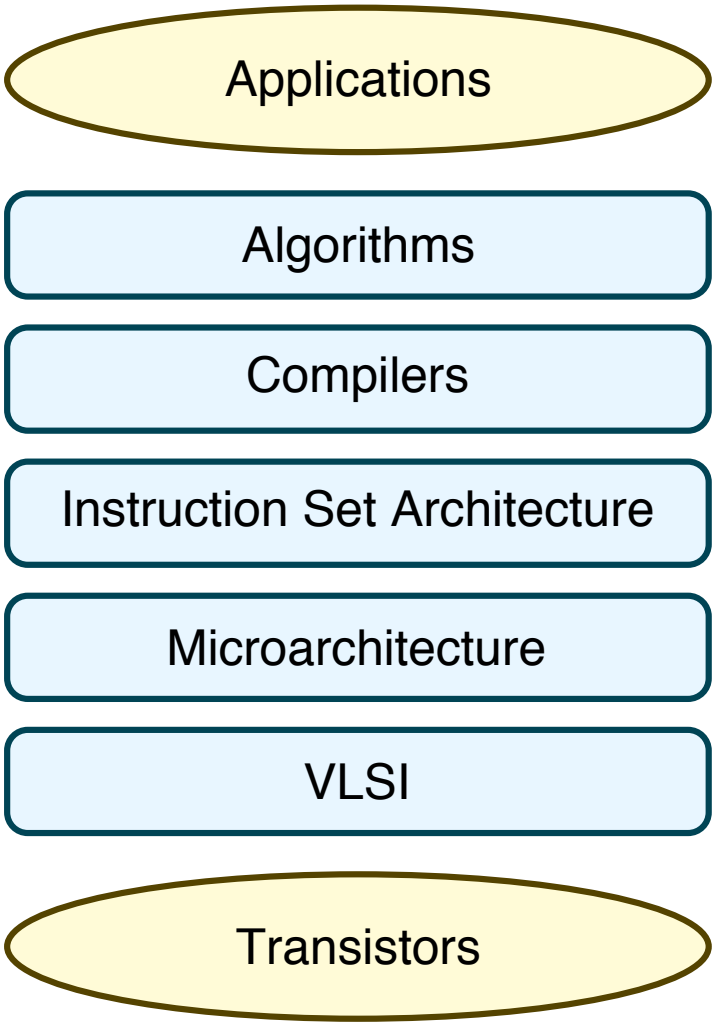
PyMTL Tutorial Schedule

1:45 – 2:00pm	Virtual Machine Installation and Setup
2:00 – 2:15pm	*Presentation:* PyMTL Overview
2:15 – 2:55pm	*Part 1:* PyMTL Basics
2:55 – 3:30pm	*Part 2:* Multi-Level Modeling with PyMTL
3:30 – 4:00pm	Coffee Break
4:00 – 4:45pm	*Part 3:* Processor Modeling with PyMTL
4:45 – 5:30pm	*Part 4:* Multi-Level Composition in PyMTL

State-of-the-Art in Multi-Level Processor Modeling



Processor FL Modeling
- ISA/SW Development
- C++ ISA Simulators
- Spike, gem5 atomic

Processor CL Modeling
- Design-Space Exploration
- C++ Simulation Framework
- SW-Focused Object-Oriented
- gem5, zsim, McPAT

Processor RTL Modeling
- Prototyping & AET Validation
- Verilog, VHDL, Chisel Languages
- HW-Focused Concurrent Structural
- Rocket, Ariane, OpenPiton

