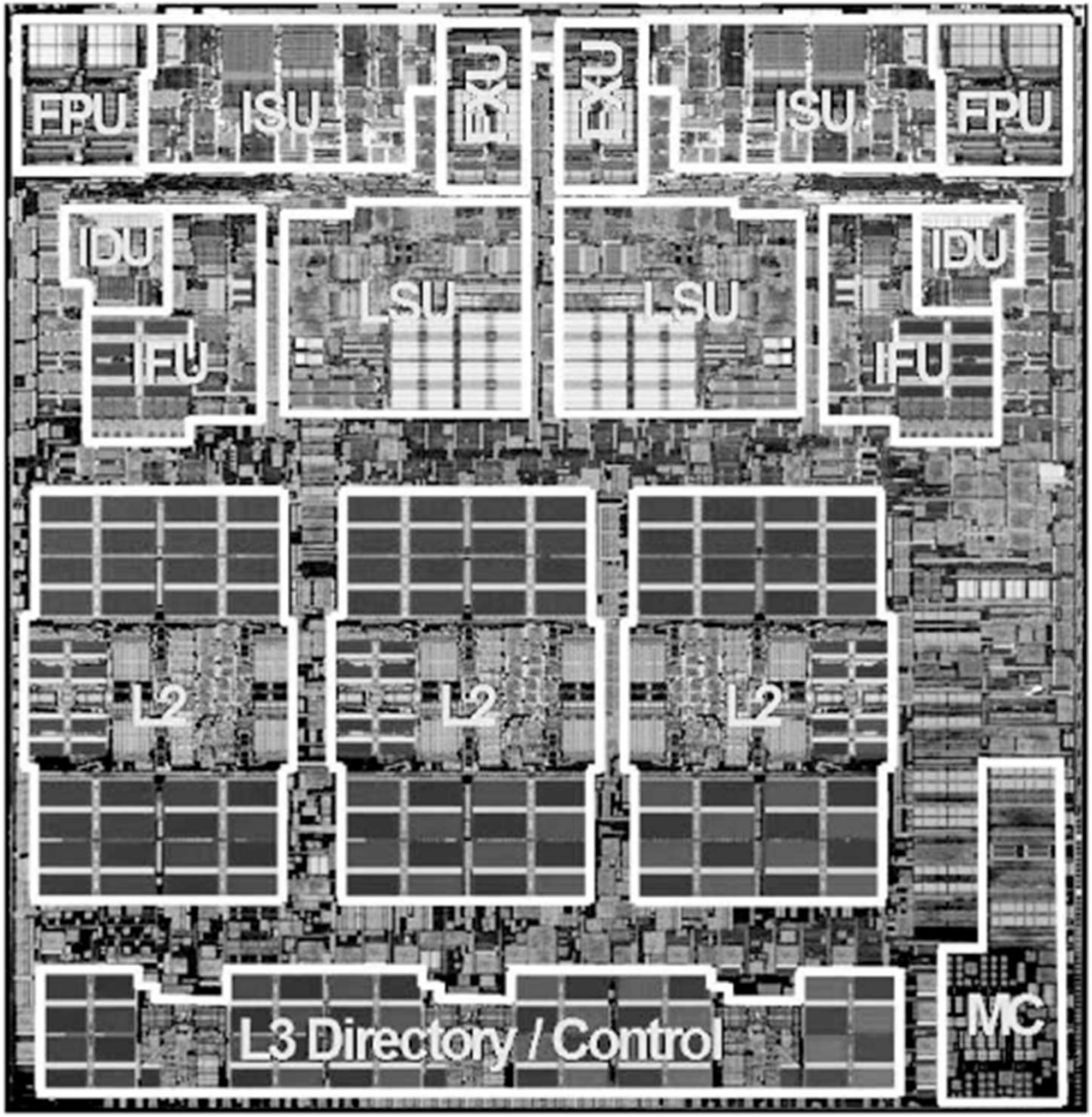
CPEG324
Computer System Design I

Instructor: Xiaoming Li

TA: Yulin Zhang

Goal



IBM Power 5

How CPUs work?

How to design and implement a CPU?

