ECE 4750 Computer Architecture
Fall 2022

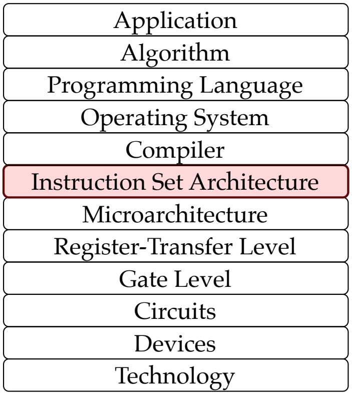
Topic 1: Fundamental Processor Concepts

School of Electrical and Computer Engineering
Cornell University

revision: 2022-08-23-23-51

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



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

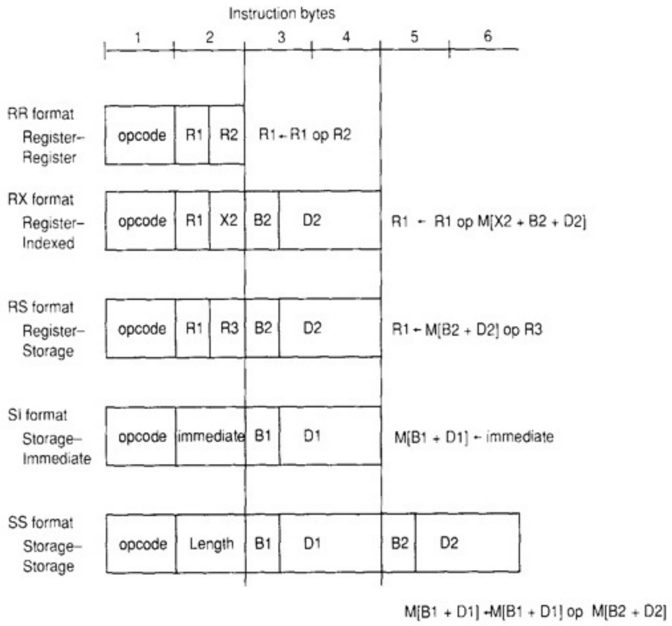
— Amdahl, Blaauw, Brooks, 1964

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
  - 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

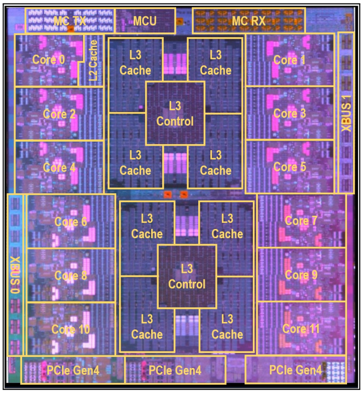
1.1. IBM 360 Instruction Set Architecture

- 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} 32-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 done 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 1µs	Conventional circuits

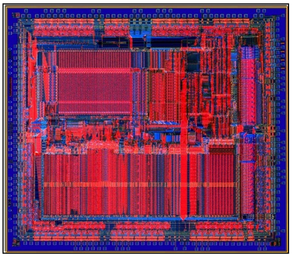
- IBM 360 instruction set architecture completely hid the underlying technological differences between various models
- Significant Milestone: The first true ISA designed as a portable hardware-software interface
- IBM 360: 60 years later ...
  The zSeries z15 Microprocessor
  - 5+ GHz in IBM 14 nm SOI
  - 9.2B transistors in 696 mm^2
  - 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!



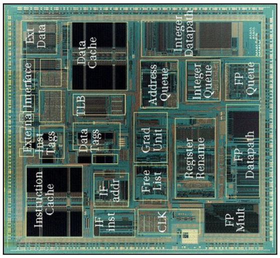
C. Berry, et al., "IBM z15: A 12-Core 5.2GHz Microprocessor," Int'l Solid-State Circuits Conference, Feb. 2020.

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} 32-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 done 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 µm, 8–15 MHz, 110K transistors, 80 mm^2



MIPS R10K: 1996, quad-issue, out-of-order, on-chip caches, 0.35 µm, 200 MHz, 6.8M transistors, 300 mm^2

31 26	25 21	20 16	15 0
ADDIU 001001	rs	rt	immediate
6	5	5	16

Format: `ADDIU rt, rs, immediate` **MIPS32**

Purpose: Add Immediate Unsigned Word

To add a constant to a 32-bit integer

Description: `GPR[rt] ← 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.

