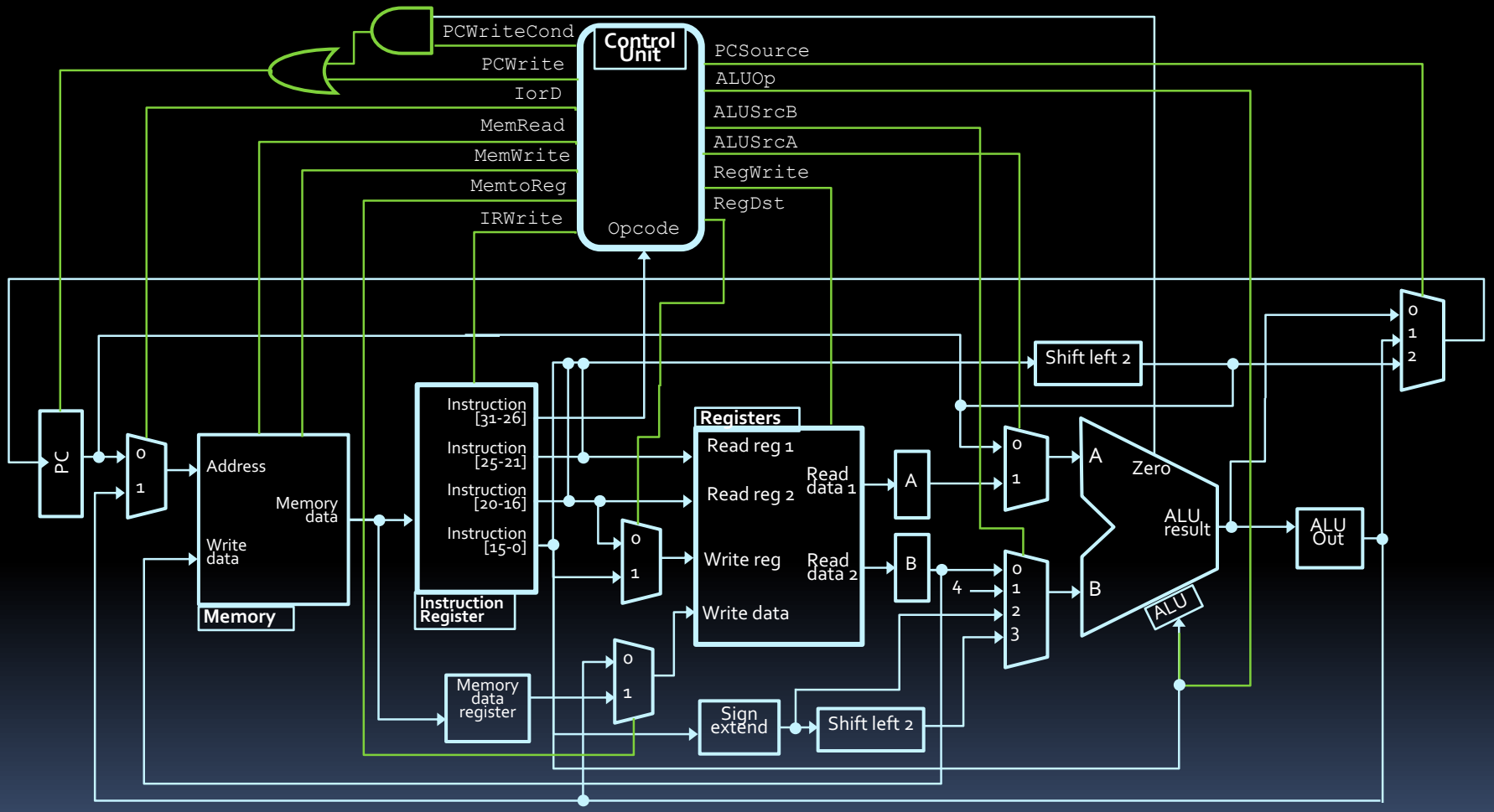
Lecture 8: Intro to Assembly Programming

The MIPS Microprocessor



Intro to Machine Code

- Now that we have a processor, operations are performed by:
  - The instruction register sends instruction components to the control unit.
  - The control unit decodes instruction according to the opcode in the first 6 bits.
  - The control unit sending a sequence of signals to the rest of the processor.
- Only questions remaining:
  - Where do these instructions come from?
  - How are they provided to the instruction memory?

