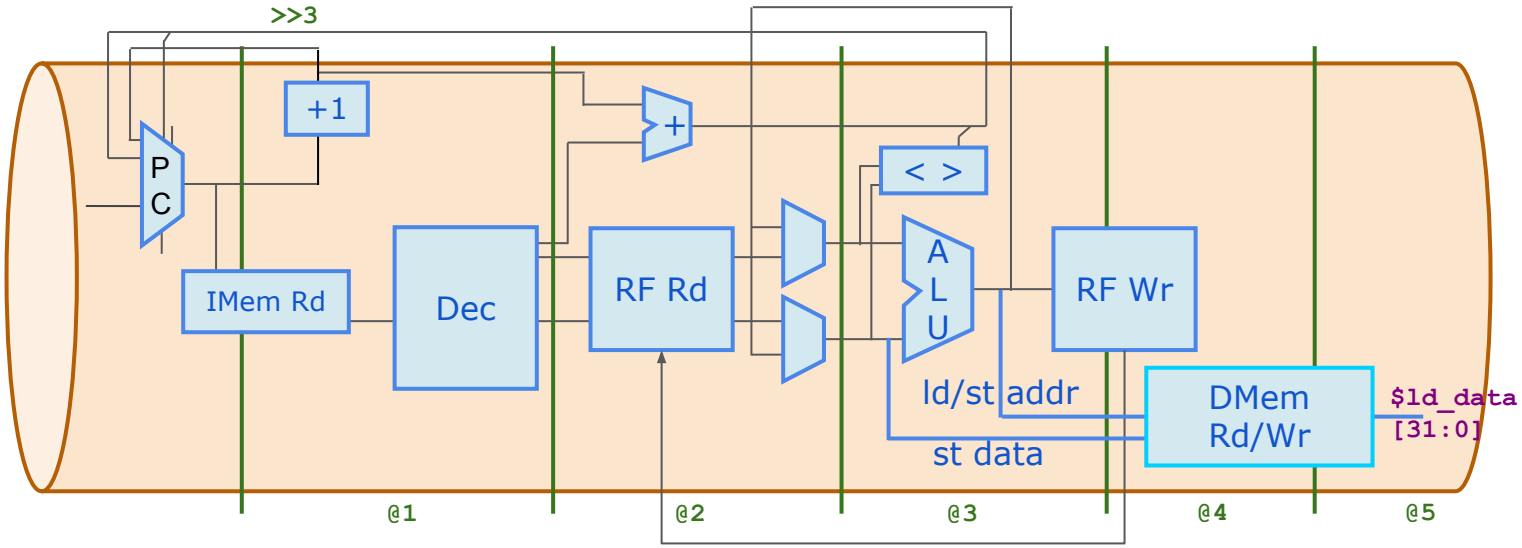
```
ANDI   $src1_value & $imm;
ORI    $src1_value | $imm;
XORI   $src1_value ^ $imm;
ADDI   $src1_value + $imm;
SLLI   $src1_value << $imm[5:0];
SRLI   $src1_value >> $imm[5:0];
