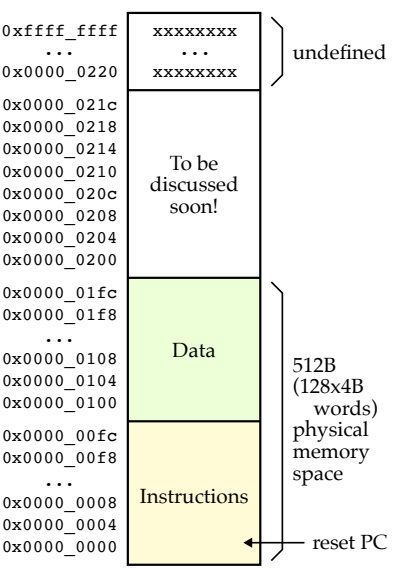
3. TinyRV1 Assembly Programming

- Assembler directives and labels enable more productive assembly programming
- Memory-mapped input/output (I/O) enable assembly programs to interact with external devices
- Conditionals, loops, arrays, and functions enable more complex assembly programs
- Compilers, assemblers, linkers, and loaders enable automatically transforming programs written in high-level languages into machine programs running on a computer

3.1. Assembler Directives and Labels

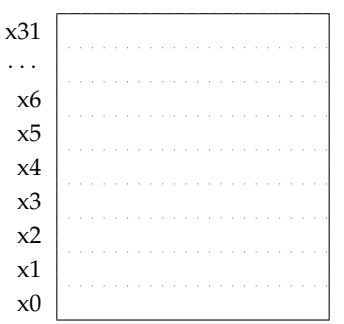
- Assembler directives tell the assembler how to assemble the program
- Assembly program is divided into many sections
  - `.text` directive specifies section for instructions (default section)
  - `.data` directive specifies section for global data
- `.word` directive specifies 4B of global data should be loaded at the corresponding memory location instead of an instruction



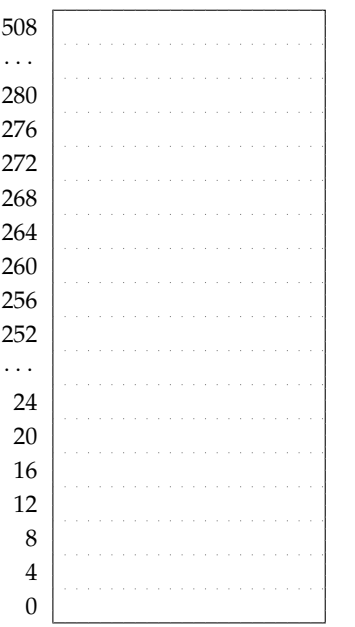
```
.text
  addi x1, x0, 256
  addi x2, x0, 260
  addi x3, x0, 264

  lw   x4, 0(x1)
  lw   x5, 0(x2)
  add  x6, x4, x5
  sw   x6, 0(x3)

.data
  .word 7
  .word 9
  .word 0
```

- Before executing assembly program
  - Reset PC to zero (if PC is shown)
  - Make sure `x0` is zero
  - Load the program (`.text` section) into memory starting at address 0
  - Load the data (`.data` section) into memory starting at address 256

