CENG331
Archlab Recitation

Optimizing the Performance of a Pipelined Processor

What are we even doing?

- You’ve been studying a bit of processor architecture in class, through toy designs called SEQ and PIPE, using the toy Y86-64 instruction set.
- In part A, you will write some programs using Y86-64. Their C equivalents are provided.
- In part B, you will modify SEQ’s control logic to add a new instruction.
- In part C, you will do your best to optimize a Y86-64 program on PIPE by modifying the code, but also PIPE’s control logic if you want to.

Why though?

- You’ve examined assembly code so far but haven’t written any. A bit of assembly coding experience will further help you understand how higher level languages can be converted to assembly.
- Better understand the basic design of a processor by playing with its control logic.
- Appreciate how code and hardware interact and how this affects the performance of your programs, and how coupled their designs have to be.

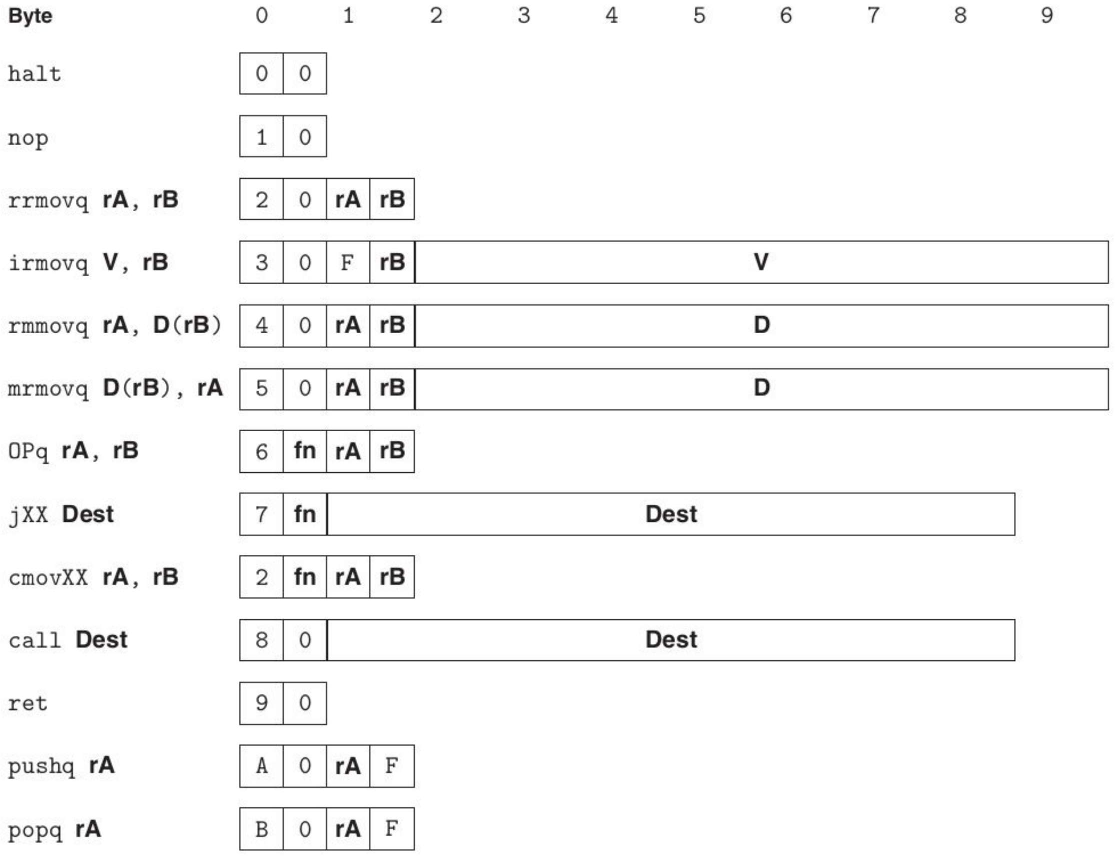
How to start working?

- The SEQ and PIPE simulators have a GUI component for debugging.
- Compilation works on the ineks, and you can use X11 forwarding for the GUI.
- Experimental instructions are available for 64-bit Ubuntu, easy.
- Experimental instructions are also available for MacOS, have to modify a few things.
- Check the final section of the homework PDF, **Installation and Usage Hints**, for details.

So many files?!