AND    $src1_value & $src2_value;
OR     $src1_value | $src2_value;
XOR    $src1_value ^ $src2_value;
```

```
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)`

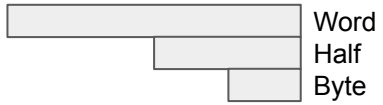
`LOAD rd, imm(rs1)`

rd <= DMem[addr]

`STORE (SW,SH,SB)`

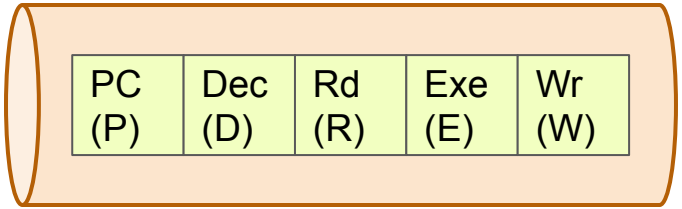
`STORE rs2, imm(rs1)`

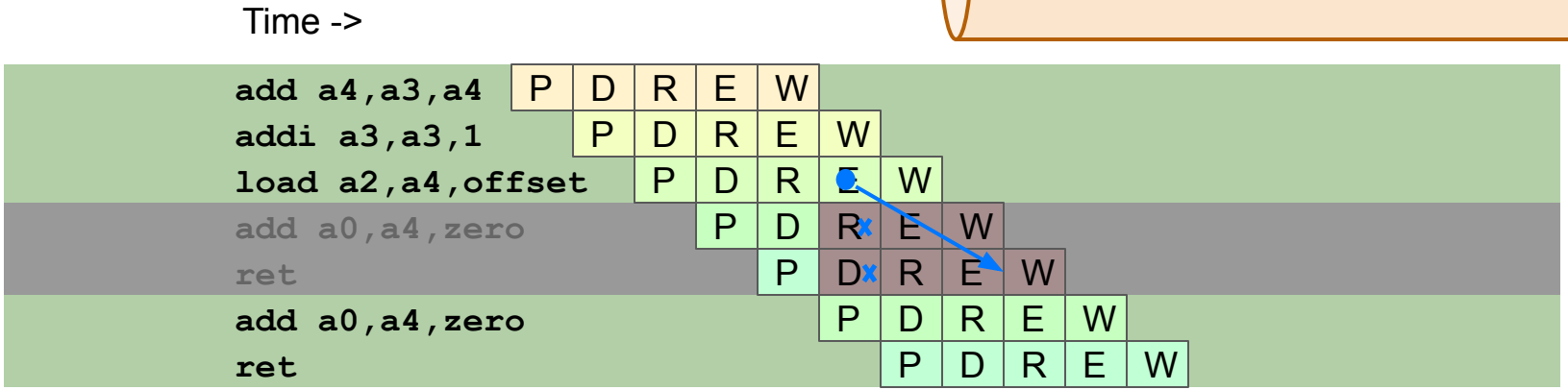
DMem[addr] <= rs2



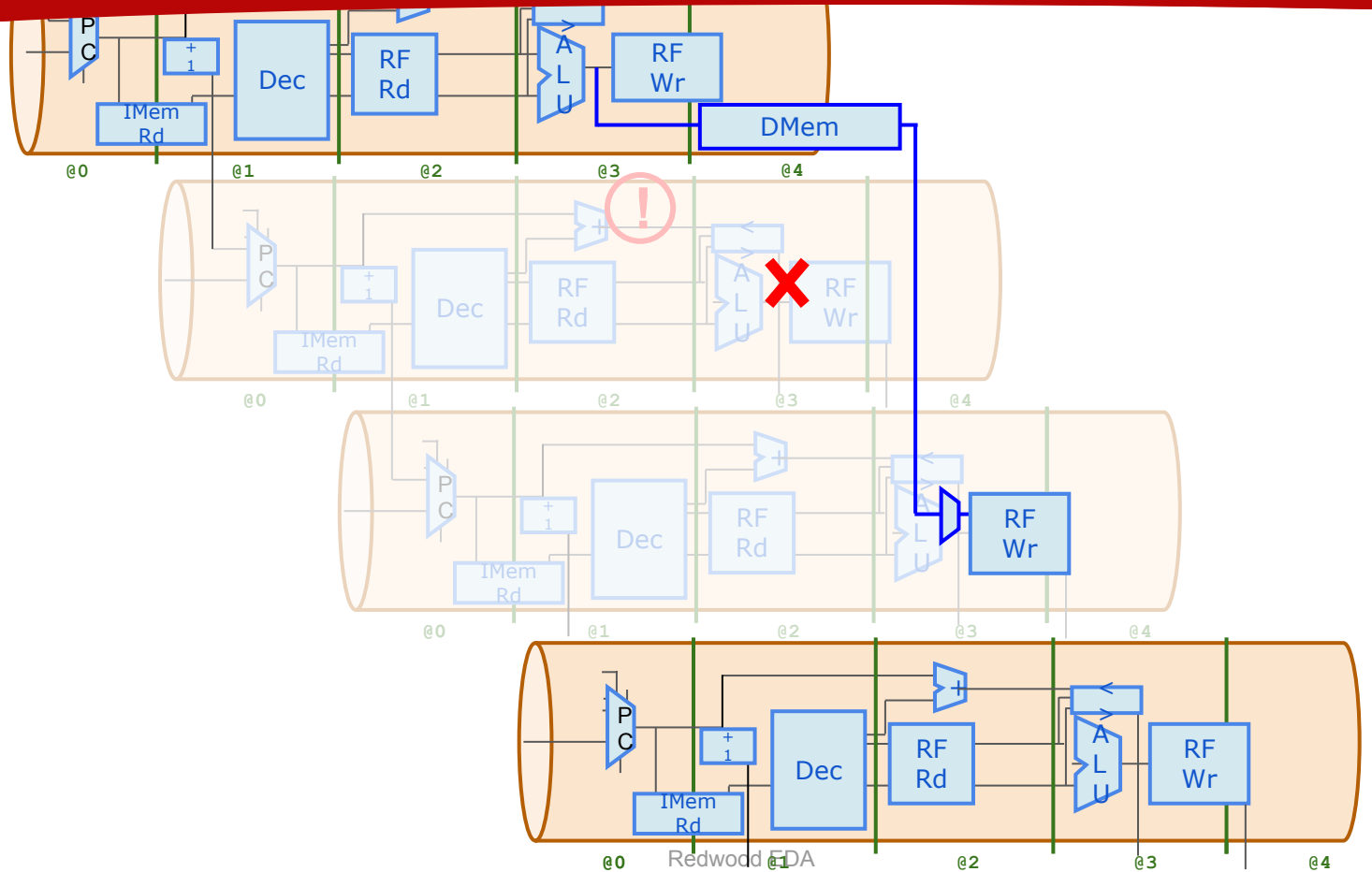
where, addr <= rs1 + imm (like addi)

