CSC258 Week 9

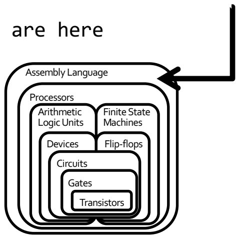
Logistics

- This week: Lab 7 is the last Logisim DE2 lab.
- Next week: Lab 8 will be assembly.
- For assembly labs you can work individually or in pairs. No matter how you do it, the important thing is that each individual of you learns from the lab.
- No prelab reports for assembly labs.
- You will be asked to submit your code to MarkUs at the end of the lab.
  - Submit as a group if you work in group (create a group on MarkUs and invite yuor partner).
  - As usual, do not plagiarize.
  - Make sure to remove your code from shared folders on the lab computers.

Quiz review

- Average: 88%
- When can A + B have overflow?
  - when they're both positive or both negative

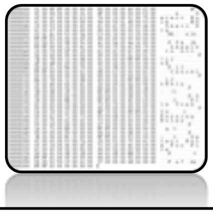
We are here



Programming the processor

- Things to learn:
  - Control unit signals to the datapath
  - Machine code instructions
  - Assembly language instructions
  - Programming in assembly language

Machine Code Instructions



Intro to Machine Code

- Machine code are the 32-bit binary instructions which the processor can understand (you can now understand, too)
- All programs (C, Java, Python) are eventually translated into machine code (by a compiler or interpreter).
- While executing, the instructions of the program are loaded into the instruction register one by one
- For each instruction loaded, the Control Unit reads the opcode and sets the signals to control the datapath, so that the processor works as instructed.

Assembly language

- Each line of assembly code corresponds to one line of 32-bit long machine code.
- Basically, assembly is a user-friendly way to write machine code.
- Example: `C = A + B`
  - Store `A` in `$t1`, `B` in `$t2`, `C` in `$t3`
  - Assembly language instruction:

```
add $t3, $t1, $t2
```

Note: There is a 1-to-1 mapping for all assembly code and machine code instructions!

  - Machine code instruction:

```
000000 01001 01010 01011 XXXXX 100000
```

Why learn assembly?

- You'll understand how your program ***really*** works.
- You'll understand your program's performance better by knowing its real "runtime".
- You'll understand how control flows (if / else / for / while) are implemented.
- You'll understand why eliminating if statements makes your code faster.
- You'll understand why pointer is such a natural concept for programming.
- You'll understand the cost of making function calls.
- You'll understand why stack can overflow
- You'll understand there is no "recursion" in the hardware, and how it's actually done.
- You'll understand why memory need to be managed.
- You'll understand why people spend so much time creating operating systems.
- You'll appreciate more the constructs in high-level programming languages.
- And much more…

And, you'll be able to read this book.

Donald Knuth "The Art of Computer Programming"

"All algorithms in this book are written in assembly for clarity."

About register names

- In machine code with have register `0` to register `31`, specified by 5 bits of the instruction.
- In assembly we have names like `$t1`, `$t2`, `$s1`, `$v0`, etc.
- What's the relation between these two?

Machine code + registers

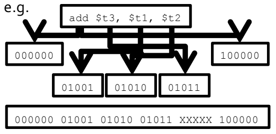
- MIPS is register-to-register.
  - Every operation operates on data in registers.
- MIPS provides 32 registers.
  - Several have special values (conventions):
    - Register `0` (`$zero`): value `0` -- always.
    - Register `1` (`$at`): reserved for the assembler.
    - Registers `2-3` (`$v0`, `$v1`): return values
    - Registers `4-7` (`$a0-$a3`): function arguments
    - Registers `8-15`, `24-25` (`$t0-$t9`): temporaries
    - Registers `16-23` (`$s0-$s7`): saved temporaries
    - Registers `28-31` (`$gp`, `$sp`, `$fp`, `$ra`): memory and function support
    - Registers `26-27`: reserved for OS kernel
  - Also three special registers (PC, HI, LO) that are not directly accessible.
    - HI and LO are used in multiplication and division, and have special instructions for accessing them.

$v0, $t2, $a3, etc are the registers' nicknames in assembly

Technically you can use any register for anything, but this is the convention

Translate assembly to machine code

- When writing machine code instructions (or interpreting them), we need to know which register values to encode (or decode).
- e.g.



