

# ECE 4750 Computer Architecture

## Fall 2024

### Topic 1: Processor Concepts

School of Electrical and Computer Engineering  
Cornell University

revision: 2024-09-03-14-29

<b>1</b>	<b>Instruction Set Architecture</b>	<b>3</b>
1.1.	IBM 360 Instruction Set Architecture . . . . .	5
1.2.	MIPS32 Instruction Set Architecture . . . . .	7
1.3.	Tiny RISC-V Instruction Set Architecture . . . . .	12
<b>2</b>	<b>Processor Functional-Level Model</b>	<b>16</b>
2.1.	Transactions and Steps . . . . .	16
2.2.	TinyRV1 Simple Assembly Example . . . . .	17
2.3.	TinyRV1 VVAdd Asm and C Program . . . . .	18
2.4.	TinyRV1 Mystery Asm and C Program . . . . .	19
<b>3</b>	<b>Processor/Laundry Analogy</b>	<b>20</b>
3.1.	Arch vs. $\mu$ Arch vs. VLSI Impl . . . . .	20
3.2.	Processor Microarchitectural Design Patterns . . . . .	21
3.3.	Transaction Diagrams . . . . .	22
<b>4</b>	<b>Analyzing Processor Performance</b>	<b>23</b>

Copyright © 2024 Anne Bracy. All rights reserved. This handout was prepared by Prof. Anne Bracy at Cornell University for ECE 4750 Computer Architecture (derived from previous handouts prepared and copyrighted by Prof. Christopher Batten). 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. Instruction Set Architecture

- By early 1960's, IBM had several incompatible lines of computers!
  - Defense : 701
  - Scientific : 704, 709, 7090, 7094
  - Business : 702, 705, 7080
  - Mid-Sized Business : 1400
  - Decimal Architectures : 7070, 7072, 7074
- Each system had its own:
  - Implementation and potentially even technology
  - Instruction set
  - I/O system and secondary storage (tapes, drums, disks)
  - Assemblers, compilers, libraries, etc
  - Application niche
- IBM 360 was the first line of machines to separate ISA from microarchitecture
  - Enabled same software to run on different current and future microarchitectures
  - Reduced impact of modifying the microarchitecture enabling rapid innovation in hardware

Application
Algorithm
Programming Language
Operating System
Compiler
Instruction Set Architecture
Microarchitecture
Register-Transfer Level
Gate Level
Circuits
Devices
Technology

... the structure of a computer that a **machine language programmer** must understand to write a correct (**timing independent**) program for that machine.

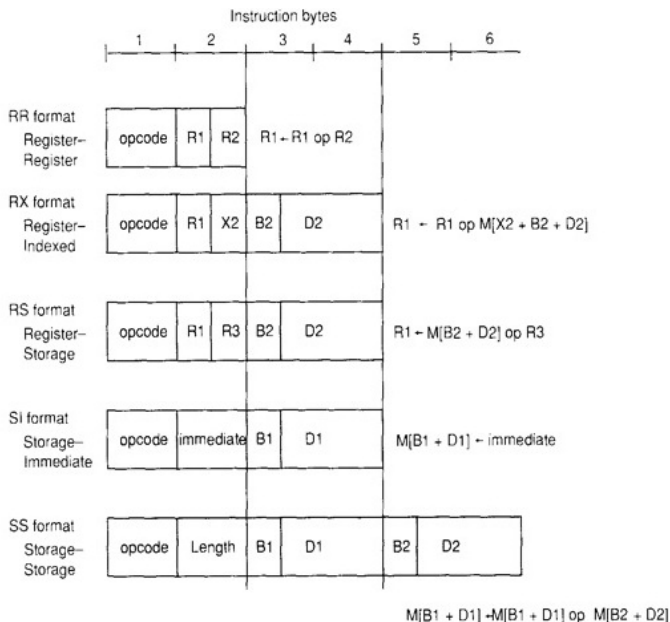
— Amdahl, Blaauw, Brooks, 1964

## ISA is the contract between software and hardware

- 1. \_\_\_\_\_
  - 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. \_\_\_\_\_
  - Registers: general-purpose, floating-point, control/status
  - Memory: different addresses spaces for heap, stack, I/O
- 3. \_\_\_\_\_
  - 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. \_\_\_\_\_
  - Integer and floating-point arithmetic instructions
  - Register and memory data movement instructions
  - Control transfer instructions
  - System control instructions
- 5. \_\_\_\_\_
  - Opcode, addresses of operands and destination, next instruction
  - Variable length vs. fixed length