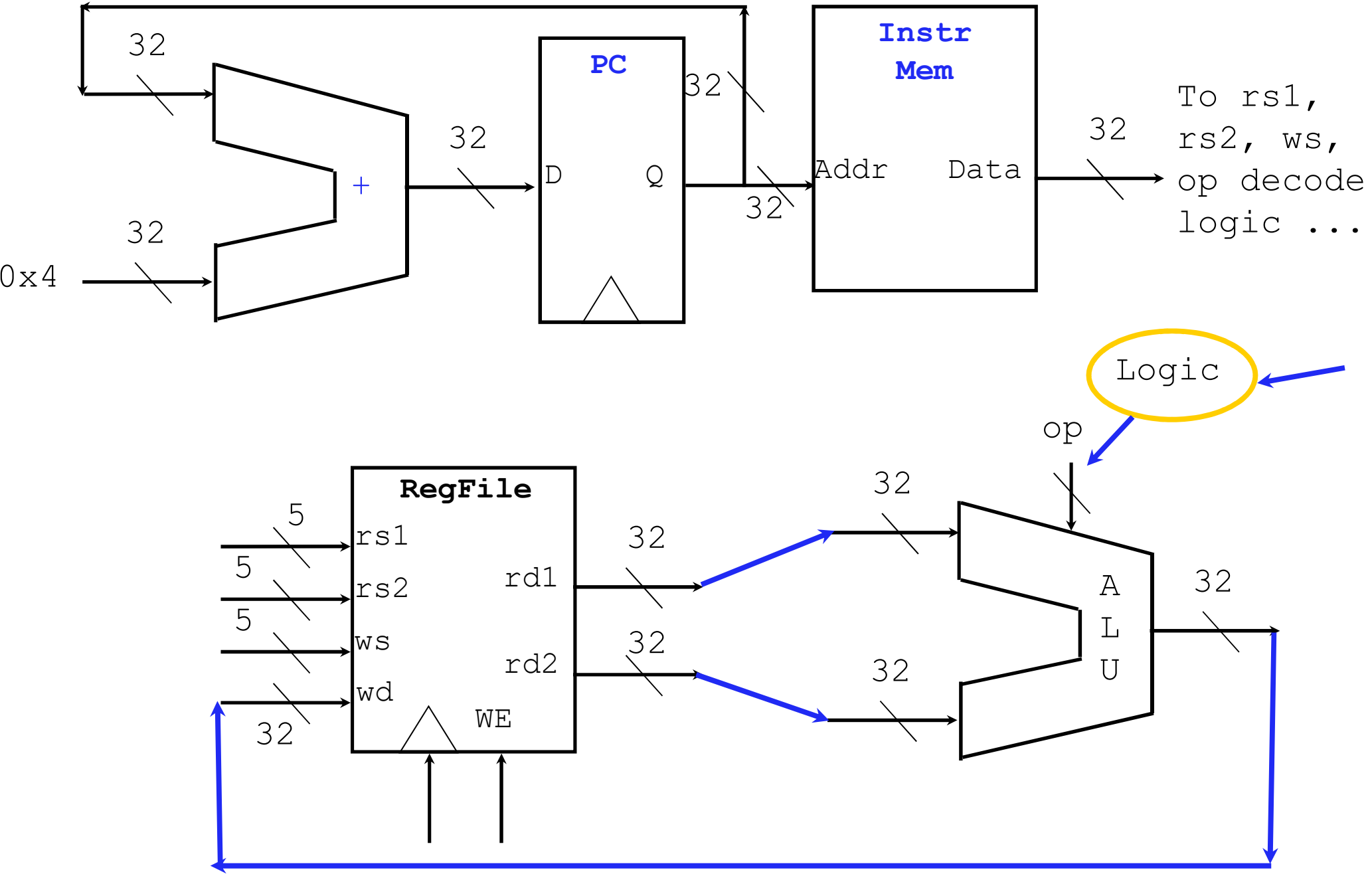
Preparation

- MIPS ISA
- Performance evaluation
- Single-cycle CPU
- Multi-cycle CPU

Goal 1: Designing a "CPU" interface

- Figure out which instructions are needed.
- Design the ISA
- Prepare you for follow-on lab projects.

