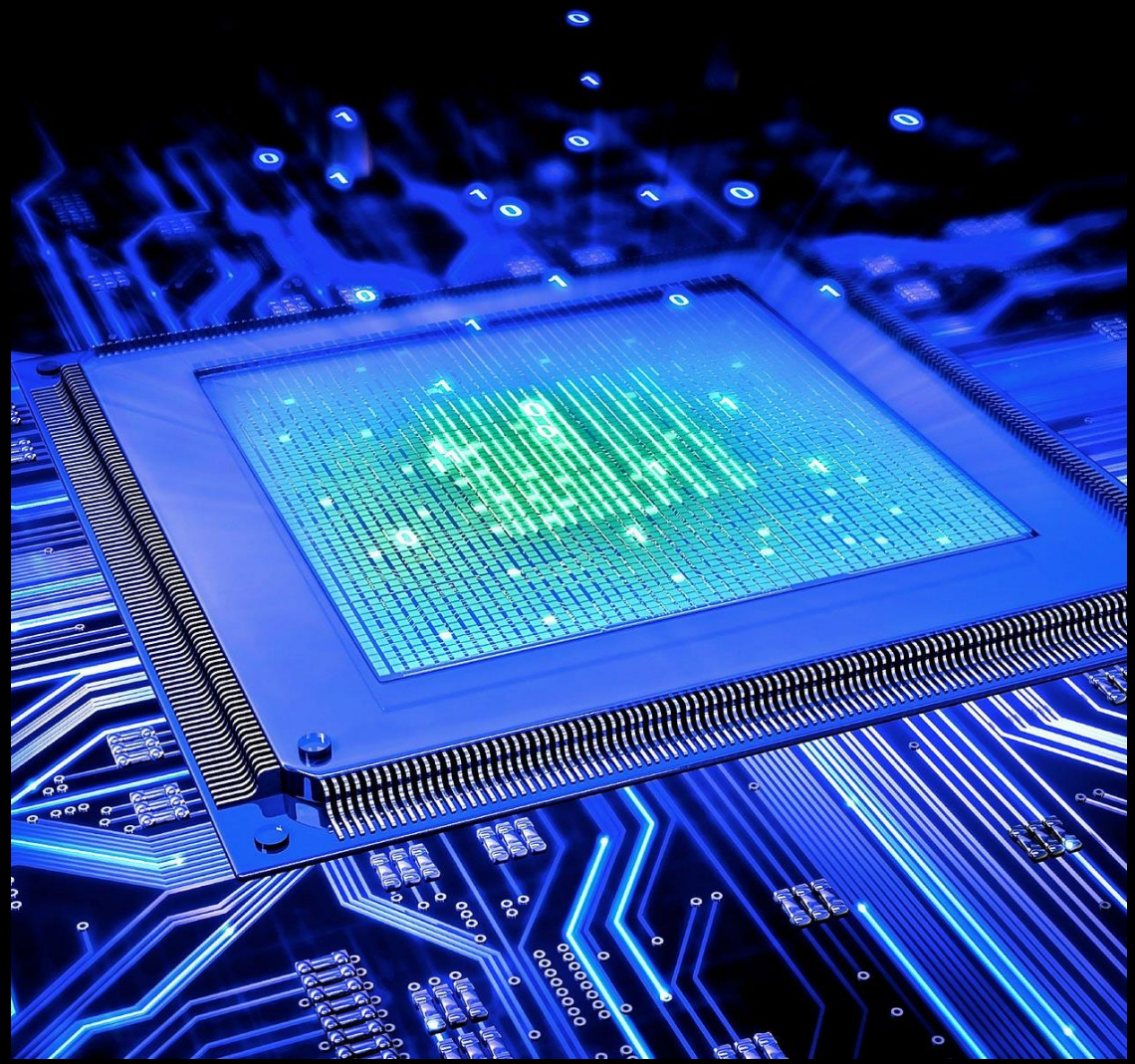
CSE3666- INTRODUCTION TO COMPUTER ARCHITECTURE