## 1.1. IBM 360 Instruction Set Architecture

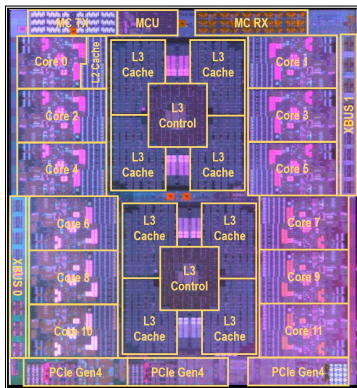
- How is data represented?
  - 8-bit bytes, 16-bit half-words, 32-bit words, 64-bit double-words
  - IBM 360 is why bytes are 8-bits long today!
- Where can data be stored?
  - $2^{24}$  8-bit memory locations
  - 16 general-purpose 32-bit registers and 4 floating-point 64-bit registers
  - Condition codes, control flags, program counter
- What operations can be performed on data?
  - Large number of arithmetic, data movement, and control instructions



	Model 30	Model 70
Storage	8–64 KB	256–512 KB
Datapath	8-bit	64-bit
Circuit Delay	30 ns/level	5 ns/level
Local Store	Main store	Transistor registers
Control Store	Read only 1 $\mu$ s	Conventional circuits

- IBM 360 instruction set architecture completely hid the underlying technological differences between various models
- **Significant Milestone: The first true ISA designed as a portable hardware-software interface**
- IBM 360: 60 years later ...  
The zSeries z15 Microprocessor

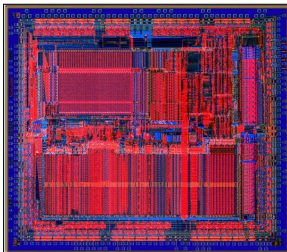
- 5+ GHz in IBM 14 nm SOI
- 9.2B transistors in 696 mm<sup>2</sup>
- 17 metal layers
- 12 cores per chip
- Aggressive out-of-order execution
- Four-level cache hierarchy
- On-chip 256MB eDRAM L3 cache
- Off-chip 960MB eDRAM L4 cache
- Can still run IBM 360 code!



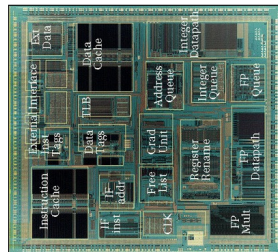
C. Berry, et al., “IBM z15: A 12-Core 5.2GHz Microprocessor,”  
Int’l Solid-State Circuits Conference, Feb. 2020.

## 1.2. MIPS32 Instruction Set Architecture

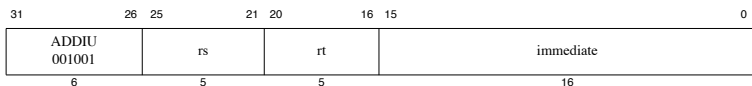
- How is data represented?
  - 8-bit bytes, 16-bit half-words, 32-bit words
  - 32-bit single-precision, 64-bit double-precision floating point
- Where can data be stored?
  - $2^{32}$  8-bit memory locations
  - 32 general-purpose 32-bit registers, 32 SP (16 DP) floating-point registers
  - FP status register, Program counter
- How can data be accessed?
  - Register, immediate, displacement
- What operations can be performed on data?
  - Large number of arithmetic, data movement, and control instructions
- How are instructions encoded?
  - Fixed-length 32-bit instructions



**MIPS R2K:** 1986, single-issue, in-order, off-chip caches, 2  $\mu\text{m}$ , 8–15 MHz, 110K transistors, 80  $\text{mm}^2$



**MIPS R10K:** 1996, quad-issue, out-of-order, on-chip caches, 0.35  $\mu\text{m}$ , 200 MHz, 6.8M transistors, 300  $\text{mm}^2$



**Format:** ADDIU *rt*, *rs*, *immediate*

MIPS32

**Purpose:** Add Immediate Unsigned Word

To add a constant to a 32-bit integer

**Description:**  $GPR[rt] \leftarrow GPR[rs] + immediate$

The 16-bit signed *immediate* is added to the 32-bit value in GPR *rs* and the 32-bit arithmetic result is placed into GPR *rt*.

No Integer Overflow exception occurs under any circumstances.