31 26	25 21	20 16	15 0
LW 100011	base	rt	offset
6	5	5	16

Format: `LW rt, offset(base)` **MIPS32**

Purpose: Load Word

To load a word from memory as a signed value

Description: `GPR[rt] ← memory[GPR[base] + 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 $vAddr_{1..0} \ne 0^2$ 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

31 26	25 21	20 16	15 0
BNE 000101	rs	rt	offset
6	5	5	16

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] ≠ 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 || $0^2$)
        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 ± 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 done on data? ____________________
  - How are inst encoded? ____________________
  - `http://www.csl.cornell.edu/courses/ece4750/handouts.html`
- **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`] ← 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 RISC-V 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 14 13	12	11 10	9 8 7	6 5	4 3 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.

Free & Open RISC-V Reference Card (riscv.org) ②

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

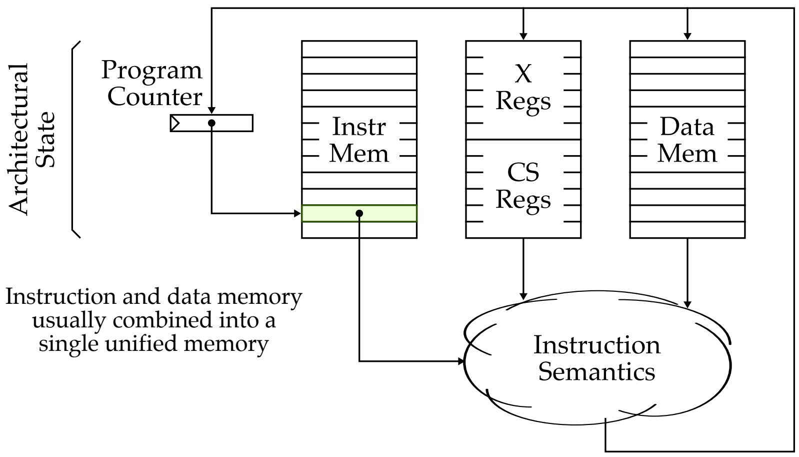
Category	Name	Fmt	RV32{F|D|Q} (HP/SP,DP,QP Fl Pt)	+RV{64,128}
Move	Move from Integer	R	`FMV.{H|S}.X rd,rs1`	`FMV.{D|Q}.X rd,rs1`
	Move to Integer	R	`FMV.X.{H|S} rd,rs1`	`FMV.X.{D|Q} rd,rs1`
Convert	Convert from Int	R	`FCVT.{H|S|D|Q}.W rd,rs1`	`FCVT.{H|S|D|Q}.{L|T} rd,rs1`
	Convert from Int Unsigned	R	`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. 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 Registers								
Register Arithmetic								
Read Memory								
Write Memory								
Write Registers								
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

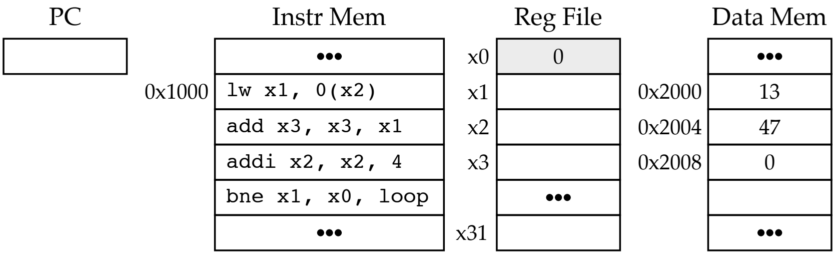


Table illustrating processor functional-level model

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`			

2.3. TinyRV1 Vector-Vector Add Assembly and C Program

C code for doing element-wise vector addition.

Equivalent TinyRV1 assembly code. Arguments are passed in `x12`–`x17`, return value is stored to `x10`, return address is stored in `x1`, and temporaries are stored in `x5`–`x7`.

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.

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.