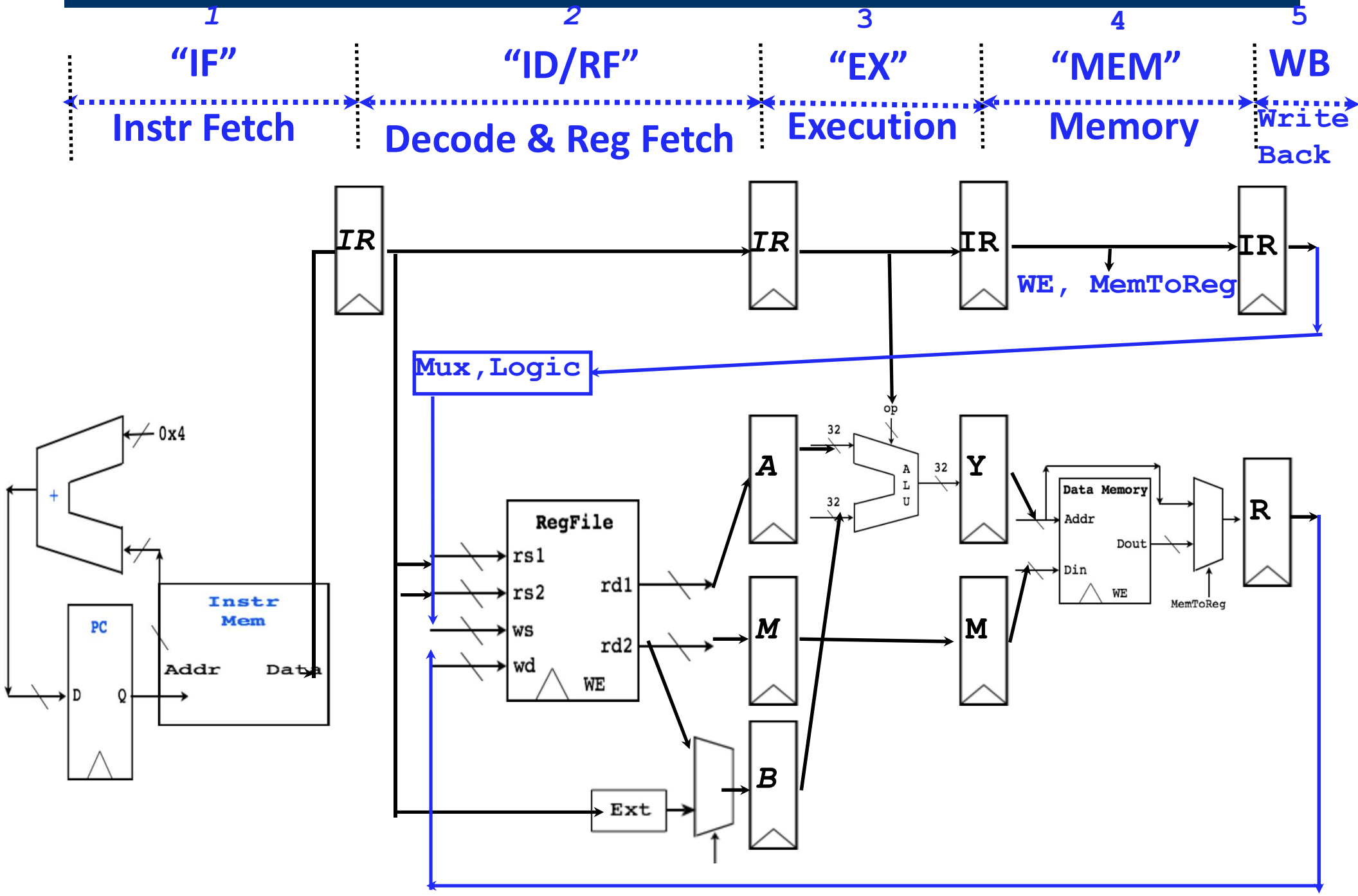
Goal 2: Single-Cycle CPU Simulator



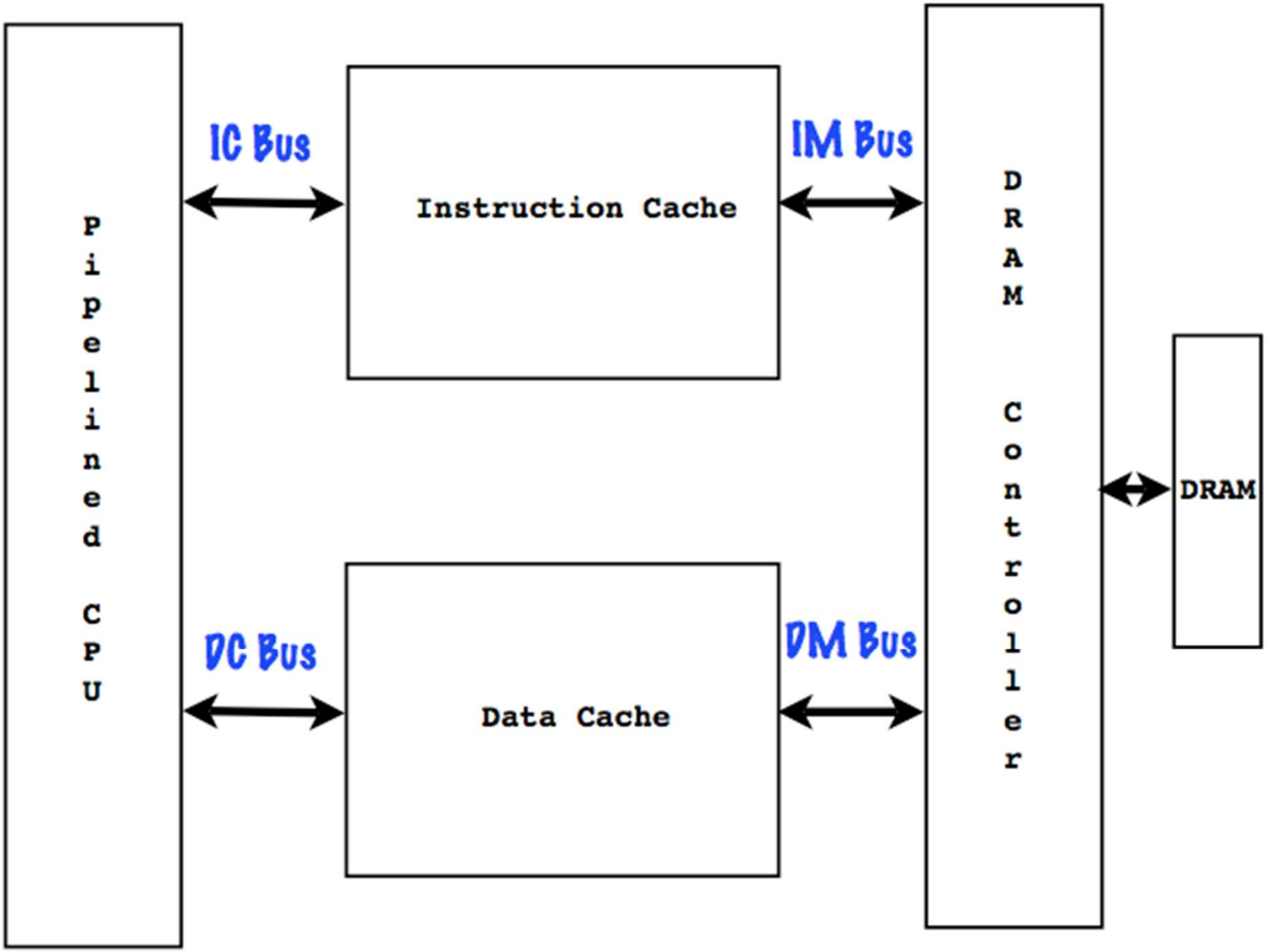
Goal 3: FPGA Implementation

- Working on the latest industry FPGA development kit.
- Understanding the difference between a FPGA simulator and a hardware board.
- Practice implementing basic FPGA functionalities.