Registers

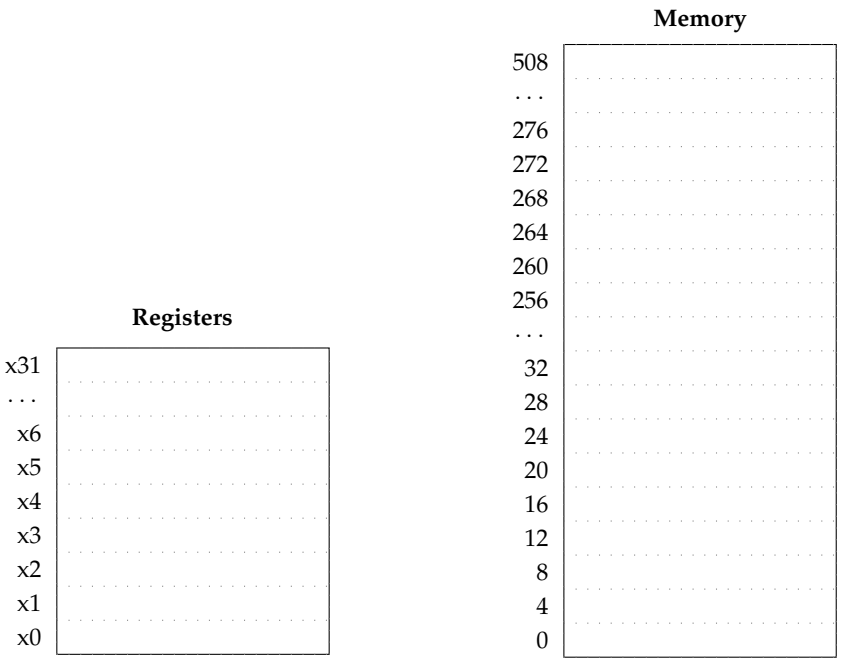


Memory



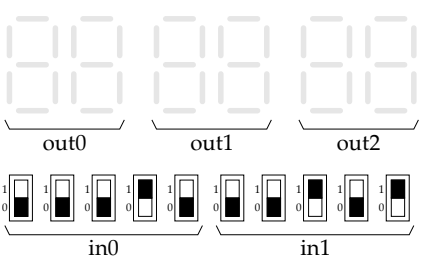
```
  addi x1, x0, 0
  addi x2, x0, 0
  addi x3, x0, 0
  jal  x0, L1
  addi x4, x0, 0

L2:
  addi x1, x0, 1
  addi x2, x0, 1

L1:
  addi x3, x0, 1
  jal  x0, L2
```

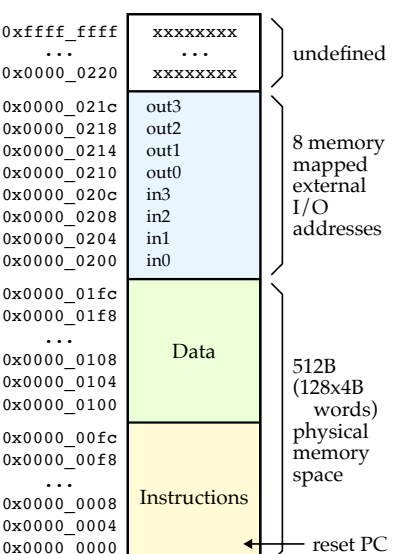
- Assembler labels ...
  - instead of absolute target addresses
  - are placed at beginning of line with colon
  - indicate instruction location in program
  - are not "executed" (no X)
  - do not take up any space in memory
  - assembler turns into PC-relative offsets



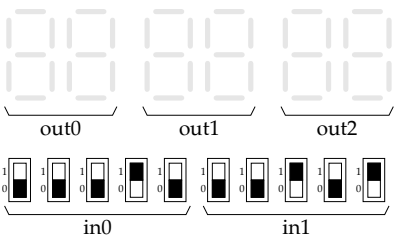
3.2. Memory-Mapped I/O

- Processor needs some way interact with the external enviroment
  - Receiving data from switches, buttons, keyboard, touch screen, network
  - Sending data to a seven-segment display, screen, speaker, network
- Memory-mapped input/output (I/O) involves choosing special memory addresses which are "mapped" to external devices
  - Loading or storing to these special memory-mapped I/O addresses does *not* read or write physical memory
  - Loading from a memory-mapped I/O address reads data from an external device (e.g., reads current value from switches)
  - Storing to a memory-mapped I/O address writes data to an external device (e.g., writes value to seven-segment display)
- TinyRV1 embedded system
  - in0 & in1 connected to switches
  - in2 connected to push button
  - in3 connected to breadboard pin
  - out0, out1, out2 connected to seven-segment displays
  - out3 connected to breadboard pin

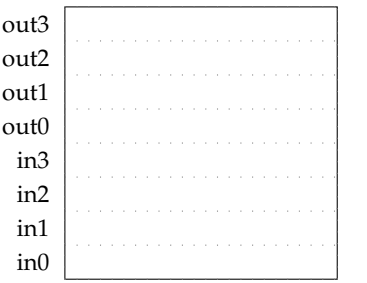




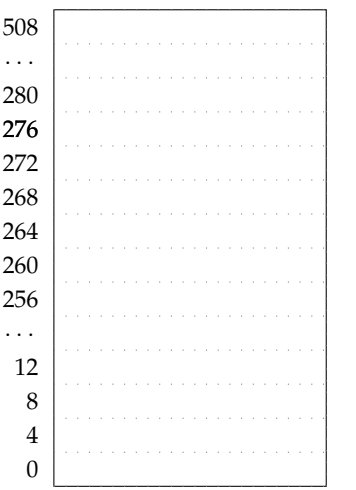
```
lw  x1, 0x200(x0)  # x1 = in0
lw  x2, 0x204(x0)  # x2 = in1

sw  x1, 0x210(x0)  # out0 = x1
sw  x2, 0x214(x0)  # out1 = x2

add x3, x1, x2     # x3 = x1 + x2

sw  x3, 0x218(x0)  # out2 = x3
```



