ETH, Design of Digital Circuits, SS17
Practice Exercises I

Instructors: Prof. Onur Mutlu, Prof. Srdjan Capkun
TAs: Jeremie Kim, Minesh Patel, Hasan Hassan, Arash Tavakkol, Der-Yeuan Yu, Francois Serre, Victoria Caparros Cabezas, David Sommer, Mridula Singh, Sinisa Matetic, Aritra Dhar, Marco Guarnieri

Note: These exercises are not graded and are optional. The points are provided just to indicate the amount of time you may want to spend working on each exercise relative to the others.

1 Big versus Little Endian Addressing [5 points]

Consider the 32-bit hexadecimal number 0xcafe2b3a.

1. What is the binary representation of this number in *little endian* format? Please clearly mark the bytes and number them from low (0) to high (3).
2. What is the binary representation of this number in *big endian* format? Please clearly mark the bytes and number them from low (0) to high (3).

2 The MIPS ISA [40 points]

2.1 Warmup: Computing a Fibonacci Number [15 points]

The Fibonacci number F_n is recursively defined as

$$F(n) = F(n-1) + F(n-2),$$

where $F(1) = 1$ and $F(2) = 1$. So, $F(3) = F(2) + F(1) = 1 + 1 = 2$, and so on. Write the MIPS assembly for the `fib(n)` function, which computes the Fibonacci number $F(\texttt{n})$:

```
int fib(int n)
{
  int a = 0;
  int b = 1;
  int c = a + b;
  while (n > 1) {
    c = a + b;
    a = b;
    b = c;
    n--;
  }
  return c;
}
```

Remember to follow MIPS calling convention and its register usage (just for your reference, you may not need to use all of these registers):

- The argument `n` is passed in register $4.
- The result (i.e., `c`) should be returned in $2.
- $8 to $15 are caller-saved temporary registers.
- $16 to $23 are callee-saved temporary registers.
- $29 is the stack pointer register.
- $31 stores the return address.

Note: A summary of the MIPS ISA is provided at the end of this handout.

2.2 MIPS Assembly for REP MOVSB [25 points]

Recall from lecture that MIPS is a Reduced Instruction Set Computing (RISC) ISA. Complex Instruction Set Computing (CISC) ISAs—such as Intel's x86—often use one instruction to perform the function of many instructions in a RISC ISA. Here you will implement the MIPS equivalent for a single Intel x86 instruction, REP MOVSB, which is specified as follows.

The REP MOVSB instruction uses three fixed x86 registers: ECX (count), ESI (source), and EDI (destination). The "repeat" (REP) prefix on the instruction indicates that it will repeat ECX times. Each iteration, it moves one byte from memory at address ESI to memory at address EDI, and then increments both pointers by one. Thus, the instruction copies ECX bytes from address ESI to address EDI.

(a) Write the corresponding assembly code in MIPS ISA that accomplishes the same function as this instruction. You can use any general purpose register. Indicate which MIPS registers you have chosen to correspond to the x86 registers used by REP MOVSB. Try to minimize code size as much as possible.

(b) What is the size of the MIPS assembly code you wrote in (a), in bytes? How does it compare to REP MOVSB in x86 (note: REP MOVSB occupies 2 bytes)?

(c) Assume the contents of the x86 register file are as follows before the execution of the REP MOVSB:

```
EAX: 0xccccaaaa
EBP: 0x00002222
ECX: 0xFEE1DEAD
EDX: 0xfeed4444
ESI: 0xdecaffff
EDI: 0xdeaddeed
EBP: 0xe0000000
ESP: 0xe0000000
```

Now, consider the MIPS assembly code you wrote in (a). How many total instructions will be executed by your code to accomplish the same fuction as the single REP MOVSB in x86 accomplishes for the given register state?

(d) Assume the contents of the x86 register file are as follows before the execution of the REP MOVSB:

```
EAX: 0xccccaaaa
EBP: 0x00002222
ECX: 0x00000000
EDX: 0xfeed4444
ESI: 0xdecaffff
EDI: 0xdeaddeed
EBP: 0xe0000000
ESP: 0xe0000000
```

Now, answer the same question in (c) for the above register values.

3 Data Flow Programs [15 points]

Draw the data flow graph for the `fib(n)` function from Question 2.1. You may use the following data flow nodes in your graph:

- + (addition)
- > (left operand is greater than right operand)
- `Copy` (copy the value on the input to both outputs)
- `BR` (branch, with the semantics discussed in class, label the True and False outputs)

You can use constant inputs (e.g., 1) that feed into the nodes. Clearly label all the nodes, program inputs, and program outputs. Try to the use fewest number of data flow nodes possible.

