Digital Design and Computer Architecture (252-0028-00L), Spring 2023
Optional HW 3: Microarchitecture, ISA, and Performance Evaluation
SOLUTIONS

Instructor: Prof. Onur Mutlu
TAs: Juan Gomez Luna, Mohammad Sadrosadati, Mohammed Alser, Ataberk Olgun, Giray Yaglikci, Can Firtina, Geraldo De Oliveira Junior, Rahul Bera, Konstantinos Kanellopoulos, Nika Mansouri Ghiasi, Nisa Bostancı, Rakesh Nadig, Joel Lindegger, İsmail Emir Yüksel, Haocong Luo, Yahya Can Tuğrul, Julien Eudine

Released: Monday, April 17, 2023

1 Big versus Little Endian Addressing

Consider the 32-bit hexadecimal number 0xcafe2b3a.

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

3a	2b	fe	ca
0	1	2	3

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

ca	fe	2b	3a
0	1	2	3

2 The MIPS ISA

2.1 Warmup: Computing a Fibonacci Number

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 $F($`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.

NOTE: More than one correct solution exists, this is just one potential solution. This solution uses callee-saved registers (r16-r18). There are correct solutions using the caller-saved registers as well.

```
fib:
addi $sp, $sp, -16 // allocate stack space
sw   $16, 0($sp)   // save r16
add  $16, $4,  $0  // r16 for arg n
sw   $17, 4($sp)   // save r17
add  $17, $0,  $0  // r17 for a, init to 0
sw   $18, 8($sp)   // save r18
addi $18, $0,  1   // r18 for b, init to 1
sw   $31, 12($sp)  // save return address
add  $2,  $17, $18 // c = a + b

branch:
slti $3,  $16, 2   // use r3 as temp
bne  $3,  $0,  done
add  $2,  $17, $18 // c = a + b
add  $17, $18, $0  // a = b
add  $18, $2,  $0  // b = c
addi $16, $16, -1  // n = n - 1
j    branch

done:
lw   $31, 12($sp)  // restore r31
lw   $18, 8($sp)   // restore r18
lw   $17, 4($sp)   // restore r17
lw   $16, 0($sp)   // restore r16
addi $sp, $sp, 16  // restore stack pointer
jr   $31           // return to caller
```

2.2 MIPS Assembly for REP MOVSB

MIPS is a simple ISA. Complex ISAs—such as Intel's x86—often use one instruction to perform the function of many instructions in a simple 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.

Assume: $1 = ECX, $2 = ESI, $3 = EDI

```
beq    $1, $0, AfterLoop          // If counter is zero, skip
CopyLoop:
lb     $4, 0($2)                  // Load 1 byte
sb     $4, 0($3)                  // Store 1 byte
addiu  $2, $2, 1                  // Increase source pointer by 1 byte
addiu  $3, $3, 1                  // Increase destination pointer by 1 byte
addiu  $1, $1, -1                 // Decrement counter
bne    $1, $0, CopyLoop           // If not zero, repeat
AfterLoop:
Following instructions
```

(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)?

The size of the MIPS assembly code is 4 bytes $\times$ 7 = 28 bytes, as compared to 2 bytes for x86 REP MOVSB.

(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?

The count (value in ECX) is 0xfee1dead = 4276215469. Therefore, the loop body is executed 4276215469 times. As there are 6 instructions in the loop body, total instructions executed = 6 * 4276215469 + 1 = 25657292814 + 1 (beq instruction outside of the loop) = 25657292815.

(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.

The count (value in ECX) is 0x00000000 = 0. Therefore, the loop body is executed 0 times. Total instructions executed = 1 (beq instruction outside of the loop).