Goal 4: Pipelined Processor Design



More ?: Caches + DRAM



This is a challenging course

- A crucial step for learning the computer architecture - designing one
- A rewarding experience:
  - How instructions are designed and implemented
    - Performance implication
  - Hand-on experience of using hardware design language
    - In this course, we use VHDL
    - FPGA

Admin

- Course website:
  - http://www1.udel.edu/canvas/
  - Check it regularly for announcements, slides, homework, …

Course work

- Homework
  - 2 homework
  - Total 10% of final grade
- Two exams
  - Each 10% of final grade
- Lab projects
  - The first programming project and the first FPGA project are individual projects
  - Other labs are team projects
    - 2 students
    - Individual teams need my approval. Good grades in CPEG323 and prior experience with VHDL or Verilog.
  - Project grades have two parts
    - Technical and presentation: 65% of final grade
    - Peer review: 5% of final grade

Peer review

- A grade from 1 to 5 for every other teammate for every cooperative lab project
- Explain the peer review grades you give
- How well a teammate fulfills his/her "social contract".
  - 100%: Meet your expectation of a good team member
  - Consider both effort and actual contribution

Lab

- All lab projects can be completed with freely available software.
  - You can use your personal computer
- Install Ubuntu on Windows machines.
- Try installing GHDL on your personal computer.