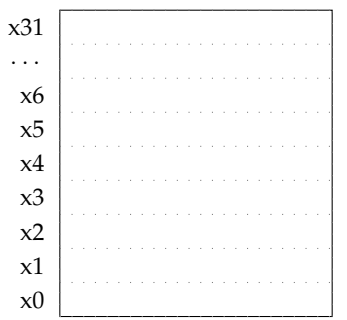
External I/O



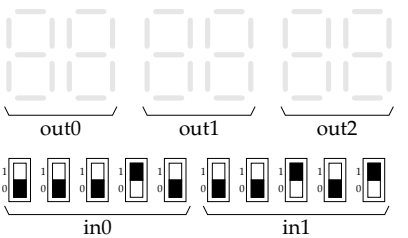
Memory



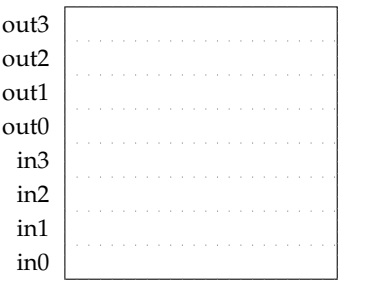
Registers



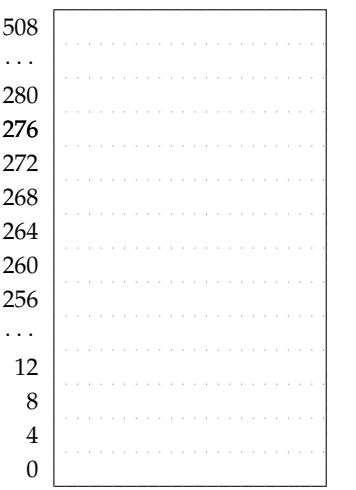
```
lw  x1, 0x200(x0)
sw  x1, 0x210(x0)

add x2, x1, 1
add x3, x1, -1

sw  x2, 0x214(x0)
sw  x3, 0x218(x0)
```



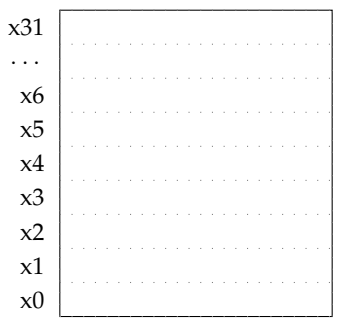
External I/O



Memory



Registers

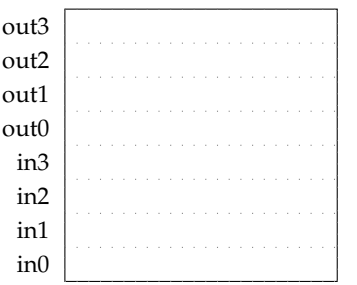


3.3. Conditionals

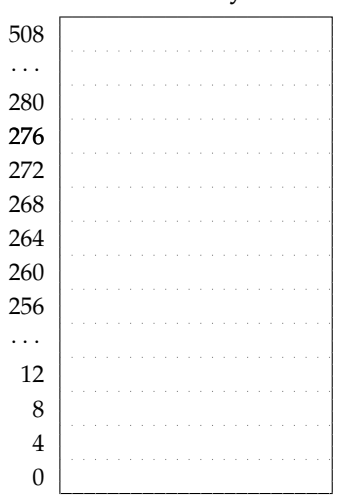
```
1 out0 = in0
2 is_zero = 0
3 if ( in0 == 0 ):
4   is_zero = 1
5 out1 = is_zero
```

```
  lw   x1, 0x200(x0)  # read in0
  sw   x1, 0x210(x0)  # out0 = in0

  addi x2, x0, 0      # is_zero = 0

  bne  x1, x0, L1     # if statement
  addi x2, x0, 1      # is_zero = 1
L1:

  sw   x2, 0x214(x0)  # out1 = is_zero
```

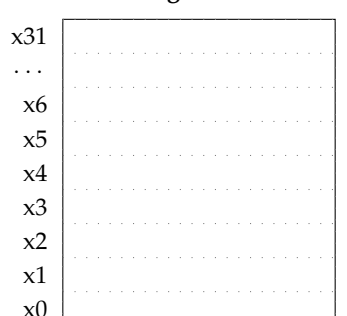
External I/O



Memory



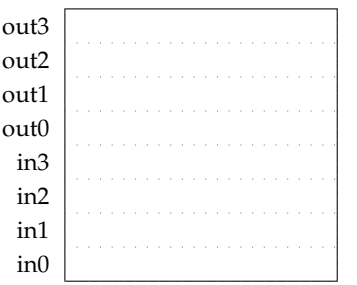
Registers



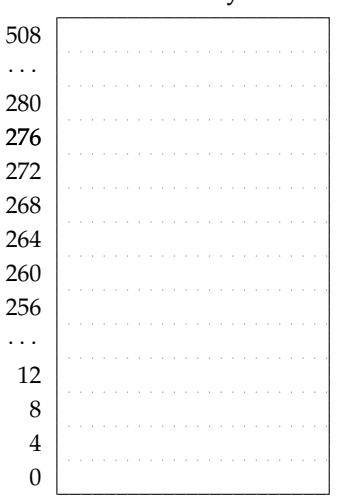
```
out0 = in0
if ( in0 == 0 ):
  is_zero = 1
else:
  is_zero = 0
out1 = is_zero
```

```
  lw   x1, 0x200(x0)
  sw   x1, 0x210(x0)

  bne  x1, x0, L1
  addi x2, x0, 1
  jal  x0, L2
L1:
  addi x2, x0, 0

L2:
  sw   x2, 0x214(x0)
```