4 Performance Metrics [10 points]

- If a given program runs on a processor with a higher frequency, does it imply that the processor always executes more instructions per second (compared to a processor with a lower frequency)? (Use less than 10 words.)
- If a processor executes more of a given program's instructions per second, does it imply that the processor always finishes the program faster (compared to a processor that executes fewer instructions per second)? (Use less than 10 words.)

5 Performance Evaluation [9 points]

Your job is to evaluate the potential performance of two processors, each implementing a different ISA. The evaluation is based on its performance on a particular benchmark. On the processor implementing ISA A, the best compiled code for this benchmark performs at the rate of 10 IPC. That processor has a 500 MHz clock. On the processor implementing ISA B, the best compiled code for this benchmark performs at the rate of 2 IPC. That processor has a 600 MHz clock.

- What is the performance in Millions of Instructions per Second (MIPS) of the processor implementing ISA A?
- What is the performance in MIPS of the processor implementing ISA B?
- Which is the higher performance processor: A B Don't know Briefly explain your answer.

6 LC-3b Microcode [80 points]

We wrote the microcode for at least one state of the LC-3b microarchitecture in class. In this homework, you will complete the microcode for all states of the LC-3b. Refer to Appendix C of Patt and Patel for the LC-3b state machine and datapath and Appendix A of Patt and Patel for the LC-3b ISA description.

Fill out the microcode in the microcode.csv file handed out with this homework. Enter a 1 or a 0 or an X as appropriate for the microinstructions corresponding to states. You do not need to fill out states 8, 10 and 11. We fill out state 18 as an example. Please turn in this CSV file electronically along with your homework.

7 Microarchitecture vs. ISA [15 points]

(a) Briefly explain the difference between the microarchitecture level and the ISA level in the transformation hierarchy. What information does the compiler need to know about the microarchitecture of the machine in order to compile the program correctly?

(b) Classify the following attributes of a machine as either a property of its microarchitecture or ISA:

(i) The machine does not have a subtract instruction.

(ii) The ALU of the machine does not have a subtract unit.

(iii) The machine does not have condition codes.

(iv) A 5-bit immediate can be specified in an `ADD` instruction.

(v) It takes n cycles to execute an `ADD` instruction.

(vi) There are 8 general purpose registers.

(vii) A 2-to-1 mux feeds one of the inputs to ALU.

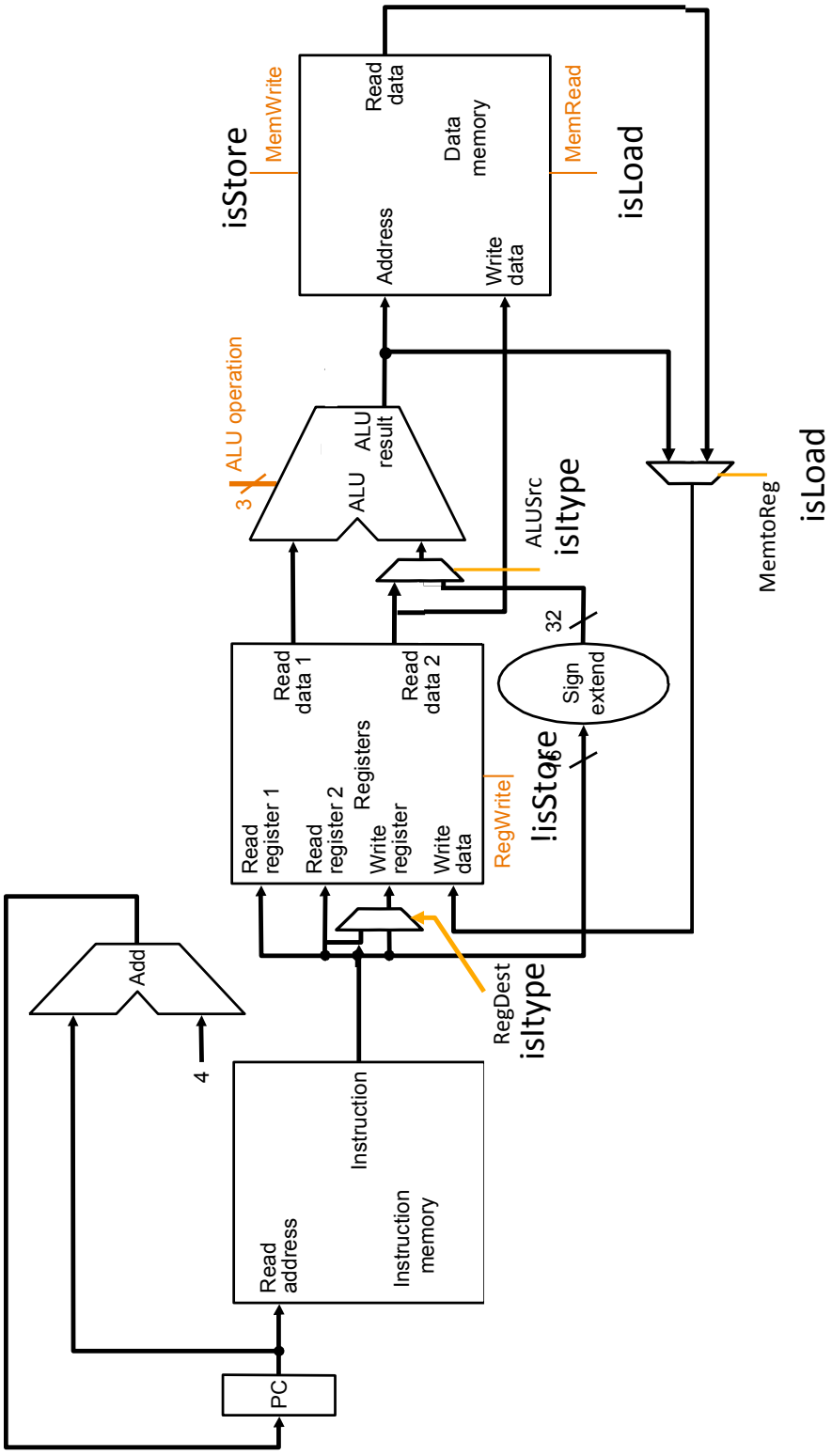
(viii) The register file has one input port and two output ports.