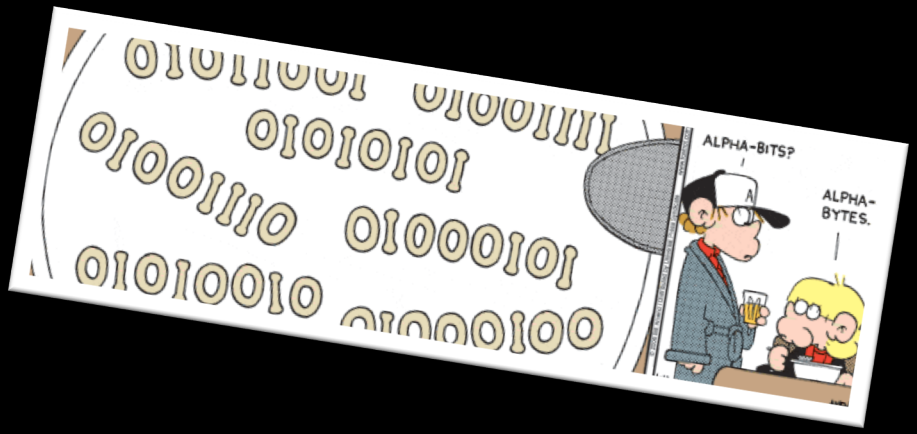
Machine Code Instructions

```
00000000  01 00 FF FF 00 00 00 00  00 00 00 00 40 00 CC 80  ............@.
00000010  0C 00 00 00 00 00 26 01  8F 00 00 00 00 00 53 00  ......&.......S.
00000020  65 00 6C 00 65 00 63 00  74 00 20 00 52 00 75 00  e.l.e.c.t. .R.u.
00000030  6C 00 65 00 00 00 08 00  00 00 00 01 4D 00 53 00  l.e.........M.S.
00000040  20 00 53 00 68 00 65 00  6C 00 6C 00 20 00 44 00   .S.h.e.l.l. .D.
00000050  6C 00 67 00 00 00 00 00  00 00 00 00 00 02 00 00  l.g.............
00000060  03 01 A1 50 53 00 3A 00  C3 00 36 00 32 25 00 00  ...PS.:...6.2%..
00000070  FF FF 83 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000080  03 00 01 50 0E 00 56 00  41 00 0A 00 4A 26 00 00  ...P..V.A...J&..
00000090  FF FF 80 00 26 00 41 00  70 00 70 00 6C 00 79 00  ....&.A.p.p.l.y.
000000a0  20 00 74 00 6F 00 20 00  61 00 6C 00 6C 00 00 00   .t.o. .a.l.l...
000000b0  00 00 00 00 00 00 00 00  00 00 00 00 01 00 01 50  ...............P
000000c0  7E 00 7D 00 32 00 0E 00  01 00 00 00 FF FF 80 00  ~.}.2...........
000000d0  4F 00 4B 00 00 00 00 00  00 00 00 00 00 00 00 00  O.K.............
000000e0  00 00 01 50 B4 00 7D 00  32 00 0E 00 02 00 00 00  ...P..}.2.......
000000f0  FF FF 80 00 43 00 61 00  6E 00 63 00 65 00 6C 00  ....C.a.n.c.e.l.
00000100  00 00 00 00 00 00 00 00  00 00 00 00 00 00 01 50  ...............P
00000110  EA 00 7D 00 32 00 0E 00  09 00 00 00 FF FF 80 00  ..}.2...........
00000120  26 00 48 00 65 00 6C 00  70 00 00 00 00 00 00 00  &.H.e.l.p.......
00000130  00 00 00 00 00 00 00 00  80 08 81 50 0E 00 3A 00  ...........P..:.
00000140  3B 00 0E 00 2F 25 00 00  FF FF 81 00 00 00 00 00  ;.../%..........
00000150  00 00 00 00 00 00 00 00  00 00 02 50 0E 00 30 00  ...........P..0.
00000160  1E 00 08 00 EE 25 00 00  FF FF 82 00 46 00 69 00  .....%......F.i.
00000170  6C 00 65 00 20 00 54 00  79 00 70 00 65 00 00 00  l.e. .T.y.p.e...
00000180  00 00 00 00 00 00 00 00  00 00 00 00 00 00 02 50  ...............P
00000190  54 00 30 00 2C 00 08 00  EF 25 00 00 FF FF 82 00  T.0.,....%......
000001a0  50 00 61 00 72 00 73 00  69 00 6E 00 67 00 20 00  P.a.r.s.i.n.g. .
000001b0  52 00 75 00 6C 00 65 00  73 00 00 00 00 00 00 00  R.u.l.e.s.......
000001c0  00 00 00 00 00 00 00 00  07 00 00 50 06 00 07 00  ...........P....
000001d0  1A 01 71 00 ED 25 00 00  FF FF 80 00 00 00 00 00  ..q..%..........
000001e0  00 00 00 00 00 00 00 00  00 00 02 50 0E 00 11 00  ...........P....
000001f0  3E 00 08 00 EC 25 00 00  FF FF 82 00 53 00 65 00  >....%......S.e.
00000200  6C 00 65 00 63 00 74 00  20 00 52 00 75 00 6C 00  l.e.c.t. .R.u.l.
00000210  65 00 20 00 46 00 6F 00  72 00 20 00 46 00 69 00  e. .F.o.r. .F.i.
00000220  6C 00 65 00 00 00 00 00  00 00 00 00 00 00 00 00  l.e.............
00000230  80 08 81 50 0E 00 1B 00  08 01 0E 00 EB 25 00 00  ...P.........%..
00000240  FF FF 81 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000250  00 00 02 50 19 00 61 00  37 00 08 00 6B 26 00 00  ...P..a.7...k&..
00000260  FF FF 82 00 00 00 00 00  |                       ........
```

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.

