4190.308 Computer Architecture

Architecture Lab Hints

Architecture lab

- Goal
  - learn about the design and implementation of a pipelined Y86 processor
  - optimize its performance on a benchmark program

Getting Started

- Environment
  - use the updated Gentoo virtual machine provided on eTL
    - all tools required to solve the lab are pre-installed in the VM
    - get it from eTL:
      Boards/Additional Material and Resources/Gentoo Virtual Machine

Warning
We will not answer any questions caused by not using this VM.

Getting Started

- Environment
  - download the `archlab-handout.tar` file. from the eTL
  - give the commands
    - `tar xvf archilab-handout.tar`
    - `cd archlab-handout/`
    - `tar xvf sim.tar`
    - `cd sim/`
    - `sim$ make clean; make`

Part A

Part A

- Work in directory `sim/seq` in this part
  - add one new instruction: `iaddl`
    (described in CS:APP textbook, practice problems 4.48 and 4.50)
  - To add this instruction, you will modify the file `seq-full.hcl`
    (it contains declarations of some constants that you will need)

Part A

- Build and Test your solution
  - Build a new simulator
    - `/seq$ make VERSION=full`
  - Test your solution on a simple Y86 program.
    For your initial testing, we recommend running a simple program such as `asumi.yo`(testing `iaddl`) in TTY mode, comparing the results against the ISA simulation:
    - `/seq$ ./ssim -t ../y86-code/asumi.yo`
  - Retest your solution using the benchmark programs.
    - `/seq$ cd ../y86-code`
    - `/y86-code$ make testssim`

Part A

- Perform regression tests
  - Run the extensive set of regression tests
    - `/y86-code$ cd ../ptest`
    - `/ptest$ make SIM=../seq/ssim`
  - test your implementation of iaddl
    - `/ptest$ make SIM=../seq/ssim TFLAGS=-i`

Part A

- `iaddl` instruction computation
  - `irmovl` instruction + OPl instruction
    - `iaddl V, rB : R[rB] ← R[rB] + V`

	`irmovl` V, rB
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_4[PC+2] valP ← PC+6
Decode	
Execute	valE ← 0 + valC
Memory	
Write back	R[rB] ← valE
PC update	PC ← valP

		OPl rA, rB
Fetch	icode,ifun	icode:ifun ← M_1[PC]
	rA,rB	rA:rB ← M_1[PC+1]
	valC	
	valP	valP ← PC+2
Decode	valA, srcA	valA ← R[rA]
	valB, srcB	valB ← R[rB]
Execute	valE	valE ← valB OP valA
	Cond code	Set CC
Memory	valM	
Write back	dstE	R[rB] ← valE
	dstM	
PC update	PC	PC ← valP

Part A

- `iaddl` instruction verification
  - ./ssim –t ../y86-code/asumi.yo

```
ISA Register != Pipeline Register File
%eax:    0x9bdddc31      0x0000000d
%ecx:    0x00001f48      0x00000014
%edx:    0x000007d1      0x00000004
%esi:    0x00000000      0x0000000d
ISA Check Fails
bjkim@ubuntu:~/ComArchi/archilab/archlab-handout/sim/seq$
```

```
%esp:    0x00000000      0x000000f0
%ebp:    0x00000000      0x000000f0
Changed Memory State:
0x00f0: 0x00000000      0x00000100
0x00f4: 0x00000000      0x00000039
0x00f8: 0x00000000      0x00000014
0x00fc: 0x00000000      0x00000004
ISA Check Succeeds
bjkim@ubuntu:~/ComArchi/archilab/archlab-handout/sim/seq$
```

Before you start Part A : Hint

Example: leave instruction

- Work in directory `sim/seq/` in this part
- To add this instruction, you will modify the file `seq-full.hcl`
- `leave` instruction can be used to prepare the stack for returning.
  - As described in Section 3.7.2. in textbook.
  - It is equivalent to the following code sequence(X86):
    - `movl %ebp, %esp` *Set stack pointer to beginning of frame*
    - `popl %ebp` *Restore saved %ebp and set stack ptr to end of caller's frame*

Example: leave instruction

- `leave` instruction computation in Y86
  - (`rrmovl %ebp, %esp`) + (`popl %ebp`)

	`rrmovl` rA, rB
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2
Decode	valA ← R[rA]
Execute	valE ← 0 + valA
Memory	
Write back	R[rB] ← valE
PC update	PC ← valP