**Restrictions:**

None

**Operation:**

```
temp ← GPR[rs] + sign_extend(immediate)
GPR[rt] ← temp
```

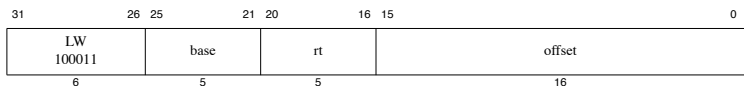
**Exceptions:**

None

**Programming Notes:**

The term “unsigned” in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. This instruction is appropriate for unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.





**Format:** LW *rt*, *offset*(*base*)

MIPS32

**Purpose:** Load Word

To load a word from memory as a signed value

**Description:**  $GPR[rt] \leftarrow \text{memory}[GPR[base] + \text{offset}]$

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, sign-extended to the GPR register length if necessary, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

**Restrictions:**

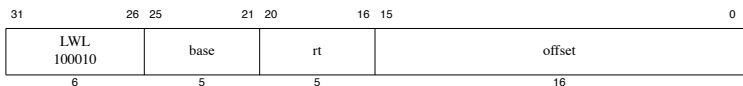
The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

**Operation:**

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[rt] ← memword
```

**Exceptions:**

TLB Refill, TLB Invalid, Bus Error, Address Error, Watch



**Format:** LWL rt, offset(base)

MIPS32

**Purpose:** Load Word Left

To load the most-significant part of a word as a signed value from an unaligned memory address

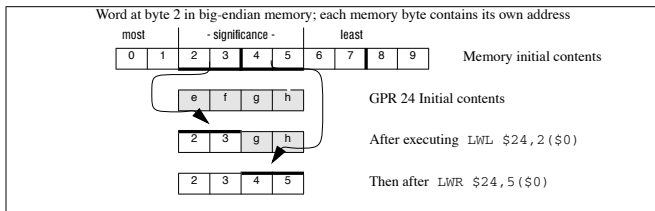
**Description:**  $GPR[rt] \leftarrow GPR[rt] \text{ MERGE } \text{memory}[GPR[base] + \text{offset}]$

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address (*EffAddr*). *EffAddr* is the address of the most-significant of 4 consecutive bytes forming a word (*W*) in memory starting at an arbitrary byte boundary.

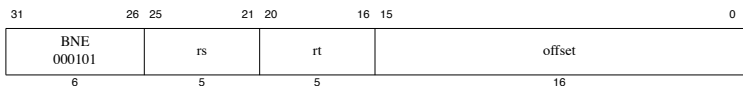
The most-significant 1 to 4 bytes of *W* is in the aligned word containing the *EffAddr*. This part of *W* is loaded into the most-significant (left) part of the word in GPR *rt*. The remaining least-significant part of the word in GPR *rt* is unchanged.

The figure below illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is in the aligned word containing the most-significant byte at 2. First, LWL loads these 2 bytes into the left part of the destination register word and leaves the right part of the destination word unchanged. Next, the complementary LWR loads the remainder of the unaligned word

**Figure 3.4 Unaligned Word Load Using LWL and LWR**



The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned word, that is, the low 2 bits of the address ( $vAddr_{1,0}$ ), and the current byte-ordering mode of the processor (big- or little-endian). The figure below shows the bytes loaded for every combination of offset and byte ordering.



**Format:** BNE *rs*, *rt*, *offset*

MIPS32

**Purpose:** Branch on Not Equal

To compare GPRs then do a PC-relative conditional branch

**Description:** if  $GPR[rs] \neq GPR[rt]$  then branch

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* and GPR *rt* are not equal, branch to the effective target address after the instruction in the delay slot is executed.

**Restrictions:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