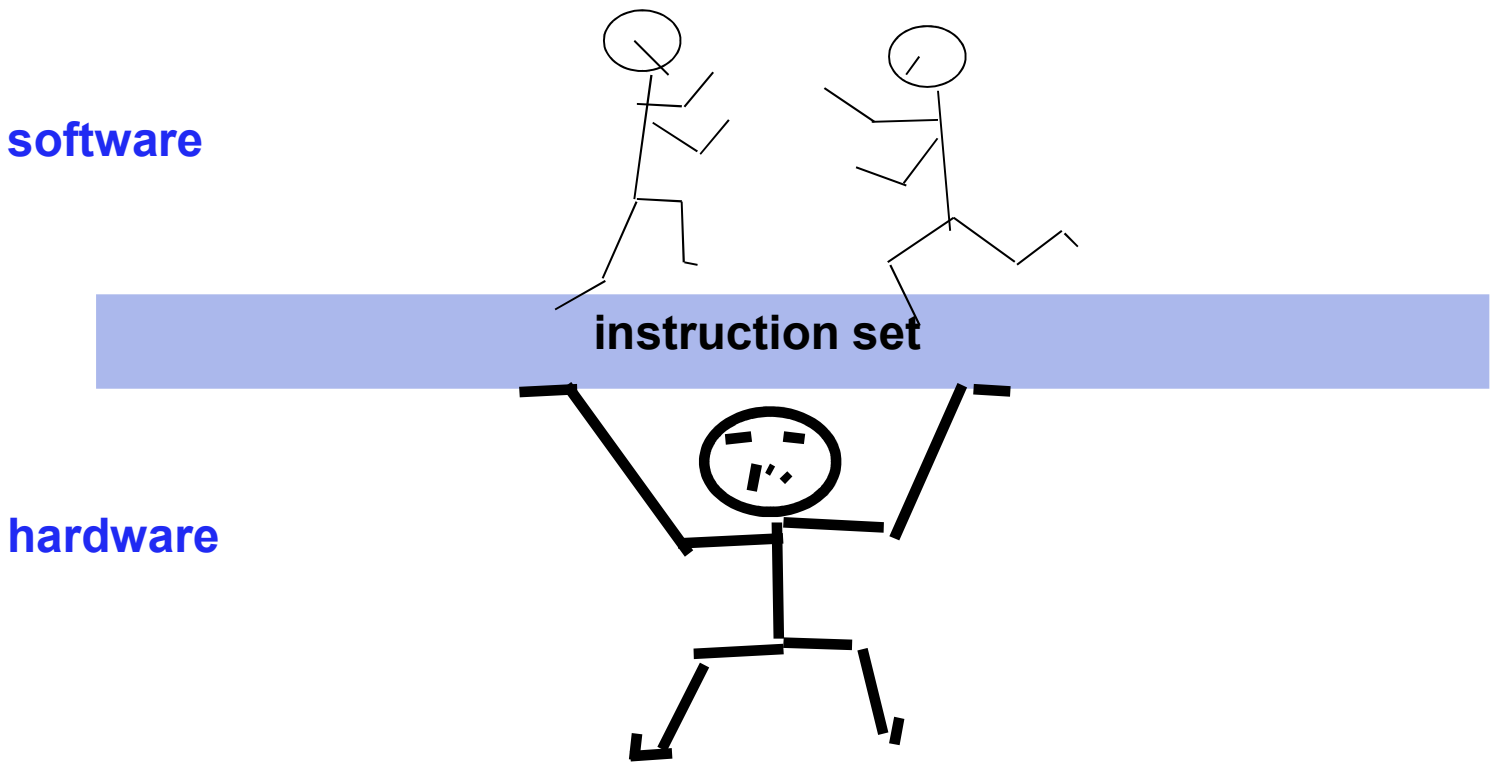
Late policy

- Following an hourly scheme. Strictly followed.
  - Up to 1 hour late, -15%
  - Up to 2 hours late, -40%
  - Up to 3 hours late, -70%
  - Zero grade after 3 hours.
- Use the submission timestamp on Canvas.
- 48 hours grace period quota for the individual projects and the homework.
  - Counted by hour
- 72 hours for the team projects.
- No other extensions will be granted.

Office hours

- Wed/Fri 2:30pm-3:30pm, and by appointment
- Lab Sessions
  - On demand

New successful instruction sets

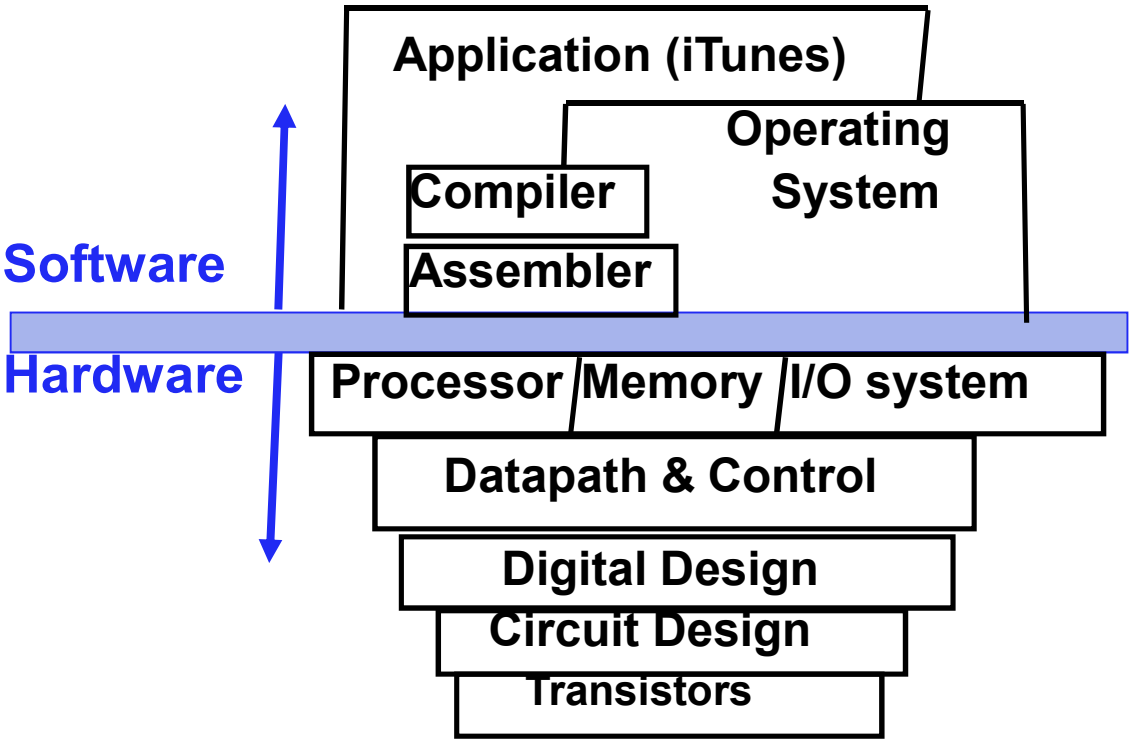


MIPS ISA

- MIPS memory model
  - data types and sizes
  - little vs. big endings
- Register conventions
- MIPS operations (3 types)
  - instruction format and fields
  - addressing mode
  - assembly vs. machine representation
- Calling conventions and stack frames
  - caller save vs. callee save
  - parameter passing

