ECE 2300 Digital Logic and Computer Organization Fall 2026

Topic 9: Instruction Set Architecture

School of Electrical and Computer Engineering
Cornell University

revision: 2026-10-27-09-34

1 Computer Organization **3**

1.1. Pico-Processor 6

1.2. Pico-Processor Instruction-Set Architecture 9

2 TinyRV1 Instruction Set Architecture **12**

2.1. Arithmetic Instructions 14

2.2. Memory Instructions 18

2.3. Control Instructions 24

2.4. TinyRV1 ISA Specification (preliminary) 27

3 TinyRV1 Assembly Programming **31**

3.1. Assembler Directives and Labels 31

3.2. Memory-Mapped I/O 34

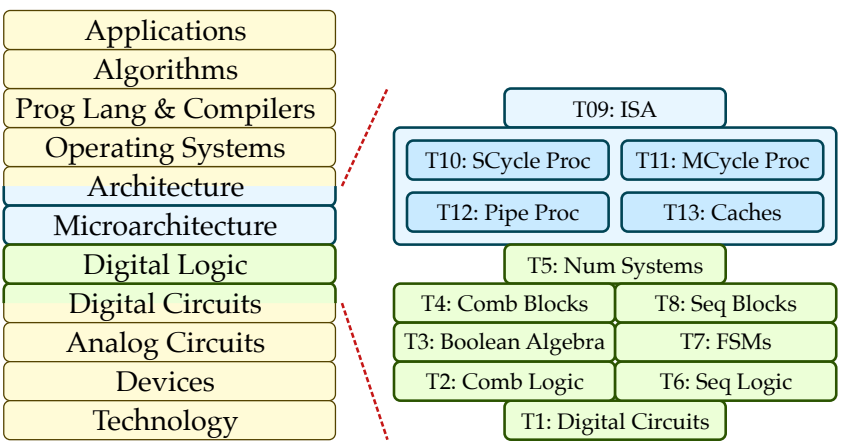
3.3. Conditionals 37

3.4. Loops 39

3.5. Arrays . . . 41
3.6. Functions . . . 43
3.7. Compilers, Assemblers, Linkers, and Loaders . . . 47
3.8. TinyRV1 ISA Specification . . . 49

4 From Architecture to Microarchitecture 51

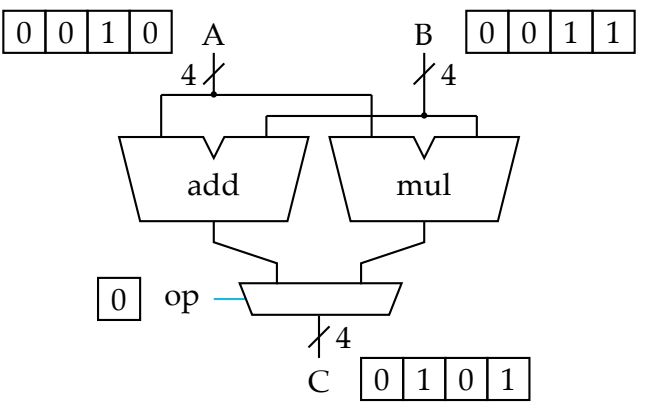
4.1. Processor Microarchitectural Design Patterns . . . 52
4.2. Transaction Diagrams . . . 53
4.3. Analyzing Processor Performance . . . 54



Copyright © 2026 Christopher Batten. All rights reserved. This handout was prepared by Prof. Christopher Batten at Cornell University for ECE 2300 / ENGRD 2300 Digital Logic and Computer Organization. Download and use of this handout is permitted for individual educational non-commercial purposes only. Redistribution either in part or in whole via both commercial or non-commercial means requires written permission.