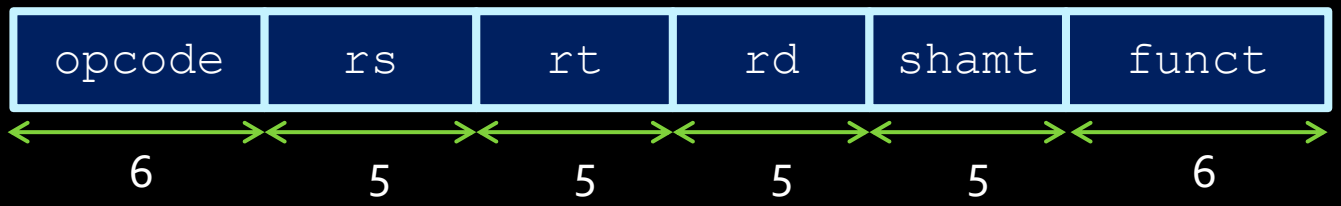
MIPS Memory and Instructions



- All memory is addressed in bytes.
- Instruction addresses are measured in bytes, starting from the instruction at address 0.
- All instructions are 32 bits (4 bytes) long
- Therefore:
  all instruction addresses are divisible by 4.

Recall: MIPS instruction types

- R-type:



- I-type:

- J-type:

MIPS Registers

- In MIPS is register-to-register (a.k.a. load-store) architecture
  - Source, destination of ALU operations are registers.
- MIPS provides 32 registers.
  - Some have special values:
    - Register 0 ($zero): value 0 – always (writes to it are discarded)
    - Register 1 ($at): reserved for the assembler.
    - Registers 28-31 ($gp, $sp, $fp, $ra): memory and function support
    - Registers 26-27: reserved for OS kernel
  - Some are used by programs as functions parameters:
    - Registers 2-3 ($v0, $v1): return values
    - Registers 4-7 ($a0-$a3): function arguments
  - Some are used by programs to store values:
    - Registers 8-15, 24-25 ($t0-$t9): temporaries
    - Registers 16-23 ($s0-$s7): saved temporaries
  - 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.

Assembly Language Introduction

```
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
```

Assembler

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

Assembly vs Machine Code

- Each processor type has its own language for representing 32-bit instructions as user-readable code words.
- <u>Example</u>: `C = A + B`
  - Assume `A` is stored in `$t1`, `B` in `$t2`, `C` in `$t3`.
  - Assembly language instruction:

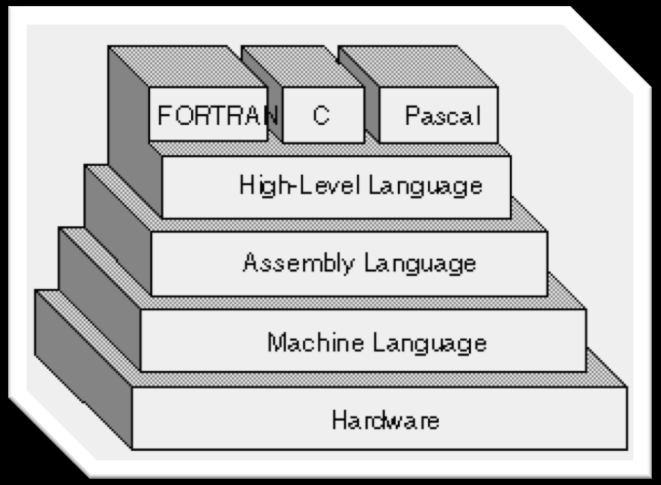
```
add $t3, $t1, $t2
```

  - Machine code instruction:

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

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

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.
- <u>Note:</u> There are multiple types of assembly language, especially for different architectures!

Why learn assembly?

- Understand how code really works
- Better analyze code (runtime, control flows, pointers, stack overflows)
- Make you appreciate constructs of high level languages
- Connect your high level programming knowledge to hardware
- It's on the exam…

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".**

Assembly → Machine Code

Operation

```
t3 = t1 + t2;
```

Assembly

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

Instruction	Opcode/Function	Syntax	Operation
add	100000	$d, $s, $t	$d = $s + $t

R-type instruction!

