ECE 2300 Digital Logic and Computer Organization
Fall 2024

Topic 9: Instruction Set Architecture

School of Electrical and Computer Engineering
Cornell University

revision: 2024-10-29-11-03

1 Computer Organization **3**

1.1. Pico-Processor 6

1.2. Pico-Processor Instruction-Set Architecture 9

1.3. From Pico-Processor to Realistic ISAs 13

2 TinyRV1 **15**

2.1. TinyRV1 Instruction-Set Architecture 17

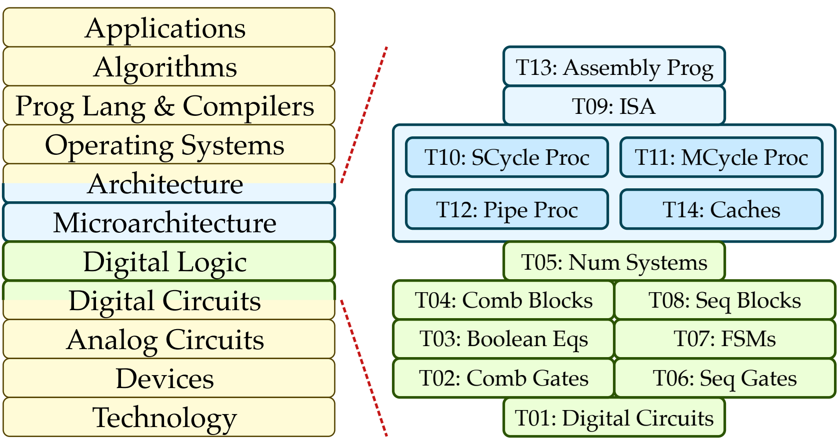
2.2. TinyRV1 Basic Assembly Programming 20

3 From Architecture to Microarchitecture **34**

3.1. Processor Microarchitectural Design Patterns 35

3.2. Transaction Diagrams 36

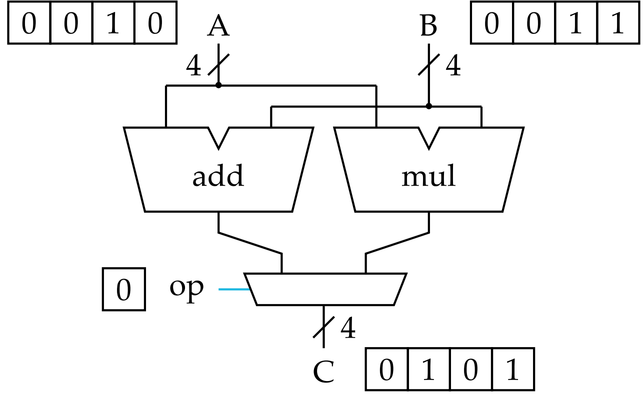
3.3. Analyzing Processor Performance 37



Copyright © 2024 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.

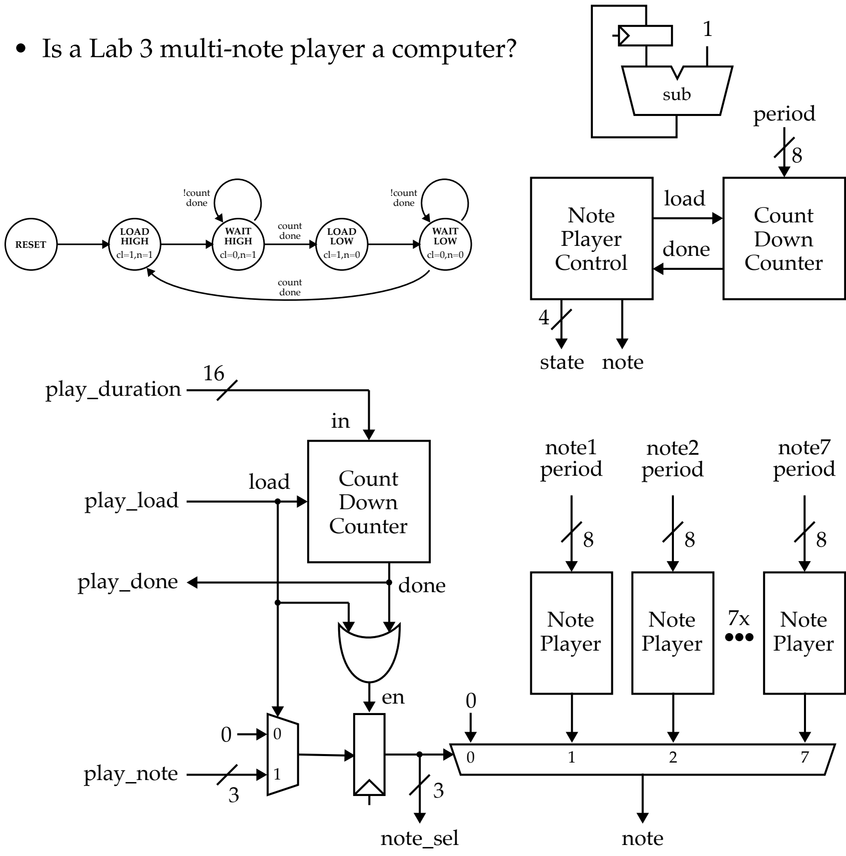
1. Computer Organization

- 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?



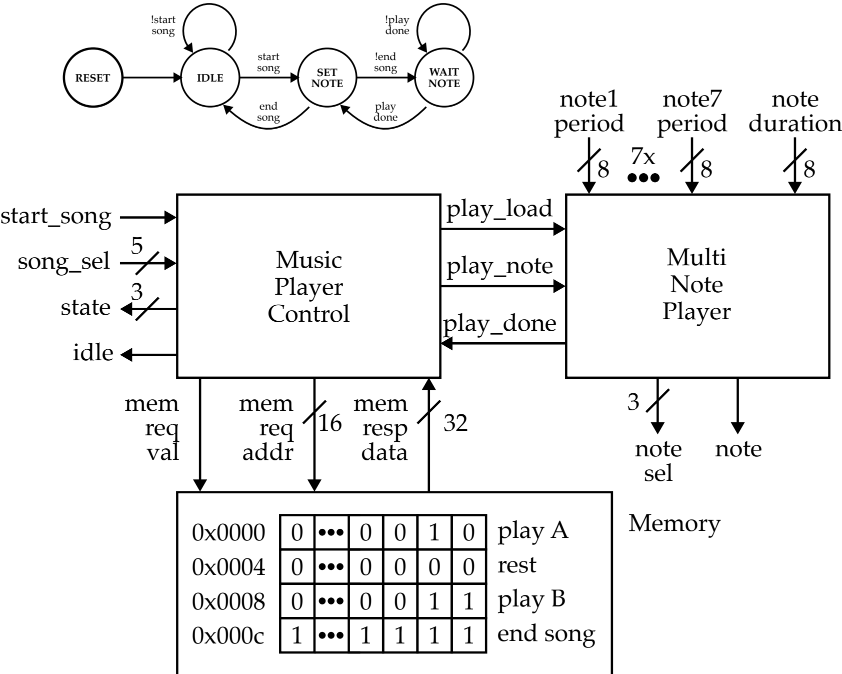
- A computer is "an electronic device for processing and storing data (typically in binary form), according to general-purpose instructions (also in binary form) stored as a program in memory"

- 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"

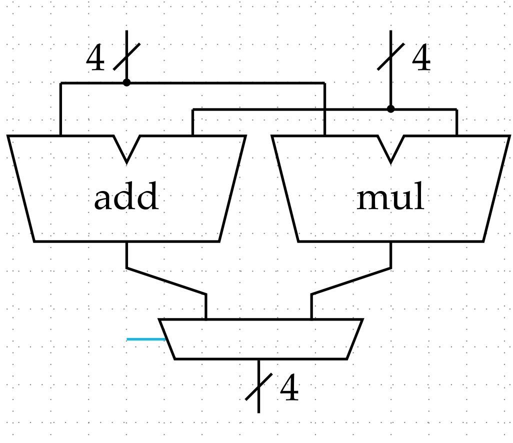
- Is a Lab 3 music player a computer?