	`popl` rA
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2
Decode	valA ← R[%esp] valB ← R[%esp]
Execute	valE ← valB + 4
Memory	valM ← M_4[valA]
Write back	R[%esp] ← valE R[rA] ← valM
PC update	PC ← valP

Example: leave instruction

- `leave` instruction computation in Y86

	`leave`
Fetch	icode:ifun ← M_1[PC] valP ← PC+1
Decode	valA ← R[%ebp] valB ← R[%ebp]
Execute	valE ← valB + 4
Memory	valM ← M_4[valA]
Write back	R[%esp] ← valE R[%ebp] ← valM
PC update	PC ← valP

Example: leave instruction

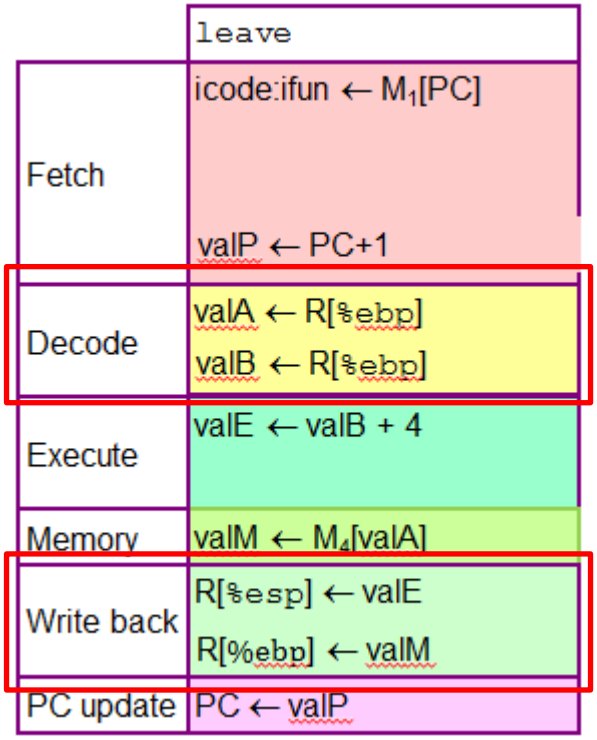
- Implementation
  - Fetch stage

	leave
Fetch	icode:ifun ← M_1[PC] valP ← PC+1
Decode	valA ← R[%ebp] valB ← R[%ebp]
Execute	valE ← valB + 4
Memory	valM ← M_4[valA]
Write back	R[%esp] ← valE R[%ebp] ← valM
PC update	PC ← valP

```
bool instr_valid = icode in
        { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
                ILEAVE,
                IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };
```

Example: leave instruction

- Implementation
  - Decode & Write-back stage




```
## What register should be used as the A source?
int srcA = [
        icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL  } : rA;
        icode in { ILEAVE } : REBP;
        icode in { IPOPL, IRET } : RESP;
        1 : RNONE; # Don't need register
];
```

```
## What register should be used as the B source?
int srcB = [
        icode in { IOPL, IRMMOVL, IMRMOVL  } : rB;
        icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
        icode in { ILEAVE  } : REBP;
        1 : RNONE;  # Don't need register
];
```

```
## What register should be used as the E destination?
int dstE = [
        icode in { IRRMOVL } && Cnd : rB;
        icode in { IIRMOVL, IOPL} : rB;
        icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
        icode in { ILEAVE } : RESP;
        1 : RNONE;  # Don't write any register
];
```

```
## What register should be used as the M destination?
int dstM = [
        icode in { IMRMOVL, IPOPL } : rA;
        icode in { ILEAVE } : REBP;
        1 : RNONE;  # Don't write any register
];
```

Example: leave instruction

- Implementation
  - Execute stage

	leave
Fetch	icode:ifun ← M_1[PC] valP ← PC+1
Decode	valA ← R[%ebp] valB ← R[%ebp]
Execute	valE ← valB + 4
Memory	valM ← M_4[valA]
Write back	R[%esp] ← valE R[%ebp] ← valM
PC update	PC ← valP

```
int aluA = [
        icode in { IRRMOVL, IOPL } : valA;
        icode in { IIRMOVL, IRMMOVL, IMRMOVL } : valC;
        icode in { ICALL, IPUSHL } : -4;
        icode in { IRET, IPOPL } : 4;
        icode in { ILEAVE } : 4;
        # Other instructions don't need ALU
];
```

```
int aluB = [
        icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
                   IPUSHL, IRET, IPOPL } : valB;
        icode in { ILEAVE } : valB;
        icode in { IRRMOVL, IIRMOVL } : 0;
        # Other instructions don't need ALU
];
```

```
## Set the ALU function
int alufun = [
        icode == IOPL : ifun;
        1 : ALUADD;
];
```