```

I:   target_offset ← sign_extend(offset || 02)
       condition ← (GPR[rs] ≠ GPR[rt])
I+1: if condition then
       PC ← PC + target_offset
       endif

```

**Exceptions:**

None

**Programming Notes:**

With the 18-bit signed instruction offset, the conditional branch range is  $\pm 128$  KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

### 1.3. Tiny RISC-V 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
- Tiny RISC-V instruction set architecture
  - Subset we use in this course
  - Small enough for teaching, powerful enough for running real C programs
  - How is data represented? \_\_\_\_\_
  - Where can data be stored? \_\_\_\_\_
  - How can data be accessed? \_\_\_\_\_
  - What ops can be performed on \_\_\_\_\_  
data?
  - How are inst encoded? \_\_\_\_\_
  - <http://www.csl.cornell.edu/courses/ece4750/handouts.shtml>
- **TinyRV1:** Small subset suitable for lecture, problems, exams
  - \_\_\_\_\_
  - \_\_\_\_\_
  - \_\_\_\_\_
- **TinyRV2:** Subset suitable for lab assignments and capable of executing simple C programs without an operating system
  - add, addi, sub, mul, and, andi, or, ori, xor, xori
  - slt, slti, sltu, sltiu
  - sra, srai, srl, srli, sll, slli
  - lui, aupic, lw, sw
  - jal, jalr, beq, bne, blt, bge, bltu, bgeu
  - csrr, csrw

## TinyRV1 instruction assembly, semantics, and encoding

### ADD

add rd, rs1, rs2

$R[rd] \leftarrow R[rs1] + R[rs2]$

$PC \leftarrow PC + 4$

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

### ADDI

addi rd, rs1, imm

$R[rd] \leftarrow R[rs1] + \text{sext}(imm)$

$PC \leftarrow PC + 4$

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

### MUL

mul rd, rs1, rs2

$R[rd] \leftarrow R[rs1] \times R[rs2]$

$PC \leftarrow PC + 4$

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

### LW

lw rd, imm(rs1)

$R[rd] \leftarrow M[R[rs1] + \text{sext}(imm)]$

$PC \leftarrow PC + 4$

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

### SW

sw rs2, imm(rs1)

$M[R[rs1] + \text{sext}(imm)] \leftarrow R[rs2]$

$PC \leftarrow PC + 4$

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

$R[rd] \leftarrow PC + 4$

$PC \leftarrow PC + \text{sext}(imm)$

31	12 11	7 6	0
imm		rd	1101111

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

### JR

jr rs1

$PC \leftarrow R[rs1]$

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

### BNE

bne rs1, rs2, imm

if ( $R[rs1] \neq R[rs2]$ )  $PC \leftarrow PC + \text{sext}(imm)$

else  $PC \leftarrow PC + 4$

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

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

<b>Base Integer Instructions: RV32I, RV64I, and RV128I</b>				<b>RV Privileged Instructions</b>						
Category	Name	Fmt	RV32I Base	+RV(64,128)	Category	Name	RV mnemonic			
<b>Loads</b>	Load Byte	I	LB rd,rs1,imm		<b>CSR Access</b>	Atomic R/W	CSRWR rd,csr,rs1			