Loads

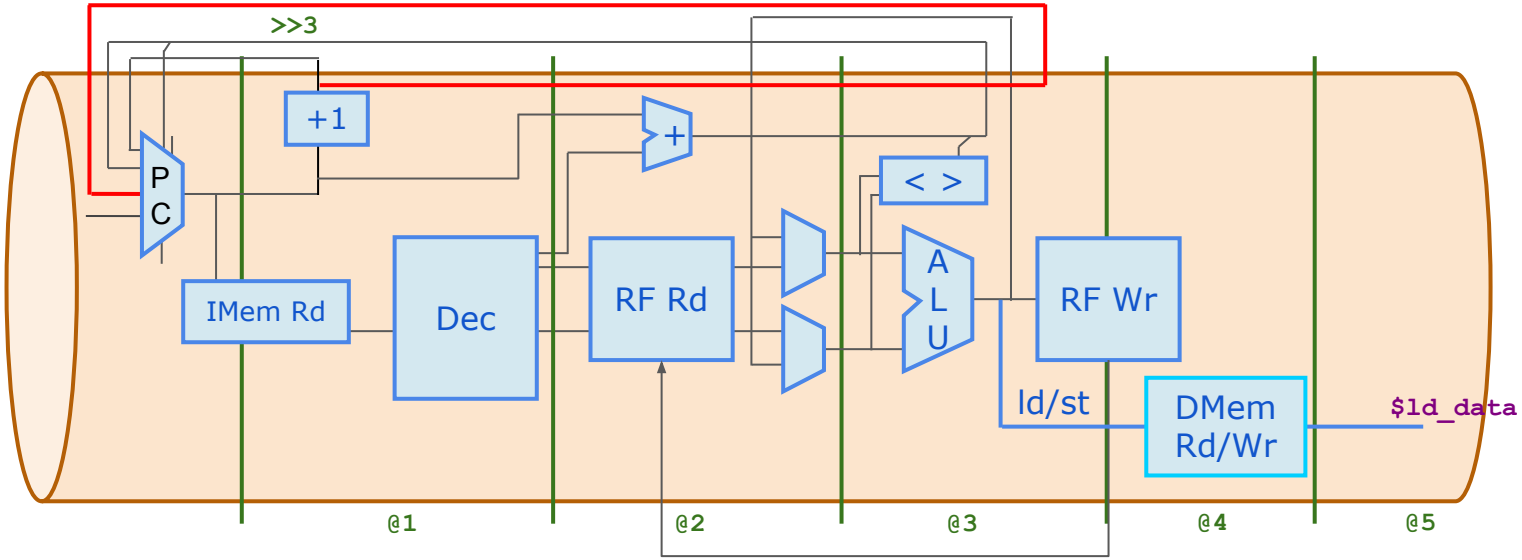

PC (P)
Dec (D)
Rd (R)
Exe (E)
Wr (W)


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

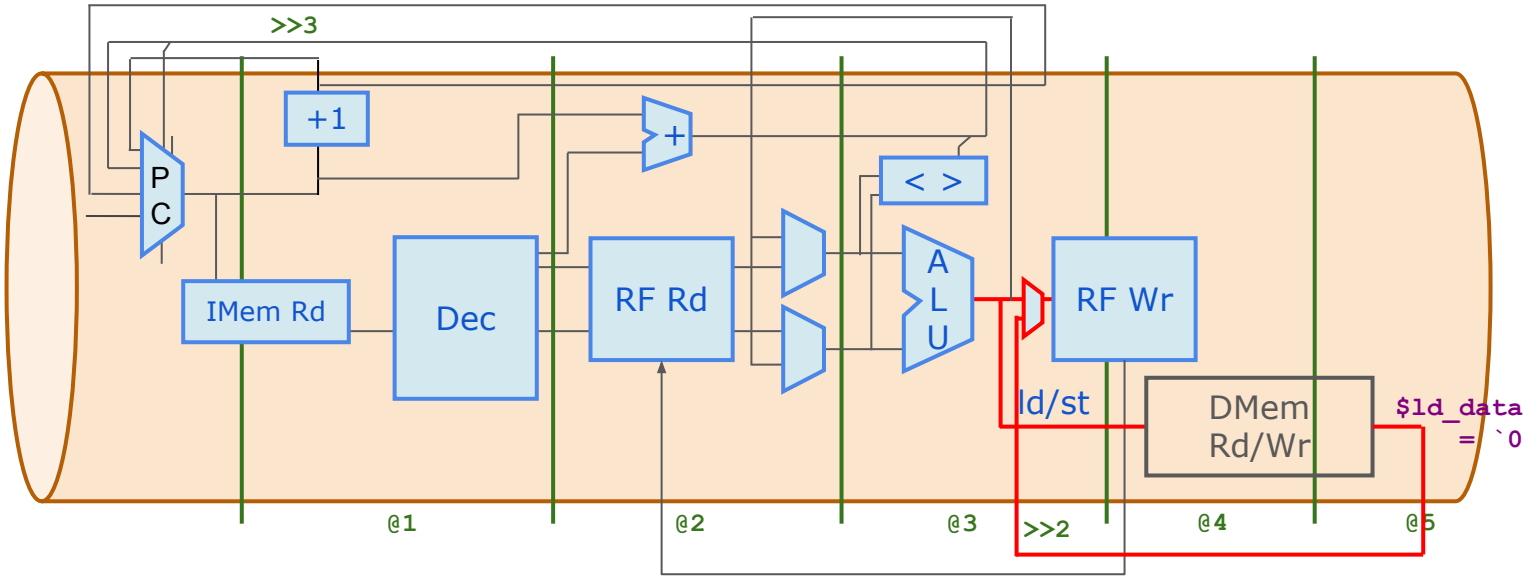


Lab: Redirect Loads



