UCLA Samueli School of Engineering

CS-M151B
Computer Systems Architecture

Blaise Tine
UCLA Computer Science

ISA Classification

- Organizing ISAs based on their characteristics
  - Instruction Complexity
  - Addressing mode
  - Encoding style
  - Parallelism

ISA Classification

- Instruction complexity
  - CISC: Complex instruction set computing (1960-)
    - System/360 - IBM - 1964
    - X86 - Intel - 1978
  - RISC: Relaxed instruction set computing (1980-)
    - RISC-I Project - UC Berkley - 1980
    - MIPS Project - Stanford - 1981
    - RISC-V - UC Berkley - 2010

ISA Classification

- CISC Characteristics
  - **Small register File (e.g. x86 – 8 registers)**
  - **Large number of instructions**
  - Some instructions are complex (can perform multiple operations, taking several cycles).
    - e.g. x86 string instructions: LODS, STOS, MOVS
  - Complex addressing modes
    - immediate, register, memory
  - Complex data types
    - string, complex numbers
  - **Variable-length instructions**
    - Enables compact representation

ISA Classification

- **CISC Example: memset implementation on x86**

```
void memset(char* dest, char value, int count) {
  while (count--) { *dest++ = value; }
}
```

```
memset:
   push  edi                ; save EDI before change
   mov edi, [esp+8]         ; load dest into EDI
   mov al, [esp+12]         ; load value into lower 8-bit of EAX
   mov ecx, [esp+16]        ; load count into ECX
   rep stosb                ; repeat storing AL into memory at [EDI]
   pop  edi                 ; restore EDI
   ret                      ; Return to caller
```

ISA Classification

- CISC Design Motivations
  - Reducing register file
    - Transistor cost was high
  - Reducing program size
    - Memory was very expensive
  - Complex instructions
    - Simplify assembly programming
    - Compilers were very primitive

ISA Classification

- RISC Characteristics
  - **Smaller set of instructions**
  - **Larger register file (e.g. RISC-V - 32 registers)**
  - Optimized instructions to execute in single clock
  - Uniform instruction format
  - Identical general-purpose registers
  - **Simple address mode**
  - Fewer data type
  - **Fixed-Length Instruction**
    - Simplify instruction decoding

ISA Classification

- **RISC Example: memset implementation on RISC-V**

```
void memset(char* dest, char value, int count) {
  while (count--) { *dest++ = value; }
}
```

```
memset:
    beqz x12, memset_done    # If count == 0, return immediately

memset_loop:
    sb x11, 0(x10)           # Store the value in x11 to the address in x10
    addi x10, x10, 1         # Increment dest pointer (dest++)
    addi x12, x12, -1        # Decrement count (count--)
    bnez x12, memset_loop    # Repeat while count != 0

memset_done:
    ret                      # Return to caller
```

ISA Classification

- RISC Design Motivations
  - Simpler was often faster
    - Complex instructions took more time than multiple simpler ones
  - Memory got cheaper and larger
    - Could store more instructions
  - Transistors got cheaper
    - Could afford larger register file
  - Compiler got better
    - Move complexity to the compiler

How did Intel address the shortcoming of CISC?

ISA Classification

- Intel solutions
  - Intel cannot drop CISC
    - Legacy software and operating systems
    - Some critical applications cannot be modified (no sources)
    - Introducing a new ISA versus licensing ARM both costly
  - Introduce RISC-like microarchitecture
    - Translate CISC into RISC micro-ops

ISA Classification

- CISC-like ISA is still relevant today
  - Domain-specific accelerators
  - e.g. Google TPU's Matrix ISA
    - Instructions are complex and execute for several cycles

```
Load(A, Address1) // Load matrix from memory into buffer A
Load(B, Address2) // Load matrix from memory into buffer B
MatrixMul(A, B, C) // Perform C = A * B
```

TPU paper: https://dl.acm.org/doi/pdf/10.1145/3079856.3080246

ISA Classification

- Parallelism
  - VLIW ISA
  - Vector ISA

ISA Classification

- VLIW ISA (Very Long Instruction Word)
  - Encode multiple operations into a single long instruction word
  - Execute each operation on a different execution unit (parallelism)
  - Example: Intel IA64 (2001)
    - 128-bit instruction bundle (3 slots)
    - 5-bit template and 3x 41-bit instruction slots
    - {LDB R1 [R4], Add R2 R7 R4, STB R3 [R7]}
  - Static instruction scheduling
    - The compiler is responsible to ensuring that all operations within the instruction have no internal dependencies and can execute in parallel.