LAB 7

Immediate Generator



EXTRA SUPPORT

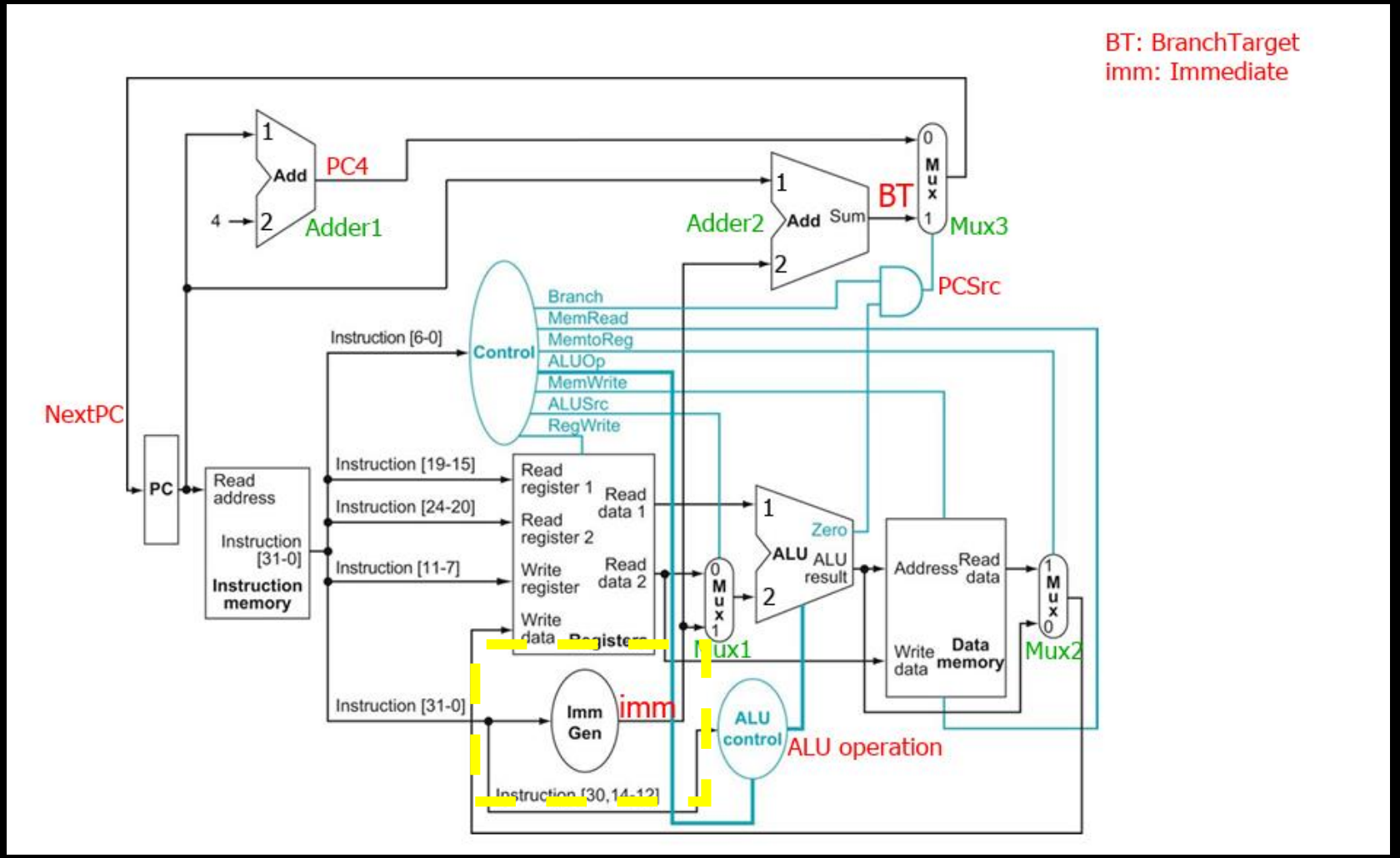
- Jacob.gerow@uconn.edu
- Find me in class discord
- Office Hours: 11:15am-12:15pm Wednesday
- BEACH (ITE 360) Drop-in Hours **Subject to change**:

  Tuesday 6:00pm-8:00pm

  Thursday 4:00pm-8:00pm

  Friday 6:00pm-8:00pm

Processor does not support all instructions



LAB 7 IMM-GEN INSTRUCTION SUPPORT

RISC-V Reference

RISC-V Instruction Set

Core Instruction Formats

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

RV32I Base Integer Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ^ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 | imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch ≥	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 >= rs2) PC += imm	zero-extends
jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm	
lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	
ecall	Environment Call	I	1110011	0x0	imm=0x0	Transfer control to OS	
ebreak	Environment Break	I	1110011	0x0	imm=0x1	Transfer control to debugger	

IMMEDIATE GENERATION

	31 25	24 20	19 15	14 12	11 7	6 0
R	funct7	rs2	rs1	funct3	rd	opcode
I	imm[11:0]		rs1	funct3	rd	opcode
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]				rd	opcode
UJ	imm[20|10:1|11|19:12]				rd	opcode

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	sB-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	uJ-immediate

STARTER CODE WALK-THROUGH

```
from myhdl import block, always_comb, intbv, concat, instances

"""
tag: CAF039043FBCCC351938AS25

The ImmGen module generates the 32-bit imm from an instruction word.

Supported instruction types:

    opcode      instructions

    000 0011    I, LW ...
    001 0011    I, ANDI, ORI, ...
    110 0111    I, JALR
    010 0011    S
    110 0011    SB
    110 1111    UJ, JAL
    011 0111    U, LUI

    See the paragraph at the end of Section 4.3 in textbook for comments about
    generating immediate for R, Load, Store, and branches.

"""
```

Comment shows you the different opcodes that need to be supported by this immediate generator.