Example: leave instruction

- Implementation
  - Memory stage

	leave
Fetch	icode:ifun ← M_1[PC] valP ← PC+1
Decode	valA ← R[%ebp] valB ← R[%ebp]
Execute	valE ← valB + 4
Memory	valM ← M_4[valA]
Write back	R[%esp] ← valE R[%ebp] ← valM
PC update	PC ← valP

```
## Set read control signal
bool mem_read = icode in { IMRMOVL, IPOPL, IRET, ILEAVE };
```

```
## Select memory address
int mem_addr = [
        icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : valE;
        icode in { IPOPL, IRET } : valA;
        icode in { ILEAVE } : valA;
        # Other instructions don't need address
];
```

Example: leave instruction

- Implementation
  - PC update

	leave
Fetch	icode:ifun ← M_1[PC] valP ← PC+1
Decode	valA ← R[%ebp] valB ← R[%ebp]
Execute	valE ← valB + 4
Memory	valM ← M_4[valA]
Write back	R[%esp] ← valE R[%ebp] ← valM
PC update	PC ← valP

```
int new_pc = [
        # Call.  Use instruction constant
        icode == ICALL : valC;
        # Taken branch.  Use instruction constant
        icode == IJXX && Cnd : valC;
        # Completion of RET instruction.  Use value from stack
        icode == IRET : valM;
        # Default: Use incremented PC
        1 : valP;
];
```

Example: leave instruction

- Implementation
  - verification
    - `/seq$ make clean; make ssim VERSION=full`
    - `/seq$ ./ssim -t ../y86-code/asuml.yo`

```
0x03a8: 0x00000000      0x00000014
0x03c0: 0x00000000      0x000003f0
0x03c4: 0x00000000      0x0000007e
0x03c8: 0x00000000      0x00000018
0x03cc: 0x00000000      0x00000003
0x03f0: 0x00000000      0x00000400
0x03f4: 0x00000000      0x00000039
0x03f8: 0x00000000      0x00000014
0x03fc: 0x00000000      0x00000004
ISA Check Succeeds
bjkim@ubuntu:~/ComArchi/archilab/archlab-handout/sim/seq$
```

Part B

Part B

- work in directory `sim/pipe` in this part
  - modify `ncopy.ys` and `pipe-full.hcl` with the goal of making `ncopy.ys` run as fast as possible.
  - Refer to HCL files for various CS:APP textbook practice problems. Each practice problem is a clue of optimization.
    - `pipe-nobypass.hcl` PIPE without bypassing
    - `pipe-full.hcl` Add iaddl instruction to PIPE
    - `pipe-nt.hcl` Implement branch not taken strategy
    - `pipe-btfnt.hcl` Implement back-taken forward-not-taken strategy
    - `pipe-lf.hcl` Implement load forwarding logic
    - `pipe-lw.hcl` Implement single ported register file

Part B

- Coding Rules
  - Your `ncopy.ys` function must work for arbitrary array sizes. You might be tempted to hardwire your solution for 64-element arrays by simply coding 64 copy instructions, but this would be a bad idea because we will be grading your solution based on its performance on arbitrary arrays.
  - Your `ncopy.ys` function must run correctly with `YIS`. By correctly, we mean that it must correctly copy the `src` block `and` return (in `%eax`) the correct number of positive integers.
  - Your `pipe-full.hcl` implementation must pass the regression tests in `../y86-code` and `../ptest` (without the `-il`flags that test `iaddl`).

Part B

- Coding Rules
  - you are free to implement the `iaddl` instruction if you think that will help.
  - You are free to alter the branch prediction behavior or to implement techniques such as load bypassing.
  - You may make any semantics preserving transformations to the `ncopy.ys` function, such as swapping instructions, replacing groups of instructions with single instructions, deleting some instructions, and adding other instructions.