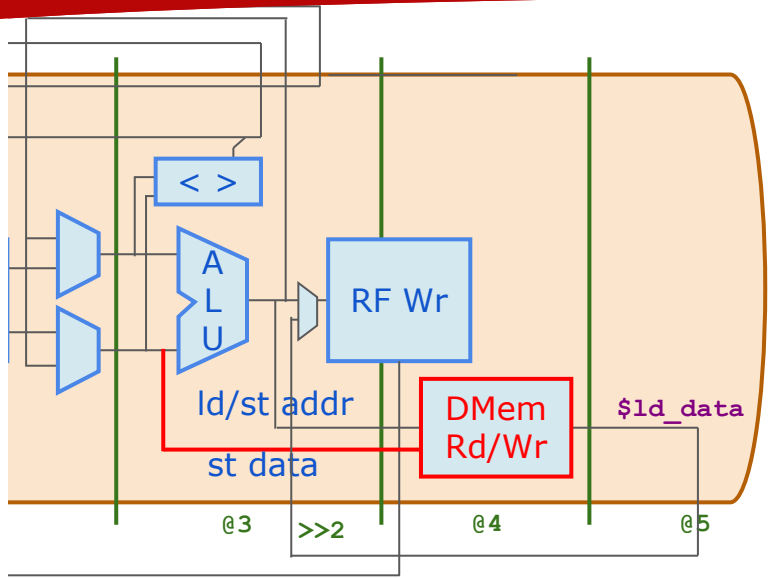
1. Clear **`$valid`** in the “shadow” of a load (like branch).
2. Select **`$inc_pc`** from 3 instructions ago for load redirect.
3. Debug. Confirm save.

Lab: Load Data

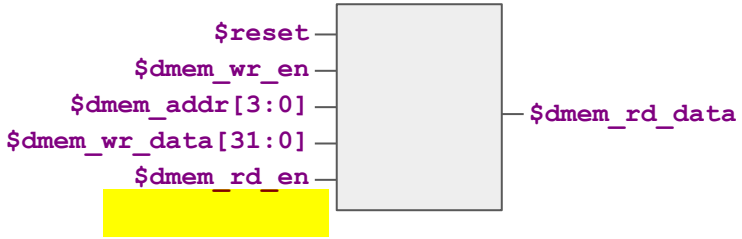


1. For loads/stores (**`$is_load`**/**`$is_s_instr`**), compute same result as for `addi`.
2. Add the RF-wr-data MUX to select **`>>2$ld_data`** and **`>>2$rd`** as RF inputs for **`!$valid`** instructions.
3. Enable write of **`$ld_data`** 2 instructions after valid **`$load`**.
4. Confirm save.

Lab: Load Data



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



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

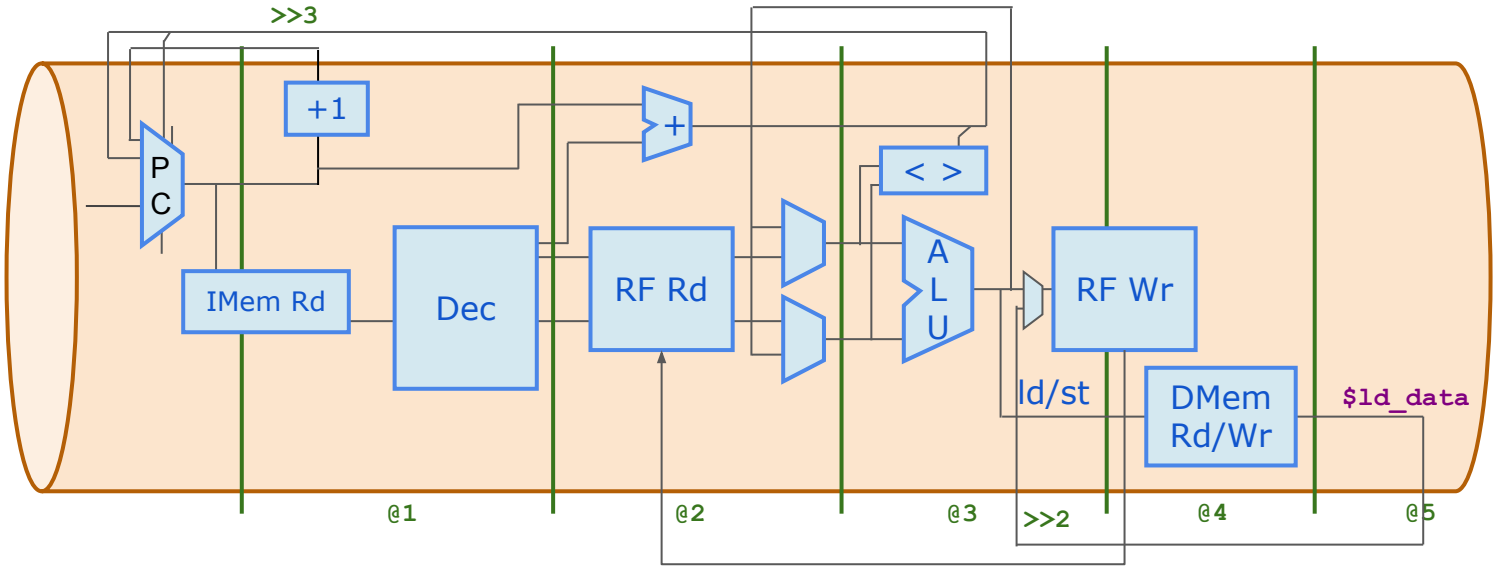
Modify the test program to store the final result value to byte address 16, then load it into x17.

```
m4_asm(SW, r0, r10, 10000)
m4_asm(LW, r17, r0, 10000)
```

Update passing condition to look in `xreg[17]`.