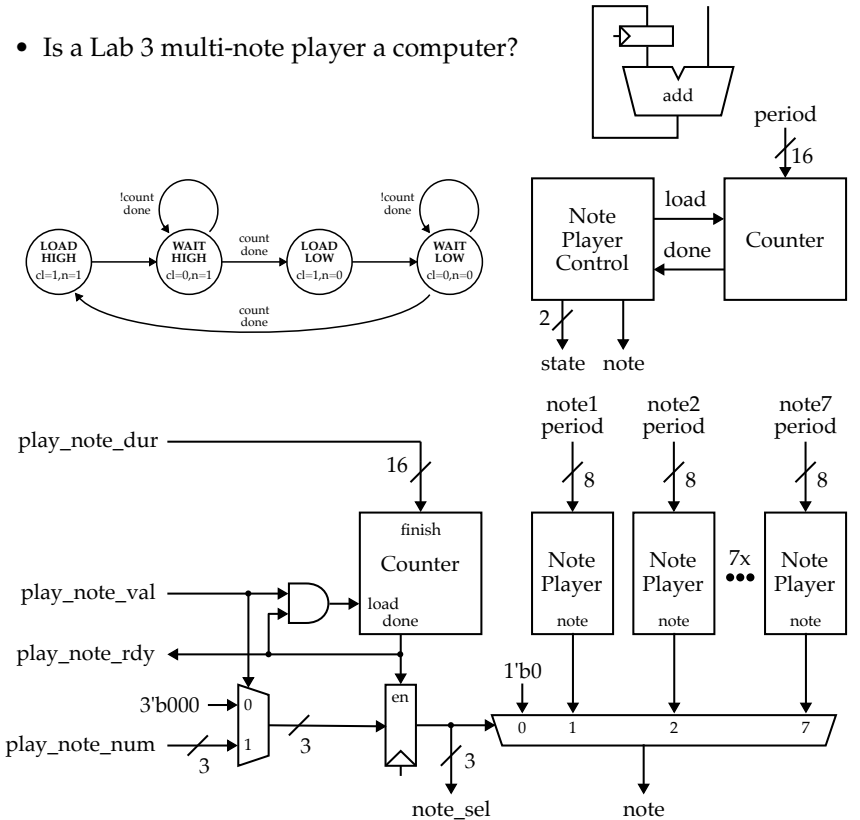
1. Computer Organization

- A **computer** is "an electronic device for processing and storing data (typically in binary form), according to general-purpose instructions (also in binary form) stored as a program in memory"
- **Computer organization** involves using number systems, combinational building blocks, and sequential building blocks to implement a computer
- Is our Lab 2 calculator a computer?



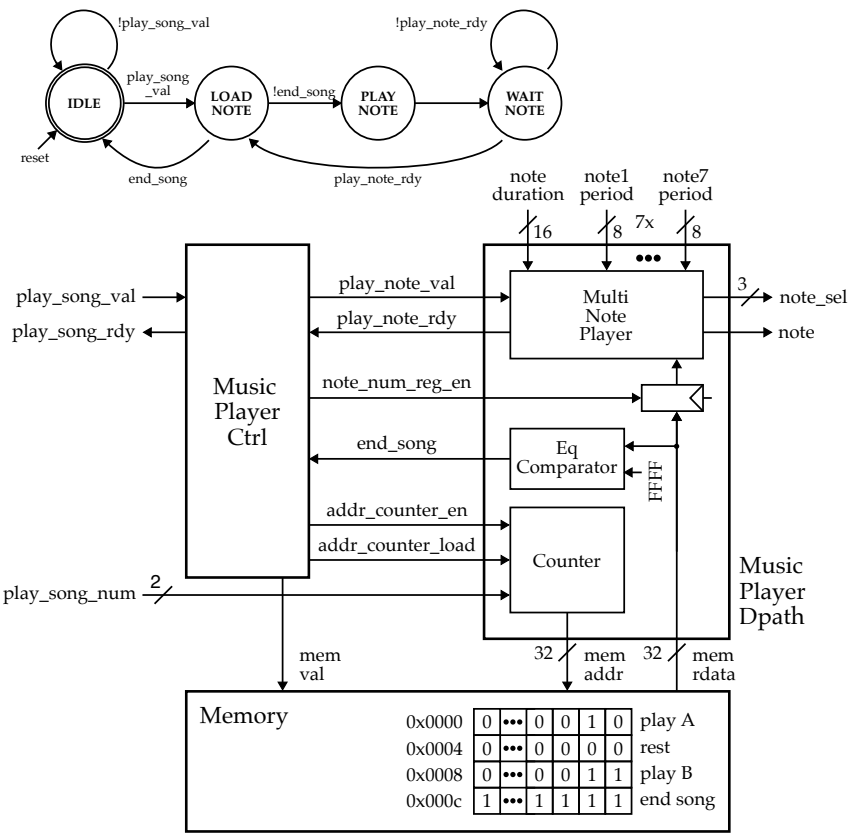
- A computer is "an electronic device for processing and storing data (typically in binary form), according to general-purpose instructions (also in binary form) stored as a program in memory"

- Is a Lab 3 multi-note player a computer?



- A computer is "an electronic device for processing and storing data (typically in binary form), according to general-purpose instructions (also in binary form) stored as a program in memory"

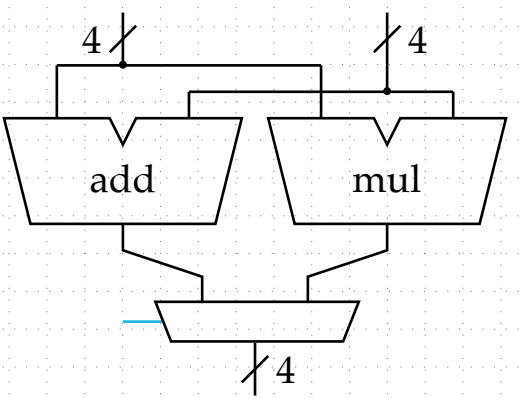
- Is a Lab 3 music player a computer?



