# PennSim Overview

CIS 240

Fall 2012

Barry Rosenberg

# Downloading PennSim

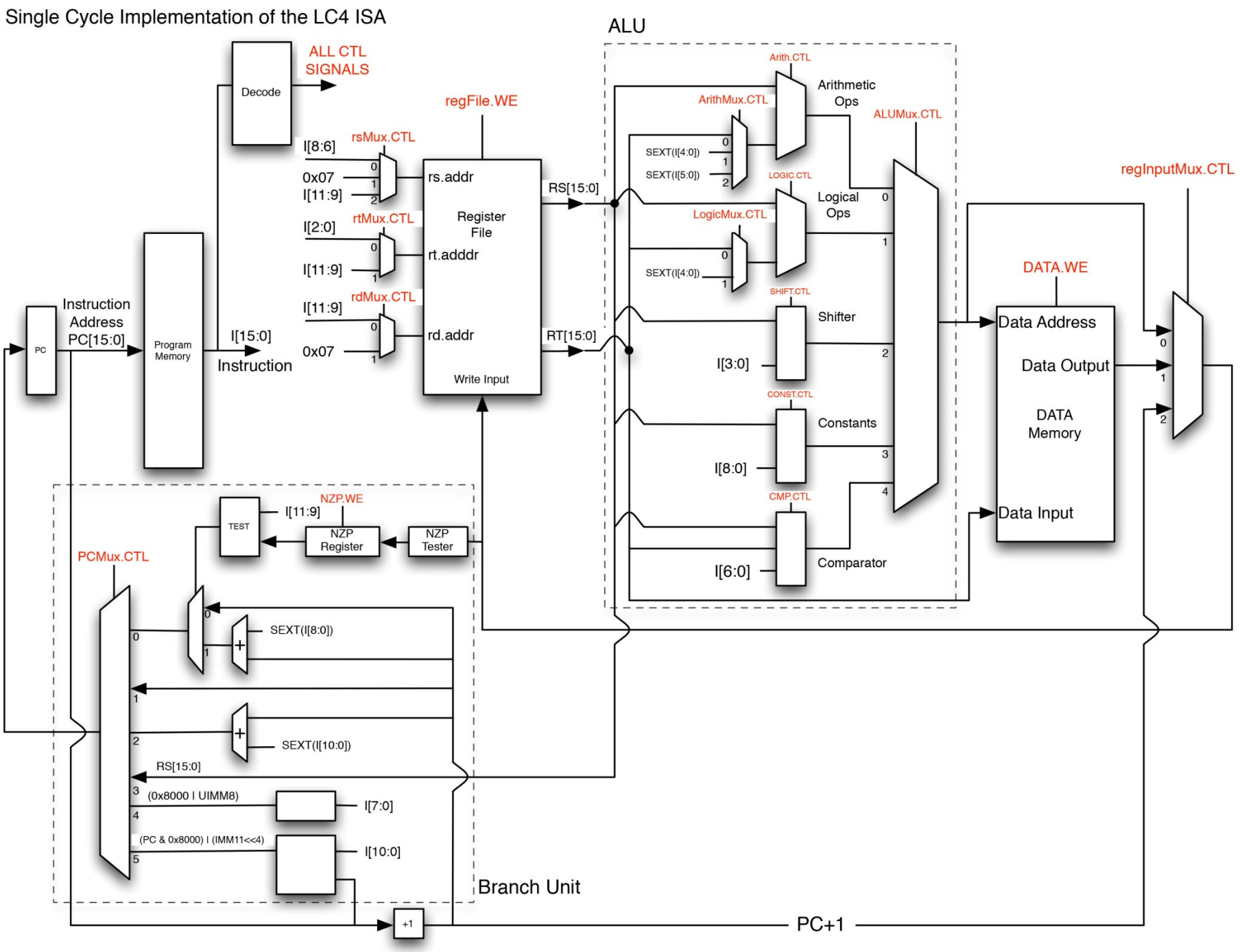
- PennSimStartGuide.zip contains:
  - PennSim.jar: the simulator executable file
  - *.html: documentation files
  - *.asm: sample assembly programs