ISA Classification

- VLIW ISA (Very Long Instruction Word)
  - Requires multiple fetches to decode long instructions?
    - Not necessarily, can support fixed-length 64/128-bit instructions
    - Otherwise, fetch will be too slow

Whiteboard: VLIW Machine

ISA Classification

- VLIW ISA Design Motivation
  - Hardware simplification
    - Move the complexity of instruction scheduling to the compiler
    - Simplify data dependency handling in hardware
  - High Parallelism
    - Enable multiple operations to execute in parallel
    - Exploit Instruction-level parallelism (ILP)

ISA Classification

- VLIW ISA Shortcomings
  - Compiler complexity
    - Expensive if hardware changes
  - Portability (ISA)
    - Too much coupling with hardware (e.g. instruction latency)
    - Process changes renders binary incompatible
  - Static Scheduling
    - Limits hardware runtime optimizations
      - OoO execution is constrained
  - Code Size
    - Instructions bundle not always full,
    - Increase memory bandwidth

ISA Classification

- Vector ISAs
  - Single Instruction Multiple Data (SIMD)
  - Fetch multiple Data at once
  - Parallel data processing
  - Examples:
    - Intel AVX
      - Vfmadd231ps y3, y2, y1, y0
      - $y3_{7\text{-}0} = y0_{7\text{-}0} * y1_{7\text{-}0} + y2_{7\text{-}0}$
    - ARM Neon
      - vmla.f32 v2, v1, v0
      - $v2_{3\text{-}0} \mathrel{+}= v0_{3\text{-}0} * v1_{3\text{-}0}$

Vector Register File

ID	DATA0	DATA1	DATA2	DATA3
0	0x00FF00FF	0x00000000	0x00FF00FF	0x00FF00FF
1	0xAA008800	0xFFFFFFFF	0x00FF00FF	0xFFFFFFFF
2	0xF0F0F0F0	0xF0F0F0F0	0xF0F0F0F0	0xF0F0F0F0
3	0x00000000	0x00000000	0xFFFFFFFF	0x00FF00FF

ISA Classification

- Vector ISAs Limitations
  - Memory bandwidth
    - Increased memory requests
  - Control-flow limitations
    - Divergent conditional values
  - Programmability
    - Vectorization not always possible
    - Data alignment restrictions

ISA Classification

- Vector ISAs Limitations
  - Memory bandwidth
    - Increased memory requests
  - Control-flow limitations
    - Divergent conditional values
  - Programmability
    - Vectorization not always possible
    - Data alignment restrictions

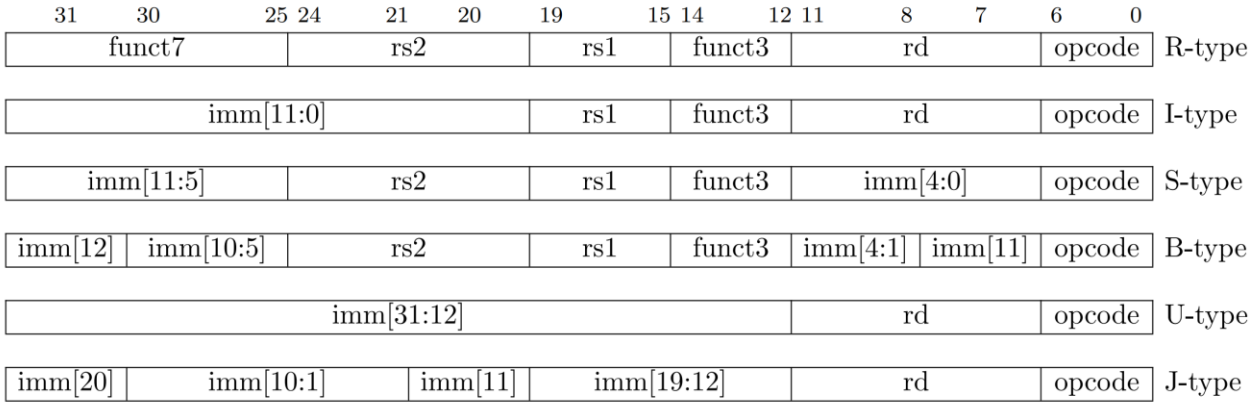
Attendance

- Please Fill the form with today’s keyword to register your attendance.