- A computer is "an electronic device for processing and storing data (typically in binary form), according to general-purpose instructions (also in binary form) stored as a program in memory"

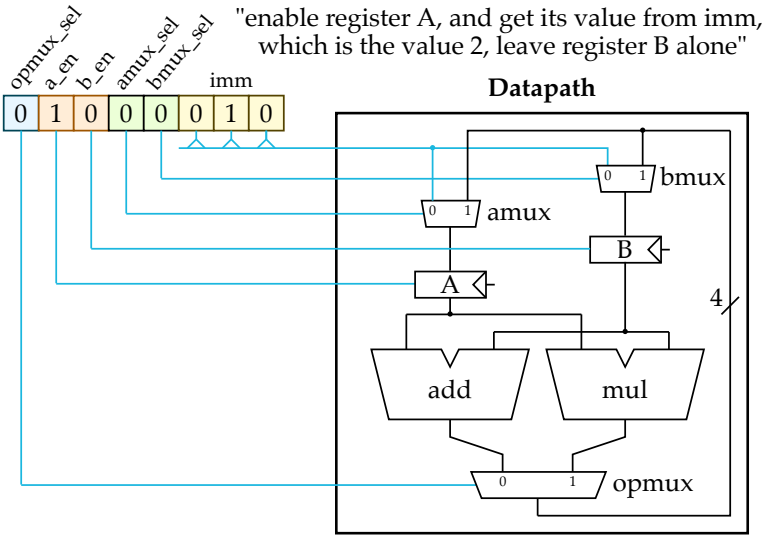
1.1. Pico-Processor

- Let's build a pico-processor [a]
  - Calculator: processing, general-purpose instructions
  - Music Player: storing, program in memory



[a] Not a micro-processor (10^{-6}), not a nano-processor (10^{-9}), but a pico-processor (10^{-12})

- An instruction is a binary value which tells a computer how to perform a specific operation



- A program is a sequence of instructions

opmux_sel	a_en	b_en	amux_sel	bmux_sel	imm		
0	1	0	0	0	0	1	0
0	0	1	0	0	0	1	1
0	1	0	1	1	0	0	0
1	0	1	1	1	0	0	0

"set register A to 2"

"set register B to 3"

"set register A to A+B"

"set register B to A×B"

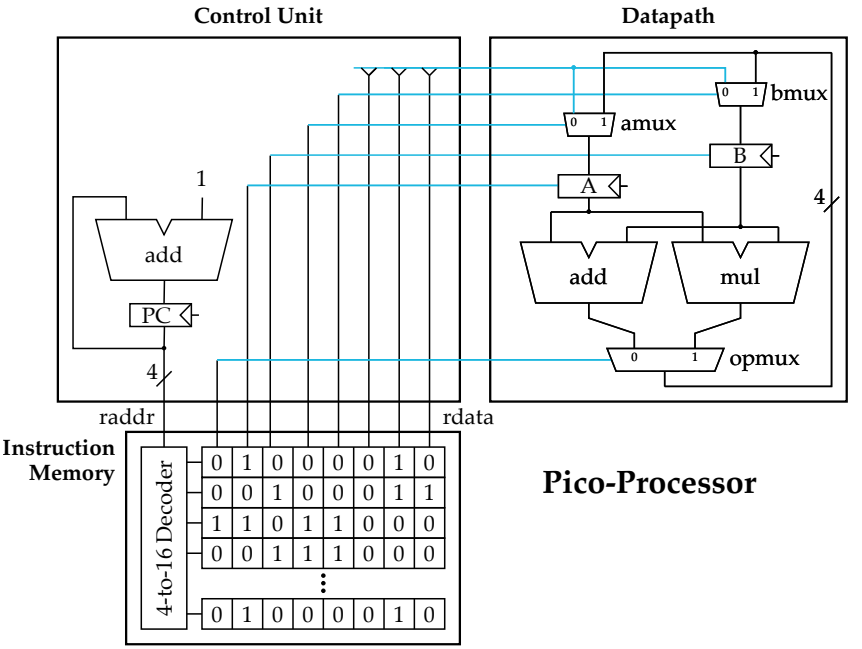
Machine Instructions make up a **Machine Program** (Program Binary)

Assembler

Disassembler

Assembly Instructions make up an **Assembly Program**

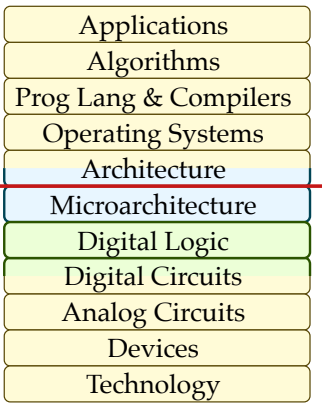
- A program stored in memory is just a sequence of machine instructions (binary values) stored in a memory array



- A computer is "an electronic device for processing and storing data (typically in binary form), according to general-purpose instructions (also in binary form) stored as a program in memory"

The pico-processor is an (extremely simple) computer!

1.2. Pico-Processor Instruction-Set Architecture



An instruction set architecture is the contract between software and hardware

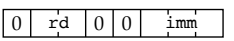
1. How is the data represented?
2. Where can the data be stored?
3. How can data be accessed?
4. What operations can be done on data?
5. How are instructions encoded?