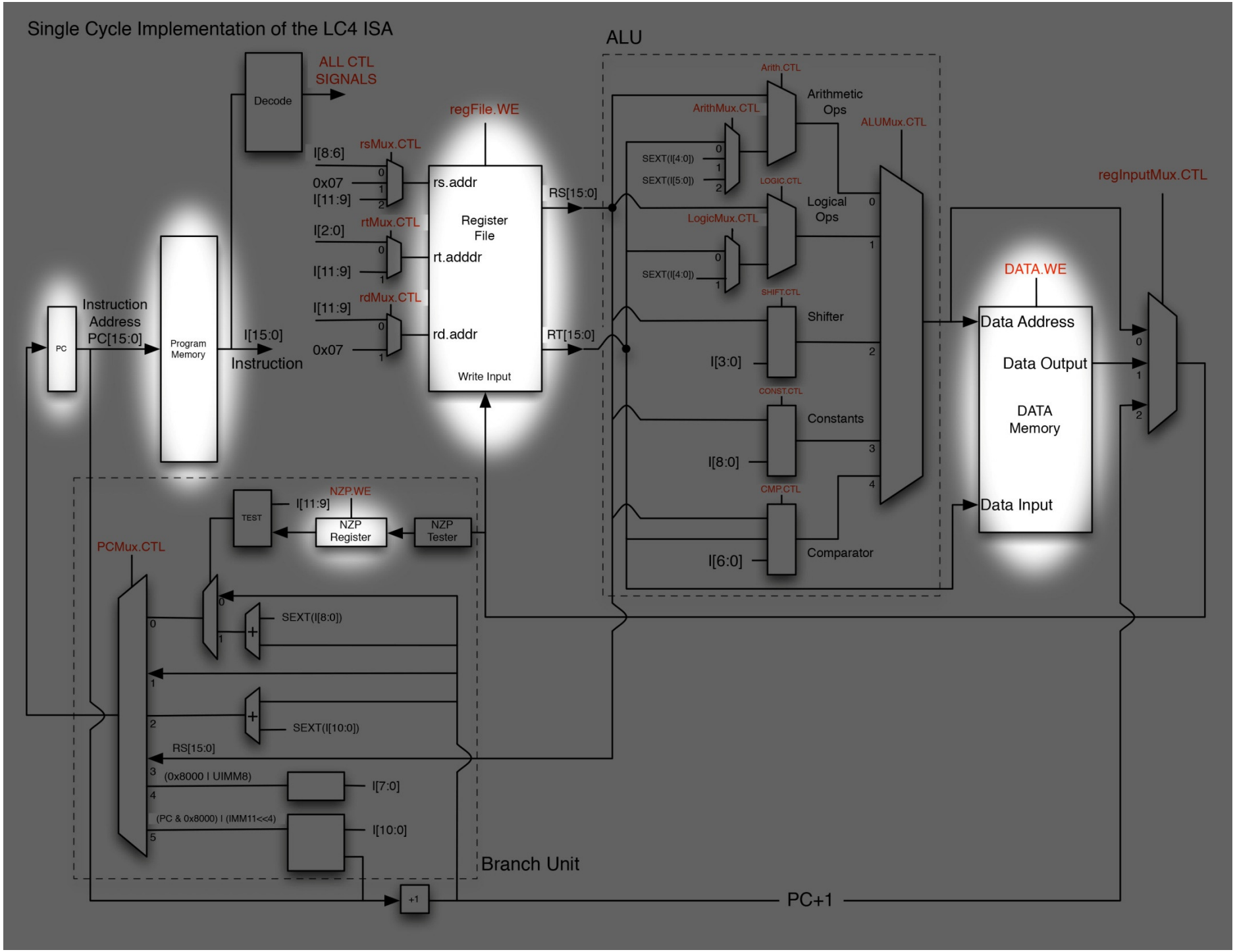
# Running PennSim

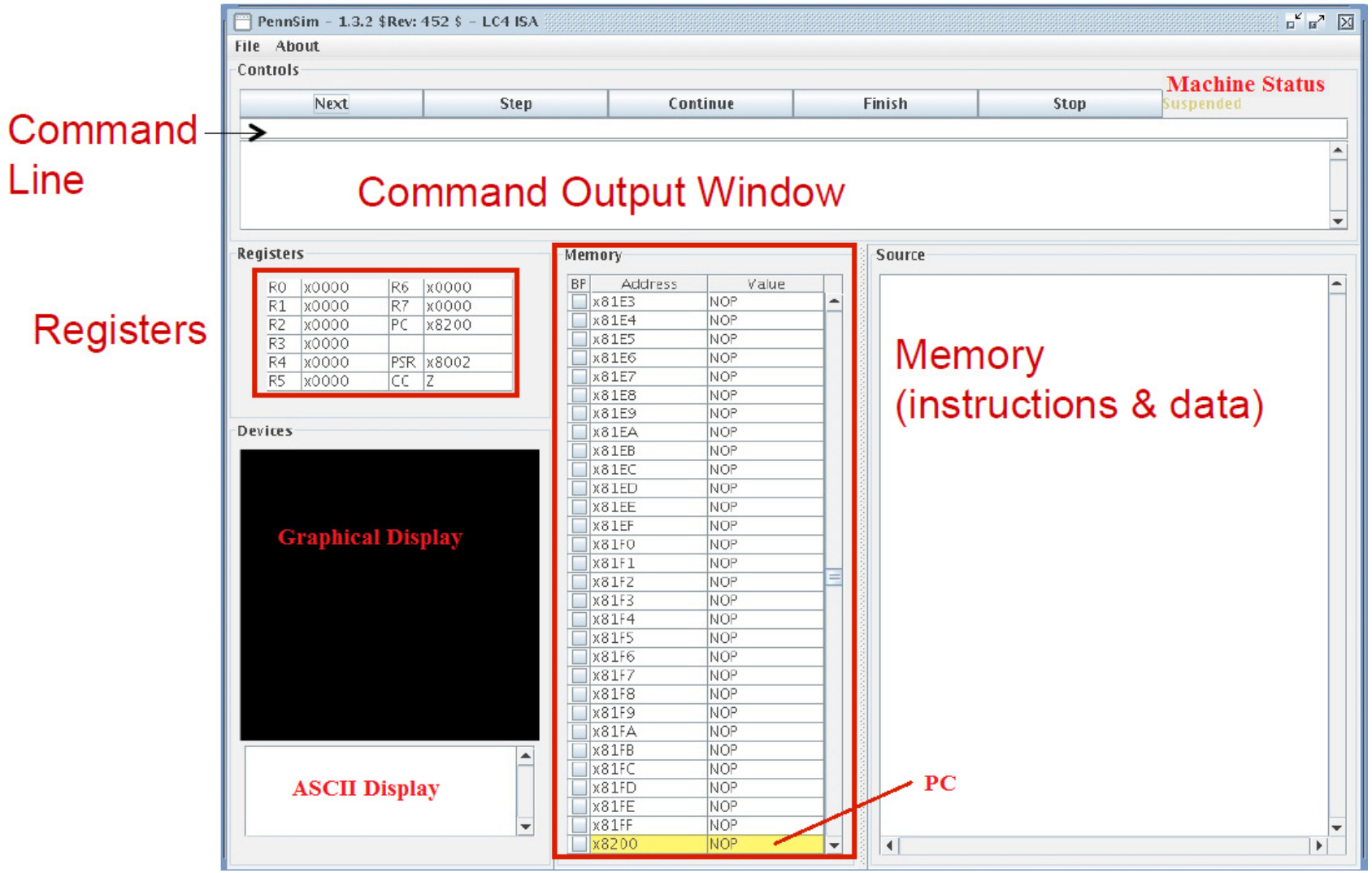
- Need to have Java (v1.5 or newer) installed
- Known bugs:
  - Listed in pennsim-dist.html
- From GUI

  double clicking on PennSim.jar should launch it
- From command line:
  - useful if you need to pass commandline flags to PennSim

```
java –jar /path/to/PennSim.jar
java –jar /path/to/PennSim.jar –d   to run in double-buffered mode
```