Machine code details

- Things to note about machine code:
  - R-type instructions have an opcode of `000000`, with a 6-bit function listed at the end.
  - Although we specify "don't care" bits as `X` values, the assembly language interpreter always assigns them to some value (like `0`)
  - In exams, we want you to write X instead of `0`, to show that you know we don't care those bits

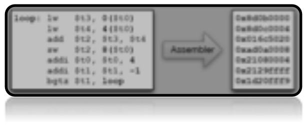
Try this at home

Below is the content of an executable file "mystery.exe", what does this program do?

```
1000 1110 0000 1000 0101 1010 1111 0001
1000 1110 0010 1001 1101 0010 0011 0010
0000 0001 0000 1001 0101 0000 0010 0000
1000 1110 0100 1011 1111 0011 0011 0111
0000 0000 0000 1100 0011 0001 0000 0000
0000 0010 0110 1010 1010 0000 0010 0010
1010 1101 1101 0100 0000 1111 0101 1010
```

Now you can totally program an executable like this (don't even need a compiler).

Assembly Language Instructions

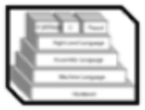


```
loop: lw   $t3, 0($t0)
      lw   $t4, 4($t0)
      add  $t2, $t3, $t4
      sw   $t2, 8($t0)
      addi $t0, $t0, 4
      addi $t1, $t1, -1
      bgtz $t1, loop
```

```
0x8d0b0000
0x8d0c0004
0x016c5020
0xad0a0008
0x21080004
0x2129ffff
0x1d20fff9
```

Assembly language

- Assembly language is the lowest-level language that you'll ever program in.
- Many compilers translate their high-level program commands into assembly commands, which are then converted into machine code and used by the processor.
- Note: There are multiple types of assembly language, especially for different architectures!



Trivia

The thing that converts assembly code to executable is NOT called a compiler.

It's called an assembler, because there is no fancy complication needed, it just assembles the lines!

You should be able to write one easily (e.g., for the processor created in Lab 7)

A little about MIPS

- MIPS
  - Short for Microprocessor without Interlocked Pipeline Stages
    - A type of RISC (Reduced Instruction Set Computer) architecture.
  - Provides a set of simple and fast instructions
    - Compiler translates instructions into 32-bit instructions for instruction memory.
    - Complex instructions are built out of simple ones by the compiler and assembler.

The layout of assembly code

Code sectioning syntax: example

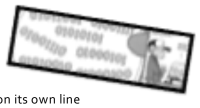
```
.data
A:      .space   400      # array of 100 integers
B:      .space   400      # array of 100 integers

.text
main:   add $t0, $zero, $zero     # load “0” into $t0
        addi $t1, $zero, 400      # load “400" into $t1
        addi $t9, $zero, B        # store address of B
        addi $t8, $zero, A        # store address of A

loop:   add $t4, $t8, $t0  # $t4 = addr(A) + i
        add $t3, $t9, $t0  # $t3 = addr(B) + i
        lw $s4, 0($t3)     # $s4 = B[i]
        addi $t6, $s4, 1   # $t6 = B[i] + 1
        sw $t6, 0($t4)     # A[i] = $t6
        addi $t0, $t0, 4   # $t0 = $t0++
        bne $t0, $t1, loop # branch back if $t0<400
end:
```

Code sectioning syntax

- `.data`
  - Indicates the start of the data declarations.
- `.text`
  - Indicates the start of the program instructions.
- `main:`
  - The initial line to run when executing the program.
- You can create other labels as needed.

MIPS Instructions

- Things to note about MIPS instructions:
  - Instruction are written as: `<instr> <parameters>`
  - Each instruction is written on its own line
  - All instructions are 32 bits (4 bytes) long
  - Instruction addresses are measured in **bytes**, starting from the instruction at address 0.
- The following tables show the most common MIPS instructions, the syntax for their parameters, and what operation they perform.