	Load Halfword	I	LH rd,rs1,imm			Atomic Read & Set Bit	CSRSR rd,csr,rs1			
	Load Word	I	LW rd,rs1,imm	L{D Q} rd,rs1,imm		Atomic Read & Clear Bit	CSRRC rd,csr,rs1			
	Load Byte Unsigned	I	LBU rd,rs1,imm			Atomic R/W Imm	CSRRWI rd,csr,imm			
	Load Half Unsigned	I	LHU rd,rs1,imm	L{W D U} rd,rs1,imm		Atomic Read & Set Bit Imm	CSRRSI rd,csr,imm			
<b>Stores</b>	Store Byte	S	SB rs1,rs2,imm		Atomic Read & Clear Bit Imm	CSRRCI rd,csr,imm				
	Store Halfword	S	SH rs1,rs2,imm		<b>Change Level</b>	Env. Call	ECALL			
	Store Word	S	SW rs1,rs2,imm	S{D Q} rs1,rs2,imm	Environment Breakpoint		EBREAK			
<b>Shifts</b>	Shift Left	R	SLL rd,rs1,rs2	SLL{W D} rd,rs1,rs2	Environment Return		ERET			
	Shift Left Immediate	I	SLLI rd,rs1,shamt	SLLI{W D} rd,rs1,shamt	<b>Trap Redirect to Supervisor</b>		MRTS			
	Shift Right	R	SRL rd,rs1,rs2	SRL{W D} rd,rs1,rs2		Redirect Trap to Hypervisor		MRTH		
	Shift Right Immediate	I	SRLI rd,rs1,shamt	SRLI{W D} rd,rs1,shamt		Hypervisor Trap to Supervisor		MRTS		
	Shift Right Arithmetic	R	SRA rd,rs1,rs2	SRA{W D} rd,rs1,rs2	<b>Interrupt</b>	Wait for Interrupt	WFI			
Shift Right Arith Imm	I	SRAI rd,rs1,shamt	SRAI{W D} rd,rs1,shamt	<b>MMU</b>	Supervisor FENCE	SFENCE.VM rs1				
<b>Arithmetic</b>	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								
<b>Logical</b>	XOR	R	XOR rd,rs1,rs2		<b>Optional Compressed (16-bit) Instruction Extension: RVC</b>					
	XOR Immediate	I	XORI rd,rs1,imm		<b>Category</b>	<b>Name</b>	<b>Fmt</b>			
	OR	R	OR rd,rs1,rs2		<b>Loads</b>	Load Word	CL C.LW rd',rs1',imm	RVC	RVI equivalent	
	OR Immediate	I	ORI rd,rs1,imm			Load Word SP	CI C.LWSP rd,imm		RV1 equivalent	
	AND	R	AND rd,rs1,rs2			Load Double	CL C.LD rd',rs1',imm			
AND Immediate	I	ANDI rd,rs1,imm			Load Double SP	CI C.LDSP rd,imm				
<b>Compare</b>	Set <	R	SLT rd,rs1,rs2			Load Quad	CI C.LQ rd',rs1',imm			
	Set < Immediate	I	SLTI rd,rs1,imm			Load Quad SP	CI C.LQSP rd,imm			
	Set < Unsigned	R	SLTU rd,rs1,rs2		<b>Stores</b>	Store Word	CS C.SW rs1',rs2',imm			
	Set < Imm Unsigned	I	SLTIU rd,rs1,imm				Store Word SP	CSS C.SWSP rs2,imm		
<b>Branches</b>	Branch =	SB	BEQ rs1,rs2,imm				Store Double	CS C.SD rs1',rs2',imm		
	Branch ≠	SB	BNE rs1,rs2,imm				Store Double SP	CSS C.SDSP rs2,imm		
	Branch <	SB	BLT rs1,rs2,imm			Store Quad	CS C.SQ rs1',rs2',imm			
	Branch ≥	SB	BGE rs1,rs2,imm			Store Quad SP	CSS C.SQSP rs2,imm			
	Branch < Unsigned	SB	BLTU rs1,rs2,imm		<b>Arithmetic</b>	ADD	CR C.ADD rd,rs1			
Branch ≥ Unsigned	SB	BGEU rs1,rs2,imm				ADD Word	CR C.ADDW rd,rs1			
<b>Jump &amp; Link</b>	J&L	UJ	JAL rd,imm				ADD Immediate	CI C.ADDI rd,imm		
	Jump & Link Register	UJ	JALR rd,rs1,imm				ADD Word Imm	CI C.ADDIW rd,imm		
<b>Synch</b>	Synch thread	I	FENCE			ADD SP Imm * 16	CI C.ADDI16SP x0,imm			
	Synch Instr & Data	I	FENCE.I			ADD SP Imm * 4	CIW C.ADDI4SPN rd',imm			