WRIMM

`wrimm rd, imm`

R[`rd`] ← { 1'b0, `imm` }

PC ← PC + 1



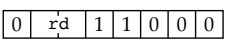
for register A, `rd` = 10
for register B, `rd` = 01

ADD

`add rd`

R[`rd`] ← R[rA] + R[rB]

PC ← PC + 1



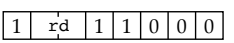
for register A, `rd` = 10
for register B, `rd` = 01

MUL

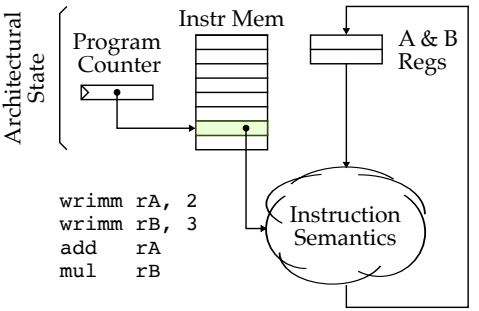
`mul rd`

R[`rd`] ← R[rA] × R[rB]

PC ← PC + 1



for register A, `rd` = 10
for register B, `rd` = 01

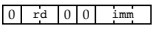


WRIMM

`wrimm rd, imm`

R[rd] ← { 1'b0, imm }

PC ← PC + 1



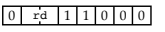
for register A, rd = 10
for register B, rd = 01

ADD

`add rd`

R[rd] ← R[rA] + R[rB]

PC ← PC + 1



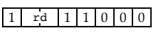
for register A, rd = 10
for register B, rd = 01

MUL

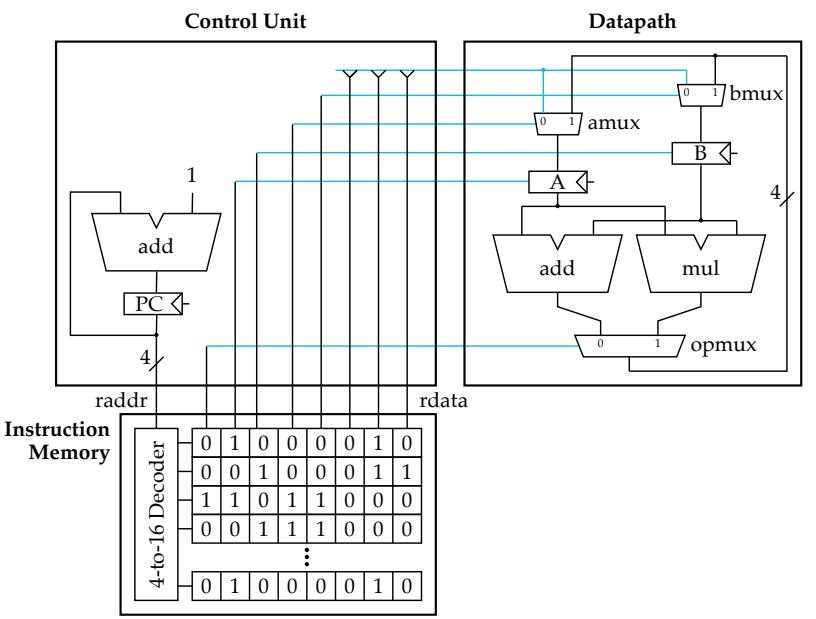
`mul rd`

R[rd] ← R[rA] × R[rB]

PC ← PC + 1



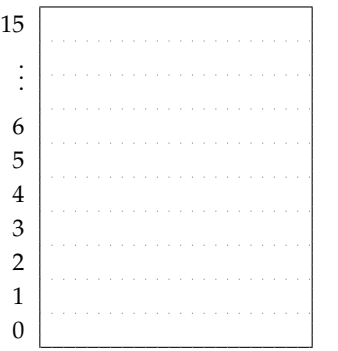
for register A, rd = 10
for register B, rd = 01