000000	01001	01010	01011	XXXXX	100000

Although we specify "don't care" bits as X values, the assembler generally assigns some value (like 0).

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).

Assembly → Machine Code II

Operation

```
t2 = t1 & 42;
```

Assembly

```
andi $t2, $t1, 42
```

Instruction	Opcode/Function	Syntax	Operation
andi	001100	$t, $s, i	$t = $s & ZE(i)

I-type instruction!

001100	01001	01010	0000000000101010

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"
`sra` = "shift right arithmetic".
The "`v`" denotes a variable number of bits, specified by `$s`.
`a` is shift amount, and is stored in `shamt` when encoding the R-type machine code instructions.

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 (for multiplication and division)

ALU instructions in RISC

- Most ALU instructions 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 principle dictate that an operation doesn't need an instruction if it can be performed through multiple existing operations.
  - Example: `addi` + `div` → `divi`

Pseudoinstructions

- Move data from $t4 to $t5?
  - move $t5,$t4 →

```
add $t5,$t4,$zero
```

- Multiply and store in $s1?
  - mul $s1,$t4,$t5 →

```
mult $t4,$t5
mflo $s1
```

Time for a Break

Bill Watterson: Calvin & Hobbes

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 ☺

Time to write our first assembly program

Compute result = $a^2 + 2b + 10$

- Set up values in registers
  - a → $t0, b → $t1
  - temp → $t6
- temp = 10
- temp = temp + b
- temp = temp + b (again!)
- result = a*a
- result = result + temp

```
addi $t0, $zero, 7
addi $t1, $zero, 9

addi $t6, $zero, 10

add $t6, $t6, $t1
add $t6, $t6, $t1

mult $t0, $t0
mflo $t4
mfhi $t5

add $t4, $t4, $t6
```

Formatting Assembly Code

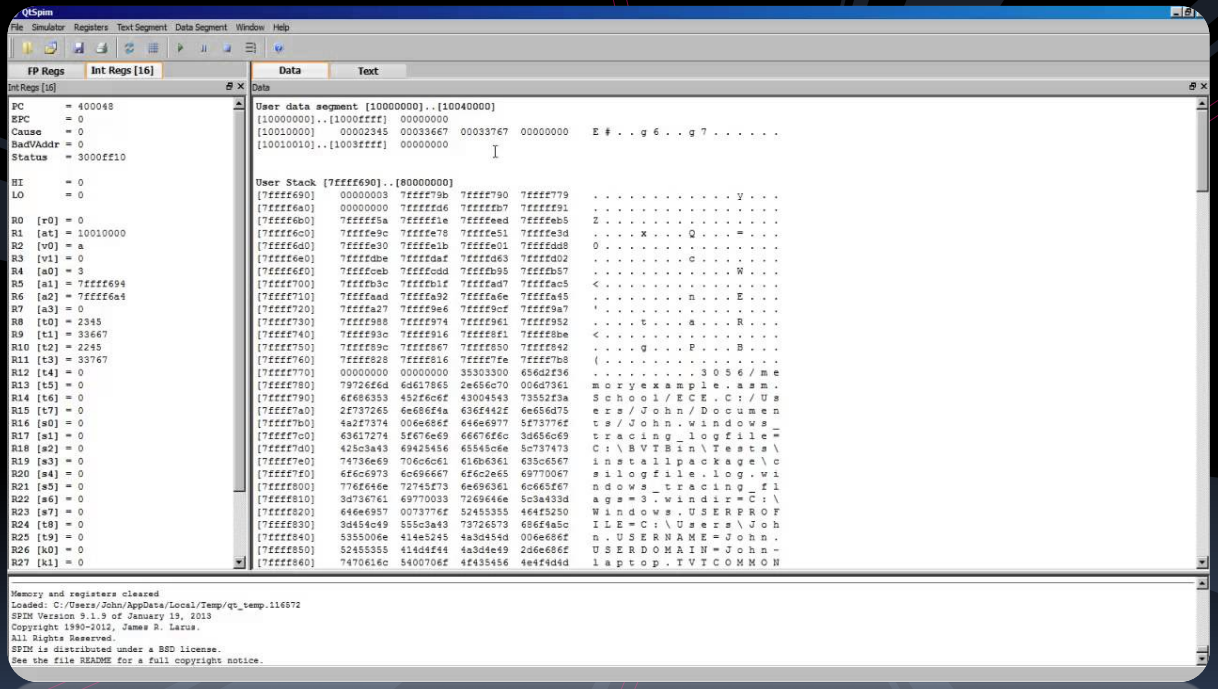
- Instruction are written as: `<instr> <parameters>`
- Each instruction is written on its own line
- 3 columns
  - Labels
  - Code
  - Comments
- Start with .text (we'll see other options later)
- First line of code to run = label: main

```
# Compute the following result: r = a^2 + 2b + 10

.text
# load up some values to test
main:  addi $t0, $zero, 7
       addi $t1, $zero, 9
# $t0 will be a,   $t1 will be b,   $t5:$t4  will be r
# $t6 will be temp
       addi $t6, $zero, 10  # add 10 to r
       add $t6, $t6, $t1    # then add b
       add $t6, $t6, $t1    # then add b again
       mult $t0, $t0        # multiply a * a
       mflo $t4             # move the low result of a^2
                            # into the low register of r
       mfhi $t5             # move the high result of a^2
                            # into the high register of r
       add $t4, $t4, $t6    # add the temporary value
                            # (2b + 10) to the low
                            # register of r
```