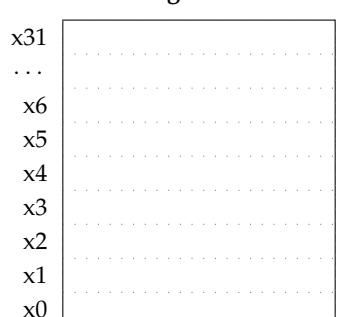
External I/O



Memory



Registers

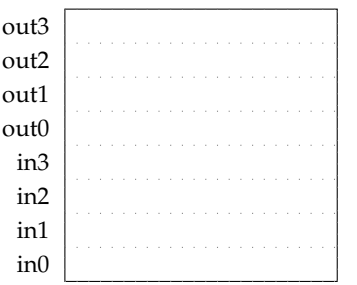


3.4. Loops

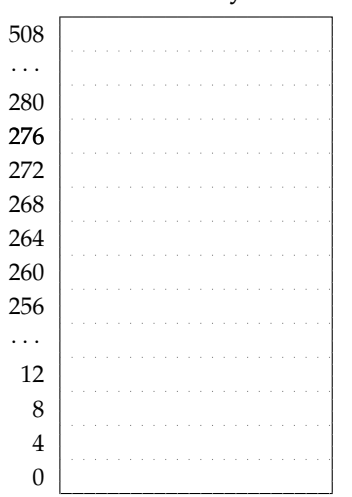
```
1 out0 = in0
2 stop = in0 + 1
3 fact = 1
4 for i in range(1,stop):
5   fact = fact * i
6 out1 = fact
```

```
  lw   x1, 0x200(x0)  # read in0
  sw   x1, 0x210(x0)  # out0 = in0

  addi x2, x1, 1      # stop = in0+1
  addi x3, x0, 1      # fact = 1
  addi x4, x0, 1      # i = 1

loop:
  mul  x3, x3, x4     # fact = fact*i
  addi x4, x4, 1
  bne  x4, x2, loop

  sw   x3, 0x214(x0)  # out1 = fact
```

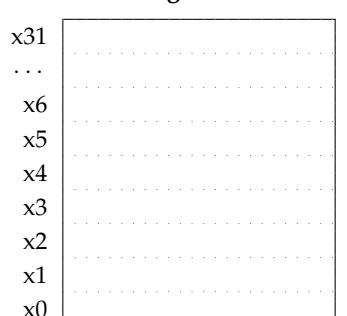
External I/O



Memory



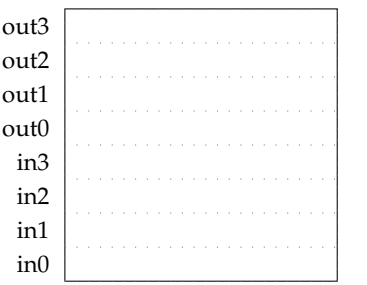
Registers



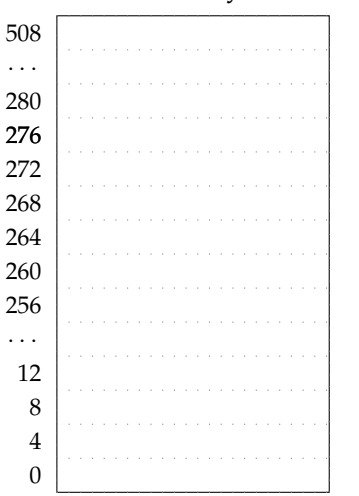
```
1 out0 = in0
2 out1 = in1
3 pow  = 1
4 for i in range(in1):
5   pow = pow * in0
6 out1 = pow
```

```
  lw   x1, 0x200(x0)
  lw   x2, 0x204(x0)
  sw   x1, 0x210(x0)
  sw   x2, 0x214(x0)

  addi x3, x0, 1

loop:
  mul  x3, x3, x1
  addi x2, x2, -1
  bne  x2, x0, loop

  sw   x3, 0x218(x0)
```