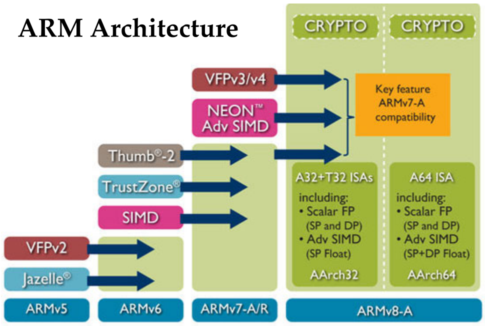
```
  addi x5,  x0,  0

 loop:
  lw    x6,  0(x12)
  bne   x6,  x14, foo
  addi  x10, x5,  0
  jr    x1

 foo:
  addi  x12, x12, 4
  addi  x5,  x5,  1
  bne   x5,  x13, 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

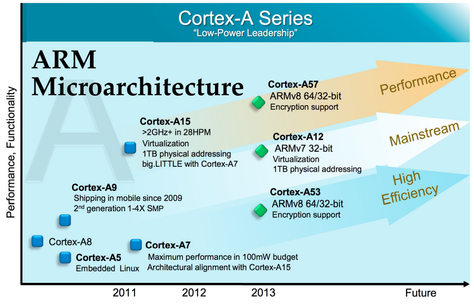
3.1. Arch vs. µArch vs. VLSI Impl

ARM Architecture

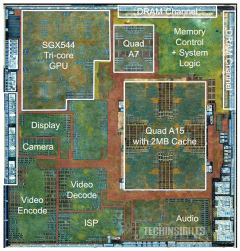


CRYPTO
CRYPTO
VFPv3/v4
NEON™ Adv SIMD
Key feature ARMv7-A compatibility
Thumb®-2
TrustZone®
SIMD
VFPv2
Jazelle®
A32+T32 ISAs including: • Scalar FP (SP and DP) • Adv SIMD (SP Float)
AArch32
A64 ISA including: • Scalar FP (SP and DP) • Adv SIMD (SP+DP Float)
AArch64
ARMv5
ARMv6
ARMv7-A/R
ARMv8-A

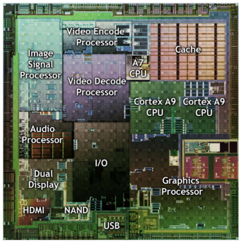
ARM Microarchitecture



ARM VLSI Implementation



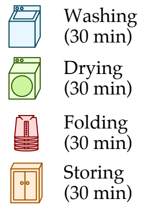
Samsung Exynos Octa



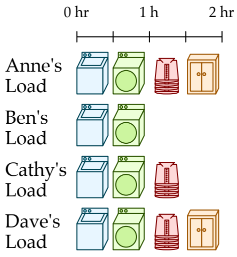
NVIDIA Tegra 2

3.2. Processor Microarchitectural Design Patterns

Transaction Steps

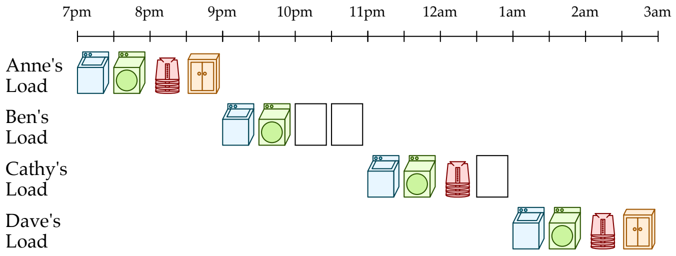


Four Types of Transactions

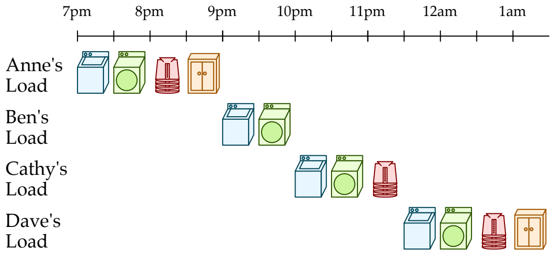


	Transaction Latency	
Anne's Load	2.0 hr	Anne requires all four steps
Ben's Load	1.0 hr	Ben is messy, leaves unfolded clothes in his laundry basket
Cathy's Load	1.5 hr	Cathy does not have a bureau, leaves folded clothes in basket
Dave's Load	2.0 hr	Dave requires all four steps

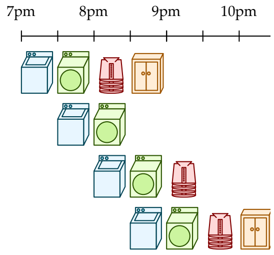
Fixed Time Slot Laundry (Single-Cycle Processors)



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

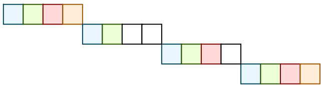
4. Analyzing Processor Performance

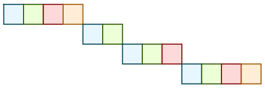
$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Avg Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

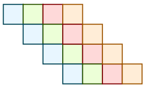
- Instructions / program depends on source code, compiler, ISA
- Avg cycles / instruction (CPI) depends on ISA, microarchitecture
- Time / cycle depends upon microarchitecture and implementation

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(x13)
 lw    x6,  0(x14)
 add   x7,  x5,  x6
 sw    x7,  0(x12)
 addi  x13, x12, 4
 addi  x14, x14, 4
 addi  x12, x12, 4
 addi  x15, x15, -1
 bne   x15, 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(x12)
 bne   x6,  x14, foo
 addi  x10, x5,  0
 jr    x1
foo:
 addi  x12, x12, 4
 addi  x5,  x5,  1
 bne   x5,  x13, loop
 addi  x10, x0,  -1
 jr    x1
```