Simulating MIPS

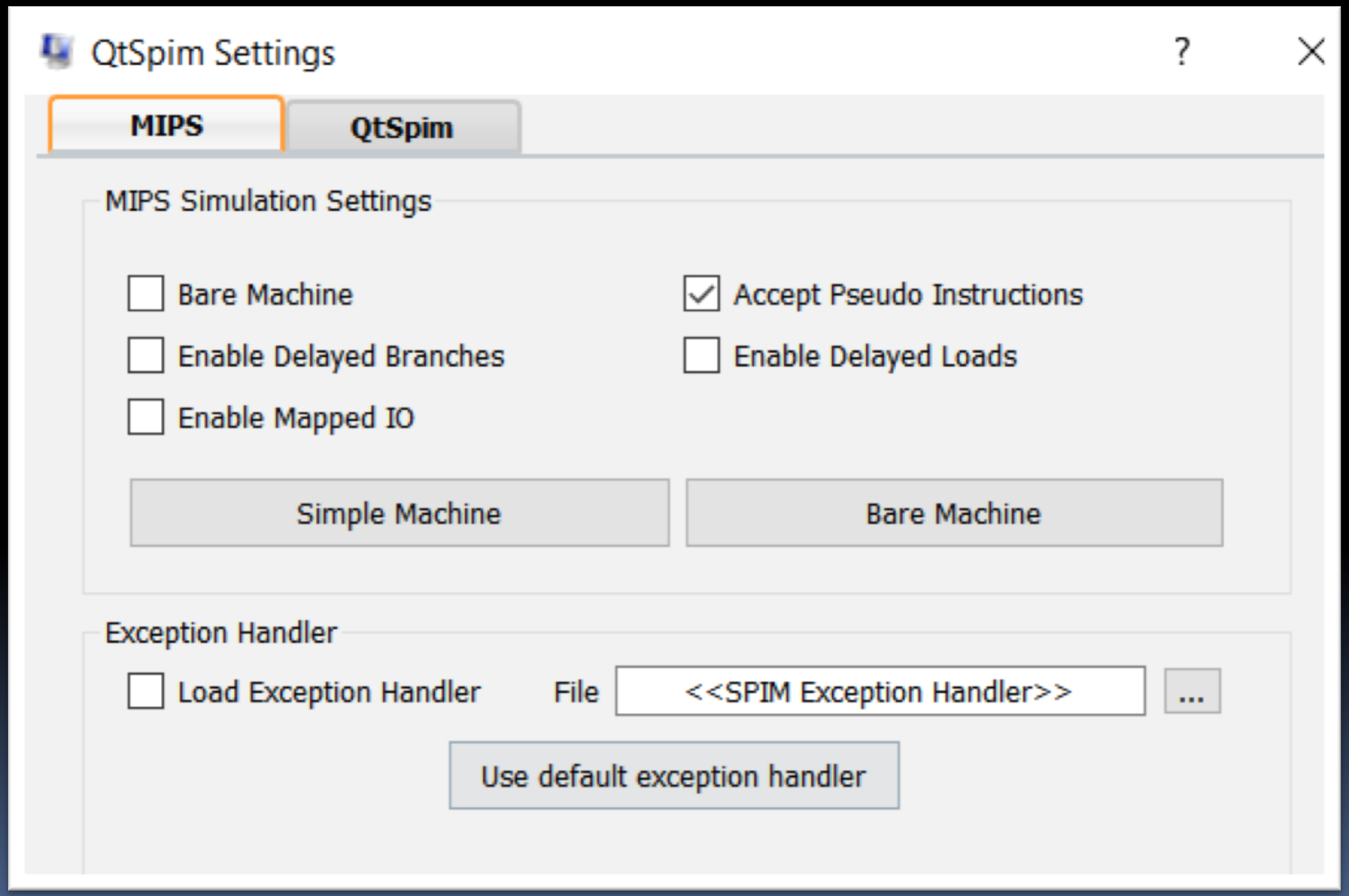
aka: QtSpim



QtSpim Simulator

- Link to download:
  - http://spimsimulator.sourceforge.net
- MIPS settings in the simulator:
  - From menu, Simulator → Settings
  - Important to not have "delayed branches" or "delayed loads" selected under Settings.
  - "Load exception handler" field should also be unchecked.
- You should view user code (Text Segment -> User text), no need for "kernel text"

QtSpim Settings



QtSpim – Quick How To

- Write a MIPS program (similar to the ones posted) in any text editor. Save it with `.asm` extension.
- In QtSpim select:
  - File -> Reinitialize and load a file
  - Single step through your program while observing (a) the Int Regs window and (b) the text window (user text).
    - As you step through, the highlighted instruction is the one about to be executed.