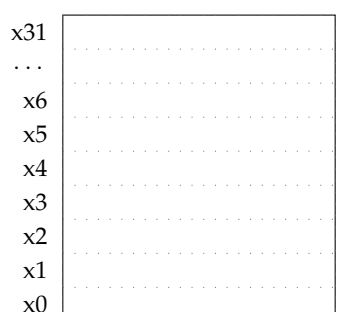
External I/O



Memory



Registers

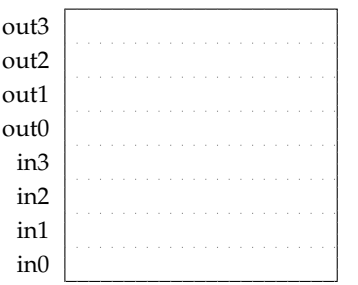


3.5. Arrays

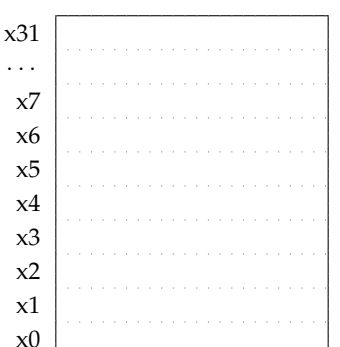
```
1 src = [ 1, 2, 3 ]
2 len = 3
3 for i in range(len):
4   out[i] = src[i] + in0
```

```
  lw   x1, 0x200(x0)  # read in0
  addi x2, x0, 256    # src array
  addi x3, x0, 0x210  # out array
  addi x4, x0, 0      # i = 0
  addi x5, x0, 12     # len = 3

loop:
  addi x6, x2, x4     #
  lw   x7, 0(x6)      # x7 = src[i]
  add  x7, x7, x1     # x7 = x7 + in0
  addi x6, x3, x4     #
  sw   x7, 0(x6)      # out[i] = x7
  addi x4, x4, 4      # i = i + 1
  bne  x4, x5, loop

.data
  .word 1
  .word 2
  .word 3
```

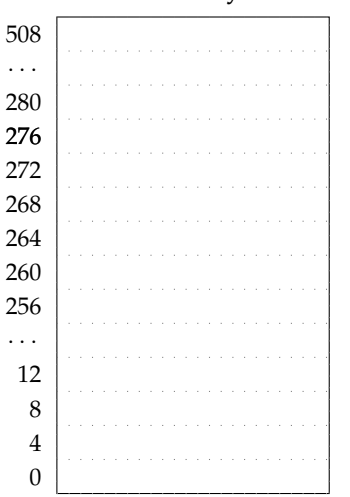
External I/O



Registers



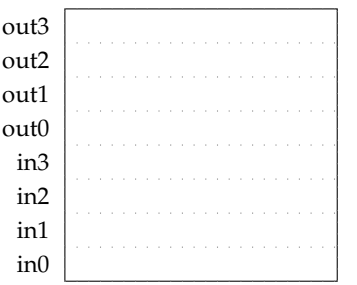
Memory



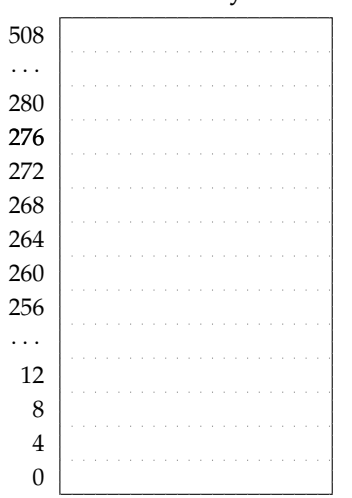
```
src = [ 1, 2, 3 ]
len = 3
for i in range(len):
  out[i] = src[i] + in0
```

```
  lw   x1, 0x200(x0)  # read in0
  addi x2, x0, 256    # src ptr
  addi x3, x0, 0x210  # out ptr
  addi x4, x0, 3      # len = 3

loop:
  lw   x5, 0(x2)      # x7 = src[i]
  add  x5, x5, x1     # x7 = x7 + in0
  sw   x5, 0(x3)      # out[i] = x7
  addi x2, x2, 4      # src ptr bump
  addi x3, x3, 4      # out ptr bump
  bne  x4, x4, -1
  bne  x4, x0, loop

.data
  .word 1
  .word 2
  .word 3
```