- The **sim** directory may feel confusing because it contains many files and directories.
- This is because it contains all the simulation, testing etc. programs. But also every file referenced in the book and its exercises...
- Focus on the files you need to use and modify, do not worry too much about the rest!
- You will only be submitting six files in the end. Feel free to modify the others for fun, but remember the originals will be used during grading.

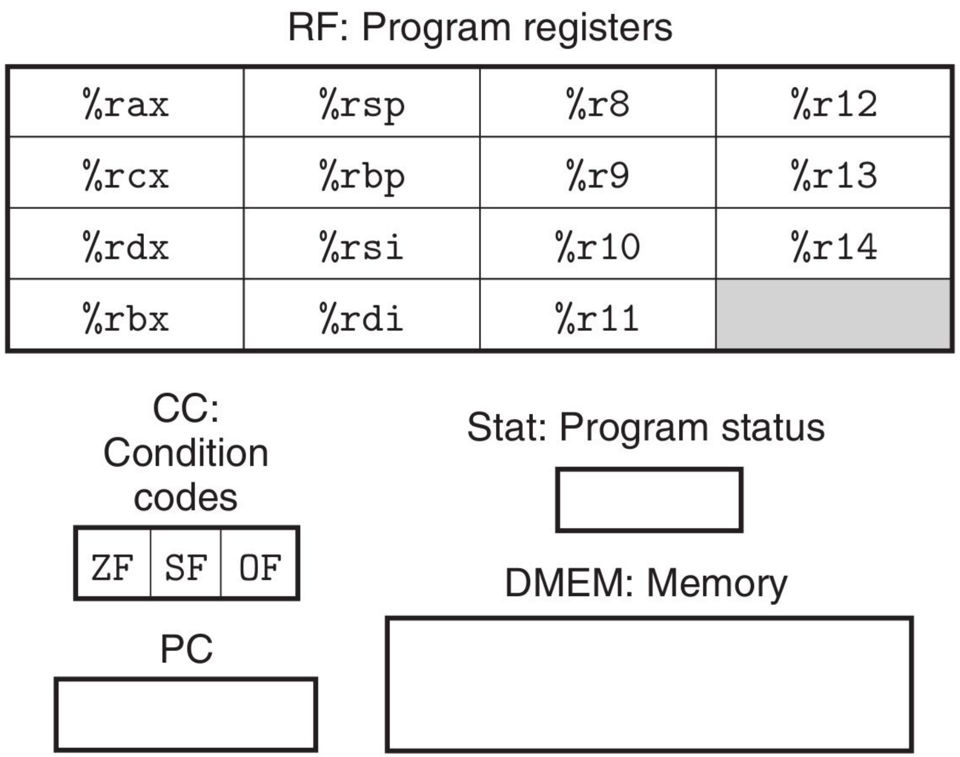
Y86-64 Overview



Byte	0	1	2	3	4	5	6	7	8	9
`halt`	0	0								
`nop`	1	0								
`rrmovq` **rA**, **rB**	2	0	**rA**	**rB**						
`irmovq` **V**, **rB**	3	0	F	**rB**	**V**					
`rmmovq` **rA**, **D**(**rB**)	4	0	**rA**	**rB**	**D**					
`mrmovq` **D**(**rB**), **rA**	5	0	**rA**	**rB**	**D**					
`OPq` **rA**, **rB**	6	**fn**	**rA**	**rB**						
`jXX` **Dest**	7	**fn**	**Dest**							
`cmovXX` **rA**, **rB**	2	**fn**	**rA**	**rB**						
`call` **Dest**	8	0	**Dest**							
`ret`	9	0								
`pushq` **rA**	A	0	**rA**	F						
`popq` **rA**	B	0	**rA**	F						

- Many conditions (e, ne, l, le, g, ge) are available for jumps and conditional moves, same as x86-64.
- Only ADD, SUB, AND and XOR operations.
- Operation instructions only work with registers. (Cannot do **andq $3, %rax** for example).
- Jump & call only constant addresses.