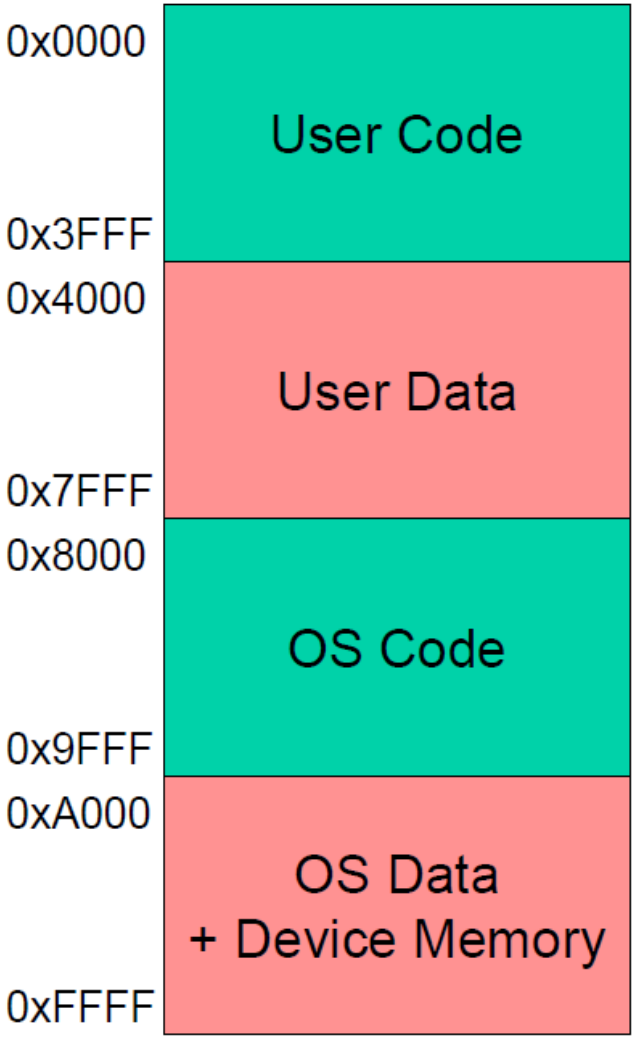

Single Cycle Implementation of the LC4 ISA
ALU
Decode
ALL CTL SIGNALS
regFile.WE
rsMux.CTL
rtMux.CTL
rdMux.CTL
I[8:6]
0x07
I[11:9]
I[2:0]
I[11:9]
I[11:9]
0x07
rs.addr
Register File
rt.adddr
rd.addr
Write Input
RS[15:0]
RT[15:0]
PC
Instruction Address PC[15:0]
Program Memory
I[15:0]
Instruction
Arith.CTL
Arithmetic Ops
ArithMux.CTL
SEXT(I[4:0])
SEXT(I[5:0])
ALUMux.CTL
LOGIC.CTL
Logical Ops
LogicMux.CTL
SEXT(I[4:0])
SHIFT.CTL
Shifter
I[3:0]
CONST.CTL
Constants
I[8:0]
CMP.CTL
Comparator
I[6:0]
regInputMux.CTL
DATA.WE
Data Address
Data Output
DATA Memory
Data Input
TEST
I[11:9]
NZP.WE
NZP Register
NZP Tester
PCMux.CTL
SEXT(I[8:0])
SEXT(I[10:0])
RS[15:0]
(0x8000 | UIMM8)
I[7:0]
(PC & 0x8000) | (IMM11<<4)
I[10:0]
Branch Unit
+1
PC+1


Single Cycle Implementation of the LC4 ISA
ALU
Decode
ALL CTL SIGNALS
regFile.WE
rsMux.CTL
I[8:6]
0x07
I[11:9]
rtMux.CTL
I[2:0]
I[11:9]
rdMux.CTL
I[11:9]
0x07
rs.addr
Register File
rt.adddr
rd.addr
Write Input
RS[15:0]
RT[15:0]
PC
Instruction Address PC[15:0]
Program Memory
I[15:0]
Instruction
Arith.CTL
Arithmetic Ops
ArithMux.CTL
SEXT(I[4:0])
SEXT(I[5:0])
ALUMux.CTL
LOGIC.CTL
Logical Ops
LogicMux.CTL
SEXT(I[4:0])
SHIFT.CTL
Shifter
I[3:0]
CONST.CTL
Constants
I[8:0]
CMP.CTL
Comparator
I[6:0]
regInputMux.CTL
DATA.WE
Data Address
Data Output
DATA Memory
Data Input
TEST
NZP.WE
I[11:9]
NZP Register
NZP Tester
PCMux.CTL
SEXT(I[8:0])
SEXT(I[10:0])
RS[15:0]
(0x8000 | UIMM8)
I[7:0]
(PC & 0x8000) | (IMM11<<4)
I[10:0]
Branch Unit
+1
PC+1


Command Line
Registers
PennSim - 1.3.2 $Rev: 452 $ - LC4 ISA
File About
Controls
Next
Step
Continue
Finish
Stop
Machine Status
Suspended
Command Output Window
Registers
R0 x0000 R6 x0000
R1 x0000 R7 x0000
R2 x0000 PC x8200
R3 x0000
R4 x0000 PSR x8002
R5 x0000 CC Z
Devices
Graphical Display
ASCII Display
Memory
BP Address Value
x81E3 NOP
x81E4 NOP
x81E5 NOP
x81E6 NOP
x81E7 NOP
x81E8 NOP
x81E9 NOP
x81EA NOP
x81EB NOP
x81EC NOP
x81ED NOP
x81EE NOP
x81EF NOP
x81F0 NOP
x81F1 NOP
x81F2 NOP
x81F3 NOP
x81F4 NOP
x81F5 NOP
x81F6 NOP
x81F7 NOP
x81F8 NOP
x81F9 NOP
x81FA NOP
x81FB NOP
x81FC NOP
x81FD NOP
x81FE NOP
x81FF NOP
x8200 NOP
PC
Source
Memory
(instructions & data)

# LC4 Memory

- The address space in LC4 ISA is split into separate areas for code and data. There are also separate regions for Operating System and User.
  - PennSim will yell at you if you try to execute data, use instructions as data, or access OS space without privilege
- Note that PennSim will display either a hex value or an instruction depending on the location
  - This is to make your life easier. Don't be fooled – underlying this is just a string of 16 bits.