Q&A

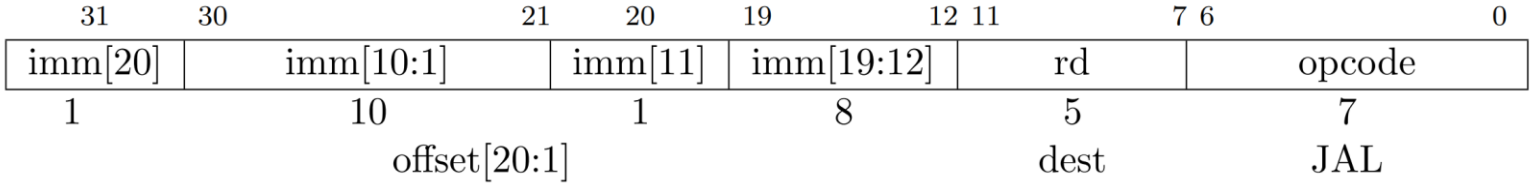
RISC-V Base Instruction Formats

- **Formats classified based on addressing mode and operation type**
  - **R-type: register**
  - **I-type: direct**
  - **S-type: store**
  - **B-type: branch**
  - **U-type: immediate**
  - **J-type: jump**



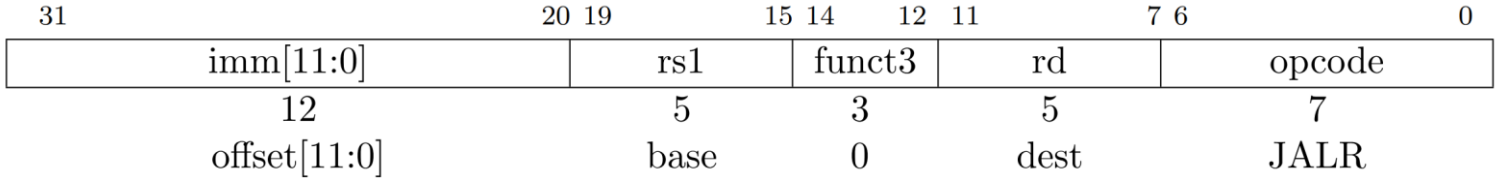
RISC-V Control transfer Instructions

- JAL: Plain unconditional jumps
  - Target address = sign-extended 20-bit offset
  - Link register *rd*
  - If (rd != 0) PC + 4 is written into *rd*



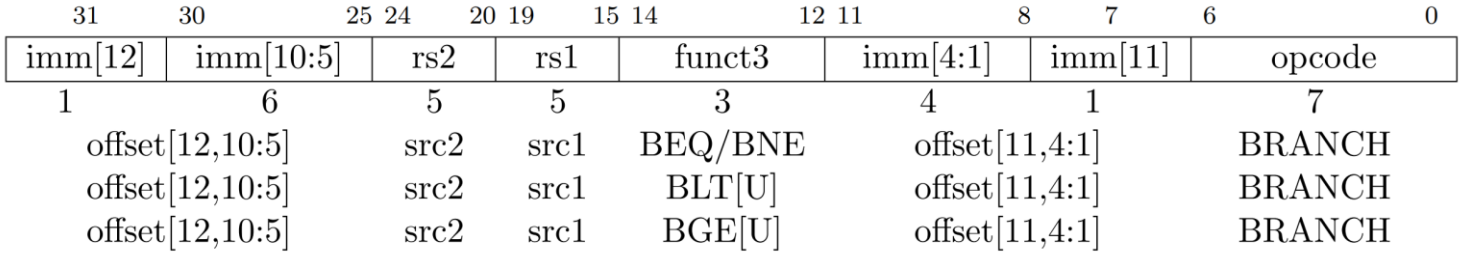
RISC-V Control transfer Instructions

- JALR: indirect jump
  - Target address = base + sign-extended 12-bit offset
  - Link register *rd*
  - If (rd != 0) PC + 4 is written into *rd*



RISC-V Control transfer Instructions

- Conditional Branches
  - Compare *rs1* and *rs2*
  - Target address = PC + sign-extended 12-bit offset
  - Funct3 = Comparison mode (EQ, NE, LT, GT, LE, GE)

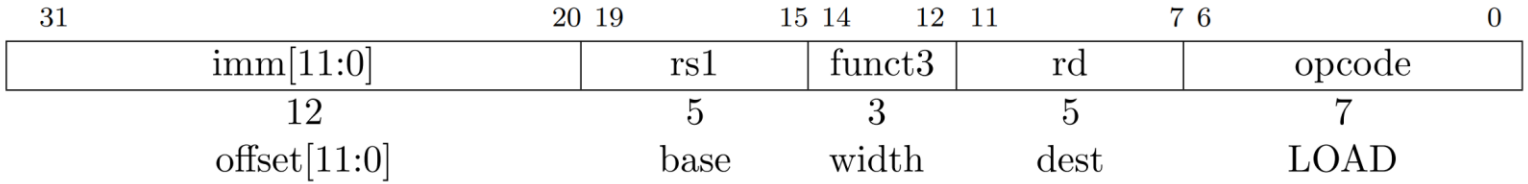


31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
offset[12,10:5]		src2	src1	BEQ/BNE	offset[11,4:1]		BRANCH
offset[12,10:5]		src2	src1	BLT[U]	offset[11,4:1]		BRANCH
offset[12,10:5]		src2	src1	BGE[U]	offset[11,4:1]		BRANCH