Arithmetic instructions

Instruction	Opcode/Function	Syntax	Operation
`add`	100000	$d, $s, $t	$d = $s + $t
`addu`	100001	$d, $s, $t	$d = $s + $t
`addi`	001000	$t, $s, i	$t = $s + SE(i)
`addiu`	001001	$t, $s, i	$t = $s + SE(i)
`div`	011010	$s, $t	lo = $s / $t; hi = $s % $t
`divu`	011011	$s, $t	lo = $s / $t; hi = $s % $t
`mult`	011000	$s, $t	hi:lo = $s * $t
`multu`	011001	$s, $t	hi:lo = $s * $t
`sub`	100010	$d, $s, $t	$d = $s - $t
`subu`	100011	$d, $s, $t	$d = $s - $t

Note: “hi” and “lo” refer to the high and low bits referred to in the register slide. “SE” = “sign extend”.

ALU instructions

- Note that for ALU instruction, most are R-type instructions.
  - The six-digit codes in the tables are therefore the function codes (opcodes are `000000`).
  - Exceptions are the I-type instructions (`addi`, `andi`, `ori`, etc.)
- Not all R-type instructions have an I-type equivalent.
  - RISC architectures dictate that an operation doesn't need an instruction if it can be performed through multiple existing operations.
  - Example: `divi $t0, 42  can be done by`
  - `addi $t1, $zero, 42`
  - `div $t0 $t1`

Logical instructions

Instruction	Opcode/Function	Syntax	Operation
`and`	100100	$d, $s, $t	$d = $s & $t
`andi`	001100	$t, $s, i	$t = $s & ZE(i)
`nor`	100111	$d, $s, $t	$d = ~($s | $t)
`or`	100101	$d, $s, $t	$d = $s | $t
`ori`	001101	$t, $s, i	$t = $s | ZE(i)
`xor`	100110	$d, $s, $t	$d = $s ^ $t
`xori`	001110	$t, $s, i	$t = $s ^ ZE(i)

Note: ZE = zero extend (pad upper bits with 0 value).

Shift instructions

Instruction	Opcode/Function	Syntax	Operation
`sll`	000000	$d, $t, a	$d = $t << a
`sllv`	000100	$d, $t, $s	$d = $t << $s
`sra`	000011	$d, $t, a	$d = $t >> a
`srav`	000111	$d, $t, $s	$d = $t >> $s
`srl`	000010	$d, $t, a	$d = $t >>> a
`srlv`	000110	$d, $t, $s	$d = $t >>> $s

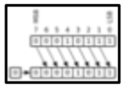
Note: `srl` = "shift right logical", and `sra` = "shift right arithmetic".
The "`v`" denotes a variable number of bits, specified by `$s`.

Logic shift vs Arithmetic shift

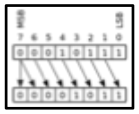
Left shift: same, fill empty spot (lower bits) with zeros
(that's why we have `sll` but no `sla`)

Right shift: different

- Logic shift fills empty spot(higher bits) with zeros
- Arithmetic shift fills empty spot (higher bits) with the MSB of the original number.



Logic



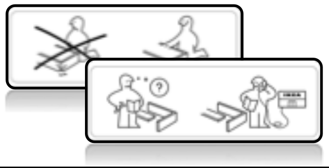
Arithmetic

Data movement instructions

Instruction	Opcode/Function	Syntax	Operation
`mfhi`	010000	$d	$d = hi
`mflo`	010010	$d	$d = lo
`mthi`	010001	$s	hi = $s
`mtlo`	010011	$s	lo = $s

- These are instructions for operating on the HI and LO registers described earlier.

Time for more instructions!



Flow control: Branch and loop

Control flow in assembly

- Not all programs follow a linear set of instructions.
  - Some operations require the code to branch to one section of code or another (if/else).
  - Some require the code to jump back and repeat a section of code again (for/while).
- For this, we have labels on the left-hand side that indicate the points that the program flow might need to jump to.
  - References to these points in the assembly code are resolved at compile time to offset values for the program counter.