Instruction Sets: A Thin Interface

Syntax: `ADD $8 $9 $10` **Semantics:** `$8 = $9 + $10`



Instruction Set Architecture

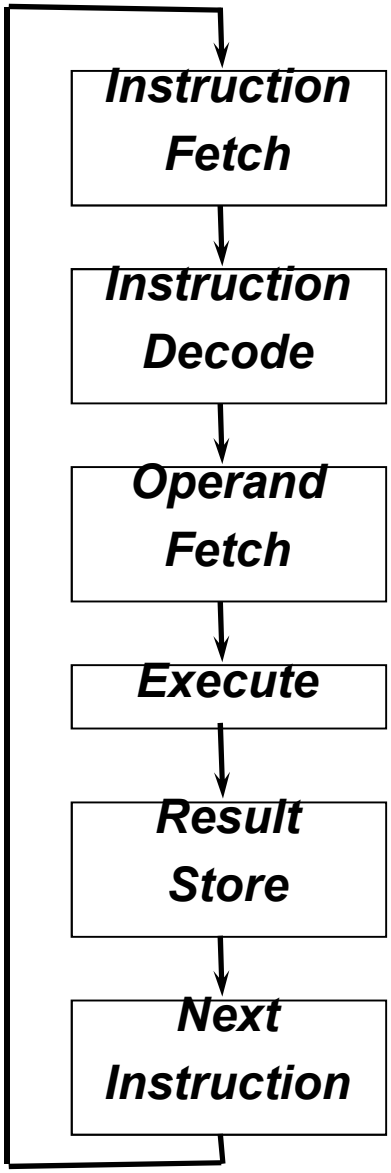
"R-Format"

Fieldsize:	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
Bitfield:	opcode	rs	rt	rd	shamt	funct
Binary:	000000	01001	01010	01000	00000	100000

In Hexadecimal: 012A4020

Hardware implements

Syntax: ADD $8 $9 $10 **Semantics:** $8 = $9 + $10



Fetch next inst from memory: 012A4020

opcode	rs	rt	rd	shamt	funct

Decode fields to get : ADD $8 $9 $10

"Retrieve" register values: $9 $10

Add $9 **to** $10

Place this sum in $8

Prepare to fetch instruction that follows the ADD in the program.

ADD

syntax & semantics, as seen in the MIPS ISA document.

Add Word **ADD**

Format: `ADD rd, rs, rt` **MIPS32**

Purpose:

To add 32-bit integers. If an overflow occurs, then trap.

Description: `rd ← rs + rt`

The 32-bit word value in GPR *rt* is added to the 32-bit value in GPR *rs* to produce a 32-bit result.

- If the addition results in 32-bit 2's complement arithmetic overflow, the destination register is not modified and an Integer Overflow exception occurs.
- If the addition does not overflow, the 32-bit result is placed into GPR *rd*.

Restrictions:

None

Operation:

```
temp ← (GPR[rs]31||GPR[rs]31..0) + (GPR[rt]31||GPR[rt]31..0)
if temp32 ≠ temp31 then
   SignalException(IntegerOverflow)
else
   GPR[rd] ← temp
endif
```

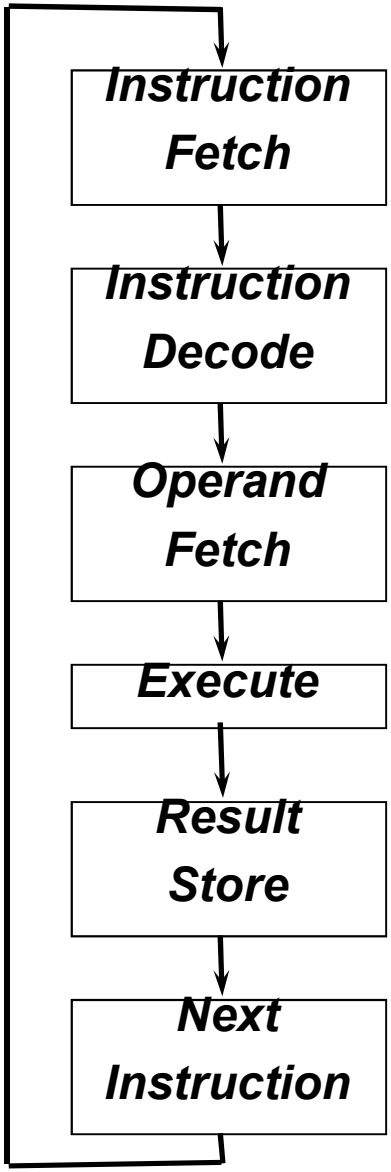
Exceptions:

Integer Overflow

Programming Notes:

ADDU performs the same arithmetic operation but does not trap on overflow.