<b>System</b>	System CALL	I	SCALL			Load Immediate	CI C.LI rd,imm			
	System BREAK	I	SBREAK			Load Upper Imm	CI C.LUI rd,imm			
<b>Counters</b>	Read CYCLE	I	RDCYCLE rd			MoVe	CR C.MV rd,rs1			
	Read CYCLE upper Half	I	RDCYLEH rd			SUB	CR C.SUB rd,rs1			
	Read TIME	I	RDTIME rd		<b>Shifts</b>	Shift Left Imm	CI C.SLLI rd,imm			
	Read TIME upper Half	I	RDTIMEH rd				Branch=0	CB C.BEQZ rs1',imm		
	Read INSTR RETired	I	RDINSTRET rd			Branch≠0	CB C.BNEZ rs1',imm			
Read INSTR upper Half	I	RDINSTRETH rd		<b>Jump</b>	Jump	CJ C.J imm				
						Jump Register	CR C.CJR rd,rs1			
<b>U</b>					<b>Jump &amp; Link</b>	J&L	CJ C.JAL imm			
							Jump & Link Register	CR C.CJALR rs1		
					<b>System</b>	Env. BREAK	CI C.EBREAK			

<b>32-bit Instruction Formats</b>																	
	31	30	25:24	21	20	19	15:14	12:11	8	7	6	5	4	3	2	1	0
<b>R</b>	func7		rs2		rs1	func3		rd		opcode							
<b>I</b>	imm[11:0]		rs2		rs1	func3		rd		opcode							
<b>S</b>	imm[11:5]		rs2		rs1	func3		imm[4:0]		opcode							
<b>SB</b>	imm[12]		imm[10:5]		rs2	rs1	func3		imm[4:1]		imm[11]						opcode
<b>U</b>	imm[20]		imm[10:1]		imm[11]		imm[19:12]		rd		opcode						
	imm[20]		imm[10:1]		imm[11]		imm[19:12]		rd		opcode						

<b>16-bit (RVC) Instruction Formats</b>																
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<b>CR</b>	func4		rd/rs1		rs2		op									
<b>CI</b>	func3		imm		rd/rs1		imm		op							
<b>CSS</b>	func3		imm		imm		rs2		op							
<b>CIW</b>	func3		imm		imm		rd'		op							
<b>CL</b>	func3		imm		rs1'		imm		rd'		op					
<b>CS</b>	func3		imm		rs1'		imm		rs2'		op					
<b>CB</b>	func3		offset		rs1'		offset		op							
<b>CJ</b>	func3		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 RV1I base of <math>f</math>-50 classic integer RISC instructions is required. Every 16-bit RVC instruction matches an existing 32-bit RV1I instruction. See risc.org.

<b>Optional Multiply-Divide Instruction Extension: RVM</b>				
Category	Name	Fmt	RV32M (Multiply-Divide)	+RV(64,128)
<b>Multiply</b>	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	
<b>Divide</b>	DIVide	R	DIV rd,rs1,rs2	DIV{W D} rd,rs1,rs2
	DIVide Unsigned	R	DIVU rd,rs1,rs2	
<b>Remainder</b>	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

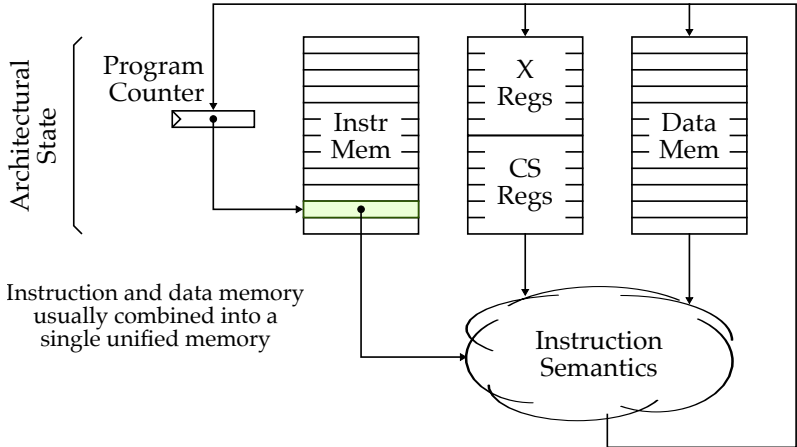
<b>Optional Atomic Instruction Extension: RVA</b>				
Category	Name	Fmt	RV32A (Atomic)	+RV(64,128)
<b>Load</b>	Load Reserved	R	LR.W rd,rs1	LR.{D Q} rd,rs1
<b>Store</b>	Store Conditional	R	SC.W rd,rs1,rs2	SC.{D Q} rd,rs1,rs2
<b>Swap</b>	SWAP	R	AMOSWAP.W rd,rs1,rs2	AMOSWAP.{D Q} rd,rs1,rs2
<b>Add</b>	ADD	R	AMOADD.W rd,rs1,rs2	AMOADD.{D Q} rd,rs1,rs2
<b>Logical</b>	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
<b>Min/Max</b>	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

<b>Three Optional Floating-Point Instruction Extensions: RVF, RVD, &amp; RVQ</b>				