Code sectioning syntax: example

```
.data
A:       .space   400      # array of 100 integers
B:       .space   400      # array of 100 integers

.text
main:    add $t0, $zero, $zero     # load "0" into $t0
         addi $t1, $zero, 400      # load "400" into $t1
         addi $t9, $zero, B        # store address of B
         addi $t8, $zero, A        # store address of A

loop:    add $t4, $t8, $t0  # $t4 = addr(A) + i
         add $t3, $t9, $t0  # $t3 = addr(B) + i
         lw $s4, 0($t3)     # $s4 = B[i]
         addi $t6, $s4, 1   # $t6 = B[i] + 1
         sw $t6, 0($t4)     # A[i] = $t6
         addi $t0, $t0, 4   # $t0 = $t0++
         bne $t0, $t1, loop # branch back if $t0<400
end:
```

Branch instructions

Instruction	Opcode/Function	Syntax	Operation
`beq`	000100	$s, $t, label	if ($s == $t) pc += i << 2
`bgtz`	000111	$s, label	if ($s > 0) pc += i << 2
`blez`	000110	$s, label	if ($s <= 0) pc += i << 2
`bne`	000101	$s, $t, label	if ($s != $t) pc += i << 2

- Branch operations are key when implementing if statements and while loops.
- The labels are memory locations, assigned to each label at compile time.
  - Note: `i` is calculated as `(label - (current PC + 4)) >> 2`

Comparison instructions

Instruction	Opcode/Function	Syntax	Operation
`slt`	101010	$d, $s, $t	$d = ($s < $t)
`sltu`	101001	$d, $s, $t	$d = ($s < $t)
`slti`	001010	$t, $s, i	$t = ($s < SE(i))
`sltiu`	001001	$t, $s, i	$t = ($s < SE(i))

Note: Comparison operation stores a one in the destination register if the less-than comparison is true, and stores a zero in that location otherwise.

Note: Real vs Pseudo instructions

What we list in the slides are all real instructions, i.e., each one has an **opcode** corresponding to it.

There are some pseudo-instructions, which don't have their own opcode, but is implemented using real instructions; they are provided for coding convenience.

For example:

- `bge $t0,$t1,Label` is actually
- `slt $t2,$t0,$t1; beq $t2,$zero,Label`

Jump instructions

Instruction	Opcode/Function	Syntax	Operation
`j`	000010	label	pc += i << 2
`jal`	000011	label	$31 = pc; pc += i << 2
`jalr`	001001	$s	$31 = pc; pc = $s
`jr`	001000	$s	pc = $s

- `jal` = "jump and link".
  - Register $31 (aka $ra) stores the address that's used when returning from a subroutine.
- Note: `jr` and `jalr` are not j-type instructions.

If/Else statements in MIPS

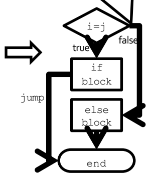
```
if ( i == j )
  i++;
else
  j--;
j += i;
```

- Strategy for if/else statements:
  - Test condition, and jump to `if` logic block whenever condition is true.
  - Otherwise, perform `else` logic block, and jump to first line after `if` logic block.
- A flowchart can be helpful here

If statement

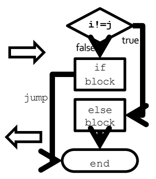
Only problem: branch instructions jump on TRUE instead of FALSE, so negate the checked condition to i != j

```
if ( i == j )
  i++;
else
  j--;
j += i;
```



If statement flowcharts

```
if ( i == j )
  i++;
else
  j--;
j += i;
```



```
#  $t1 = i, $t2 = j
main:    bne  $t1, $t2, ELSE
         addi $t1, $t1, 1
         j END
ELSE:    addi $t2, $t2, -1
END:     add $t2, $t2, $t1
```

Translated if/else statements

```
#  $t1 = i, $t2 = j
main:    bne  $t1, $t2, ELSE    # branch if ! ( i == j )
         addi $t1, $t1, 1      # i++
         j END                 # jump over ELSE
ELSE:    addi $t2, $t2, -1     # j--
END:     add $t2, $t2, $t1     # j += i
```

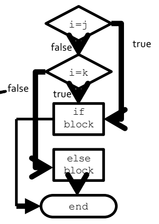
- If we change BNE to BEQ, then we also need to swap the IF and ELSE blocks

```
#  $t1 = i, $t2 = j
main:    beq  $t1, $t2, IF     # branch if ( i == j )
         addi $t2, $t2, -1    # j--
         j END                # jump over IF
IF:      addi $t1, $t1, 1     # i++
END:     add $t2, $t2, $t1    # j += i
```

Multiple if conditions

```
if ( i == j || i == k )
   i++ ;  // if-body
else
   j-- ;  // else-body
j = i + k ;
```

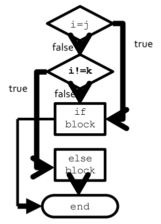
Branch on FALSE!