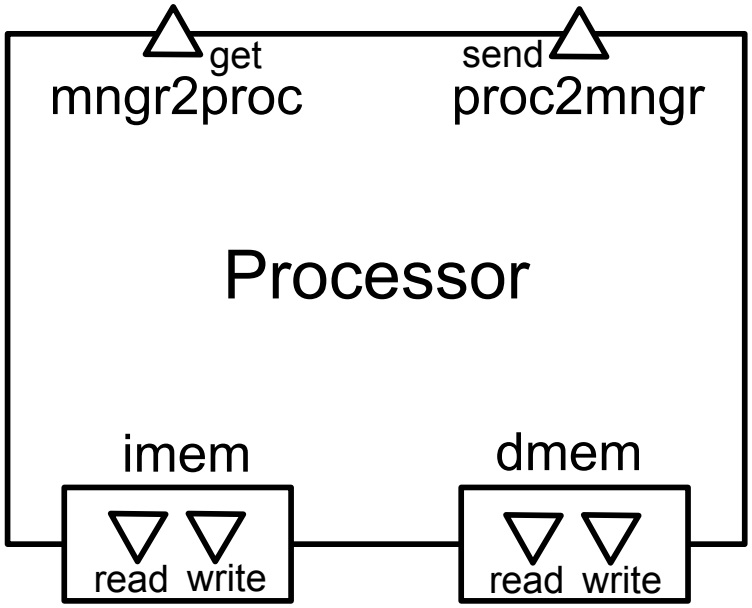
TinyRV0 Instruction Set Architecture (ISA)

In this tutorial, we are going to work with a TinyRV0 in-order processor. The TinyRV0 ISA is a slim (ten-instruction) version of RISC-V ISA.

Name	Syntax	Semantics
`CSRR`	`csrr rd, csr`	move value in control/status register to GPR
`CSRW`	`csrw csr, rs1`	move value in GPR to control/status register
`ADD`	`add rd, rs1, rs2`	addition with 3 GPRs, no overflow exception
`AND`	`and rd, rs1, rs2`	bitwise logical AND with 3 GPRs
`SLL`	`sll rd, rs1, rs2`	shift left logical by register value (append zeroes)
`SRL`	`srl rd, rs1, rs2`	shift right logical by register value (append zeroes)
`ADDI`	`addi rd, rs1, imm`	add constant, no overflow exception
`LW`	`lw rd, imm(rs1)`	load word from memory
`SW`	`sw rs2, imm(rs1)`	store word into memory
`BNE`	`bne rs1, rs2, imm`	branch if 2 GPRs are not equal

`https://tiny.cc/rv0`

Functional-Level Processor: Interface




```
1  class ProcFL( Component ):
2    def construct( s ):
3      s.imem = MemMasterIfcFL()
4      s.dmem = MemMasterIfcFL()
5
6      s.proc2mngr = SendIfcFL()
7      s.mngr2proc = GetIfcFL()
```

```
1  class MemMasterIfcFL( Interface ):
2    def construct( s ):
3      s.read  = CallerPort()
4      s.write = CallerPort()
```