Part B

- Build and Run your solution
  - build a driver program that calls your `ncopy` function

  - We have provided you with the `gen-driver.pl` program that generates two driver programs for arbitrary sized input arrays.
    - `/pipe$ make drivers`
    - `sdriver.yo`: `A small driver program` that tests an `ncopy` function on small arrays with 4 elements. If your solution is correct, then this program will halt with a value of 2 in register `%eax` after copying the `src` array.
    - ldriver.yo: A large driver program that tests an ncopy function on larger arrays with 63 ele-ments. If your solution is correct, then this program will halt with a value of 62 (`0x1f`) in register%eaxafter copying the `src` array.

Part B

- Build and Run your solution
  - Each time you modify your ncopy.ys program, you can rebuild the driver programs by typing
    - `/pipe$ make drivers`
  - Each time your modify your `pipe-full.hcl` file you can rebuild the simulator by typing
    - `/pipe$ make psim`
  - If you want to rebuild the simulator and the driver programs, type
    - `/pipe$ make`

Part B

- Build and Run your solution
  - To test your solution on a small 4-element array, type
    - `/pipe$ ./psim sdriver.yo`
  - To test your solution on a larger 63-element array, type
    - `/pipe$ ./psim ldriver.yo`

Part B

- Additional tests
  - Testing your driver file on the ISA simulator. Make sure that your ncopy.ys function works properly with YIS:
    - `/pipe$ make`
    - `/pipe$ ../misc/yis sdriver.yo`
  - Testing your code on a range of block lengths with the ISA simulator.
    - `/pipe$ correctness.pl`

Part B

- Additional tests
  - generate a driver file for that length that includes checking code, and where the result varies randomly
    - `/pipe$  ./gen-driver.pl -f ncopy.ys K  -rc > driver.ys`
    - `/pipe$  make driver.yo`
    - `/pipe$  ../misc/yis driver.yo`
  - The program will end with register %eax having value:
    - `0xaaaa` : All tests pass
    - `0xbbbb` : Incorrect count
    - `0xcccc` : Function ncopy is more than 1000 bytes long.
    - `0xdddd` : Some of the source data was not copied to its destination.
    - `0xeeee` : Some word just before or just after the destination region was corrupted.
  - In printing register and memory values, it only prints out words that change during simulation, either in registers or in memory

Part B

- Additional tests
  - Testing your simulator on the benchmark programs.
    - `/pipe$ (cd ../y86-code; make testpsim)`
  - Testing your pipeline simulator with extensive regression tests.
    - `/pipe$ (cd ../ptest; make SIM=../pipe/psim TFLAGS=-i)`

Before you start Part B: Hint

Example: nobypass

- Modify Pipeline Register F
  - `pipe-nobypass.hcl`

```
# Should I stall or inject a bubble into Pipeline Register F?
# At most one of these can be true.
bool F_bubble = 0;
bool F_stall =
        # Modify the following to stall the update of pipeline register F
        0 ||
        # Stalling at fetch while ret passes through pipeline
        IRET in { D_icode, E_icode, M_icode };
```

  - hint

```
# Should I stall or inject a bubble into Pipeline Register F?
# At most one of these can be true.
bool F_bubble = 0;
bool F_stall =
        # Stall if either operand source is destination of
        # instruction in execute, memory, or write-back stages




        # Stalling at fetch while ret passes through pipeline
        IRET in { D_icode, E_icode, M_icode };
```

Example: nobypass

- Modify Pipeline Register D
  - `pipe-nobypass.hcl`

```
# Should I stall or inject a bubble into Pipeline Register D?
# At most one of these can be true.
bool D_stall =
        # Modify the following to stall the instruction in decode
        0;
```

  - hint

```
# Should I stall or inject a bubble into Pipeline Register D?
# At most one of these can be true.
bool D_stall =
        # Stall if either operand source is destination of
        # instruction in execute, memory, or write-back stages
        # but not part of mispredicted branch
```

Example: nobypass

- Modify Pipeline Register E
  - `pipe-nobypass.hcl`

```
# Should I stall or inject a bubble into Pipeline Register E?
# At most one of these can be true.
bool E_stall = 0;
bool E_bubble =
        # Mispredicted branch
        (E_icode == IJXX && !e_Cnd) ||
        # Modify the following to inject bubble into the execute stage
        0;
```

  - hint

```
bool E_bubble =
        # Mispredicted branch
        (E_icode == IJXX && !e_Cnd) ||
          # Inject bubble if either operand source is destination of
          # instruction in execute, memory, or write back stages
```

Now, it’s your turn!

Good Luck!