# Displayed Values in PennSim

- Addresses and data values are displayed in hex
  - Make sure you are comfortable converting between hex, binary, and decimal

| Dec | Bin | Hex | Dec | Bin | Hex |
|---|---|---|---|---|---|
| 0 | 0000 | x0 | 8 | 1000 | x8 |
| 1 | 0001 | x1 | 9 | 1001 | x9 |
| 2 | 0010 | x2 | 10 | 1010 | xA |
| 3 | 0011 | x3 | 11 | 1011 | xB |
| 4 | 0100 | x4 | 12 | 1100 | xC |
| 5 | 0101 | x5 | 13 | 1101 | xD |
| 6 | 0110 | x6 | 14 | 1110 | xE |
| 7 | 0111 | x7 | 15 | 1111 | xF |

# Aside: PennSim Commands

- PennSim command line understands hex and decimal values, but you need to specify which you are using
  - Decimal – just write the number
    - e.g. `set pc 10` → pc is set to x000A
    - In assembly code, you will see decimal values written as #10
  - Hex – precede the number with an x
    - e.g. `set pc x10` → pc is set to x0010

# PSR

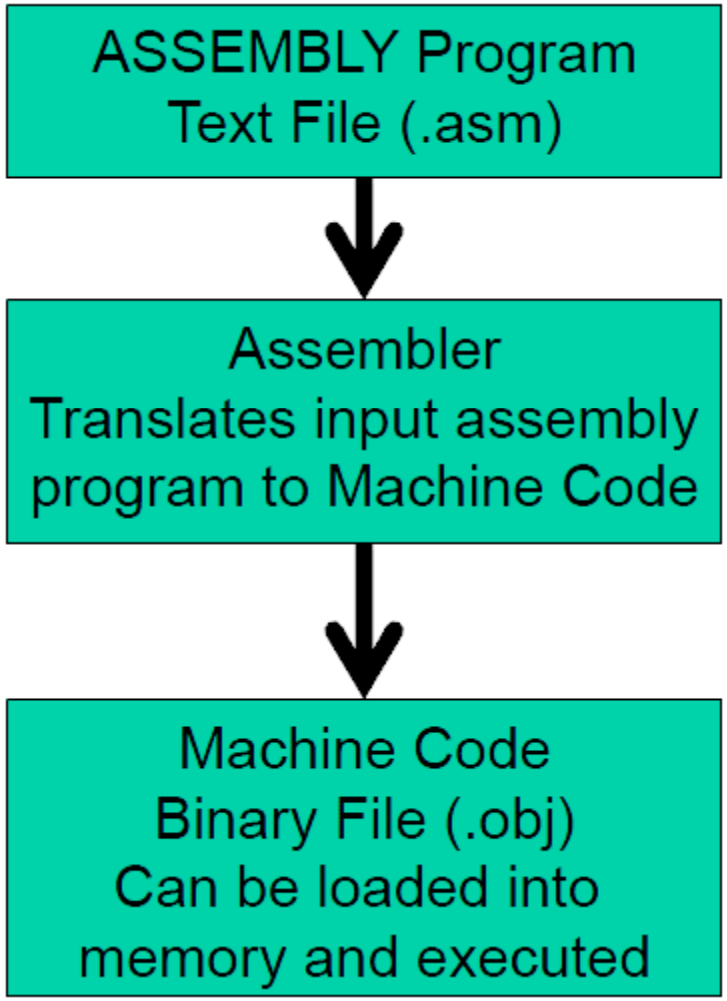
- PSR contains two pieces of information
  - The MSB is OS privilege bit (more on this in future lectures)
  - PSR[2:0] are the NZP bits
    - These are from the last instruction that set NZP
    - For convenience, the NZP value is displayed as text also

# PC

- PC holds the address of the *next* instruction which will be executed. (NOT the one which has just been executed)
- In PennSim, that instruction is highlighted yellow
- Value of PC is displayed in the registers area
- When PennSim is launched, PC defaults to x8200. This is normally where OS instructions will be.

# Writing LC4 Programs

- Assembly programs are written as text files. **Hint: MS Word does not produce text files!**
  - Use a text editor. Notepad will work, there are more powerful ones freely available.
- Assembler translates text to machine code (and does some convenient replacements)
- Output from the assembler is a .obj file, which is loaded and executed by PennSim

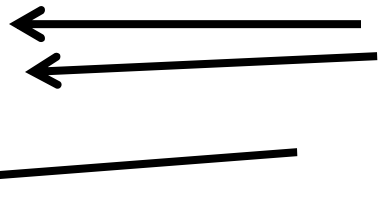


# Counter.asm

```
.CODE
.ADDR x0000

.FALIGN
main_start
        CONST R0, #0
        CONST R1, #1
infinite_loop
        JSR increment
        JMP infinite_loop

.FALIGN
increment
        ADD R0, R0, R1
        RET

;; End demo counter
```