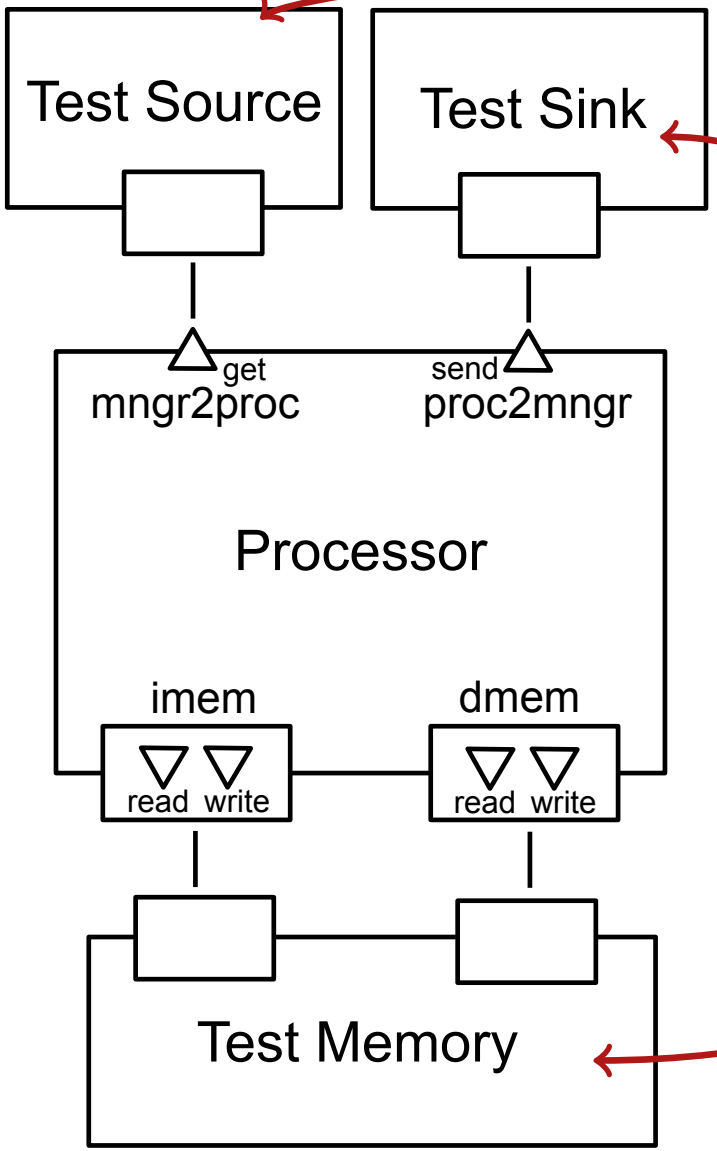
FL Processor: Implementation of ADD, LW, BNE

```
1  @s.update
2  def upblk_fl_proc():
3    ...
4    s.raw_inst = s.imem.read( s.PC, 4 )
5    inst       = TinyRV0Inst( s.raw_inst )
6    inst_name  = inst.name
7    ...
8    elif inst_name == "add":
9      s.R[inst.rd] = s.R[inst.rs1] + s.R[inst.rs2]
10     s.PC += 4

11   elif inst_name == "lw":
12     addr = s.R[inst.rs1] + sext( inst.i_imm, 32 )
13     s.R[inst.rd] = s.dmem.read( addr, 4 )
14     s.PC += 4

15   elif inst_name == "bne":
16     if s.R[inst.rs1] != s.R[inst.rs2]:
17       s.PC = s.PC + sext( inst.b_imm, 32 )
18     else:
19       s.PC += 4
```

How Assembly Tests Work




```
1  def gen_add_basic_test():
2    return """
3      csrr x1, mngr2proc < 5
4      csrr x2, mngr2proc < 4
5      add x3, x1, x2
6      csrw proc2mngr, x3 > 9
7    """
8  ...
9  asm_snippet = gen_add_basic_test()
10
11 from tinrv0_encoding import assemble
12 image = assemble( asm_snippet )
13 test_harness.load( image )
```

How Assembly Tests Work (Cont'd)

- Line tracing output of the FL processor running `add_basic_test`

```
% pytest ProcFL_test.py -k add_basic_test -s

0: 00000005 > [#                                ] > .
1: #        > [00000204 csrr   x01, 0xfc0       ] > .
2: 00000004 > [#                                ] > .
3: #        > [00000208 csrr   x02, 0xfc0       ] > .
4:          > [#                                ] > .
5:          > [0000020c add    x03, x01, x02    ] > .
6:          > [#                                ] > .
7:          > [00000210 csrw   0x7c0, x03       ] > 00000009
8:          > [#                                ] > .
```

- Note that the common strategy is to use FL models as a golden reference model to help develop tests for CL/RTL models.