QtSpim Help => MIPS reference

- QtSpim help (Help -> View Help) contains
  - "Appendix A (Assemblers, Linkers, and the SPIM Simulator)" from *Patterson and Hennessey, Computer Organization and Design: The Hardware/Software Interface, Third Edition*
  - Useful reference for MIPS R2000 Assembly Language
    - Look at "Arithmetic and Logical Instructions".

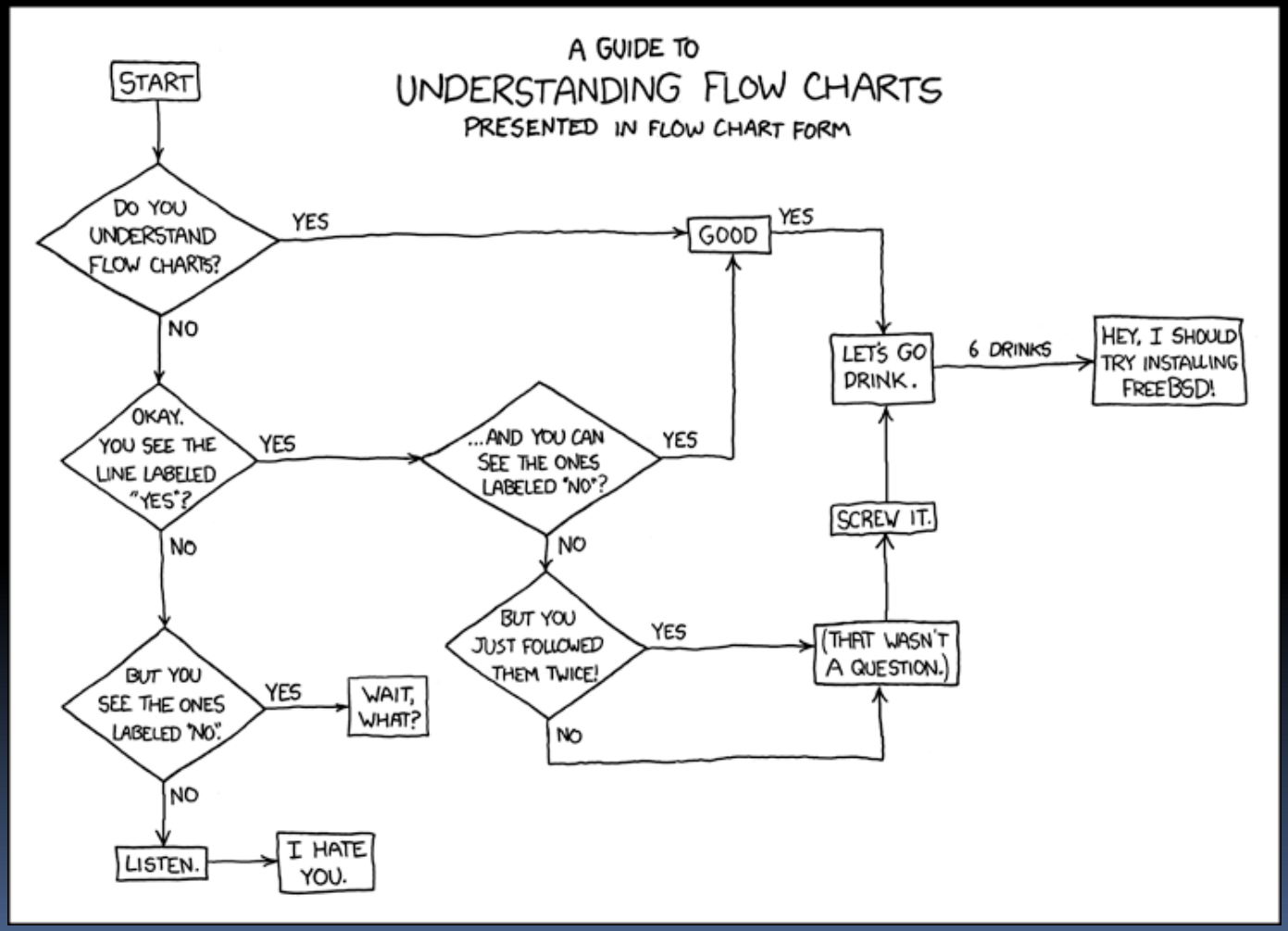
r = (2a + 5) * (7b)

```
.text
# $t0 = a, $t1 = b, $t4 = r
# $t7 = left side, $t8 = right side
main:  addi $t0, $zero, 7  # load up some values to test
       addi $t1, $zero, 9
# calculate left side
calc_left:   add $t7, $t0, $t0   # ls <- 2a
             addi $t7, $t7, 5    # ls <- ls + 5

# calculate right side
calc_right:  addi $t8, $zero, 7  # rs <- 7
             mult $t8, $t1       # multiply b * 7
             mflo $t8            # put result back into rs

# multiply left * right and put result into r
mulitply:    mult $t7, $t8
             mflo $t4
```