- A slightly modified version of the one that comes with pennsim

# Counter.asm

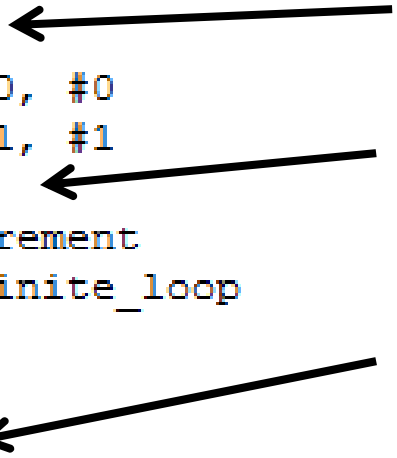


```
.CODE
.ADDR x0000

.FALIGN
main_start
        CONST R0, #0
        CONST R1, #1
infinite_loop
        JSR increment
        JMP infinite_loop

.FALIGN
increment
        ADD R0, R0, R1
        RET

;; End demo counter
```

- Assembly directives
  - Tell the assembler where code and data should be placed when loaded in LC4 memory

# Counter.asm

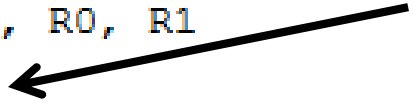
```
.CODE
.ADDR x0000

.FALIGN
main_start
        CONST R0, #0
        CONST R1, #1
infinite_loop
        JSR increment
        JMP infinite_loop

.FALIGN
increment
        ADD R0, R0, R1
        RET

;; End demo counter
```



- Labels
  - Instead of manually calculating addresses and offsets for branches and jumps
  - Give meaningful names to functions, loops, and other important locations in your code

# Counter.asm

```
.CODE
.ADDR x0000

.FALIGN
main_start
        CONST R0, #0
        CONST R1, #1
infinite_loop
        JSR increment
        JMP infinite_loop

.FALIGN
increment
        ADD R0, R0, R1
        RET

;; End demo counter
```



- Pseudo Instructions
  - Look like regular assembly instructions, but they are translated by the assembler
  - The assembler replaces RET with JMPR R7

# Assembling and Loading

- PennSim should be in the same folder as your assembly files
- `as counter counter`
  - Looks for `counter.asm`, runs it through the assembler, and creates `counter.obj`
    - Success: "Assembly completed without errors or warnings."
- `ld counter`
  - Loads counter.obj into PennSim memory
    - Success: "Loading object file counter.obj: code and data ... symbols ... file and line numbers ..."
- Note that we omit the file extensions

# Controlling Execution

- Step
  - Step executes one instruction at a time
  - PC points at the instruction that will be executed the next time you click Step
- Next
  - Next will start executing the current instruction, and stop when PC = currentPC+1
  - This is used to run through functions / subroutines without stopping
- Continue
  - Continue will keep going (FAST!) until you hit an error or click stop
- Breakpoints
  - `break set <address or label>`
  - Will stop execution when you reach the instruction (before it is executed)
  - Note that this command appears to be case sensitive if you are using a label (?!?)

# Script files

- A great way to avoid repetitive strain
- Put a series of PennSim commands into a text file, one per line
- Then run `script script_name.txt`
  - Note that the script command DOES require the file extension .txt
  - PennSim has many quirks like this. Don’t let error messages scare you! They’re usually friendly.

# Writing code

- It often helps to do the following on paper first:
  - Write your algorithm in pseudocode
  - Register allocation - decide which register will hold each variable
  - Draw a flowchart