Assembly Program

```
wrimm rA, 2
wrimm rB, 3
add   rA
mul   rB
```

Memory



Program Counter

Registers

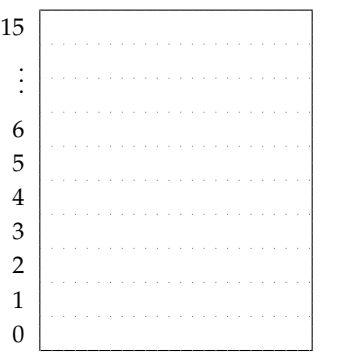
rB

rA

Assembly Program

```
wrimm rA, 2
wrimm rB, 0
add   rB
mul   rA
mul   rA
```

Memory



Program Counter

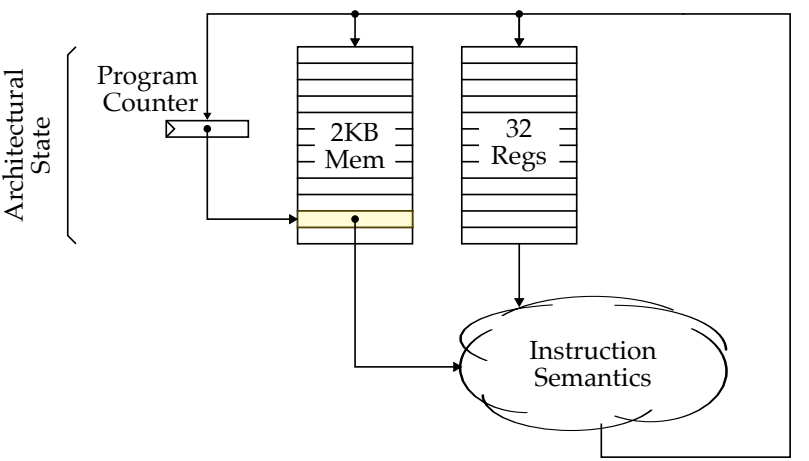
Registers

rB

rA

2. TinyRV1 Instruction Set Architecture

- RISC-V instruction set architecture
  - Brand new free, open instruction set architecture
  - Significant excitement around RISC-V hardware/software ecosystem
  - Helping to energize “open-source hardware”
  - Specifically designed to encourage subsetting and extension
  - Link to official ISA manual on course webpage
- TinyRV1 instruction set architecture
  - Subset we use in this course
  - Small enough for teaching
- TinyRV1 functional processor model



1. How is the data represented?

2. Where can the data be stored?

3. How can data be accessed?

4. What operations can be done on data?

5. How are instructions encoded?

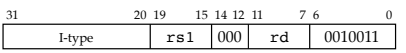
2.1. Arithmetic Instructions

ADDI

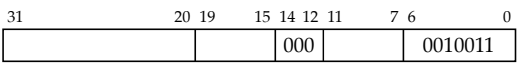
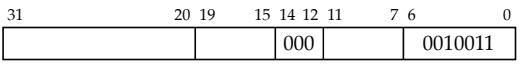
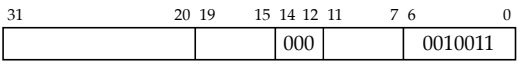
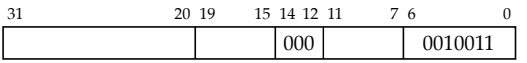
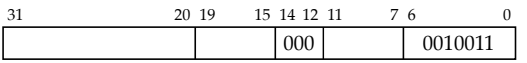
`addi rd, rs1, imm`

R[`rd`] ← R[`rs1`] + sext(`imm`)

PC ← PC + 4



- Instruction set architecture specification includes:
  - ① Instruction name
  - ② Instruction assembly format (how to represent instructions as strings)
  - ③ Instruction encoding (how to represent instructions as bits)
  - ④ Instruction semantics (what does the instruction actually do)

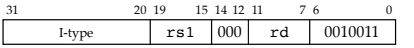
Assembly Instruction	Machine Instruction
`addi x1, x0, 9`	
`addi x2, x1, 7`	
`addi x1, x0, 13`	
`addi x1, x1, 2`	
`addi x6, x1, 0`	

ADDI

`addi rd, rs1, imm`

R[`rd`] ← R[`rs1`] + sext(`imm`)

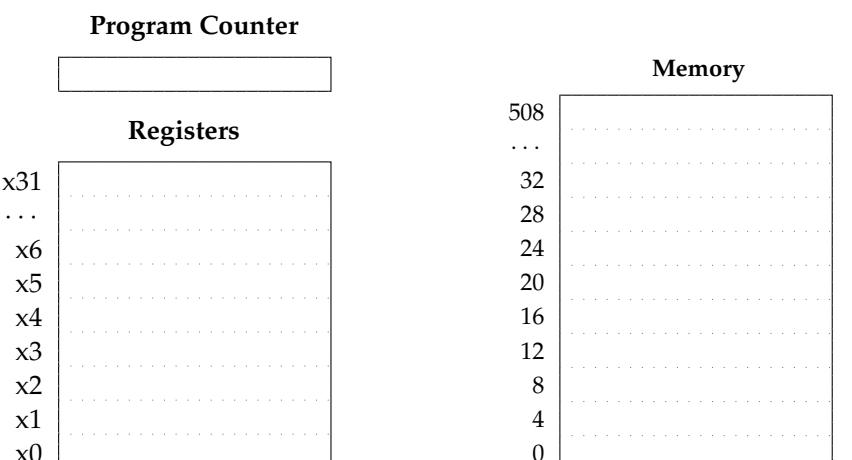
PC ← PC + 4



I-type Immediate

- Explanation of instruction semantics
  - `imm` : immediate value stored in the instruction
  - R[`r`] : 4-byte (32-bit) value stored in general-purpose register `r`
  - sext(`V`) : sign extend the value `V` to 32-bits (fill with immediate's sign bit)
  - Note that R[`x0`] is always the constant zero

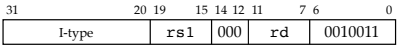
```
 0 addi x1, x0,  9  # R[x1] = R[x0] + 9
 4 addi x2, x1,  7  # R[x2] = R[x1] + 7
 8 addi x1, x0, 13  # R[x1] = R[x0] + 13
12 addi x1, x1,  2  # R[x1] = R[x1] + 2
16 addi x6, x1,  0  # R[x6] = R[x1] + 0
```