Multiple if conditions

```
if ( i == j || i == k )
   i++ ;  // if-body
else
   j-- ;  // else-body
j = i + k ;
```

```
#  $t1 = i, $t2 = j, $t3 = k
main:  beq $t1, $t2, IF
       bne $t1, $t3, ELSE
IF:    addi $t1, $t1, 1
       j END
ELSE:  addi $t2, $t2, -1
END:   add $t2, $t1, $t3
```



Loops

Loops in MIPS (while loop)

- Example of a simple loop, in assembly:

```
#  $t0 = i, $t1 = n
main:    add $t0, $zero, $zero   # i = 0
         addi $t1, $zero, 100    # n = 100
START:   beq $t0, $t1, END       # if i == n, END
         addi $t0, $t0, 1        # i++
         j START
END:
```

- ...which is the same as saying (in C):

```
while (i != 100) {
   i++;
}
```

For loop

```
for ( <init> ; <cond> ; <update> ) {
    <for body>
}
```

- For loops (such as above) are usually implemented with the following structure:

```
main:    <init>
START:   if (!<cond>) branch to END
         <for-body>
UPDATE:  <update>
         jump to START
END:
```

Exercise:

```
j = 0
for ( _____ ; _____ ; _____ )
{
     j = j + i;
}
```

```
#  $t0 = i, $t1 = j
main:    add $t1, $zero, $zero        # set j = 0
         add $t0, $zero, $zero        # set i = 0
         addi $t9, $zero, 100         # set $t9 to 100
START:   beq $t0, $t9, EXIT           # branch if i==100
         add $t1, $t1, $t0            # j = j + i
UPDATE: addi $t0, $t0, 1              # i++
         j START
EXIT:
```

Answer

```
j = 0
for ( i=0 ; i!=100 ; i++ )
{
     j = j + i;
}
```

- This translates to:

```
#  $t0 = i, $t1 = j
main:    add $t1, $zero, $zero        # set j = 0
         add $t0, $zero, $zero        # set i = 0
         addi $t9, $zero, 100         # set $t9 to 100
START:   beq $t0, $t9, EXIT           # branch if i==100
         add $t1, $t1, $t0            # j = j + i
UPDATE: addi $t0, $t0, 1              # i++
         j START
EXIT:
```

- `while` loops are the same, without the initialization and update sections.

Another exercise

- Fibonacci sequence:
  - How would you convert this into assembly?

```
int fib(void) {
   int n = 10;
   int f1 = 1, f2 = -1;

   while (n != 0) {
      f1 = f1 + f2;
      f2 = f1 - f2;
      n = n - 1;
   }
   return f1;
}
```

Assembly code example

- Fibonacci sequence in assembly code:

```
int fib(void) {
   int n = 10;
   int f1 = 1, f2 = -1;

   while (n != 0) {
      f1 = f1 + f2;
      f2 = f1 - f2;
      n = n - 1;
   }
   return f1;
}
```

```
# fib.asm
# register usage: $t3=n, $t4=f1, $t5=f2
# RES refers to memory address of result
FIB:  addi $t3, $zero, 10     # initialize n=10
      addi $t4, $zero, 1      # initialize f1=1
      addi $t5, $zero, -1     # initialize f2=-1
LOOP: beq $t3, $zero, END     # done loop if n==0
      add $t4, $t4, $t5       # f1 = f1 + f2
      sub $t5, $t4, $t5       # f2 = f1 - f2
      addi $t3, $t3, -1       # n = n - 1
      j LOOP                  # repeat until done
END:  sb $t4, RES             # store result
```

Making an assembly program

- Assembly language programs typically have structure similar to simple Python or C programs:
  - They set aside registers to store data.
  - They have sections of instructions that manipulate this data.
- It is always good to decide at the beginning which registers will be used for what purpose!
  - More on this later ☺