Y86-64 Overview



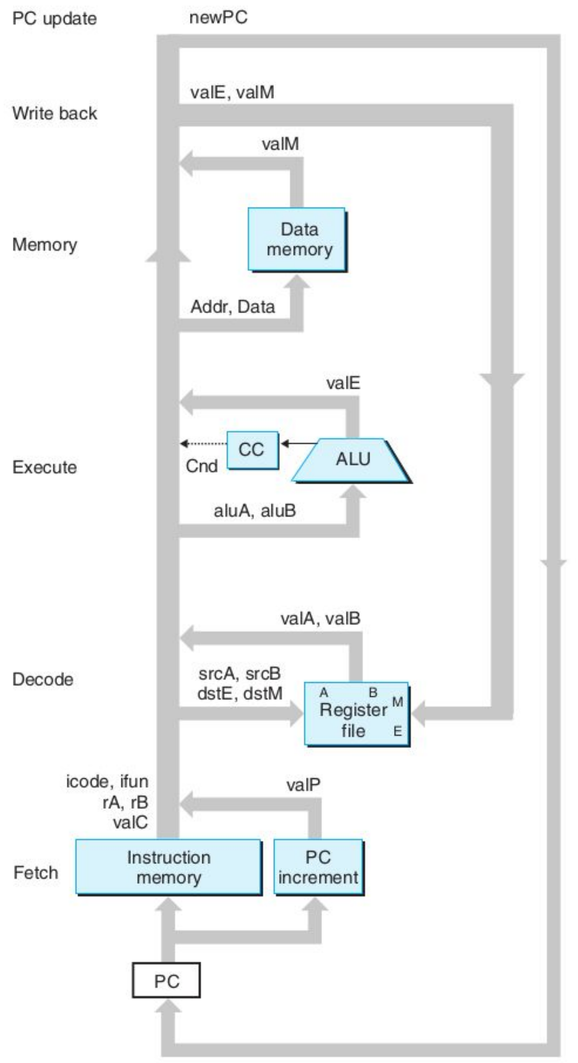
- Woohoo! Plenty of registers.
- Naming is pretty much the same as x86-64. **%rsp** holds the stack pointer, for example.
- Use the same calling conventions.
- Memory is simple, data and code are kept together, no segments.

Part A Quickstart

- Make sure to compile the homework first.
- Head into **sim/misc**.
- Check **examples.c** to see the C versions of the functions.
- You will write one Y86-64 program for each function. Details in the PDF!
- Fire up your favorite editor and write code for the first function in **rev.ys**.
- The **yas** program is the assembler, run it with **rev.ys** to obtain **rev.yo**, an object file.
- **yis** is the simulator, run it with **rev.yo** and observe the results.
- When you think your function works, move on to the next.

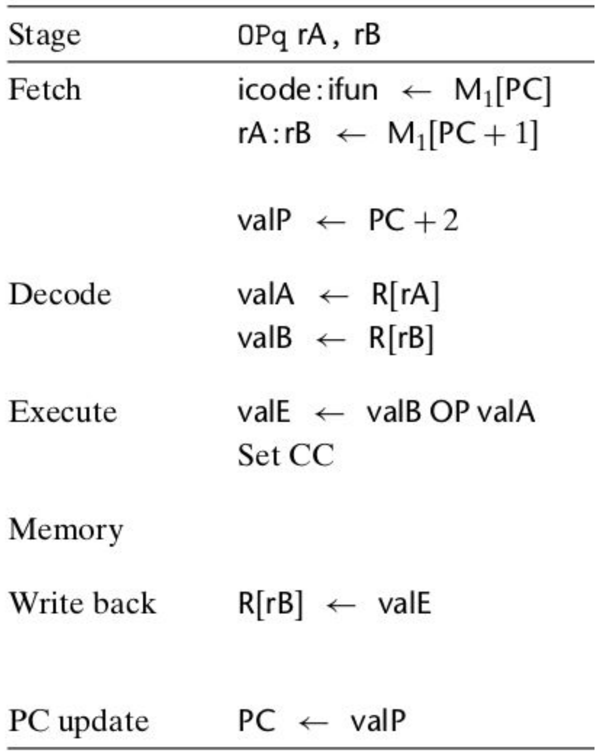
SEQ Overview

- Figure 4.22 in the book.
- Fairly simple, one instruction is performed per cycle, goes through the whole CPU. The multiple stages are well defined. Not too many signals.
- Since there is no pipeline, there are no hazards! No need for branch predictions, nothing.
- Do your best to understand how it works.
- You will only change the control logic using the HCL (Hardware Control Language) descriptions. No complete Verilog style design!



Computation Description

- Description is from Figure 4.18 in the book.
- Simply states what is done at each stage, leaving stages where nothing is done empty, for a given instruction.
- You will have to figure out the computation for the new instruction you will add in Part B, and add it to your file as a comment.
- Use this figure as a reference!