8 Single-Cycle Processor Datapath [30 points]

In this problem, you will modify the single-cycle datapath we built up in lecture to support the `JAL` instruction. The datapath that we will start with is provided below. Your job is to implement the necessary data and control signals to support the `JAL` instruction, which we define to have the following semantics:

$$\begin{aligned} \texttt{JAL}: \quad & \texttt{R31} \leftarrow \texttt{PC} + 4 \\ & \texttt{PC} \leftarrow \texttt{PC}_{31\ldots28} \mid\mid \texttt{Immediate} \mid\mid 0^2 \end{aligned}$$

Add to the datapath on the next page the necessary data and control signals to implement the `JAL` instruction. Draw and label all components and wires very clearly (give control signals meaningful names; if selecting a subset of bits from many, specify exactly which bits are selected; and so on).


PC
Add
4
Instruction memory
Read address
Instruction
Registers
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
RegWrite
RegDest
isItype
!isStore
Sign extend
32
ALU
ALU result
ALU operation
3
ALUSrc
isItype
Data memory
Address
Write data
Read data
MemWrite
isStore
MemRead
isLoad
MemtoReg
isLoad

9 Pipelining [30 points]

Here is a program:

```
MUL R3  ← R1, R2
ADD R5  ← R4, R3
ADD R6  ← R4, R1
MUL R7  ← R8, R9
ADD R4  ← R3, R7
MUL R10 ← R5, R6
```

Note: Each instruction is specified with the destination register first.

Calculate the number of cycles it takes to execute the given code on the following models:

(a) A non-pipelined machine

(b) A pipelined machine with scoreboarding and five adders and five multipliers without data forwarding

(c) A pipelined machine with scoreboarding and five adders and five multipliers with data forwarding

(d) A pipelined machine with scoreboarding and one adder and one multiplier without data forwarding

(e) A pipelined machine with scoreboarding and one adder and one multiplier with data forwarding

For all machine models, use the basic instruction cycle as follows:

1. Fetch (one clock cycle)
2. Decode (one clock cycle)
3. Execute (`MUL` takes 6, `ADD` takes 4 clock cycles). The multiplier and the adder are not pipelined.
4. Write-back (one clock cycle)

Do not forget to list any assumptions you make about the pipeline structure (e.g., how is data forwarding done between pipeline stages)

10 Hardware vs Software Interlocking [30 points]

Consider two pipelined machines, I and II:

Machine I implements interlocking in hardware. On detection of a flow dependence, it stalls the instruction in the decode stage of the pipeline (blocking fetch/decode of subsequent instructions) until all of the instruction's sources are available. Assume internal register file forwarding (an instruction writes into a register in the first half of a cycle and another instruction can access the same register in the next half of the cycle). No other data forwarding is implemented. However, there are three execute units with adders, and independent instructions can be executed in separate execution units and written back out-of-order. There is one write-back stage per execute unit, so an instruction can write-back as soon as it finishes execution.