- 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”

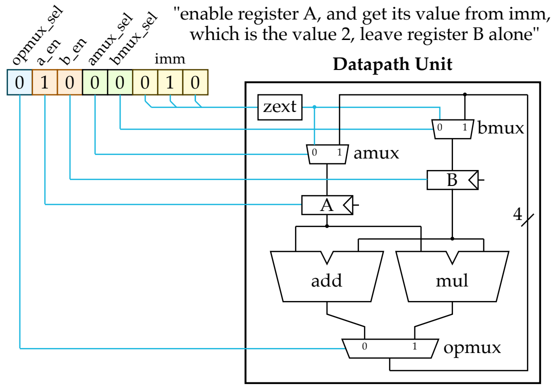
1.1. Pico-Processor

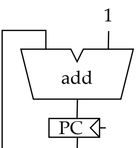
- Let's build a pico-processor [a]
  - Calculator: processing, general-purpose instructions
  - Music Player: storing, program in memory



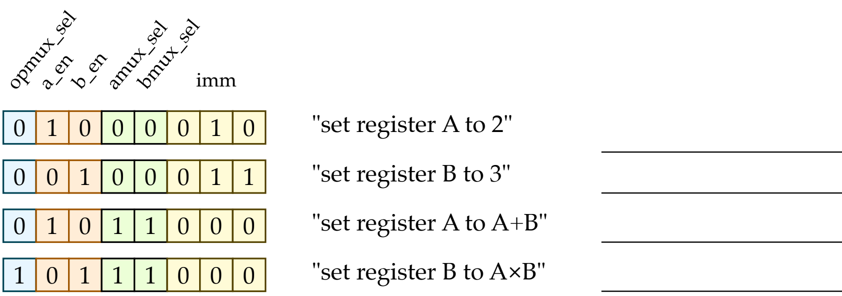
[a] Not a micro-processor (10^{-6}), not a nano-processor (10^{-9}), but a pico-processor (10^{-12})

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





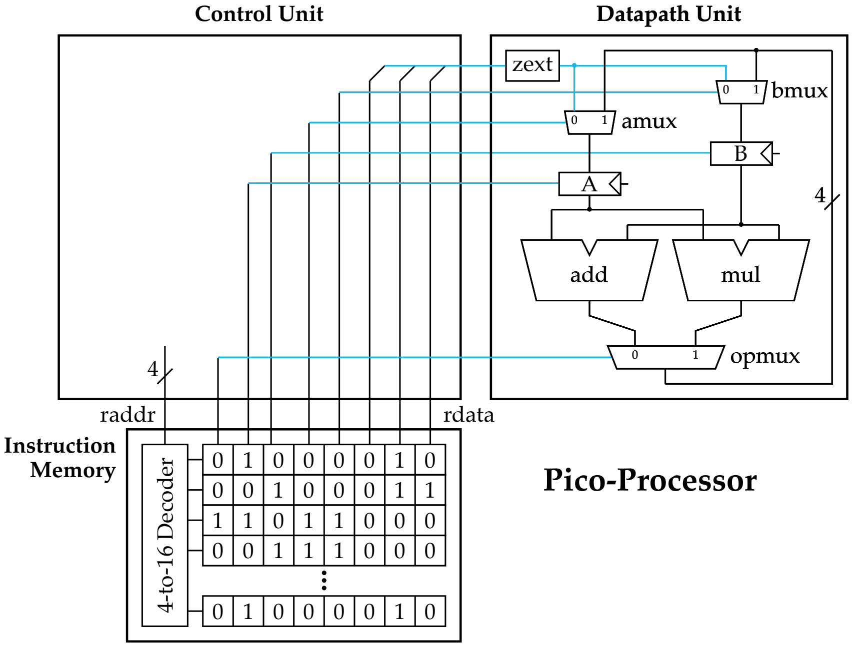
- A program is a sequence of instructions



Machine Instructions
make up a
Machine Program (Program Binary)

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

Applications
Algorithms
Prog Lang & Compilers
Operating Systems
Architecture
Microarchitecture
Digital Logic
Digital Circuits
Analog Circuits
Devices
Technology

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`] ← zext(`imm`)
PC ← PC + 1

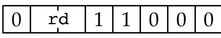
0	`rd`	0	0	`imm`

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

ADD

`add rd`

R[`rd`] ← R[rA] + R[rB]
PC ← PC + 1

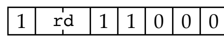


for register A, `rd` = 10
for register B, `rd` = 01

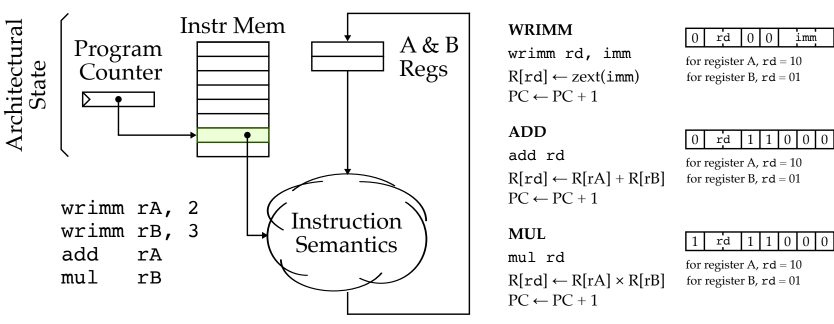
MUL

`mul rd`

R[`rd`] ← R[rA] × R[rB]
PC ← PC + 1



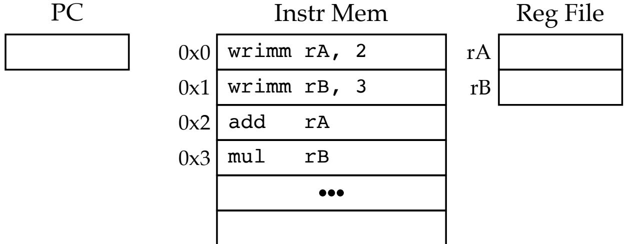
for register A, `rd` = 10
for register B, `rd` = 01