ADDI

`addi rd, rs1, imm`

R[`rd`] ← R[`rs1`] + sext(`imm`)
PC ← PC + 4

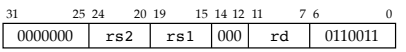


I-type Immediate

ADD

`add rd, rs1, rs2`

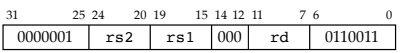
R[`rd`] ← R[`rs1`] + R[`rs2`]
PC ← PC + 4



MUL

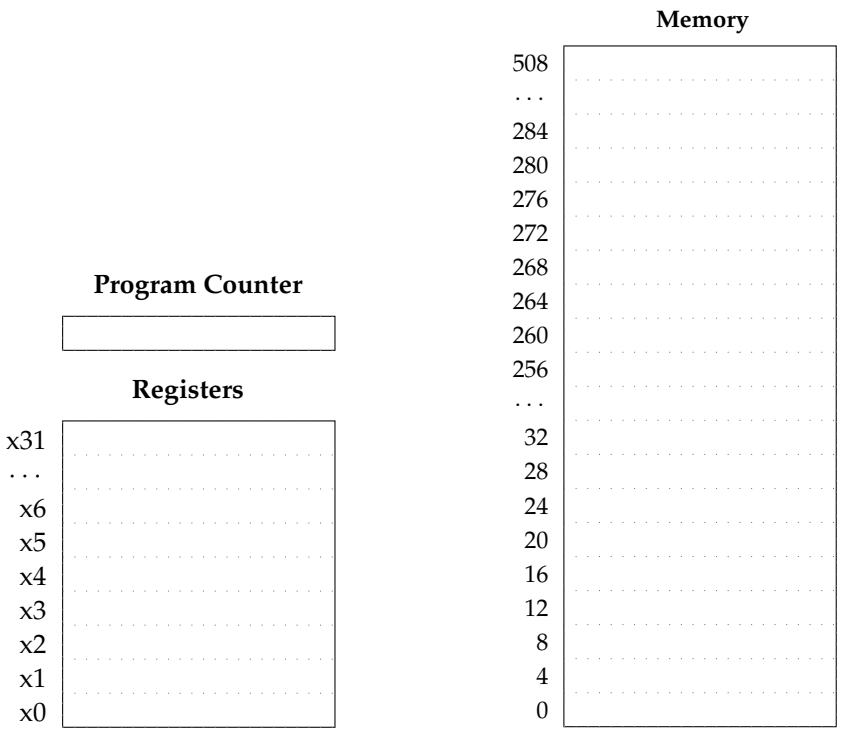
`mul rd, rs1, rs2`

R[`rd`] ← R[`rs1`] × R[`rs2`]
PC ← PC + 4



- Instruction set architecture specification includes:
  - ① Instruction name
  - ② Instruction assembly format (how to represent instructions as strings)
  - ③ Instruction encoding (how to represent instructions as bits)
  - ④ Instruction semantics (what does the instruction actually do)
- Explanation of instruction semantics
  - `imm` : immediate value stored in the instruction
  - R[`rX`] : 4-byte (32-bit) value stored in general-purpose register `rX`
  - sext(`V`) : sign extend the value `V` to 32-bits (fill with immediate's sign bit)
  - Note that R[`x0`] is always the constant zero
- *Arithmetic Instructions*
  - Read a 4-byte (32-bit) value from general-purpose registers or immediates
  - Perform a 4-byte (32-bit) arithmetic operation
  - Write the resulting 4-byte (32-bit) value to a general-purpose register

```
0  addi x1, x0,  9  # R[x1] = R[x0] + 9
4  addi x2, x0,  7
8  add  x3, x1, x2
12 addi x4, x0,  2
16 mul  x1, x2, x4
```



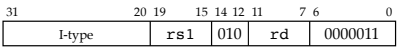
2.2. Memory Instructions

LW

`lw rd, imm(rs1)`

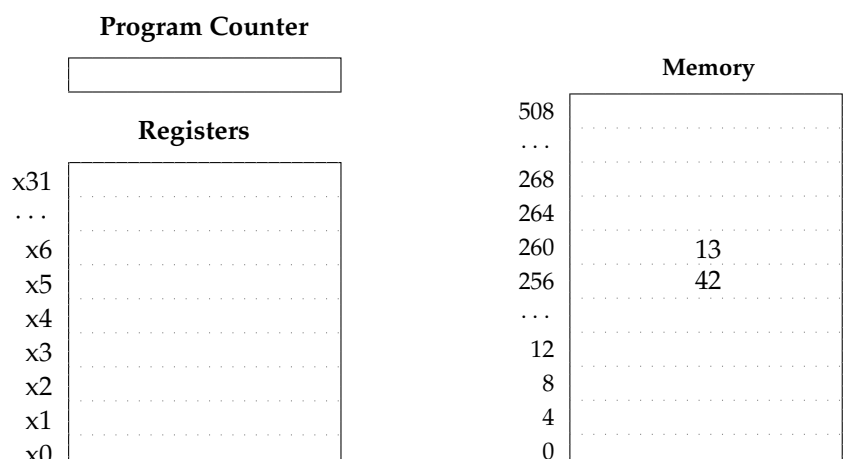
R[rd] ← M[R[rs1] + sext(imm)]