Note I-type has 3 different Opcodes

Opcodes are used to decode what instruction type the instruction is

```python
@block
def ImmGen(imm, inst):

    # internal signals
    I, S, SB, U, UJ = [Signal(False) for _ in range(5)]

    @always_comb
    def  generate_instr_type():
        # only use the higher 5 bits in opcode
        # a is inst[6], b is inst[5], and so on
        a, b, c, d, e = inst[7:2]

        # If you are meticulous, you can add "and inst[1] and inst[0]"
        # to each term

        # As an example, S is generated as follows.
        # 010 0011    S
        S.next = not a and b and not c and not d and not e

        # TODO
        # Generate I, SB, UJ, and U below

        # I
        # 000 0011    I, LW ...
        # 001 0011    I, ANDI, ORI, ...
        # 110 0111    I, JALR


    @always_comb
    def  set_imm():
        # logic expression for imm[0]
        imm.next[ 0] = I and inst[20] or S and inst[7]

        # TODO
        # Generate imm[1] to imm[30]


        # imm[31] is always inst[31]
        imm.next[31] = inst[31]

    return instances()
```

- Your code here
- **Combinational Design**
  - Signals are set as logic equations **NOT** with if statements or any other **behavioral** logic
- Opcode bit 1 & 0 are always 1 (so we ignore them)

The instruction is a store instruction when the opcode top 5 bits are 01000 since the opcode for a store instruction is 0100011

Repeat for other instruction types here and then we have a way to identify what type of instruction is coming into the processor

The 0^{th} bit of immediate should be a 1 if it is an I type instruction and inst[20] is a 1 or when it is an S type instruction and inst[7] is a 1

Repeat for next 30 bits

```python
# do not need to change any lines below
@block
def ImmGen0(imm, instruction):
    """Generate imm from instruction.

    Behavior description.
    """

    @always_comb
    def comb_logic():

        # 12-bit imm
        imm12 = intbv(0)[12:]
        # 20 sign bits, all 0 or all 1
        sign20 = intbv(0-instruction[31])[20:]

        is_imm12 = True
        i32 = 0             # Python int type

        opcode = instruction[7:]
        if opcode in [0x03, 0x13, 0x67]: # I
            imm12 = instruction[32:20]
        elif opcode == 0x23: # S
            imm12 = concat(instruction[32:25], instruction[12:7])
        elif opcode == 0x63: # SB
            imm12 = concat(instruction[7], instruction[31:25], instruction[12:8], bool(0))
        elif opcode == 0x6F: # UJ
            is_imm12 = False
            i32 = concat(sign20[12:], instruction[20:12], instruction[20],
                         instruction[31:25], instruction[25:21], bool(0))
        else:  # U
            is_imm12 = False
            i32 = instruction[32:12] << 12

        # set the imm signal
        if is_imm12:
            imm.next = concat(sign20, imm12)
        else:
            imm.next = i32

    return comb_logic
```

- Professor's implementation of a behavioral version of the **SAME** immediate generator
- You do not need to do anything here
- In practice 2 or more versions (usually also completed by 2 or more separate engineers) of the same design are often compared against each other to ensure they are both functioning correctly

```python
if __name__ == "__main__":
    from myhdl import intbv, delay, instance, Signal, StopSimulation, bin
    import argparse

    # testbench itself is a block
    @block
    def test_comb(args):

        instruction = Signal(intbv(0x33)[32:])
        imm_a = Signal(intbv(0)[32:])
        imm_b = Signal(intbv(0)[32:])

        # instantiating a block
        dut_a = ImmGen0(imm_a, instruction)
        dut_b = ImmGen(imm_b, instruction)

        @instance
        def stimulus():

            for i in args.instruction:
                print(i)

                # set the input
                instruction.next = int(i, 0)

                # wait
                yield delay(10)

                # compare the results
                print(bin(imm_a, 32))
                print(bin(imm_b, 32))

                if imm_a != imm_b:
                    s = bin(imm_a ^ imm_b, 32)
                    print(s.replace("0", " ").replace("1", "^"))

            # stop simulation
            raise StopSimulation()

        return dut_a, dut_b, stimulus

    parser = argparse.ArgumentParser(description='Testing ImmGen. Generate imm from instruction word.')
    parser.add_argument('instruction', nargs="+", help='instruction word')
    parser.add_argument('--trace', action='store_true', help='generate trace')
    parser.add_argument('--verbose', '-v', action='store_true', help='verbose')

    args = parser.parse_args()
    if args.verbose:
        print(args)

    tb = test_comb(args)
    tb.config_sim(trace=args.trace)
    tb.run_sim()
```

- This is the test bench that feeds an instruction to both designs and compares the output

- Can run the program with:
  python imgen.py {instruction starting with 0x}
- Make sure it's a valid instruction or you might get two different results

- Alternatively you can use professors test file

IMPORTANT RESOURCES

Discord (must join, change name to first and last**, and set role):**
https://discord.gg/xE4fTGRS

Github Repo: https://github.uconn.edu/zhs04001/cse3666-2025spring

Lab Presentation Repo: https://github.com/Jacob-Gerow/CSE_3666