0	1	0	0	0	0	1	0
0	0	1	0	0	0	1	1
1	1	0	1	1	0	0	0
0	0	1	1	1	0	0	0
⋮							
0	1	0	0	0	0	1	0

Static Asm Sequence	Instruction Semantics
`wrimm rA, 2`	
`wrimm rB, 3`	
`add rA`	
`mul rB`	

Worksheet illustrating processor functional-level model

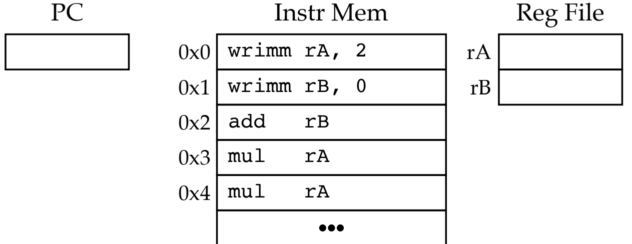


Simulation Table

PC	Dynamic Asm Sequence	rA	rB
	`wrimm rA, 2`		
	`wrimm rB, 3`		
	`add rA`		
	`mul rB`		
	`???`		

Static Asm Sequence	Instruction Semantics
`wrimm rA, 2`	
`wrimm rB, 0`	
`add rB`	
`mul rA`	
`mul rA`	

Worksheet illustrating processor functional-level model



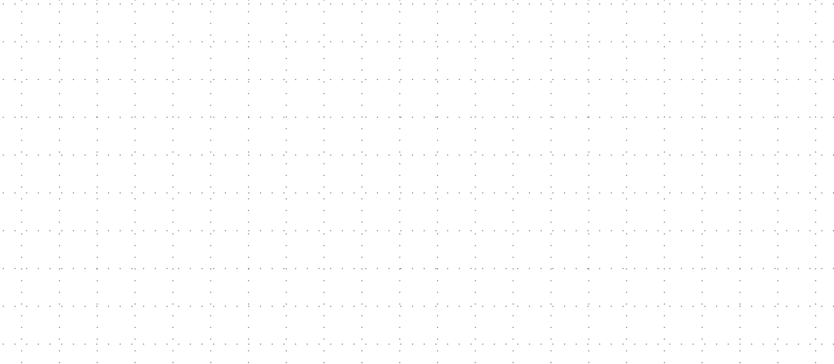
Simulation Table

PC	Dynamic Asm Sequence	rA	rB
	`wrimm rA, 2`		
	`wrimm rB, 0`		
	`add rB`		
	`mul rA`		
	`mul rA`		
	`???`		

1.3. From Pico-Processor to Realistic ISAs

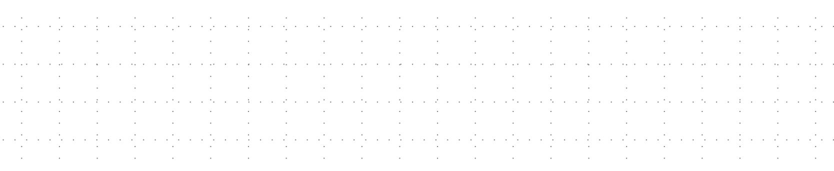
1. How is the data represented?
   - Representations for characters, integers, floating-point
   - Integer formats can be signed or unsigned
   - Floating-point formats can be single- or double-precision
   - Byte addresses can ordered within a word as either little- or big-endian

2. Where can the data be stored?
   - Registers: general-purpose, floating-point, control
   - Memory: different addresses spaces for heap, stack, I/O



3. How can data be accessed?
   - Register: operand stored in registers
   - Immediate: operand is an immediate in the instruction
   - Direct: address of operand in memory is stored in instruction
   - Register Indirect: address of operand in memory is stored in register
   - Displacement: register indirect, addr is added to immediate
   - Autoincrement/decrement: register indirect, addr is automatically adj
   - PC-Relative: displacement is added to the program counter

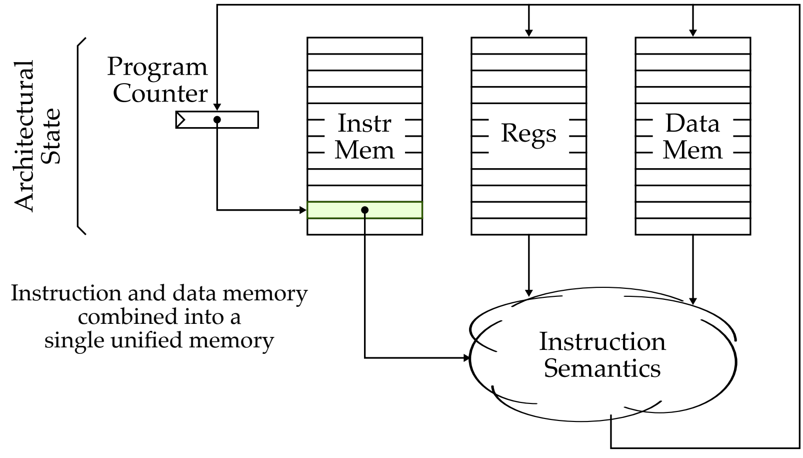
4. What operations can be done on data?
   - Integer and floating-point arithmetic instructions
   - Register and memory data movement instructions
   - Control transfer instructions
   - System control instructions



5. How are instructions encoded?
   - Opcode, addresses of operands and destination, next instruction
   - Variable length vs. fixed length

2. TinyRV1

- 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. TinyRV1 Instruction-Set Architecture

ADD

`add rd, rs1, rs2`

R[rd] ← R[rs1] + R[rs2]
PC ← PC + 4

31–25	24–20	19–15	14–12	11–7	6–0
0000000	rs2	rs1	000	rd	0110011

ADDI

`addi rd, rs1, imm`

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

31–20	19–15	14–12	11–7	6–0
imm	rs1	000	rd	0010011

MUL

`mul rd, rs1, rs2`

R[rd] ← R[rs1] × R[rs2]
PC ← PC + 4

31–25	24–20	19–15	14–12	11–7	6–0
0000001	rs2	rs1	000	rd	0110011

LW

`lw rd, imm(rs1)`

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

31–20	19–15	14–12	11–7	6–0
imm	rs1	010	rd	0000011

SW

`sw rs2, imm(rs1)`

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

31–25	24–20	19–15	14–12	11–7	6–0
imm	rs2	rs1	010	*imm*	0100011

`imm = { inst[31:25], inst[11:7] }`

JAL

`jal rd, imm`

R[rd] ← PC + 4
PC ← PC + sext(imm)

31–12	11–7	6–0
imm	rd	1101111

`imm = { inst[31], inst[19:12], inst[20], inst[30:21], 0 }`

JR

`jr rs1`

PC ← R[rs1]