★ Task 3.1: Implement & Test AND in FL Processor ★

```
% cd $HOME/pymtl-tut/examples/build
% pytest ../ex03_proc/test/ProcFL_test.py -v
% geany ../ex03_proc/ProcFL.py
```

```
73      ...
74      elif inst_name == "and":
75        s.R[inst.rd] = s.R[inst.rs1] & s.R[inst.rs2]
76        s.PC += 4
77      ...
```

```
% geany ../ex03_proc/test/inst_and.py
```

```
14  def gen_and_basic_test():
15    return """
16      csrr x1, mngr2proc < 0x0f0f0f0f
17      csrr x2, mngr2proc < 0x00ff00ff
18      and x3, x1, x2
19      csrw proc2mngr, x3 > 0x000f000f
20    """
```

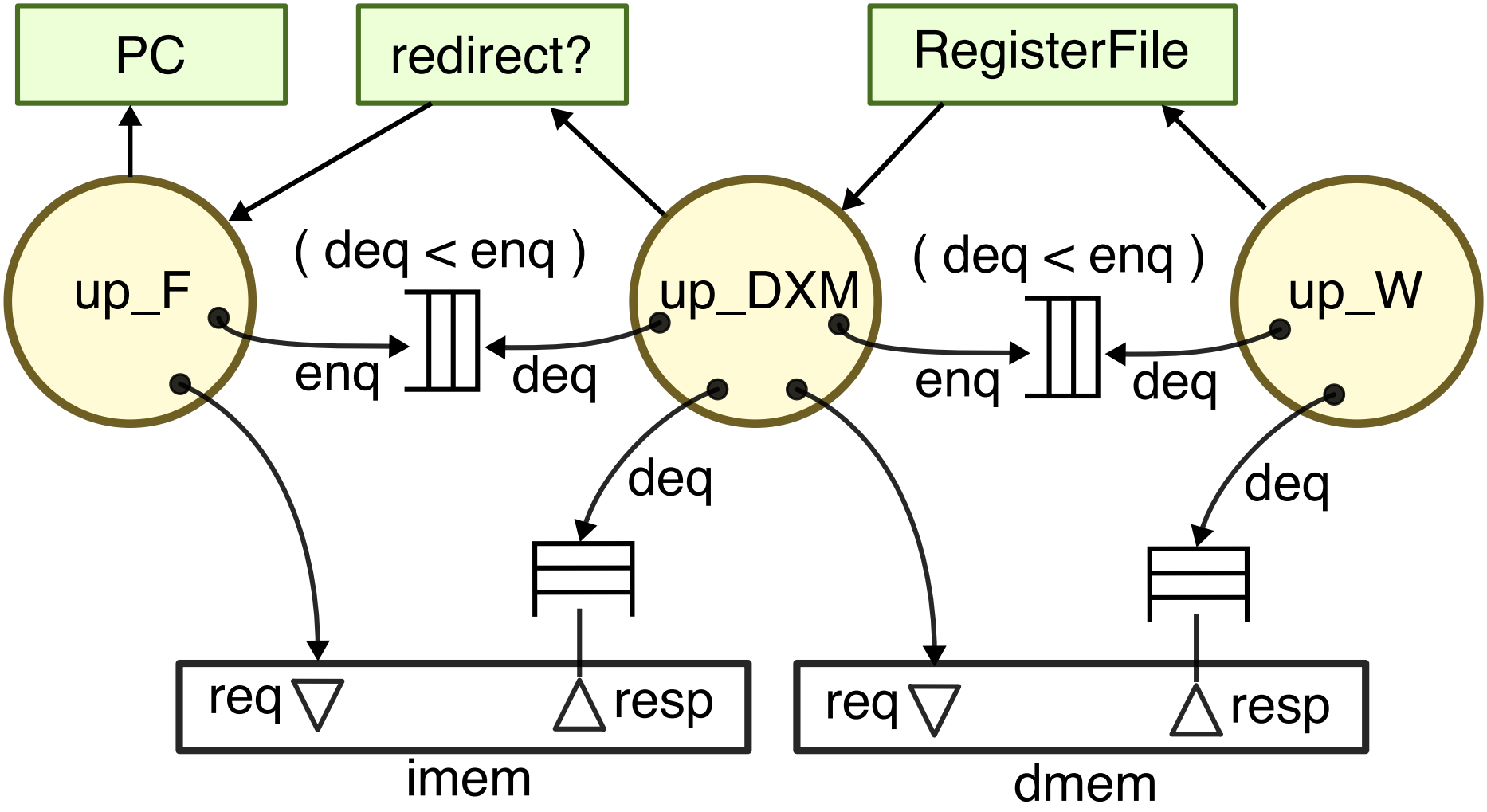
```
% pytest ../ex03_proc/test/ProcFL_test.py -svk and_basic
```