Machine II does not implement interlocking in hardware. It assumes all instructions are independent and relies on the compiler to order instructions such that there is sufficient distance between dependent instructions. The compiler either moves other independent instructions between two dependent instructions, if it can find such instructions, or otherwise, inserts `NOP`s. Assume internal register file forwarding (an instruction writes into a register in the first half of a cycle and another instruction can access the same register in the next half of the cycle).

Both machines have the following *four pipeline stages* and *three adders*:

1. Fetch (one clock cycle)
2. Decode (one clock cycle)

3. Execute (`ADD` takes 3 clock cycles. Each `ADD` unit is not pipelined, but an instruction can be executed if an unused execute (`ADD`) unit is available.)
4. Write-back (one clock cycle). There is one write-back stage per execute (`ADD`) unit.

Consider the following 2 code segments:

Code segment *A*

```
ADD R5  ← R6, R7
ADD R3  ← R5, R4
ADD R6  ← R3, R8
ADD R9  ← R6, R3
```

Code segment *B*

```
ADD R3  ← R1, R2
ADD R8  ← R9, R10
ADD R4  ← R5, R6
ADD R7  ← R1, R4
ADD R12 ← R8, R2
```

(a) Calculate the number of cycles it takes to execute each of these two code segments on Machines I and Machine II.

(b) Calculate the machine code size of each of these two code segments on machines I and II, assuming fixed-length ISA, where each instruction is encoded as 4 bytes.

(c) Which machine takes a smaller number of cycles to execute each code segment I and II?

(d) Does the machine that takes the smaller number of cycles for code segment A also take the smaller number of cycles than the other machine for code segment B? Why or why not?

(e) Would you say that the machine that provides a smaller number of cycles as compared to the other machine has higher performance (taking into account all components of the Iron Law of Performance)?

(f) Which machine incurs lower code size for each code segment A and B?

(g) Does the same machine incur lower code sizes for both code segments A and B? Why or why not?

MIPS Instruction Summary

Opcode	Example Assembly	Semantics
add	`add $1, $2, $3`	`$1 = $2 + $3`
sub	`sub $1, $2, $3`	`$1 = $2 - $3`
add immediate	`addi $1, $2, 100`	`$1 = $2 + 100`
add unsigned	`addu $1, $2, $3`	`$1 = $2 + $3`
subtract unsigned	`subu $1, $2, $3`	`$1 = $2 - $3`
add immediate unsigned	`addiu $1, $2, 100`	`$1 = $2 + 100`
multiply	`mult $2, $3`	`hi, lo = $2 * $3`
multiply unsigned	`multu $2, $3`	`hi, lo = $2 * $3`
divide	`div $2, $3`	`lo = $2/$3, hi = $2 mod $3`
divide unsigned	`divu $2, $3`	`lo = $2/$3, hi = $2 mod $3`
move from hi	`mfhi $1`	`$1 = hi`
move from low	`mflo $1`	`$1 = lo`
and	`and $1, $2, $3`	`$1 = $2 & $3`
or	`or $1, $2, $3`	`$1 = $2 | $3`
and immediate	`andi $1, $2, 100`	`$1 = $2 & 100`
or immediate	`ori $1, $2, 100`	`$1 = $2 | 100`
shift left logical	`sll $1, $2, 10`	`$1 = $2 << 10`
shift right logical	`srl $1, $2, 10`	`$1 = $2 >> 10`
load word	`lw $1, 100($2)`	`$1 = memory[$2 + 100]`
store word	`sw $1, 100($2)`	`memory[$2 + 100] = $1`
load upper immediate	`lui $1, 100`	`$1 = 100 << 16`
branch on equal	`beq $1, $2, label`	`if ($1 == $2) goto label`
branch on not equal	`bne $1, $2, label`	`if ($1 != $2) goto label`
set on less than	`slt $1, $2, $3`	`if ($2 < $3) $1 = 1 else $1 = 0`
set on less than immediate	`slti $1, $2, 100`	`if ($2 < 100) $1 = 1 else $1 = 0`
set on less than unsigned	`sltu $1, $2, $3`	`if ($2 < $3) $1 = 1 else $1 = 0`
set on less than immediate	`sltui $1, $2, 100`	`if ($2 < 100) $1 = 1 else $1 = 0`
jump	`j label`	`goto label`
jump register	`jr $31`	`goto $31`
jump and link	`jal label`	`$31 = PC + 4; goto label`