31–20	19–15	14–12	11–7	6–0
000000000000	rs1	000	00000	1100111

BNE

`bne rs1, rs2, imm`

if (R[rs1] != R[rs2]) PC ← PC + sext(imm)
else PC ← PC + 4

31–25	24–20	19–15	14–12	11–7	6–0
imm	rs2	rs1	001	*imm*	1100011

`imm = { inst[31], inst[7], inst[30:25], inst[11:8], 0 }`

Free & Open Reference Card ①

Base Integer Instructions: RV32I, RV64I, and RV128I

Category	Name	Fmt	RV32I Base		+RV{64,128}	
Loads	Load Byte	I	LB	rd,rs1,imm		
	Load Halfword	I	LH	rd,rs1,imm		
	Load Word	I	LW	rd,rs1,imm	L{D|Q}	rd,rs1,imm
	Load Byte Unsigned	I	LBU	rd,rs1,imm		
	Load Half Unsigned	I	LHU	rd,rs1,imm	L{W|D}U	rd,rs1,imm
Stores	Store Byte	S	SB	rs1,rs2,imm		
	Store Halfword	S	SH	rs1,rs2,imm		
	Store Word	S	SW	rs1,rs2,imm	S{D|Q}	rs1,rs2,imm
Shifts	Shift Left	R	SLL	rd,rs1,rs2	SLL{W|D}	rd,rs1,rs2
	Shift Left Immediate	I	SLLI	rd,rs1,shamt	SLLI{W|D}	rd,rs1,shamt
	Shift Right	R	SRL	rd,rs1,rs2	SRL{W|D}	rd,rs1,rs2
	Shift Right Immediate	I	SRLI	rd,rs1,shamt	SRLI{W|D}	rd,rs1,shamt
	Shift Right Arithmetic	R	SRA	rd,rs1,rs2	SRA{W|D}	rd,rs1,rs2
	Shift Right Arith Imm	I	SRAI	rd,rs1,shamt	SRAI{W|D}	rd,rs1,shamt
Arithmetic	ADD	R	ADD	rd,rs1,rs2	ADD{W|D}	rd,rs1,rs2
	ADD Immediate	I	ADDI	rd,rs1,imm	ADDI{W|D}	rd,rs1,imm
	SUBtract	R	SUB	rd,rs1,rs2	SUB{W|D}	rd,rs1,rs2
	Load Upper Imm	U	LUI	rd,imm		
	Add Upper Imm to PC	U	AUIPC	rd,imm		
Logical	XOR	R	XOR	rd,rs1,rs2		
	XOR Immediate	I	XORI	rd,rs1,imm		
	OR	R	OR	rd,rs1,rs2		
	OR Immediate	I	ORI	rd,rs1,imm		
	AND	R	AND	rd,rs1,rs2		
	AND Immediate	I	ANDI	rd,rs1,imm		
Compare	Set <	R	SLT	rd,rs1,rs2		
	Set < Immediate	I	SLTI	rd,rs1,imm		
	Set < Unsigned	R	SLTU	rd,rs1,rs2		
	Set < Imm Unsigned	I	SLTIU	rd,rs1,imm		
Branches	Branch =	SB	BEQ	rs1,rs2,imm		
	Branch ≠	SB	BNE	rs1,rs2,imm		
	Branch <	SB	BLT	rs1,rs2,imm		
	Branch ≥	SB	BGE	rs1,rs2,imm		
	Branch < Unsigned	SB	BLTU	rs1,rs2,imm		
	Branch ≥ Unsigned	SB	BGEU	rs1,rs2,imm		
Jump & Link	J&L	UJ	JAL	rd,imm		
	Jump & Link Register	UJ	JALR	rd,rs1,imm		
Synch	Synch thread	I	FENCE			
	Synch Instr & Data	I	FENCE.I			
System	System CALL	I	SCALL			
	System BREAK	I	SBREAK			
Counters	ReaD CYCLE	I	RDCYCLE	rd		
	ReaD CYCLE upper Half	I	RDCYCLEH	rd		
	ReaD TIME	I	RDTIME	rd		
	ReaD TIME upper Half	I	RDTIMEH	rd		
	ReaD INSTR RETired	I	RDINSTRET	rd		
	ReaD INSTR upper Half	I	RDINSTRETH	rd		

RV Privileged Instructions

Category	Name	RV mnemonic	
CSR Access	Atomic R/W	CSRRW	rd,csr,rs1
	Atomic Read & Set Bit	CSRRS	rd,csr,rs1
	Atomic Read & Clear Bit	CSRRC	rd,csr,rs1
	Atomic R/W Imm	CSRRWI	rd,csr,imm
	Atomic Read & Set Bit Imm	CSRRSI	rd,csr,imm
	Atomic Read & Clear Bit Imm	CSRRCI	rd,csr,imm
Change Level	Env. Call	ECALL	
	Environment Breakpoint	EBREAK	
	Environment Return	ERET	
Trap Redirect	to Supervisor	MRTS	
	Redirect Trap to Hypervisor	MRTH	
	Hypervisor Trap to Supervisor	HRTS	
Interrupt	Wait for Interrupt	WFI	
MMU	Supervisor FENCE	SFENCE.VM	rs1

Optional Compressed (16-bit) Instruction Extension: RVC

Category	Name	Fmt	RVC		RVI equivalent	
Loads	Load Word	CL	C.LW	rd′,rs1′,imm	LW	rd′,rs1′,imm*4
	Load Word SP	CI	C.LWSP	rd,imm	LW	rd,sp,imm*4
	Load Double	CL	C.LD	rd′,rs1′,imm	LD	rd′,rs1′,imm*8
	Load Double SP	CI	C.LDSP	rd,imm	LD	rd,sp,imm*8
	Load Quad	CL	C.LQ	rd′,rs1′,imm	LQ	rd′,rs1′,imm*16
	Load Quad SP	CI	C.LQSP	rd,imm	LQ	rd,sp,imm*16
Stores	Store Word	CS	C.SW	rs1′,rs2′,imm	SW	rs1′,rs2′,imm*4
	Store Word SP	CSS	C.SWSP	rs2,imm	SW	rs2,sp,imm*4
	Store Double	CS	C.SD	rs1′,rs2′,imm	SD	rs1′,rs2′,imm*8
	Store Double SP	CSS	C.SDSP	rs2,imm	SD	rs2,sp,imm*8
	Store Quad	CS	C.SQ	rs1′,rs2′,imm	SQ	rs1′,rs2′,imm*16
	Store Quad SP	CSS	C.SQSP	rs2,imm	SQ	rs2,sp,imm*16
Arithmetic	ADD	CR	C.ADD	rd,rs1	ADD	rd,rd,rs1
	ADD Word	CR	C.ADDW	rd,rs1	ADDW	rd,rd,imm
	ADD Immediate	CI	C.ADDI	rd,imm	ADDI	rd,rd,imm
	ADD Word Imm	CI	C.ADDIW	rd,imm	ADDIW	rd,rd,imm
	ADD SP Imm * 16	CI	C.ADDI16SP	x0,imm	ADDI	sp,sp,imm*16
	ADD SP Imm * 4	CIW	C.ADDI4SPN	rd',imm	ADDI	rd',sp,imm*4
	Load Immediate	CI	C.LI	rd,imm	ADDI	rd,x0,imm
	Load Upper Imm	CI	C.LUI	rd,imm	LUI	rd,imm
	MoVe	CR	C.MV	rd,rs1	ADD	rd,rs1,x0
	SUB	CR	C.SUB	rd,rs1	SUB	rd,rd,rs1
Shifts	Shift Left Imm	CI	C.SLLI	rd,imm	SLLI	rd,rd,imm
Branches	Branch=0	CB	C.BEQZ	rs1′,imm	BEQ	rs1',x0,imm
	Branch≠0	CB	C.BNEZ	rs1′,imm	BNE	rs1',x0,imm
Jump	Jump	CJ	C.J	imm	JAL	x0,imm
	Jump Register	CR	C.JR	rd,rs1	JALR	x0,rs1,0
Jump & Link	J&L	CJ	C.JAL	imm	JAL	ra,imm
	Jump & Link Register	CR	C.JALR	rs1	JALR	ra,rs1,0
System	Env. BREAK	CI	C.EBREAK		EBREAK	