Cycle-Level Processor: Interface

```
1 class ProcCL( Component ):
2   def construct( s ):
3     s.imem = MemMasterIfcCL( mk_mem_msg(8,32,32) )
4     s.dmem = MemMasterIfcCL( mk_mem_msg(8,32,32) )
5
6     s.proc2mngr = NonBlockingCallerIfc()
7     s.mngr2proc = NonBlockingCalleeIfc()
```

```
1 class MemMasterIfcCL( Interface ):
2   def construct( s, req_class, resp_class, resp, resp_rdy ):
3     s.req_class  = req_class
4     s.resp_class = resp_class
5     s.req  = NonBlockingCallerIfc( req_class )
6     s.resp = NonBlockingCalleeIfc( resp_class, resp, resp_rdy )
```

Block Diagram of CL Procecssor



CL Processor Fetch Implementation

```
1  @s.update
2  def up_F():
3    if s.imem.req.rdy() and s.F_DXM_queue.enq.rdy():
4      if s.redirected_pc_DXM >= 0:
5        req = MemReqClass( MemMsgType.READ, 0, s.redirected_pc_DXM )
6        s.imem.req( req )
7        s.pc = s.redirected_pc_DXM
8      else:
9        req = MemReqClass( MemMsgType.READ, 0, s.pc )
10       s.imem.req( req )
11
12     s.F_DXM_queue.enq( s.pc )
13     s.F_status = PipelineStatus.work
14     s.pc += 4
15   else:
16     s.F_status = PipelineStatus.stall
```

Line Trace of The CL Processor

- Line tracing output of the FL processor running `add_basic_test`

```
% pytest ProcCL_test.py -k add_basic_test -s

0: 00000005 > [00000204|                          |   ] > .
1: 00000004 > [00000208|csrr   x01, 0xfc0         |   ] > .
2:          > [0000020c|csrr   x02, 0xfc0         |x01] > .
3:          > [00000210|add    x03, x01, x02      |x02] > .
4:          > [00000214|csrw   0x7c0, x03         |x03] > .
5:          > [00000218|                          |x--] > 00000009
```

★ Task 3.2: Implement AND in CL Processor ★

```
% cd $HOME/pymtl-tut/examples/build
% geany ../ex03_proc/ProcCL.py
```

```
130  ...
131    elif inst_name == "and":
132      result = s.R[inst.rs1] & s.R[inst.rs2]
133      s.DXM_W_queue.enq( (inst.rd, result, DXM_W.arith) )
134  ...
```