# Multiplication Algorithm

```
;; C = A*B
C = 0
While (B > 0) {
    C = C + A
    B = B-1;
}
```

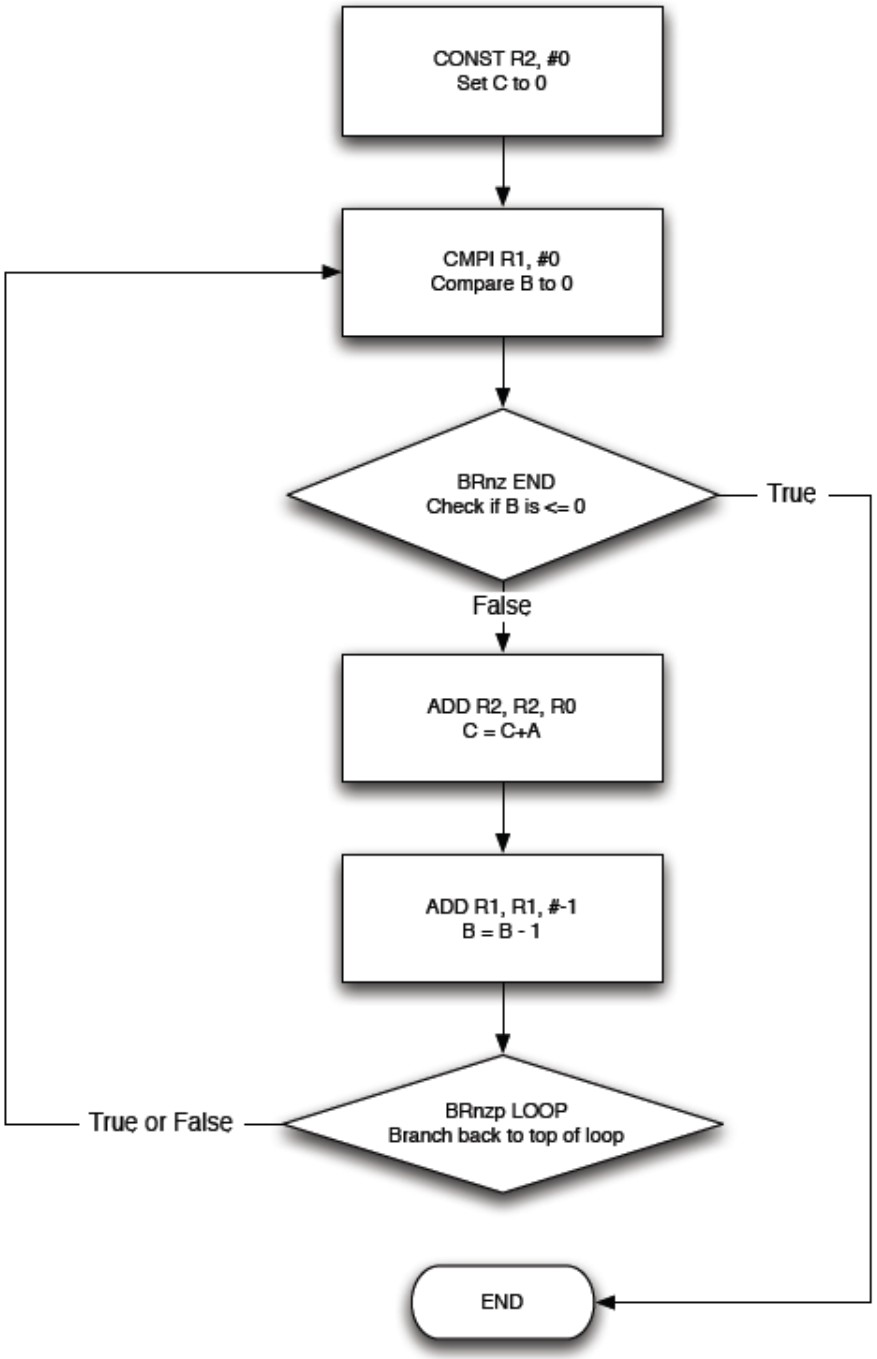
*Note there is a typo on the original slides*

Registers:

A → R0

B → R1

C → R2



# Multiply.asm

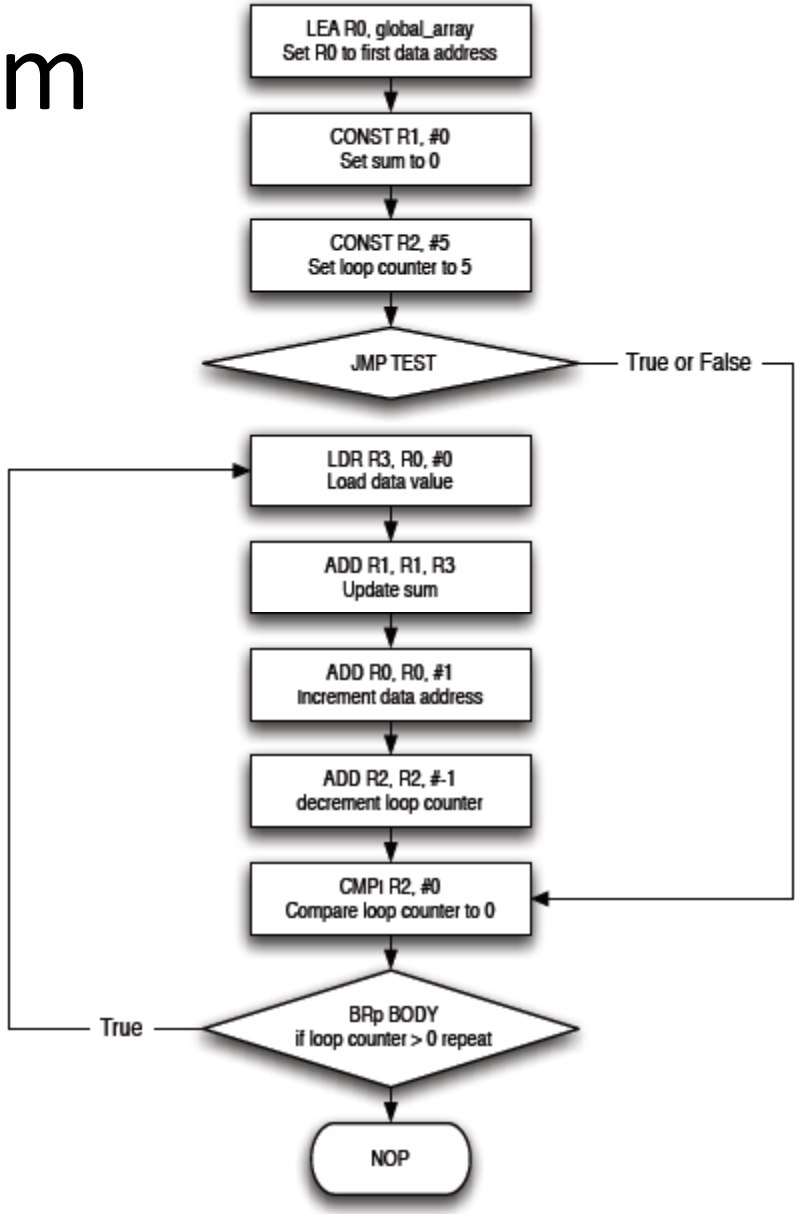
```
;; Multiplication program
;; C = A*B
;; R0 = A, R1 = B, R2 = C

        .CODE           ; This is a code segment
        .ADDR  0x0000   ; Start filling in instructions at address 0x00

        CONST R2, #0    ; Initialize C to 0

LOOP

        CMPI R1, #0     ; Compare B to 0
        BRnz END        ; if (B <= 0) Branch to the end

        ADD R2, R2, R0  ; C = C + A
        ADD R1, R1, #-1 ; B =  B - 1

        BRnzp LOOP      ; Go back to the beginning of the loop

END
```