32-bit Instruction Formats

	31	30–25	24–21	20	19–15	14–12	11–8	7	6–0
R	funct7		rs2		rs1	funct3	rd		opcode
I	imm[11:0]				rs1	funct3	rd		opcode
S	imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode
SB	imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode
U	imm[31:12]						rd		opcode
UJ	imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode

16-bit (RVC) Instruction Formats

	15–13	12	11–10	9–7	6–5	4–2	1–0
CR	funct4		rd/rs1		rs2		op
CI	funct3	imm	rd/rs1		imm		op
CSS	funct3	imm			rs2		op
CIW	funct3	imm				rd′	op
CL	funct3	imm		rs1′	imm	rd′	op
CS	funct3	imm		rs1′	imm	rs2′	op
CB	funct3	offset		rs1′	offset		op
CJ	funct3	jump target					op

RISC-V Integer Base (RV32I/64I/128I), privileged, and optional compressed extension (RVC). Registers x1-x31 and the pc are 32 bits wide in RV32I, 64 in RV64I, and 128 in RV128I (x0=0). RV64I/128I add 10 instructions for the wider formats. The RVI base of <50 classic integer RISC instructions is required. Every 16-bit RVC instruction matches an existing 32-bit RVI instruction. See risc.org.

Optional Multiply-Divide Instruction Extension: RVM

Category	Name	Fmt	RV32M (Multiply-Divide)	+RV{64,128}
Multiply	MULtiply	R	MUL rd,rs1,rs2	MUL{W|D} rd,rs1,rs2
	MULtiply upper Half	R	MULH rd,rs1,rs2	
	MULtiply Half Sign/Uns	R	MULHSU rd,rs1,rs2	
	MULtiply upper Half Uns	R	MULHU rd,rs1,rs2	
Divide	DIVide	R	DIV rd,rs1,rs2	DIV{W|D} rd,rs1,rs2
	DIVide Unsigned	R	DIVU rd,rs1,rs2	
Remainder	REMainder	R	REM rd,rs1,rs2	REM{W|D} rd,rs1,rs2
	REMainder Unsigned	R	REMU rd,rs1,rs2	REMU{W|D} rd,rs1,rs2

Optional Atomic Instruction Extension: RVA

Category	Name	Fmt	RV32A (Atomic)	+RV{64,128}
Load	Load Reserved	R	LR.W rd,rs1	LR.{D|Q} rd,rs1
Store	Store Conditional	R	SC.W rd,rs1,rs2	SC.{D|Q} rd,rs1,rs2
Swap	SWAP	R	AMOSWAP.W rd,rs1,rs2	AMOSWAP.{D|Q} rd,rs1,rs2
Add	ADD	R	AMOADD.W rd,rs1,rs2	AMOADD.{D|Q} rd,rs1,rs2
Logical	XOR	R	AMOXOR.W rd,rs1,rs2	AMOXOR.{D|Q} rd,rs1,rs2
	AND	R	AMOAND.W rd,rs1,rs2	AMOAND.{D|Q} rd,rs1,rs2
	OR	R	AMOOR.W rd,rs1,rs2	AMOOR.{D|Q} rd,rs1,rs2
Min/Max	MINimum	R	AMOMIN.W rd,rs1,rs2	AMOMIN.{D|Q} rd,rs1,rs2
	MAXimum	R	AMOMAX.W rd,rs1,rs2	AMOMAX.{D|Q} rd,rs1,rs2
	MINimum Unsigned	R	AMOMINU.W rd,rs1,rs2	AMOMINU.{D|Q} rd,rs1,rs2
	MAXimum Unsigned	R	AMOMAXU.W rd,rs1,rs2	AMOMAXU.{D|Q} rd,rs1,rs2

Three Optional Floating-Point Instruction Extensions: RVF, RVD, & RVQ

Category	Name	Fmt	RV32{F|D|Q} (HP/SP,DP,QP Fl Pt)	+RV{64,128}
Move	Move from Integer	R	FMV.{H|S}.X rd,rs1	FMV.{D|Q}.X rd,rs1
	Move to Integer	R	FMV.X.{H|S} rd,rs1	FMV.X.{D|Q} rd,rs1
Convert	Convert from Int	R	FCVT.{H|S|D|Q}.W rd,rs1	FCVT.{H|S|D|Q}.{L|T} rd,rs1
	Convert from Int Unsigned	R	FCVT.{H|S|D|Q}.WU rd,rs1	FCVT.{H|S|D|Q}.{L|T}U rd,rs1
	Convert to Int	R	FCVT.W.{H|S|D|Q} rd,rs1	FCVT.{L|T}.{H|S|D|Q} rd,rs1
	Convert to Int Unsigned	R	FCVT.WU.{H|S|D|Q} rd,rs1	FCVT.{L|T}U.{H|S|D|Q} rd,rs1
Load	Load	I	FL{W,D,Q} rd,rs1,imm	
Store	Store	S	FS{W,D,Q} rs1,rs2,imm	
Arithmetic	ADD	R	FADD.{S|D|Q} rd,rs1,rs2	
	SUBtract	R	FSUB.{S|D|Q} rd,rs1,rs2	
	MULtiply	R	FMUL.{S|D|Q} rd,rs1,rs2	
	DIVide	R	FDIV.{S|D|Q} rd,rs1,rs2	
	SQuare RooT	R	FSQRT.{S|D|Q} rd,rs1	
Mul-Add	Multiply-ADD	R	FMADD.{S|D|Q} rd,rs1,rs2,rs3	
	Multiply-SUBtract	R	FMSUB.{S|D|Q} rd,rs1,rs2,rs3	
	Negative Multiply-SUBtract	R	FNMSUB.{S|D|Q} rd,rs1,rs2,rs3	
	Negative Multiply-ADD	R	FNMADD.{S|D|Q} rd,rs1,rs2,rs3	
Sign Inject	SiGN source	R	FSGNJ.{S|D|Q} rd,rs1,rs2	
	Negative SiGN source	R	FSGNJN.{S|D|Q} rd,rs1,rs2	
	Xor SiGN source	R	FSGNJX.{S|D|Q} rd,rs1,rs2	
Min/Max	MINimum	R	FMIN.{S|D|Q} rd,rs1,rs2	
	MAXimum	R	FMAX.{S|D|Q} rd,rs1,rs2	
Compare	Compare Float =	R	FEQ.{S|D|Q} rd,rs1,rs2	
	Compare Float <	R	FLT.{S|D|Q} rd,rs1,rs2	
	Compare Float ≤	R	FLE.{S|D|Q} rd,rs1,rs2	
Categorization	Classify Type	R	FCLASS.{S|D|Q} rd,rs1	
Configuration	Read Status	R	FRCSR rd	
	Read Rounding Mode	R	FRRM rd	
	Read Flags	R	FRFLAGS rd	
	Swap Status Reg	R	FSCSR rd,rs1	
	Swap Rounding Mode	R	FSRM rd,rs1	
	Swap Flags	R	FSFLAGS rd,rs1	
	Swap Rounding Mode Imm	I	FSRMI rd,imm	
	Swap Flags Imm	I	FSFLAGSI rd,imm	