Memory Instructions: LW



Fetch the load inst from memory

"I-Format"

Decode fields to get : LW $1, 32($2)

"Retrieve" register value: $2

Compute memory address: 32 + $2

Load memory address contents into: $1

Prepare to fetch instr that follows the LW **in the program. Depending on load semantics, new** $1 **is visible to that instr, or not until the following instr ("delayed loads").**

Load Word LW

Format: `LW rt, offset(base)` **MIPS32**

Purpose:

To load a word from memory as a signed value

Description: `rt ← memory[base+offset]`

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, sign-extended to the GPR register length if necessary, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

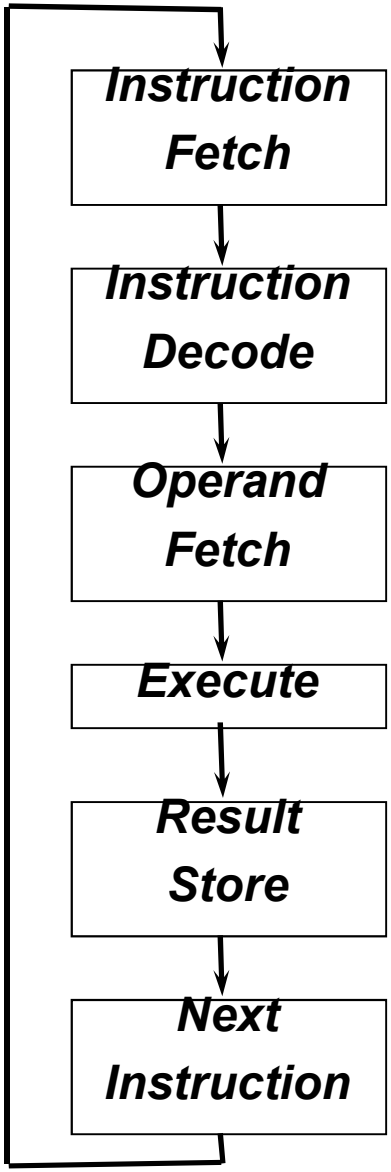
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
   SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
memword← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[rt]← memword
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error, Watch

Branch Instructions: BEQ



Fetch branch inst from memory

"I-Format"

Decode fields to get: BEQ $1, $2, 25

"Retrieve" register values: $1, $2

Compute if we take branch: $1 == $2 ?

ALWAYS prepare to fetch instr that follows the BEQ **in the program ("delayed branch"). IF we take branch, the instr we fetch AFTER that instruction is PC + 4 + 100.**

PC == "Program Counter"

Branch on Equal — BEQ

Format: `BEQ rs, rt, offset` **MIPS32**

Purpose:

To compare GPRs then do a PC-relative conditional branch

Description: `if rs = rt then branch`

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* and GPR *rt* are equal, branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Operation:

```
I:      target_offset ← sign_extend(offset || 0^2)
           condition ← (GPR[rs] = GPR[rt])
I+1:    if condition then
           PC ← PC + target_offset
        endif
```

Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 Kbytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

BEQ r0, r0 offset, expressed as B offset, is the assembly idiom used to denote an unconditional branch.

What is ISA?

- A contract between hardware design and software requirement
- To the program, it appears that instructions execute in the correct order defined by the ISA.
- As each instruction completes, the machine state (regs, mem) appears to the program to obey the ISA.
- What the machine actually does is up to the hardware designers, as long as the contract is kept.

Lab 0: Testing the MIPS contract

- Find out 3 of total 16 instructions that are buggy.
  - The 3 buggy instructions work fine for most input cases
- Use a broken version of SPIM. SPIM is treated as a black-box.
- Write a set of diagnostic programs to test where are wrong.
  - Big problem: Can't write tests with confidence, as your test code might be using one of the buggy instructions!

Lab 1: Design an ISA

- Define the functionality of a calculator
  - Operation types, operands, alignments…
- Implement the ISA in a simulator
- Write a set of diagnostic programs to verify the functionality of the simulator.
  - Your test code will be used to test your own VHDL and FPGA implementation!

Todo

- Refresh what was learned in CPEG323
  - ISA
  - MIPS
  - Single cycle