- We reuse the test we developed previously to test the CL processor

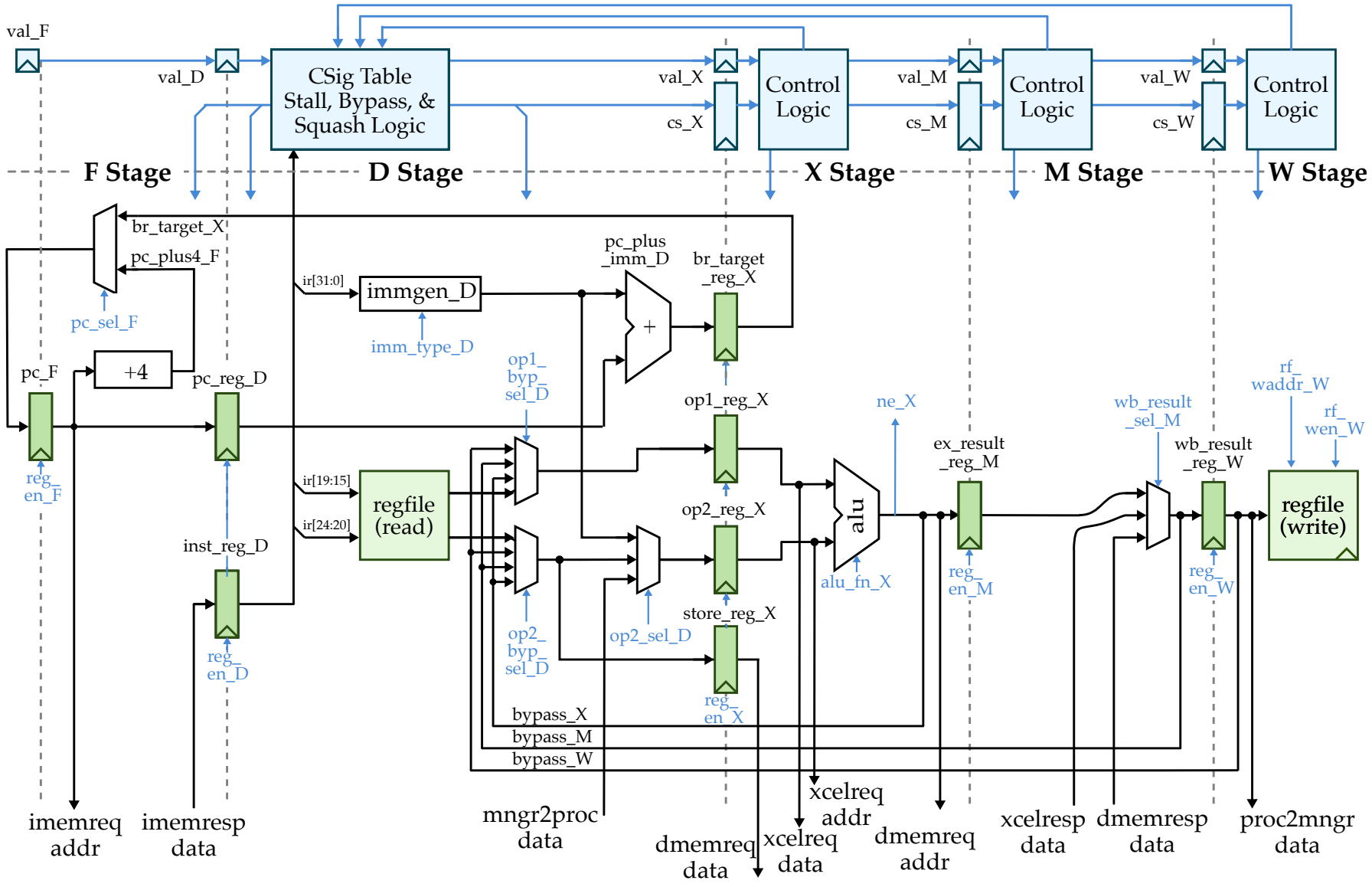
```
% pytest ../ex03_proc/test/ProcCL_test.py -svk and_basic
```

Register-Transfer Level Processor: Interface

```
1 class ProcRTL( Component ):
2   def construct( s ):
3     s.imem = MemMasterIfcRTL( mk_mem_msg(8,32,32) )
4     s.dmem = MemMasterIfcRTL( mk_mem_msg(8,32,32) )
5
6     s.mngr2proc = RecvIfcRTL( Bits32 )
7     s.proc2mngr = SendIfcRTL( Bits32 )
```

```
1 class MemMasterIfcRTL( Interface ):
2   def construct( s, req_class, resp_class ):
3     s.req_class  = req_class
4     s.resp_class = resp_class
5     s.req  = SendIfcRTL( req_class  )
6     s.resp = RecvIfcRTL( resp_class )
```