RISC-V Calling Convention

Register	ABI Name	Saver	Description
x0	zero	---	Hard-wired zero
x1	ra	Caller	Return address
x2	sp	Callee	Stack pointer
x3	gp	---	Global pointer
x4	tp	---	Thread pointer
x5-7	t0-2	Caller	Temporaries
x8	s0/fp	Callee	Saved register/frame pointer
x9	s1	Callee	Saved register
x10-11	a0-1	Caller	Function arguments/return values
x12-17	a2-7	Caller	Function arguments
x18-27	s2-11	Callee	Saved registers
x28-31	t3-t6	Caller	Temporaries
f0-7	ft0-7	Caller	FP temporaries
f8-9	fs0-1	Callee	FP saved registers
f10-11	fa0-1	Caller	FP arguments/return values
f12-17	fa2-7	Caller	FP arguments
f18-27	fs2-11	Callee	FP saved registers
f28-31	ft8-11	Caller	FP temporaries

RISC-V calling convention and five optional extensions: 10 multiply-divide instructions (RV32M); 11 optional atomic instructions (RV32A); and 25 floating-point instructions each for single-, double-, and quadruple-precision (RV32F, RV32D, RV32Q). The latter add registers f0-f31, whose width matches the widest precision, and a floating-point control and status register fcsr. Each larger address adds some instructions: 4 for RVM, 11 for RVA, and 6 each for RVF/D/Q. Using regex notation, `{}` *means set, so* `L{D|Q}` *is both* `LD` *and* `LQ`*. See risc.org. (8/21/15 revision)*

2.2. TinyRV1 Basic Assembly Programming

Arithmetic Operations

Pseudo-Code

```
1
2 a = 0
3 a = 13
4 a = b + 13
5 a = b + c
6 a = b * c
```

Assembly Code

```
1 # a:x1, b:x2, c:x3
2 addi x1, x0, 0
3 addi x1, x0, 13
4 addi x1, x2, 13
5 add  x1, x2, x3
6 mul  x1, x2, x3
```

Static Asm Sequence	Instruction Semantics
`addi x1, x0, 0`	$R[x1] \leftarrow R[x0] + 0$
`addi x1, x0, 13`	$R[x1] \leftarrow R[x0] + 13$
`addi x1, x2, 13`	$R[x1] \leftarrow R[x2] + 13$
`add x1, x2, x3`	$R[x1] \leftarrow R[x2] + R[x3]$
`mul x1, x2, x3`	$R[x1] \leftarrow R[x2] \times R[x3]$

Worksheet illustrating processor functional-level model

PC: [ ]

Instr Mem

Address	Instruction
	•••
0x1000	`addi x1, x0, 0`
0x1004	`addi x1, x0, 13`
0x1008	`addi x1, x2, 13`
0x100c	`add x1, x2, x3`
0x1010	`mul x1, x2, x3`

Reg File

Register	Value
x0	0
x1	
x2	2
x3	3
	•••
x31	

Data Mem

Address	Value
	•••
0x2000	
0x2004	
0x2008	
0x200c	
	•••

Simulation Table

PC	Dynamic Asm Sequence	x1	x2	x3
	`addi x1, x0, 0`		2	3
	`addi x1, x0, 13`			
	`addi x1, x2, 13`			
	`add x1, x2, x3`			
	`mul x1, x2, x3`			
	`opA`			

Memory Operations
Reading/writing an array

Pseudo-Code

```
1
2 a = d[0]
3 a = d[1]
4 d[0] = a
5 d[2] = a
```

Assembly Code

```
1 # a:x1, addr(d):x2
2 lw x1, 0(x2)
3 lw x1, 4(x2)
4 sw x1, 0(x2)
5 sw x1, 8(x2)
```

Memory Operations
Iterating through an array (mul/add indexing)

Pseudo-Code

```
1
2 i = 0
3
4 a = d[i]
5
6
7
8 i = i + 1
9 a = d[i]
```

Assembly Code

```
 1 # i:x1, a:x2, addr(d):x3, addr(d[i]):x4
 2 addi x1, x0, 0
 3
 4 mul  x4, x1, 4
 5 add  x4, x3, x4
 6 lw   x2, 0(x4)
 7
 8 addi x1, x1, 1
 9 mul  x4, x1, 4
10 add  x4, x3, x4
11 lw   x2, 0(x4)
```

Memory Operations
Iterating through an array (iterative indexing)

Pseudo-Code

```
1 i = 0
2 a = d[i]
3 i = i + 1
4 a = d[i]
```

Assembly Code

```
1 # a:x1, addr(d[i]):x2
2 lw   x1, 0(x2)
3 addi x2, x2, 4
4 lw   x1, 0(x2)
```

Static Asm Sequence	Instruction Semantics
`lw x1, 0(x2)`	R[x1] ← M[R[x2] + 0]
`lw x1, 4(x2)`	R[x1] ← M[R[x2] + 4]
`sw x1, 0(x2)`	M[R[x2] + 0] ← R[x1]
`sw x1, 8(x2)`	M[R[x2] + 8] ← R[x1]

Worksheet illustrating processor functional-level model

PC: (blank)

Instr Mem Addr	Instr Mem	Reg	Reg File	Data Mem Addr	Data Mem
	•••	x0	0		•••
0x1000	`lw x1, 0(x2)`	x1		0x2000	13
0x1004	`lw x1, 4(x2)`	x2	0x2000	0x2004	42
0x1008	`sw x1, 0(x2)`	x3		0x2008	
0x100c	`sw x1, 8(x2)`		•••	0x200c	
0x1010	`opA`	x31			•••

Simulation Table

PC	Dynamic Asm Sequence	x1	x2
	`lw x1, 0(x2)`		0x2000
	`lw x1, 4(x2)`		
	`sw x1, 0(x2)`		
	`sw x1, 8(x2)`		
	`opA`		

Conditional Operations
If conditional

Pseudo-Code

```
1
2 if ( a == 0 ):
3   b = 13
```

Assembly Code

```
1   # a:x1, b:x2
2   bne  x1, x0, L1
3   addi x2, x0, 13
4 L1:
```