3 Dataflow

- We define the *switch node* in Figure 1 to have 2 inputs (**I**, **Ctrl**) and 1 output (**O**). The *Ctrl* input always enters perpendicularly to the switch node. If the *Ctrl* input has a *True* token (i.e., a token with a value of 1), the **O** wire propogates the value on the **I** wire. Else, the 2 input tokens (**I**, **Ctrl**) are consumed, and no token is generated at the output (**O**).
- We define the *inverter node* in Figure 2 to have 1 input (**I**) and 1 output (**O**). The node negates the input token (i.e., O = !I).
- We define the *TF node* in Figure 3 to have 3 inputs (I_F, I_T, **Ctrl**) and 1 output (**O**). When **Ctrl** is set to True, **O** takes I_T. When **Ctrl** is set to False, **O** takes I_F.
- The $\geq$ node outputs True only when the left input is greater than or equal to the right input.
- The +1 node outputs the input plus one.
- The + node outputs the sum of the two inputs.
- A node generates an output token when tokens exist at *every* input, and *all* input tokens are consumed.
- Where a single wire splits into multiple wires, the token travelling on the wire is replicated to all wires.

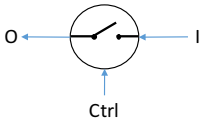


Figure 1: Switch Node

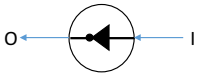


Figure 2: Inverter Node

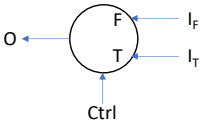


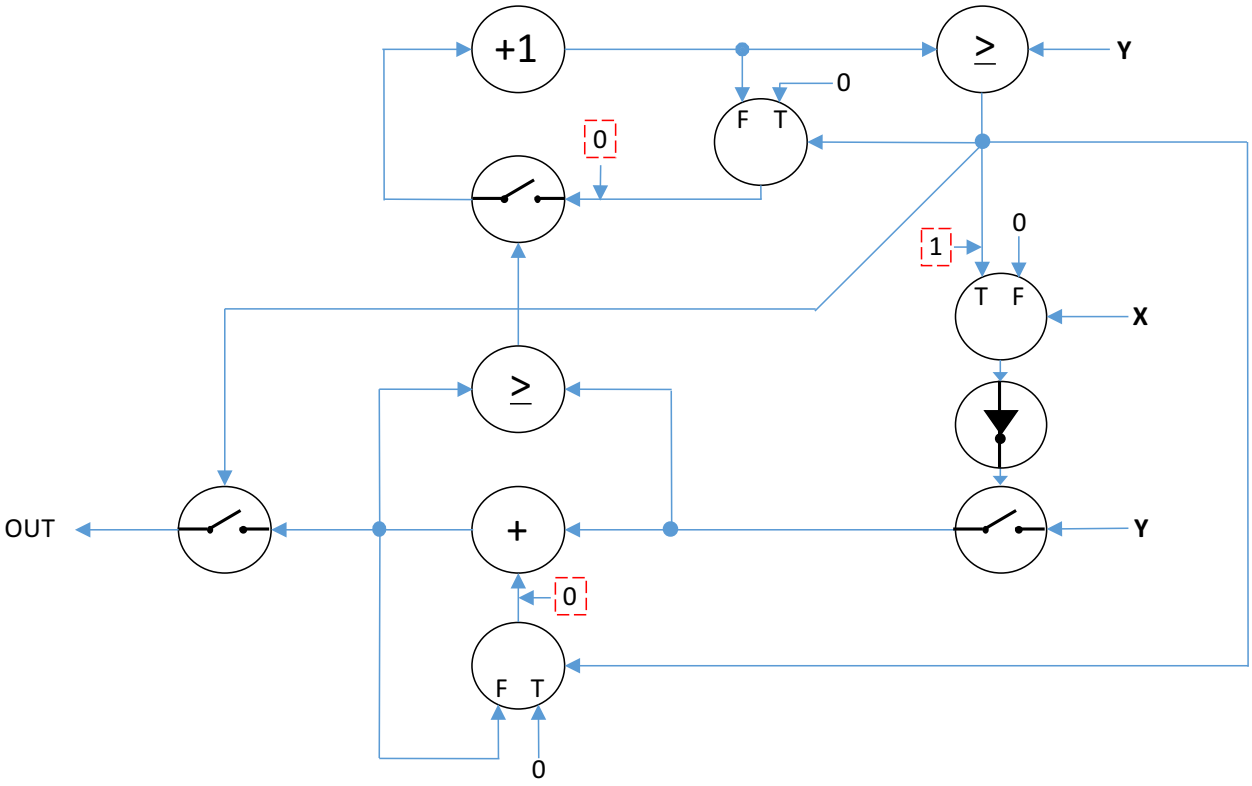
Figure 3: TF Node

Consider the dataflow graph on the following page. Numbers in dashed boxes represent tokens (with the value indicated by the number) in the initial state. The **X** and **Y** inputs automatically produce tokens as soon as the previous token on the wire is consumed. The order of these tokens follows the pattern (*note, the following are all single digit values spaced appropriately for the reader to easily notice the pattern*):

X: 0 01 011 0111 01111
Y: 1 22 333 4444 55555

Consider the dataflow graph on the following page. Please clearly describe the sequence of tokens generated at the output (OUT).

1, 4, 9, 16, 25


+1
≥
Y
0
F T
0
1
0
T F
X
≥
OUT
+
Y
0
F T
0

4 Microarchitecture vs. ISA (I)

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 a given program correctly?

The ISA level is the interface a machine exposes to the software. The microarchitecture is the actual underlying implementation of the machine. Therefore, the microarchitecture and changes to the microarchitecture are transparent to the compiler/programmer (except in terms of performance), while changes to the ISA affect the compiler/programmer. The compiler does not 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:

Microarchitecture?	**ISA?**	**Attribute**
	✓	The machine does not have a subtract instruction
✓		The ALU of the machine does not have a subtract unit
	✓	The machine does not have condition codes
	✓	A 5-bit immediate can be specified in an ADD instruction
✓		It takes n cycles to execute an ADD instruction
	✓	There are 8 general purpose registers
✓		A 2-to-1 mux feeds one of the inputs to ALU
✓		The register file has one input port and two output ports

5 Microarchitecture vs. ISA (II)

A new CPU has two comprehensive user manuals available for purchase as shown in Table 1.

Manual Title	Cost	Description
`the_isa.pdf`	CHF 1 million	describes the ISA in detail
`the_microarchitecture.pdf`	CHF 10 million	describes the microarchitecture in detail

Table 1: Manual Costs

Unfortunately, the manuals are extremely expensive, and you can only afford one of the two. If both manuals might be useful, you would prefer the cheaper one.

For each of the following questions that you would like to answer, decide which manual is more likely to help. *Note: we will subtract 1 point for each* ***incorrect*** *answer.*

1. The latency of a branch predictor misprediction.

   1. `the_isa.pdf` [2. `the_microarchitecture.pdf`]

2. The size of a physical memory page.

   1. `the_isa.pdf` [2. `the_microarchitecture.pdf`]

3. The memory-mapped locations of exception vectors.

   [1. `the_isa.pdf`] 2. `the_microarchitecture.pdf`

4. The function of each bit in a programmable branch-predictor configuration register.

   [1. `the_isa.pdf`] 2. `the_microarchitecture.pdf`

5. The bit-width of the interface between the CPU and the L1 cache.

   1. `the_isa.pdf` [2. `the_microarchitecture.pdf`]

6. The number of pipeline stages in the CPU.

   1. `the_isa.pdf` [2. `the_microarchitecture.pdf`]

7. The order in which loads and stores are executed by a multi-core CPU.

   [1. `the_isa.pdf`] 2. `the_microarchitecture.pdf`

8. The memory addressing modes available for arithmetic operations.

   [1. `the_isa.pdf`] 2. `the_microarchitecture.pdf`

9. The program counter width.

   [1. `the_isa.pdf`] 2. `the_microarchitecture.pdf`

10. The number of cache sets at each level of the cache hierarchy.

    1. `the_isa.pdf` [2. `the_microarchitecture.pdf`]

6 Performance Evaluation (I)

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?