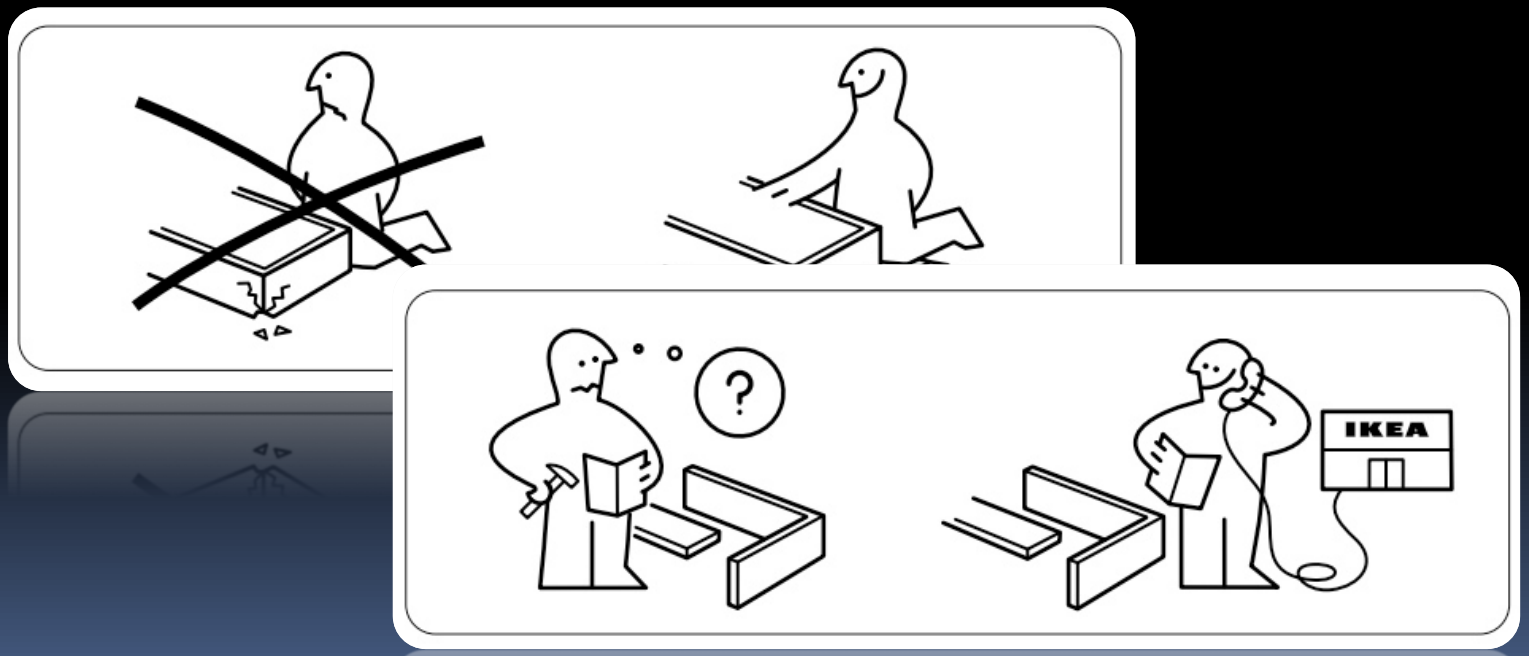
Control Flow



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 by the assembler at compile time to offset values for the program counter.

Time for more instructions!



Branch instructions

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

- Branch operations are key when implementing if statements and while loops.
- The labels are memory locations, assigned to each label at compile time.

Branch instructions

- How does a branch instruction work?

```
.text

main:   beq $t0, $t1, end  # check if $t0 == $t1
        ...                # if $t0 != $t1, then
        ...                 # execute these lines

 end:   ...                # if $t0 == $t1, then
        ...                # execute these lines
```

Branch instructions

- Alternate implementation using `bne`:

```
.text

main:   bne $t0, $t1, end  # check if $t0 == $t1
        ...                # if $t0 == $t1, then
        ...                 # execute these lines

 end:   ...                # if $t0 != $t1, then
        ...                # execute these lines
```

- Used to produce `if` statement behaviour.