PC ← PC + 4

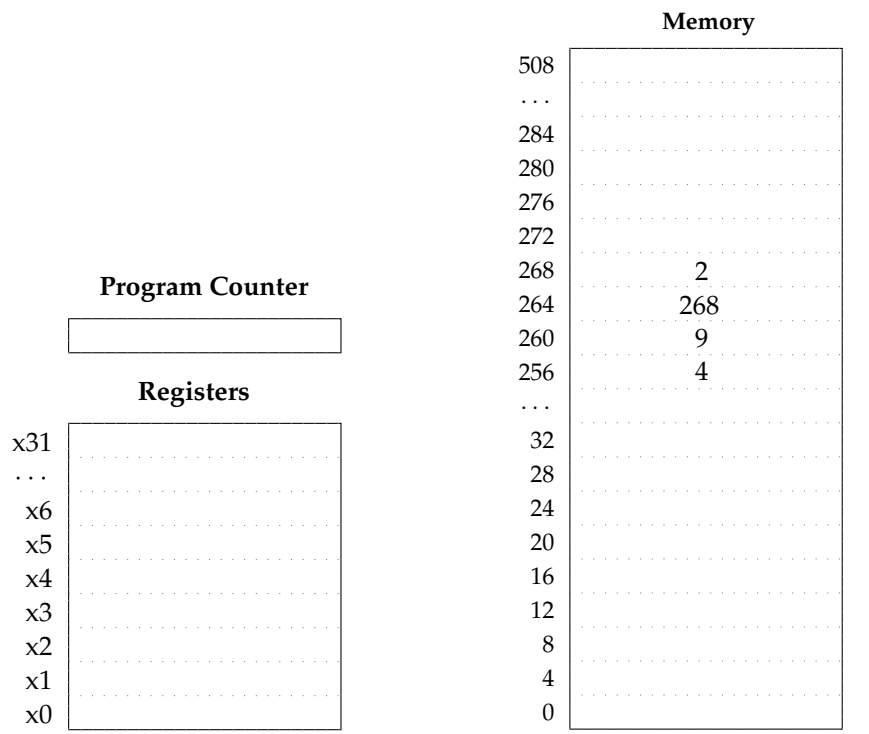


- Explanation of instruction semantics
  - M[A] : 4-byte (32-bit) memory value at byte address A
- *Load Instructions*
  - Generate address using values from a general-purpose reg and immediate
  - Read a 4-byte (32-bit) value from the corresponding memory address
  - Write this 4-byte (32-bit) value to a general-purpose register

```
0 addi x1, x0, 256   # R[x1] = 256
4 lw   x2, 0(x1)     # R[x2] = M[ R[x1] ]
8 lw   x3, 4(x1)     # R[x3] = M[ R[x1]+4 ]
```



```
0   addi x1, x0, 256
4   lw   x2, 0(x1)
8   addi x3, x0, 260
12  lw   x4, 0(x3)
16  add  x5, x2, x3
20  addi x1, x0, 256
24  lw   x2, 8(x1)
28  lw   x3, 0(x2)
32  mul  x6, x5, x3
```

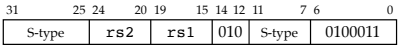


SW

`sw rs2, imm(rs1)`

M[R[`rs1`] + sext(`imm`)] ← R[`rs2`]