- Note that we don't initialize A and B in our code.
- They can be thought of as external inputs to our function, whose return value is left in R2 when the loop terminates

# Sum_numbers.asm

```
;; sum an array of 5 numbers
A = number_address
SUM = 0
COUNT = 5
While (COUNT > 0){
   SUM = SUM + data[A]
   A = A + 1
   COUNT = COUNT – 1
}
```

Registers:
R0 → current number address
R1 → SUM
R2 → COUNT
R3 → current number value



# Sum_numbers.asm

```
        ;; This is the data section
        .DATA
        .ADDR x4000             ; Start the data at address 0x4000

global_array
        .FILL #11
        .FILL #7
        .FILL #6
        .FILL #2
        .FILL #-5

        ;; Start of the code section
        .CODE
        .ADDR 0x0000            ; Start the code at address 0x0000
INIT
        LEA R0, global_array    ; R0 contains the address of the data
        CONST R1, 0             ; R1 stores the running sum init to 0
        CONST R2, 5             ; R2 is our loop counter init to 5
        JMP TEST
BODY
        LDR R3, R0, #0          ; Load the data value into R3
        ADD R1, R1, R3          ; update the sum
        ADD R0, R0, #1          ; increment the address
        ADD R2, R2, #-1         ; decrement the loop counter

TEST
        CMPI R2, #0             ; check if the loop counter is zero yet
        BRp BODY
END
        NOP
```

# PennSim Commands

- as <outfile> <infile>
  - Assembles <infile>.asm to <outfile>.obj
- ld <filename>
  - Loads <filename>.obj into memory
- set <register> <value>
  - Load a specific value into the register. Can set PC or PSR. Values can be in hex or decimal
- help <command>
  - Get help on a specific topic
- reset
  - Resets the simulator, clearing memory and registers
- clear
  - Clears the command line output window
- step / next / continue / finish / stop
  - Same as clicking the button of the same name
- break <set | clear> <label | address>
  - Sets or clears a breakpoint at the given location
- Script <filename.txt>
  - Runs commands from a file

# Assembler directives

- .CODE
- .DATA
  - Next instructions are either code or data
- .ADDR
  - Begin loading the following items at the given address
- .FALIGN
  - Address of next instruction should be a multiple of 16. Subroutines need to start at a multiple of 16 due to the semantics of JSR
- .FILL IMM16
  - Set value at the current address to the given value
- .BLKW UIMM16
  - Reserve UIMM16 words at the current address
- .CONST IMM16 and .UCONST IMM16
  - Associate IMM16 or UIMM16 with the preceding label

# Pseudo Instructions

- RET – to return from a subroutine
  - Equivalent to JMPR R7
- LEA R1, <label>
  - Load the address associated with label into a register
  - Equivalent to a CONST, HICONST pair
- LC R3, <label>
  - Load a constant value associated with the label (via .CONST or .UCONST) into a register
  - Equivalent to a CONST, HICONST pair