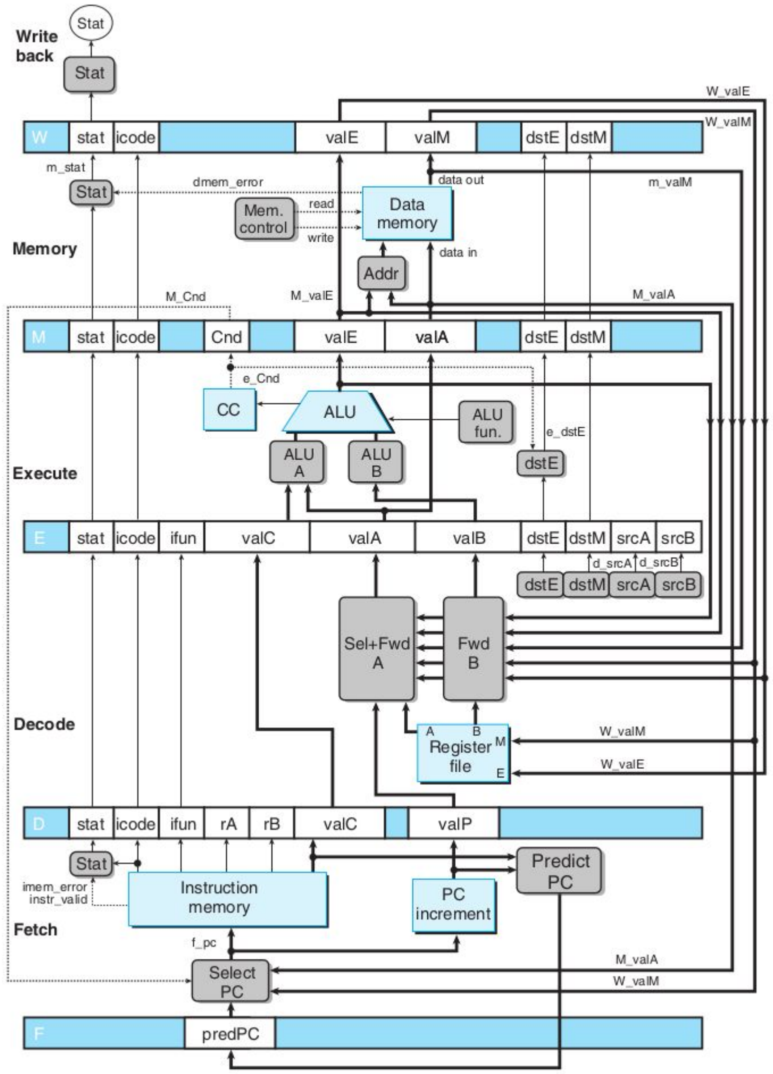
Stage	OPq rA, rB
Fetch	icode:ifun $\leftarrow$ $M_1[PC]$ rA:rB $\leftarrow$ $M_1[PC+1]$ valP $\leftarrow$ PC + 2
Decode	valA $\leftarrow$ R[rA] valB $\leftarrow$ R[rB]
Execute	valE $\leftarrow$ valB OP valA Set CC
Memory	
Write back	R[rB] $\leftarrow$ valE
PC update	PC $\leftarrow$ valP

Part B Quickstart

- Head into **sim/seq**.
- Look inside **seq-full.hcl**. This is the control logic description for the SEQ processor you will be modifying.
- Every time you change **seq-full.hcl**, you have to compile a new simulator based on your description using **make**. The sim has a GUI mode.
- Remember that Y86-64 could only jump to constant addresses. Goal is to add a new instruction **leaq** that will add to a register and store the result in another.
- Symbols are already defined, you simply have to modify the control logic to make the instruction work.
- How to test? See the PDF!

PIPE Overview

- Figure 4.52 in the book.
- Now that's hardware! What's even going on?!
- Lots of considerations for control and data hazards, exceptions etc.
- Understanding it will help make your code faster. Load/use hazards are an example, branch prediction is another.
- Just like SEQ, you will be able to access its control logic. Modification is not mandatory, but can be very useful!



Part C Quickstart

- Head into **sim/pipe**.
- The **absrev.ys** file contains the reverse copying program you will be optimizing.
- In a way similar to part B, you have **pipe-full.hcl**. This is the control logic description for the PIPE processor you *can* modify. Same considerations for the simulator.
- Check correctness with **./correctness.pl -p**, benchmark with **./benchmark.pl**, check size using **./check-len.pl < absrev.yo**, must be at most 1000 bytes!
- Can debug with the pipeline simulator if your code does not work and you do not understand why, using drivers generated by **gen-driver.pl**.
- Don't be scared of **pipe-full.hcl**, modifying it can help with performance!