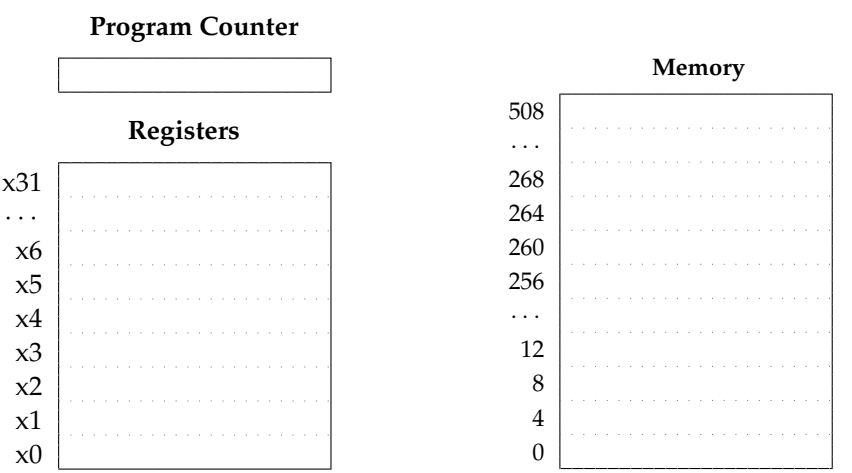
PC ← PC + 4



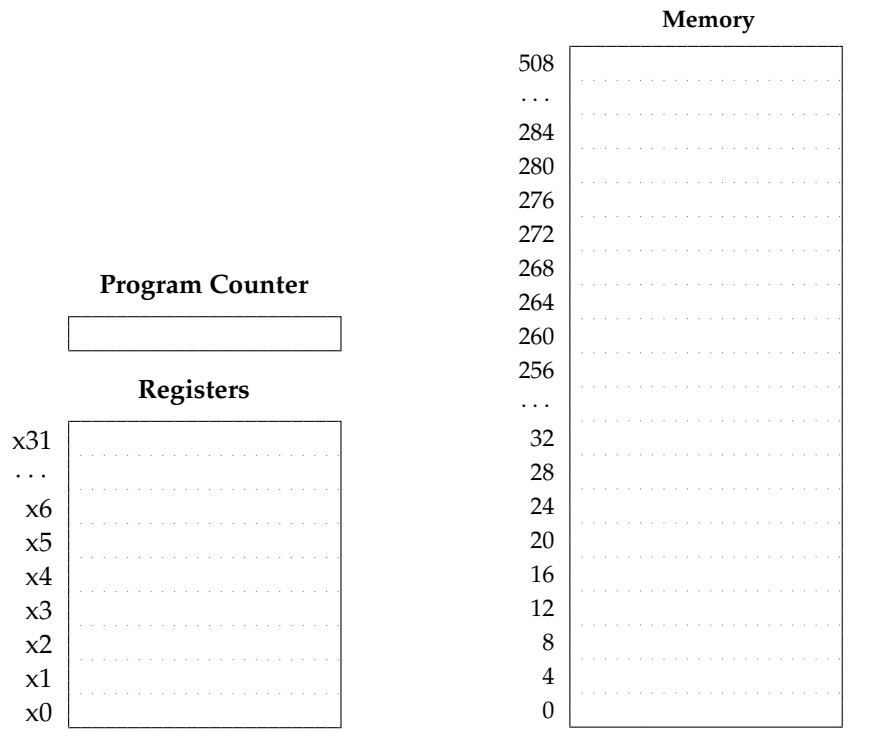
S-type Immediate

- Explanation of instruction semantics
  - M[`A`] : 4-byte (32-bit) memory value at byte address `A`
- *Store Instructions*
  - Generate address using values from a general-purpose reg and immediate
  - Read a 4-byte (32-bit) value from a general-purpose register
  - Write this 4-byte (32-bit) value to the corresponding memory address

```
 0 addi x1, x0, 256  # R[x1] = 256
 4 addi x2, x0, 13   # R[x2] = 13
 8 sw   x2, 0(x1)    # M[ R[x1]   ] = R[x2]
12 sw   x2, 4(x1)    # M[ R[x1]+4 ] = R[x2]
```



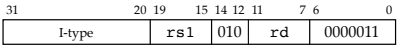
```
0   addi x1, x0, 5
4   addi x2, x0, 3
8   add  x3, x1, x2
12  addi x4, x0, 256
16  sw   x3, 0(x4)
20  addi x4, x4, 4
24  sw   x3, 0(x4)
```



LW

`lw rd, imm(rs1)`

R[`rd`] ← M[R[`rs1`] + sext(`imm`)]
PC ← PC + 4

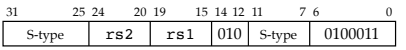


I-type Immediate

SW

`sw rs2, imm(rs1)`

M[R[`rs1`] + sext(`imm`)] ← R[`rs2`]
PC ← PC + 4



S-type Immediate

- Explanation of instruction semantics
  - M[`A`] : 4-byte (32-bit) memory value at byte address `A`
- *Load Instructions*
  - Generate address using values from a general-purpose reg and immediate
  - Read a 4-byte (32-bit) value from the corresponding memory address
  - Write this 4-byte (32-bit) value to a general-purpose register
- *Store Instructions*
  - Generate address using values from a general-purpose reg and immediate
  - Read a 4-byte (32-bit) value from a general-purpose register
  - Write this 4-byte (32-bit) value to the corresponding memory address
- Data memory addresses
  - Memory addresses will always be *byte* addresses (i.e., 4-byte word aligned), thus bottom two bits of the address must always be zero!
  - Unaligned memory accesses (i.e., bottom two bits of the address are non-zero) results in undefined behavior
  - *Virtual address space* is the entire range of addressable memory locations (2^{32} or 4 billion bytes = 1 billion 4-byte words)
  - *Physical address space* is the range of memory locations that can actually be used for storage (2^{9} or 512 bytes = 128 4-byte words)

```
 0 addi x1, x0, 256
 4 addi x2, x0, 268
 8 lw   x3, 0(x1)
12 sw   x3, 0(x2)
16 lw   x3, 4(x1)
20 sw   x3, 4(x2)
24 lw   x3, 8(x1)
28 sw   x3, 8(x2)
```