Category	Name	Fmt	RV32{F D Q} (HP/SP,DP,QP FP Pt)	+RV(64,128)
<b>Move</b>	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
<b>Convert</b>	Convert from Int	R	FCVTE.{H S D Q}.W rd,rs1	FCVTE.{H S D Q}.L{T} rd,rs1
	Convert from Int Unsigned	R	FCVTE.{H S D Q}.WU rd,rs1	FCVTE.{H S D Q}.L{T}U rd,rs1
	Convert to Int	R	FCVTE.W.{H S D Q} rd,rs1	FCVTE.L{T}.W.{H S D Q} rd,rs1
	Convert to Int Unsigned	R	FCVTE.WU.{H S D Q} rd,rs1	FCVTE.L{T}U.W.{H S D Q} rd,rs1

<b>RISC-V Calling Convention</b>					
Load	Store	Register	ABI Name	Saver	Description
<b>Arithmetic</b>	ADD	x0	zero	---	Hard-wired zero
	SUBtract	x1	ra	Caller	Return address
	MULTiply	x2	sp	Callee	Stack pointer
	DIVide	x3	gp	---	Global pointer
	SQuare RoOt	x4	tp	---	Thread pointer
<b>Mul-Add</b>	Multiply-ADD	x5-7	t0-2	Caller	Temporaries
	Multiply-SUBtract	x8	s0/fp	Callee	Saved register/frame pointer
	Negative Multiply-SUBtract	x9	s1	Callee	Saved register
	Negative Multiply-ADD	x10-11	a0-1	Caller	Function arguments/return values
<b>Sign Injct</b>	SIGN source	x12-17	a2-7	Caller	Function arguments
	Negative SIGN source	x18-27	s2-11	Callee	Saved registers
	Xor SIGN source	x28-31	t3-t6	Caller	Temporaries
<b>Min/Max</b>	MINimum	f0-7	ft0-7	Caller	FP temporaries
	MAXimum	f8-9	fs0-1	Callee	FP saved registers
<b>Compare</b>	Compare Float =	f10-11	fa0-1	Caller	FP arguments/return values
	Compare Float <	f12-17	fa2-7	Caller	FP arguments
	Compare Float ≤	f18-27	fs2-11	Callee	FP saved registers
<b>Categorization</b>	Classify Type	f28-31	ft8-11	Caller	FP temporaries
<b>Configuration</b>	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 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 riscv.org. (8/21/15 revision)

## 2. Processor Functional-Level Model



### 2.1. Transactions and Steps

- We can think of each instruction as a **transaction**
- Executing a transaction involves a sequence of **steps**

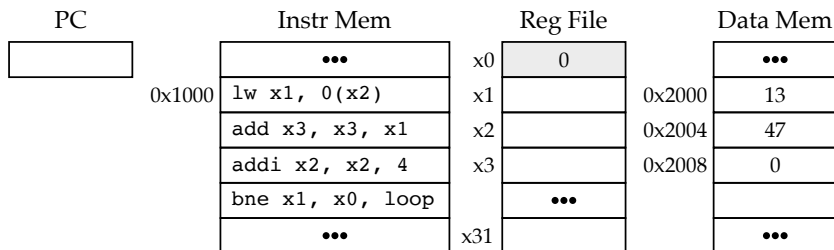
	add	addi	mul	lw	sw	jal	jr	bne
Fetch Instruction								
Decode Instruction								
Read Register File								
Register Arithmetic								
Read Memory								
Write Memory								
Write Register File								
Update PC								



## 2.2. TinyRV1 Simple Assembly Example

Static Asm Sequence	Instruction Semantics
loop: lw x1, 0(x2)	
add x3, x3, x1	
addi x2, x2, 4	
bne x1, x0, loop	

### Worksheet illustrating processor functional-level model



### Table illustrating processor functional-level model

PC	Dynamic Asm Sequence	x1	x2	x3
	lw x1, 0(x2)			
	add x3, x3, x1			
	addi x2, x2, 4			
	bne x1, x0, loop			
	lw x1, 0(x2)			
	add x3, x3, x1			



## 2.4. TinyRV1 Mystery Assembly and C Program

What is the C code corresponding to the TinyRV1 assembly shown below? Assume assembly implements a function.

---

---

---

---

---

---

---

---

---

---