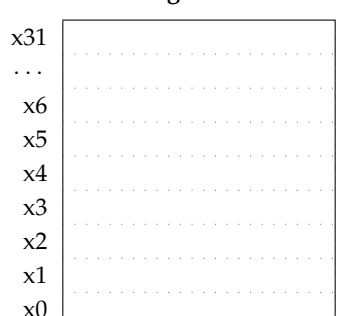
External I/O



Memory



Registers

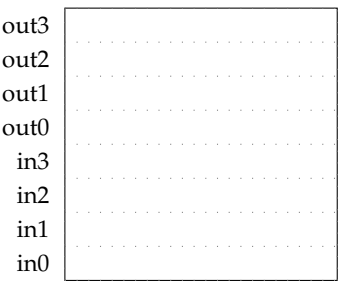


3.6. Functions

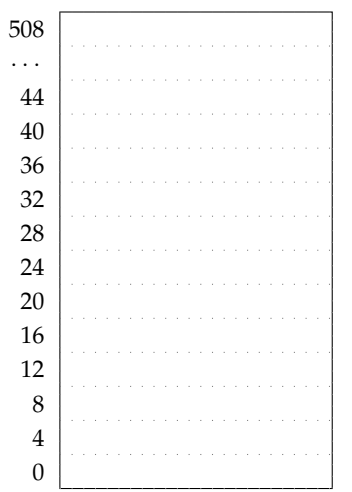
```
1 def sum( a, b ):
2   result = a + b
3   return result
4
5 out0 = sum( in0, 2 )
6 out1 = sum( in1, 3 )
```

```
  jal  x0, start

sum:
  add  x3, x1, x2    # result = a + b
  jr   x4            # return result

start:
  lw   x1, 0x200(x0) # x1 = in0
  addi x2, x0, 2     # x2 = 2
  jal  x4, sum       # tmp = sum(x1,x2)
  sw   x3, 0x210(x0) # out0 = tmp

  lw   x1, 0x204(x0) # x1 = in0
  addi x2, x0, 3     # x2 = 2
  jal  x4, sum       # tmp = sum(x1,x2)
  sw   x3, 0x214(x0) # out1 = tmp
```

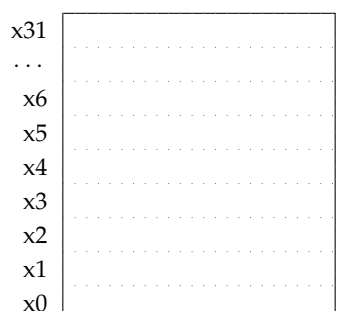
External I/O



Memory



Registers

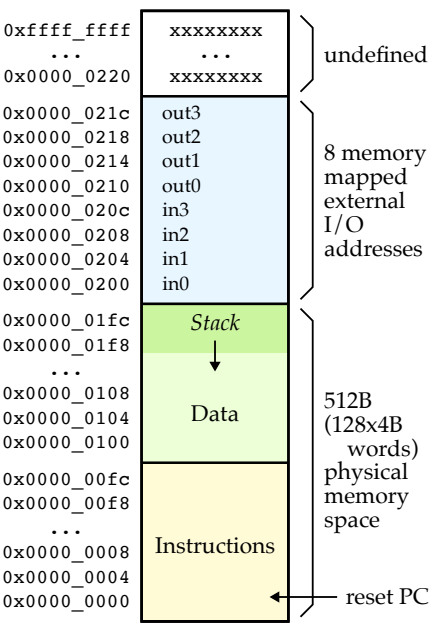


- How do we know ...
  - what registers are used for arguments? return values? return address?
  - what registers the function can use as temporaries?
  - what registers the function must save to memory before using?
  - what memory locations can be used for temporary data?
- Calling convention is standard interface for calling functions

Register	Saver	Description
x28–x31	caller	Temporary values
x18–x27	callee	Saved registers
x11–x17	caller	Function arguments
x10	caller	Function arg & return value
...		
x5–x7	caller	Temporary values
...		
x2	callee	Stack pointer
x1	caller	Function return address
x0	—	Always zero

```
sum:
  add  x10, x11, x12   # result = a + b
  jr   x1              # return result

start:
  lw   x11, 0x200(x0)  # x11 = in0
  addi x12, x0, 2      # x12 = 2
  jal  x1,  sum        # tmp = sum(x11,x12)
  sw   x10, 0x210(x0)  # out0 = tmp

  lw   x11, 0x204(x0)  # x11 = in0
  addi x12, x0, 3      # x12 = 2
  jal  x1,  sum        # tmp = sum(x11,x12)
  sw   x10, 0x214(x0)  # out1 = tmp
```

- Stack is memory region dedicated for functions to use as scratch space (i.e., for temporary data)
  - Stack expands (uses more memory) as program needs more scratch space
  - Stack contracts (uses less memory) as program needs less scratch space
  - In TinyRV1, stack is located at address 0x1fc and expands "downwards"
- Stack pointer is a register that points to the top of the stack
  - Functions can allocate temporary data on the stack and then decrement the stack pointer so the next function knows where it can allocate its own temporary data
  - Before a function returns it needs to deallocate all of its stack data by incrementing the stack pointer back to its original value



We will use a "vector-vector-add" function throughout the course as a simple microbenchmark for performance analysis