Conditional Operations
If/else conditional

Pseudo-Code

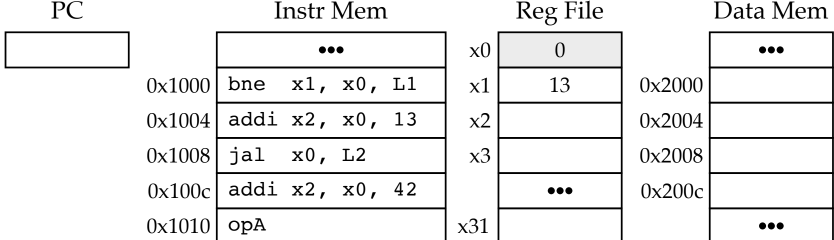
```
1
2 if ( a == 0 ):
3   b = 13
4 else:
5   b = 42
```

Assembly Code

```
1   # a:x1, b:x2
2   bne  x1, x0, L1
3   addi x2, x0, 13
4   jal  x0, L2
5 L1:
6   addi x2, x0, 42
7 L2:
```

Static Asm Sequence	Instruction Semantics
`bne x1, x0, L1`	if (R[x1] != R[x2]) PC ← L1
`addi x2, x0, 13`	R[x1] ← R[x0] + 13
`jal x0, L2`	R[x0] ← PC + 4; PC ← L2
`L1: addi x2, x0, 42`	R[x1] ← R[x0] + 42
`L2: opA`	

Worksheet illustrating processor functional-level model



Simulation Table

PC	Dynamic Asm Sequence	x1	x2
	`bne x1, x0, L1`	13	
	`opA`		

Static Asm Sequence	Instruction Semantics
`bne x1, x0, L1`	if (R[x1] != R[x2]) PC ← L1
`addi x2, x0, 13`	R[x1] ← R[x0] + 13
`jal x0, L2`	R[x0] ← PC + 4; PC ← L2
`L1: addi x2, x0, 42`	R[x1] ← R[x0] + 42
`L2: opA`	

Worksheet illustrating processor functional-level model

PC

	Instr Mem
	•••
0x1000	`bne x1, x0, L1`
0x1004	`addi x2, x0, 13`
0x1008	`jal x0, L2`
0x100c	`addi x2, x0, 42`
0x1010	`opA`

	Reg File
x0	0
x1	0
x2	
x3	
	•••
x31	

	Data Mem
	•••
0x2000	
0x2004	
0x2008	
0x200c	
	•••

Simulation Table

PC	Dynamic Asm Sequence	x1	x2
	`bne x1, x0, L1`	0	
	`addi x2, x0, 13`		
	`jal x0, L2`		
	`opA`		

Loop

Multiplying a value by itself n times (counting up)

Pseudo-Code

```
1
2 a = 2
3
4 for i in range(n):
5   a = a * a
```

Assembly Code

```
1   # a:x1, i:x2, n:x3
2   addi x1, x0, 2
3   addi x2, x0, 0
4 loop:
5   mul  x1, x1, x1
6   addi x2, x2, 1
7   bne  x2, x3, loop
```

Loop

Multiplying a value by itself n times (counting down)

Pseudo-Code

```
1
2 a = 2
3 for i in range(n):
4   a = a * a
```

Assembly Code

```
1   # a:x1, (n-i):x2
2   addi x1, x0, 2
3 loop:
4   mul  x1, x1, x1
5   addi x2, x2, -1
6   bne  x2, x0, loop
```

Static Asm Sequence	Instruction Semantics
`addi x1, x0, 2`	R[x1] ← R[x0] + 2
`loop: mul x1, x1, x1`	R[x1] ← R[x1] + R[x1]
`addi x2, x2, -1`	R[x2] ← R[x2] + (-1)
`bne x2, x0, loop`	if (R[x2] != R[x0]) PC ← loop

Worksheet illustrating processor functional-level model

PC

	Instr Mem
	•••
0x1000	`addi x1, x0, 2`
0x1004	`mul x1, x1, x1`
0x1008	`addi x2, x2, -1`
0x100c	`bne x2, x0, loop`
0x1010	`opA`

	Reg File
x0	0
x1	
x2	10
x3	
	•••
x31	

	Data Mem
	•••
0x2000	
0x2004	
0x2008	
0x200c	
	•••

Simulation Table

PC	Dynamic Asm Sequence	x1	x2
	`addi x1, x0, 2`		10
	`mul x1, x1, x1`		
	`addi x2, x2, -1`		
	`bne x2, x0, loop`		
	`mul x1, x1, x1`		
	`addi x2, x2, -1`		
	`bne x2, x0, loop`		

Loop

Accumulating elements in array

Pseudo-Code

```
1
2
3 sum = 0
4 for i in range(n):
5   a = d[i]
6   sum = sum + a
```

Assembly Code

```
1   # sum:x1, (n-i):x2
2   # a:x3, addr(d[i]):x4
3   addi x1, x0, 0
4 loop:
5   lw   x3, 0(x4)
6   add  x1, x1, x3
7   addi x4, x4, 4
8   addi x2, x2, -1
9   bne  x2, x0, loop
```

Static Asm Sequence	Instruction Semantics
`addi x1, x0, 0`	R[x1] ← R[x0] + 0
`loop: lw x3, 0(x4)`	R[x3] ← M[R[x4] + 0]
`add x1, x1, x3`	R[x1] ← R[x1] + R[x3]
`addi x4, x4, 4`	R[x4] ← R[x4] + 4
`addi x2, x2, -1`	R[x2] ← R[x2] + (-1)
`bne x2, x0, loop`	if (R[x2] != R[x0]) PC ← loop

Worksheet illustrating processor functional-level model

PC

Instr Mem

Address	Instruction
	•••
0x1000	`addi x1, x0, 0`
0x1004	`lw x3, 0(x4)`
0x1008	`add x1, x1, x3`
0x100c	`addi x4, x4, 4`
0x1010	`addi x2, x2, -1`
0x1014	`bne x2, x0, loop`

Reg File

Register	Value
x0	0
x1	
x2	10
x3	
x4	0x2000
	•••
x31	

Data Mem

Address	Value
	•••
0x2000	13
0x2004	42
0x2008	7
0x200c	9
	•••

Static Asm Sequence	Instruction Semantics
`addi x1, x0, 0`	R[x1] ← R[x0] + 0
`loop: lw x3, 0(x4)`	R[x3] ← M[R[x4] + 0]
`add x1, x1, x3`	R[x1] ← R[x1] + R[x3]
`addi x4, x4, 4`	R[x4] ← R[x4] + 4
`addi x2, x2, -1`	R[x2] ← R[x2] + (-1)
`bne x2, x0, loop`	if (R[x2] != R[x0]) PC ← loop

Simulation Table

PC	Dynamic Asm Sequence	x1	x2	x3	x4
	`addi x1, x0, 0`		10		0x2000
	`lw x3, 0(x4)`				
	`add x1, x1, x3`				
	`addi x4, x4, 4`				
	`addi x2, x2, -1`				
	`bne x2, x0, loop`				

Pseudo-code for elementwise vector-vector addition.

```
for i in range(n):
  dest[i] = src0[i] + src1[i]
```

Implement this pseudo-code using TinyRV1 assembly code.

```
# addr(dest[i]):x1, addr(src0[i]):x2, addr(src1[i]):x3, n:x4
```

Note that we are ignoring the fact that our assembly code will not function correctly if `n` <= 0. Our assembly code would need an additional check before entering the loop to ensure that `n` > 0. Unless otherwise stated, we will assume in this course that array bounds are greater than zero to simplify our analysis.