ISA A: $10 \frac{instructions}{cycle} * 500,000,000 \frac{cycle}{second} = 5000$ MIPS

- What is the performance in MIPS of the processor implementing ISA B?

ISA B: $2 \frac{instructions}{cycle} * 600,000,000 \frac{cycle}{second} = 1200$ MIPS

- Which is the higher performance processor: A B Don't know
  Briefly explain your answer.

Don't know.
The best compiled code for each processor may have a different number of instructions.

7 Performance Evaluation (II)

A multi-cycle processor $P1$ executes *load instructions* in **10 cycles**, *store instructions* in **8 cycles**, *arithmetic instructions* in **4 cycles**, and *branch instructions* in **4 cycles**. Consider an application A where 20% of all instructions are load instructions, 20% of all instructions are store instructions, 50% of all instructions are arithmetic instructions, and 10% of all instructions are branch instructions.

(a) What is the CPI of application A when executing on processor $P1$? Show your work.

$CPI = 0.2 \times 10 + 0.2 \times 8 + 0.5 \times 4 + 0.1 \times 4$
$CPI = 6$

(b) A new design of the processor doubles the clock frequency of $P1$. However, the latencies of the load, store, arithmetic, and branch instructions increase by 2, 2, 2, and 1 cycles, respectively. We call this new processor $P2$. The compiler used to generate instructions for $P2$ is the same as for $P1$. Thus, it produces the same number of instructions for program A. What is the CPI of application A when executing on processor $P2$? Show your work.

$CPI = 0.2 \times 12 + 0.2 \times 10 + 0.5 \times 6 + 0.1 \times 5$
$CPI = 7.9$

(c) Which processor is faster ($P1$ or $P2$)? By how much? Show your work.

$P2$ is 1.52× faster than $P1$.

Explanation.
$Execution\_Time\_P1 = instructions \times CPI_{P1} \times clock\_rate$
$Execution\_Time\_P2 = instructions \times CPI_{P2} \times \frac{clock\_rate}{2}$
$clock\_rate = \frac{1}{clock\_frequency}$

Assuming that $Execution\_Time\_P2 < Execution\_Time\_P1 \implies \frac{Execution\_Time\_P1}{Execution\_Time\_P2} > 1$. Thus:
$\implies \frac{instructions \times CPI_{P1} \times clock\_rate}{instructions \times CPI_{P2} \times \frac{clock\_rate}{2}}$
$\implies \frac{6 \times clock\_rate}{7.9 \times \frac{clock\_rate}{2}}$
$\implies \frac{6}{3.95}$
$\implies 1.52$

(d) There is some extra area available in the chip of processor $P1$, where extra hardware can fit. You can decide to include in your processor a faster branch execution unit or a faster memory device. The faster branch execution unit reduces the latency of branch instructions by a factor of 4. The memory device reduces the latency of the memory operations by a factor of 2. Which design do you choose? Show your work.

A faster memory device.

Explanation.
Application A executes 10% of branch operations and 40% of memory operations (load and stores).
By Amdahl's Law, we have:

$Speedup_{branch} = \frac{1}{(1-0.1)+\frac{0.1}{4}} = 1.08$
$Speedup_{memory} = \frac{1}{(1-0.4)+\frac{0.4}{2}} = 1.25$

Therefore, the new memory device provides more speedup than the faster branch execution unit, for this particular application.

Alternative Solution.
In case we decide to reduce the latency of the branch operations, the new CPI of processor $P1$ will be:
$CPI_{branch} = 0.2 \times 10 + 0.2 \times 8 + 0.5 \times 4 + 0.1 \times \frac{4}{4}$
$CPI_{branch} = 5.7$

In case we decide to reduce the latency of the memory operations, the new CPI of processor $P1$ will be:
$CPI_{memory} = 0.2 \times \frac{10}{2} + 0.2 \times \frac{8}{2} + 0.5 \times 4 + 0.1 \times 4$
$CPI_{memory} = 4.2$

Since $CPI_{memory} < CPI_{branch}$, improving the memory device will provide shorter cycles-per-instructions.

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`