Goals and Scope

Welcome to the Workshop on Simple CPU Design by Example - Architecture to Implementation

The goal of this 14-day workshop is to give you end-to-end CPU Design exposure

- RTL Design, Verification (both Formal and Design Verification (DV)), Synthesis, PnR.
- We will understand basic concepts of CPU Architecture and Micro-Architecture.
- In addition, we will cover basics on how on-chip communication protocols are implemented. Examples: APB, AXI, CHI.

Expectations from the Participants

This is an intense course designed to give you an overall end-to-end view in one place.

1. We expect you to:
    a. ask questions, basic questions are okay.
    b. make errors, type code in the lab, make mistakes and learn.
    c. Try to close the practice work on the same day.
2. Setup a github repository account.
    a. Some examples and exercises will be at: git@github.com:techaarvam/karpagam.1.git
    b. To clone: **git clone git@github.com:techaarvam/karpagam.1.git**
    c. We will share examples, exercises, handouts using this repository.

3. Please try to follow the lab-setup instructions in this handout on your laptops and the lab machines and check all the tools are working correctly.

Every day there will be a new topic introduced. We have tried to bring basic solved examples to speed up the learning, but the instruction will also be in the lab to help understand the ideas and help with issues.

The topics covered in this workshop are worth many months to build true expertise. In this workshop to goal is a overall breadth, to give you the Aha!. To get the idea on how chip design happens.

At the end of the course, you will have example code, architectural understanding, the stages and tools involved in chip design, familiarity with some hard terminologies.

What the course is not?

Note: We will not be building a working CPU by the end of the course. We will give you all the tools and the information, the architecture and the know-how. With what you learn here, it is possible to continue to debug and get your code to work.

Lab Setup Instructions:

Common prerequisites (Ubuntu)

```
sudo apt update
sudo apt-get install build-essential clang bison flex \
                     libreadline-dev gawk tcl-dev libffi-dev git 
\
                     graphviz xdot pkg-config python3 zlib1g-dev
sudo apt install python3-pip
python3 -m pip install click
```

Verilator

```
sudo apt-get install verilator
```

Reference:
https://verilator.org/guide/latest/install.html

OpenROAD Flow Scripts

```
git clone --recursive https://github.com/The-OpenROAD-
Project/OpenROAD-flow-scripts
cd OpenROAD-flow-scripts
sudo ./setup.sh
```

Reference:
https://openroad-flow-scripts.readthedocs.io/en/latest/user/BuildLocally.html

---

Yosys

```
git clone --recurse-submodules
https://github.com/YosysHQ/yosys.git
cd yosys
git submodule update --init --recursive
make
sudo make install
```

Reference:
https://yosyshq.readthedocs.io/projects/yosys/en/latest/getting_started/installation.html

---

SymbiYosys

```
git clone https://github.com/YosysHQ/sby
cd sby
sudo make install
```

```
sudo apt install z3
```

Reference:
https://symbiyosys.readthedocs.io/en/latest/install.html

GtkWave

```
sudo apt install gtkwave
```

Reference:

https://github.com/gtkwave/gtkwave/blob/master/README.md

Exercise description for Day-1,2,3

The students will be split into sub-teams and each team can take up 1-3 exercises and complete.

Goals:

1. Warm up on Verilog coding for synthesizable design.
2. Basic understanding of the sub-units in a CPU.
3. Keep things simple. No advanced uArch concepts - hazards, forwarding, pipeline, pipeline stalls, exceptions, interrupts.

Exercises:

Program Counter:

Output (Register) : PC (program-counter)
Input (override) : boolean: change the PC value.
Input (new_pc): 28 bit input for the new PC value
Default operation: Increment by 4 - PC = PC+4
If override is required set the PC to the new value.
Hand the reset/clock correctly. i.e in the TB generate a clock.
Generate a synchronous reset.
reset polarity is active low. reset_n = 0 means reset is asserted. reset_n = 1 means reset is deasserted.
Reset PC value must be zero.
priority order: reset > override > PC+4

Understand about active low vs active high resets and why

Instruction Fetch: (with a 64 byte FIFO)

prerequisite: valid/ready understanding

Data transfer happens only when both valid and ready are both high in the same cycle. (This will be covered in the class.)

IRAM is a 1 MB memory (implemented in a TB code for this unit)

Read side signals:
output : rd_addr (16 bits), rd_valid,
input: rd_ready, rd_data (32 bits)

Program-counter:
input: pc (input from the TB in this module, will come from the program counter)

Fetch to decode signals:
Every cycle if the FIFO is not empty and no stall signal pop 1 item from the FIFO and output that.
output: instruction (32-bit) -> TOP of the FIFO.
output: inst_valid: boolean -> indicates if a valid instruction is output.
input: stall (indicates the receiver has not consumed the output).

Only when !stall && inst_valid an instruction is sent to the decode unit.

Description: Fetch ahead into a FIFO that is 64 bytes deep (16 32-bit words deep)

Instruction Decode:

Input: instruction (32-bit)
outputs (Instruction type):

alu_enable
branch_enable
ls_enable
outputs (rs1, rs2, rd, funct3, funct7, immediate)
outputs (operation_code)

Register File

inputs : rs1, rs2
outputs: register_val1, register_val2

inputs: rd, write_data, write_valid

There are 32 registers (x0 to x31).
If write_valid is one , modify the register that is selected.
Always output the requested registers for the read side. i.e depending on the sel1, sel2 output the correct register values. for example if register_sel1 is 4, output r4's value in register_val1.

LS Unit

Prerequisite: valid/ready understanding

SW: Store Word
LW: Load Word

DMEM interface:
output : addr, write, valid, wdata
input: ready, rdata

Process the opcode and instruction type and generate the access to the DMEM.
Result will go to write_data and write_valid.
The register file can alway write the register (no ready signal). Valid must be a 1 clock-

cycle pulse and pulled down after the cycle.

Instruction decode and the register file are the interfacing units to the LS unit.

ALU Unit:
Have a case to implement the instructions listed.
The instructions are all register to register simpler sub-set to start with.

Instruction decode and the register file are the other interfacing units to the ALU.
Note: Make the unit clock sensitive. Internal ALU computations can be combinational. But the inputs must be sampled and the outputs registered on the clock edge. ALU will have the required signals and most logic will be combinational. But the changes must be sensitive to the posedge of clock to exhibit proper sequential behaviour for the overall CPU pipeline.

Branch Unit:
redirecting the frontend program counter is the result of Branch instructions.
use case statements or nested ifs to implement the various instructions.
send override, new_pc to the program counter unit.
Branch unit will drive the override and new_pc, which will be observed by the program counter testbench.

Instruction decode and register file are the other interfaces to the branch unit.

Protocols: (Not for day-1) :
Request/Ack, valid/ready, valid/credit