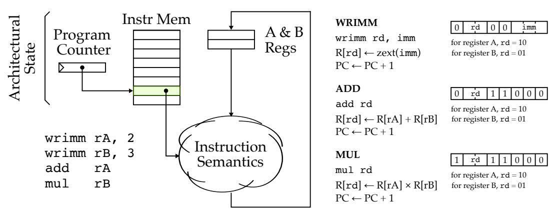
TinyRV1 assembly code for mystery program.

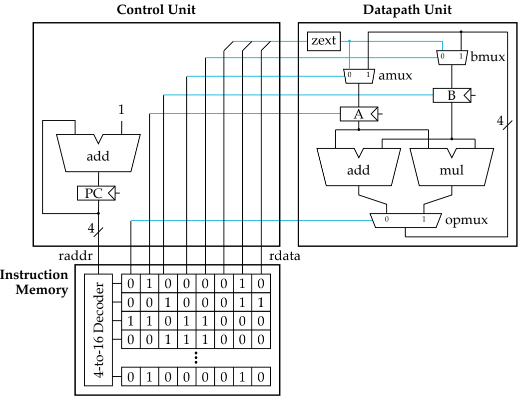
```
  addi  x5, x0, 0

 loop:
  lw    x4, 0(x1)
  bne   x4, x3, foo
  addi  x5, x0, 1

 foo:
  addi  x1, x1, 4
  addi  x2, x2, -1
  bne   x2, x0, loop
```

Show the pseudo-code for this mystery program.

3. From Architecture to Microarchitecture

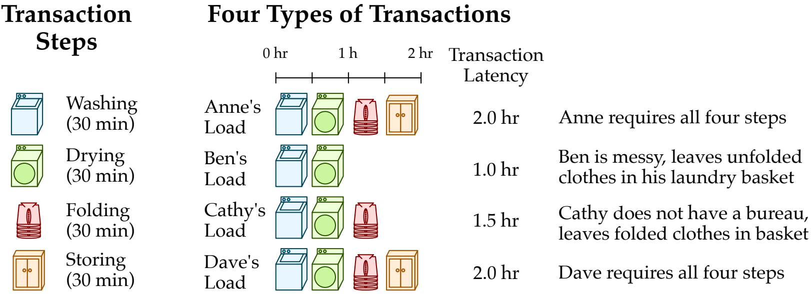
- Processor
  - Instructions are "transactions" that execute on a processor
  - Architecture: defines the hardware/software interface
  - Microarchitecture: how hardware executes sequence of instructions

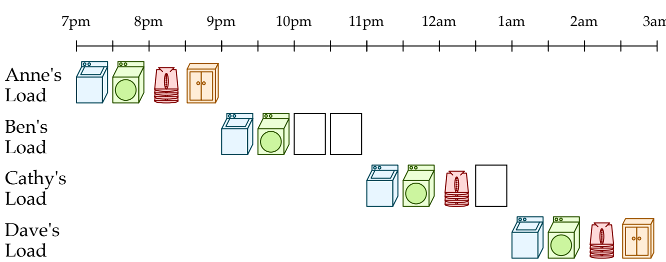


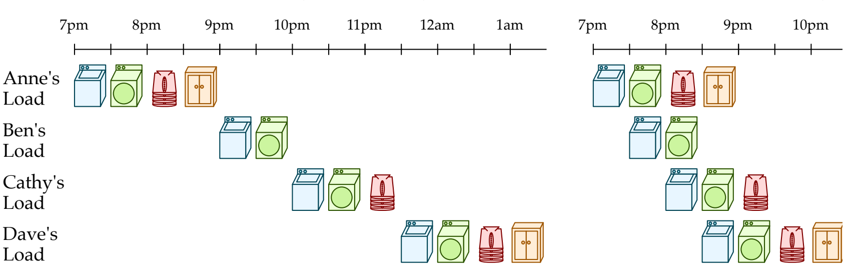


- Laundry
  - Cleaning a load of laundry is a "transaction"
  - Architecture: high-level specification, dirty clothes in, clean clothes out
  - Microarchitecture: how laundry room actually processes multiple loads

3.1. Processor Microarchitectural Design Patterns

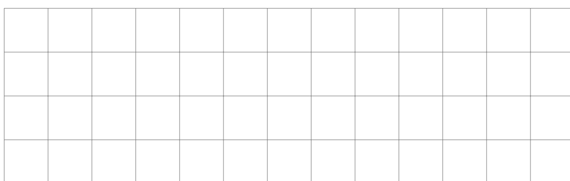


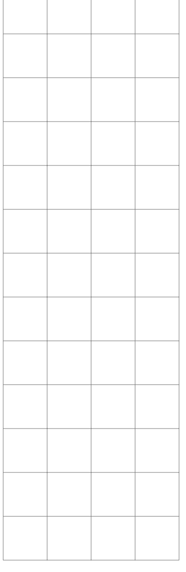




3.2. Transaction Diagrams

W: Washing **D**: Drying **F**: Folding **S**: Storing





Key Concepts

- Transaction latency is the time to complete a single transaction
- Execution time or total latency is the time to complete a sequence of transactions
- Throughput is the number of transactions executed per unit time

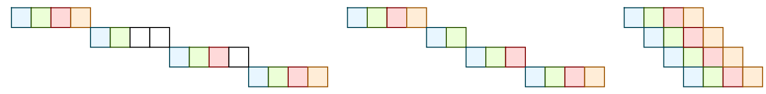
3.3. Analyzing Processor Performance

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Avg Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

- Instructions / program depends on source code, compiler, ISA
- Avg cycles / instruction (CPI) depends on ISA, microarchitecture
- Time / cycle depends upon microarchitecture and implementation

Using our first-order equation for processor performance and a functional-level model, the execution time is just the number of dynamic instructions.

Microarchitecture	**CPI**	**Cycle Time**
Single-Cycle Processor	1	long
Multi-Cycle Processor	>1	short
Pipelined Processor	≈ 1	short



Students often confuse "Cycle Time" with the execution time of a sequence of transactions measured in cycles. "Cycle Time" is the clock period or the inverse of the clock frequency.

Estimating dynamic instruction count

Estimate the dynamic instruction count for the vector-vector add example assuming `n` is 64?

```
1  loop:
2   lw    x5, 0(x1)
3   lw    x6, 0(x2)
4   add   x7, x5, x6
5   sw    x7, 0(x3)
6   addi  x1, x1, 4
7   addi  x2, x2, 4
8   addi  x3, x3, 4
9   addi  x4, x4, -1
10  bne   x4, x0, loop
```

Estimate the dynamic instruction count for the mystery program assuming `n` is 64.

```
1   addi  x5, x0, 0
2
3  loop:
4   lw    x4, 0(x1)
5   bne   x4, x3, foo
6   addi  x5, x0, 1
7
8  foo:
9   addi  x1, x1, 4
10  addi  x2, x2, -1
11  bne   x2, x0, loop
```