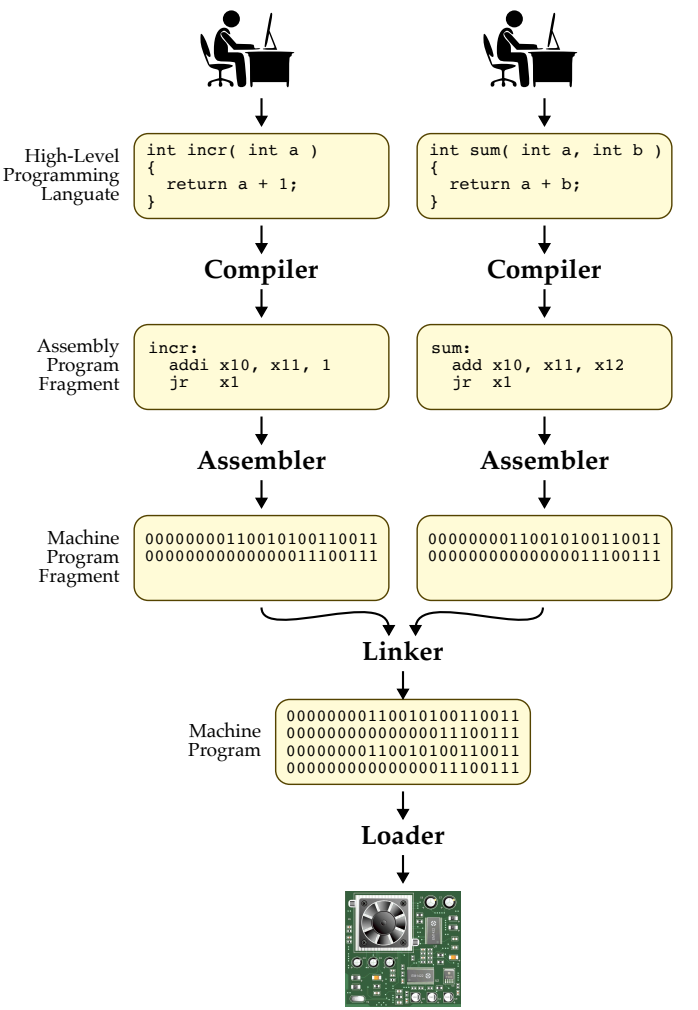
Pseudo-Code

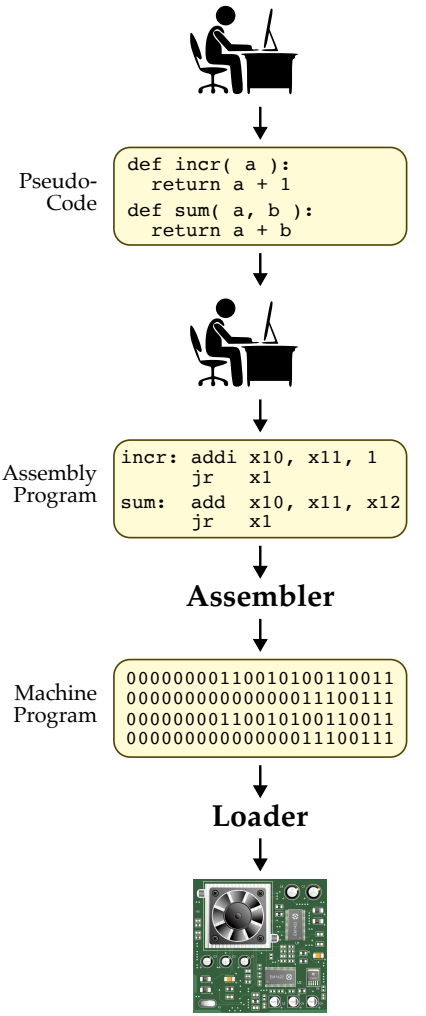
```
def vvadd( dest, src0, src1, size ):
  for i in range(size):
    dest[i] = src0[i] + src1[i]
```

Assembly Program

```
  jal x0, start

# wait_for_button()

wait_for_button:
  addi x5, x0, 1
loop1:
  lw   x6, 0x20c(x0)
  bne  x5, x6, loop1
  jr   x1

# vvadd( dest, src0, src1, size )

vvadd:
loop:
  lw    x5,  0(x11)   # x5 = src0[i]
  lw    x6,  0(x12)   # x6 = src1[i]
  add   x7,  x5, x6   # x7 = x5 + x6
  sw    x7,  0(x10)   # dest[i] = x7
  addi  x11, x11, 4   # src0 ptr bump
  addi  x12, x12, 4   # src1 ptr bump
  addi  x10, x10, 4   # src2 ptr bump
  addi  x13, x13, -1
  bne   x13, x0, loop
  jr    x1            # return

start:
  # setup vvadd arguments
  addi  x11, x0, 256
  addi  x12, x0, 370
  addi  x10, x0, 454
  addi  x13, x0, 21

  # call vvadd and trigger
  # breadboard pin for timing
  addi  x18, x0, 1
  sw    x18, 0x21c(x0)
  jal   x1,  vvadd
  sw    x0,  0x21c(x0)

  # display results
  addi  x18, x0, 454
  addi  x19, x0, 21
loop2:
  lw    x20, 0(x18)
  sw    x20, 0x210(x0)
  jal   x1,  wait_for_button
  addi  x18, x18, 4
  addi  x19, x19, -1
  bne   x19, x0, loop2

done:
  jal   x0,  done

.data
  # src0 @256, 21 elm
  .word 3
  .word 2
  ...

  # src1 @370, 21 elm
  .word 7
  .word 4
  ...

  # dest @454, 21 elm
  .word 0
  .word 0
  ...
```