RTL Processor: Data Path



RTL Processor: Control Unit

- Control unit uses a “table” to set the control signals for each instruction

```
320  @s.update
321  def comb_control_table_D():
322    inst = s.inst_type_decoder_D.out
323    #                                      br     rs1 imm    op2    rs2 alu      dmm wbmux rf  cs cs
324    #                                 val type    en type   muxsel  en fn       typ sel   wen rr rw
325    if   inst == NOP  : s.cs = concat( y, br_na,  n, imm_x, bm_x,   n, alu_x,   nr, wm_a, n,  n, n )
326    elif inst == CSRRX: s.cs = concat( y, br_na,  n, imm_i, bm_imm, n, alu_cp1, nr, wm_c, y,  y, n )
327    elif inst == CSRR : s.cs = concat( y, br_na,  n, imm_i, bm_csr, n, alu_cp1, nr, wm_a, y,  y, n )
328    elif inst == CSRW : s.cs = concat( y, br_na,  y, imm_i, bm_imm, n, alu_cp0, nr, wm_a, n,  n, y )
329    elif inst == ADD  : s.cs = concat( y, br_na,  y, imm_x, bm_rf,  y, alu_add, nr, wm_a, y,  n, n )
330    elif inst == SLL  : s.cs = concat( y, br_na,  y, imm_x, bm_rf,  y, alu_sll, nr, wm_a, y,  n, n )
331    elif inst == SRL  : s.cs = concat( y, br_na,  y, imm_x, bm_rf,  y, alu_srl, nr, wm_a, y,  n, n )
332    elif inst == ADDI : s.cs = concat( y, br_na,  y, imm_i, bm_imm, n, alu_add, nr, wm_a, y,  n, n )
333    elif inst == LW   : s.cs = concat( y, br_na,  y, imm_i, bm_imm, n, alu_add, ld, wm_m, y,  n, n )
334    elif inst == SW   : s.cs = concat( y, br_na,  y, imm_s, bm_imm, y, alu_add, st, wm_m, n,  n, n )
335    elif inst == BNE  : s.cs = concat( y, br_ne,  y, imm_b, bm_rf,  y, alu_x,   nr, wm_x, n,  n, n )
336    else:               s.cs = concat( n, br_x,   n, imm_x, bm_x,   n, alu_x,   nr, wm_x, n,  n, n )
```

★ Task 3.3: Implement AND in RTL Processor ★

```
% cd $HOME/pymtl-tut/examples/build
% geany ../ex03_proc/ProcCtrlRTL.py
```

- Implement this in a single line to make the table look pretty!

```
341 ...
342   elif inst == AND  : s.cs = concat( y, br_na,  y, imm_x, bm_rf, y,
343                                      alu_and, nr, wm_a, y,  n, n )
344 ...
```

- Edit the ALU so it supports logical AND operations

```
% geany ../ex03_proc/MiscRTL.py
```

```
142   ...
143   elif s.fn == b4(5): s.out = s.in0 & s.in1
144   ...
```

```
% pytest ../ex03_proc/test/ProcRTL_test.py -svk and_basic
```

★ Task 3.4: Running Checksum Microbenchmark ★

- Take a look at the microbenchmark assembly

```
% cd $HOME/pymtl-tut/examples/build
% geany ../ex03_proc/ubmark/proc_ubmark_cksum_roll.py
```

```
lw    x12, 0(x2)          # load 2 16-bit ints
and   x13, x12, x6        # src[i+j] = word & 0xffff
srl   x12, x12, x5        # src[i+j+1] = word >> 16
add   x7,  x7,  x13       # sum1 += src[i+j]
and   x7,  x7,  x6        # sum1 &= 0xffff
add   x8,  x8,  x7        # sum2 += sum1
and   x8,  x8,  x6        # sum2 &= 0xffff
```

- Execute the microbenchmark on all three processor models

```
% ../ex03_proc/proc-sim --bmark cksum --impl fl  --trace
% ../ex03_proc/proc-sim --bmark cksum --impl cl  --trace
% ../ex03_proc/proc-sim --bmark cksum --impl rtl --trace
```

- How many cycles does this microbenchmark require? _____ cycles