```
addi x5, x0, 0

loop:
lw    x6, 0(x10)
bne   x6, x12, foo
addi  x10, x5, 0
jr    x1

foo:
addi  x10, x10, 4
addi  x5, x5, 1
bne   x5, x11, loop

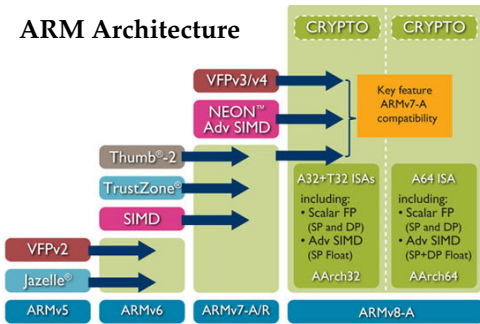
addi  x10, x0, -1
jr    x1
```

### 3. Processor/Laundry Analogy

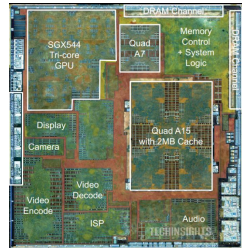
- Processor
  - Instructions are “transactions” that execute on a processor
  - Architecture: defines the hardware/software interface
  - Microarchitecture: how hardware executes sequence of instructions
- Laundry
  - Cleaning a load of laundry is a “transaction”
  - Architecture: high-level specification, dirty clothes in, clean clothes out
  - Microarchitecture: how laundry room actually processes multiple loads

### 3.1. Arch vs. $\mu$ Arch vs. VLSI Impl

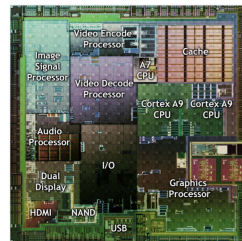
#### ARM Architecture



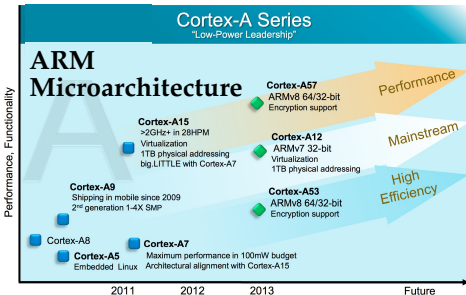
#### ARM VLSI Implementation



Samsung Exynos Octa




NVIDIA Tegra 2
















### 3.2. Processor Microarchitectural Design Patterns

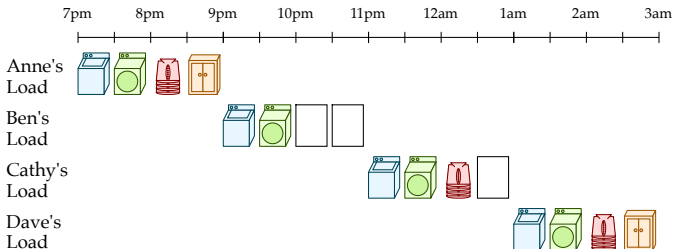
#### Transaction Steps

-  Washing (30 min)
-  Drying (30 min)
-  Folding (30 min)
-  Storing (30 min)

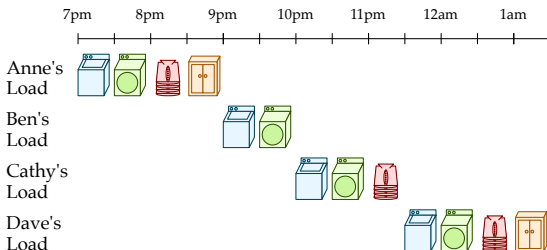
#### Four Types of Transactions

	0 hr	1 h	2 hr	Transaction Latency		
Anne's Load					2.0 hr	Anne requires all four steps
Ben's Load					1.0 hr	Ben is messy, leaves unfolded clothes in his laundry basket
Cathy's Load					1.5 hr	Cathy does not have a bureau, leaves folded clothes in basket
Dave's Load					2.0 hr	Dave requires all four steps

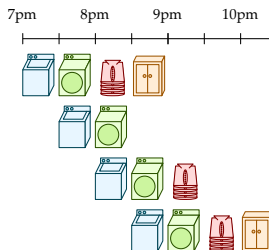
#### Fixed Time Slot Laundry (Single-Cycle Processors)



#### Variable Time Slot Laundry (FSM Processors)



#### Pipelined Laundry



### 3.3. 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

## 4. 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
FSM 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?

```
loop:
    lw    x5, 0(x11)
    lw    x6, 0(x12)
    add   x7, x5, x6
    sw    x7, 0(x10)
    addi  x11, x11, 4
    addi  x12, x12, 4
    addi  x10, x10, 4
    addi  x13, x13, -1
    bne   x13, x0, loop
    jr    x1
```

Estimate the dynamic instruction count for the mystery program assuming  $n$  is 64 and that we find a match on the final element.

```
    addi x5, x0, 0
loop:
    lw    x6, 0(x10)
    bne   x6, x12, foo
    addi  x10, x5, 0
    jr    x1
foo:
    addi  x10, x10, 4
    addi  x5, x5, 1
    bne   x5, x11, loop
    addi  x10, x0, -1
    jr    x1
```