3.7. Compilers, Assemblers, Linkers, and Loaders




Pseudo-Code
def incr(a):
return a + 1
def sum(a, b):
return a + b
Assembly Program
incr: addi x10, x11, 1
jr x1
sum: add x10, x11, x12
jr x1
Assembler
Machine Program
00000000110010100110011
00000000000000011100111
00000000110010100110011
00000000000000011100111
Loader

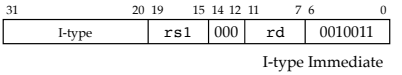
3.8. TinyRV1 ISA Specification

ADDI

`addi rd, rs1, imm`

R[rd] ← R[rs1] + sext(imm)

PC ← PC + 4

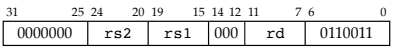


ADD

`add rd, rs1, rs2`

R[rd] ← R[rs1] + R[rs2]

PC ← PC + 4

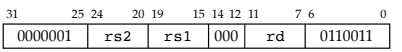


MUL

`mul rd, rs1, rs2`

R[rd] ← R[rs1] × R[rs2]

PC ← PC + 4

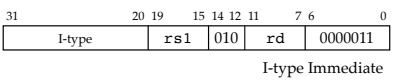


LW

`lw rd, imm(rs1)`

R[rd] ← M[R[rs1] + sext(imm)]

PC ← PC + 4

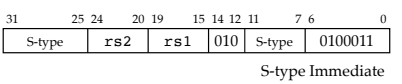


SW

`sw rs2, imm(rs1)`

M[R[rs1] + sext(imm)] ← R[rs2]

PC ← PC + 4

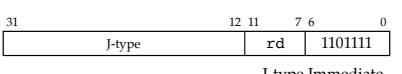


JAL

`jal rd, addr`

R[rd] ← PC + 4

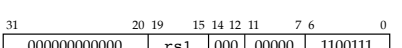
PC ← `addr`



JR

`jr rs1`

PC ← R[rs1]

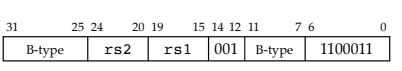


BNE

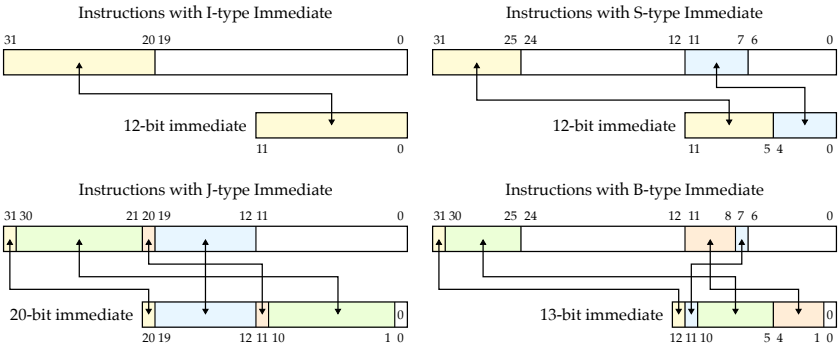
`bne rs1, rs2, addr`

if (R[rs1] != R[rs2]) PC ← `addr`

else PC ← PC + 4



TinyRV1 Immediate Types



TinyRV1 Calling Convention

Register	Saver	Description
`x28–x31`	caller	Temporary values
`x18–x27`	callee	Saved registers
`x11–x17`	caller	Function arguments
`x10`	caller	Function arg & return value
...		
`x5–x7`	caller	Temporary values
...		
`x2`	callee	Stack pointer
`x1`	caller	Function return address
`x0`	—	Always zero

Notes

- R[`x0`] is always zero
- PC reset value is `0x00000000`
- All addresses must be word aligned (i.e., bottom two address bits are zero)
- Unaligned or invalid memory accesses result in undefined behavior
- `nop` is pseudo-instruction for `addi x0, x0, 0`

TinyRV1 Virtual Address Map