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

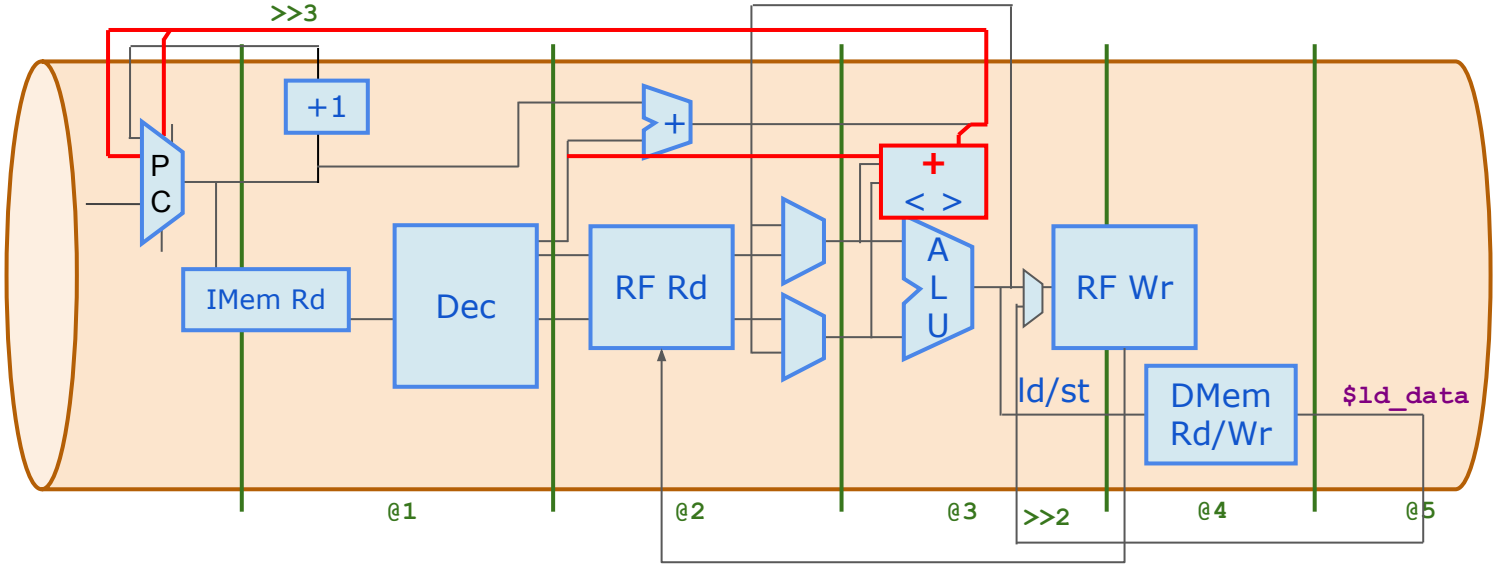
Jumps



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

JALR: Jumps to SRC1 + IMM.

Lab: Jumps

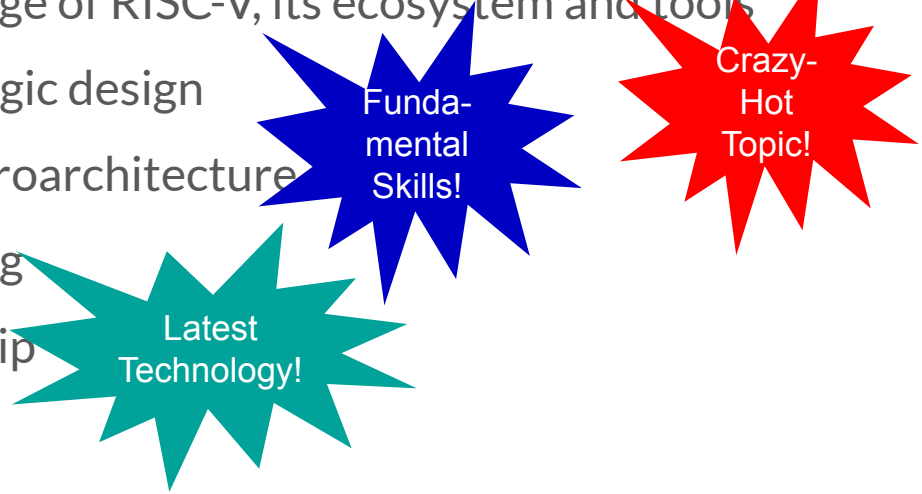


1. Define **`$is_jump`** (JAL or JALR), and, like **`$taken_br`**, create invalid cycles.
2. Compute **`$jalr_tgt_pc`** (SRC1 + IMM).
3. Select correct **`$pc`** for JAL (**`>>3$br_tgt_pc`**) and JALR (**`>>3$jalr_tgt_pc`**).
4. Save.

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… <u>you are leading it</u>!!!

Final Steps

- Submit your work for certification
- Brag about your accomplishments