Conditional Branch Terms

- When the branch condition is met, we say the branch is taken.
- When the branch condition is not met, we say the branch is not taken.
  - What is the next PC in this case?
    - It's the usual `PC+4`
- How far can a processor branch? Are there any constraints?

Jump instructions

Instruction	Opcode/Function	Syntax	Operation
`j`	000010	label	pc ← label
`jal`	000011	label	$ra = pc; pc ← label
`jalr`	001001	$s	$ra = 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.

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))

- "slt" = "Set Less Than"
- Comparison operation stores a one in the destination register if the less-than comparison is true, and stores a zero in that location otherwise.
- Signed: `0x8000000` is less than all numbers
- Unsigned: `0 - 0x7FFFFFFF` are less than `0x8000000`

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.

Translated if/else statements

```
#  $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
```

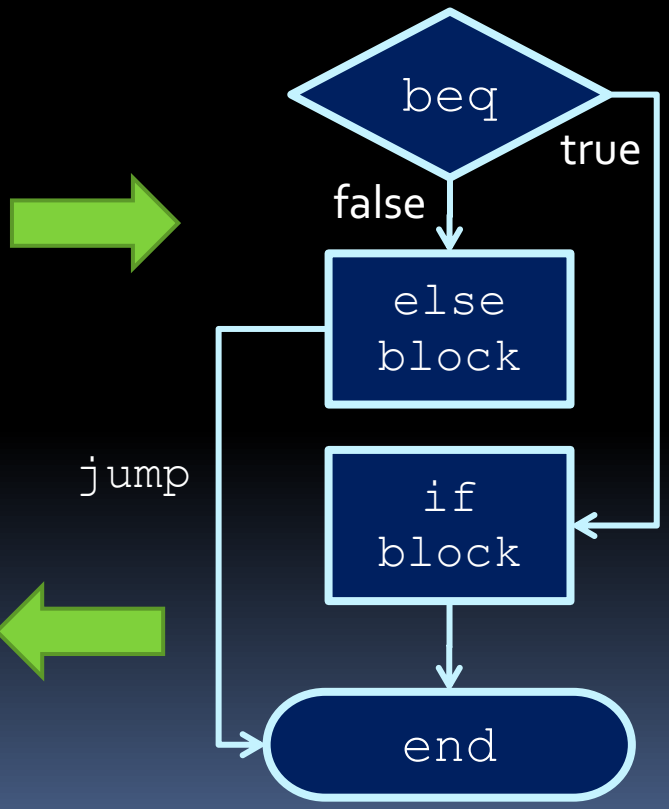
- Alternately, you can branch on the else condition first:

```
#  $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
```

A trick with if statements

- Use flow charts to help you sort out the control flow of the code:

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




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

Multiple Conditions Inside If

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

Multiple Conditions Inside If

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

- Branch statement for each condition:

```
#  $t1 = i, $t2 = j, $t3 = k
main:  beq $t1, $t2, IF      # cond1: branch if ( i == j )
       bne $t1, $t3, ELSE    # cond2: branch if ( i != k )
IF:    addi $t1, $t1, 1      # if (i==j|i==k) → i++
       j END                 # jump over else
ELSE:  addi $t2, $t2, -1     # else-body: j--
END:   add $t2, $t1, $t3     # j = i + k
```

Multiple if conditions

- How would this look if the condition changed?

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

```
#  $t1 = i, $t2 = j, $t3 = k
main:  bne $t1, $t2, ELSE   # cond1: branch if ( i != j )
       bne $t1, $t3, ELSE   # cond2: branch if ( i != k )
IF:    addi $t1, $t1, 1     # if (i==j|i==k) → i++
       j END                # jump over else
ELSE:  addi $t2, $t2, -1    # else-body: j--
END:   add $t2, $t1, $t3    # j = i + k
```

```
main:    add $t0, $zero, $zero
         addi $t1, $zero, 100
START:   beq $t0, $t1, END
         addi $t0, $t0, 1
         j START
END:
```

Loops in MIPS

- Example of a simple loop, in assembly:

```
main:     add $t0, $zero, $zero
          addi $t1, $zero, 100
START:    beq $t0, $t1, END
          addi $t0, $t0, 1
          j START
END:
```

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

```
int i = 0;
while (i < 100) {
   i++;
}
```

Loops in MIPS

```
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:
```

Loop example in MIPS

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

- This translates to:

```
#  $t0 = i, $t1 = j
main:    add $t0, $zero, $zero        # set i to 0
         add $t1, $zero, $zero        # set j to 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.

Homework

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

```
int n = 10;
int f1 = 1, f2 = 1;

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