RISC-V Load and Store instructions

- LOAD: load value from memory
  - Rs1 = base address
  - Rd = MEM[base + offset]
  - Funct3 = word size
    - LB: 000
    - LH: 001
    - LW: 010
    - LBU: 100 (unsigned)
    - LHU: 101 (unsigned)
    - LWU: 110 (unsigned)

←Notice anything about MSB value?



RISC-V Load and Store instructions

- LOAD: Alignment constraints
  - Alignment should respect the data width size
  - LB, LBU: 1-byte alignment
  - LH, LHU: 2-byte alignment
  - LW, LWU: 4-byte alignment

e.g. Loading a 16-bit value from address 0x70000001 is not supported!

Loading a 32-bit value from address 0x70000002 is not supported!

RISC-V Load and Store instructions

- STORE: Alignment constraints
  - Should respect the width size
  - SB: 1-byte alignment
  - SH: 2-byte alignment
  - SW: 4-byte alignment

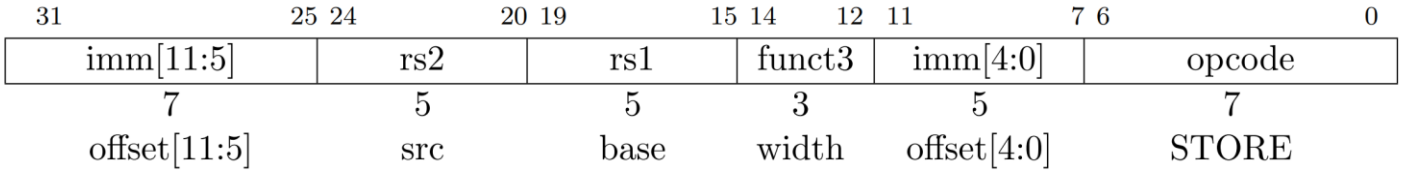
e.g. Storing a 16-bit value to address 0x50000003 is not supported!

Storing a 32-bit value to address 0x50000003 is not supported!

RISC-V Load and Store instructions

- STORE: store value to memory
  - MEM[base + offset] = src
  - Width = size of word
    - SB: 000
    - SH: 001
    - SW: 010

<- Why no unsigned variants?



Whiteboard: RISC-V Memory Alignment

RISC-V Load and Store instructions

- **Exercise:**
  - Given a 32-bit RISC-V CPU, consider you have a memory segment starting at address 0x10010000, and you need to load an unsigned byte located at address 0x1001002A into register x5. Use x2 as your base address.
  - Q1: Write the assembly code for this load instruction
  - Q2: Write the instruction code for this load instruction

RISC-V Load and Store instructions

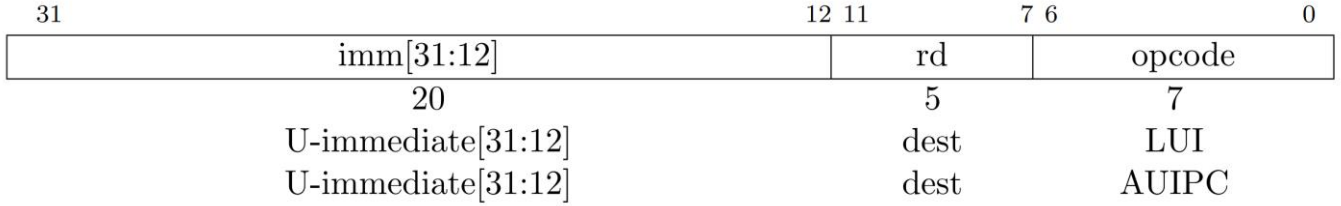
- **Q1 Solution:**
  - Target address: 0x1001002A
  - Base address: 0x10010000
  - Offset: 0x1001002A - 0x10010000 = 0x2A = 42
  - Which width mode to use?
    - LBU (100)
  - Assembly:
    - **lbu x5, 42(x2)**

RISC-V Load and Store instructions

- **Q1 Solution:**
  - We also need to update x2 with base address 0x10010000
  - Which format can we choose?
    - Let's try Load upper immediate

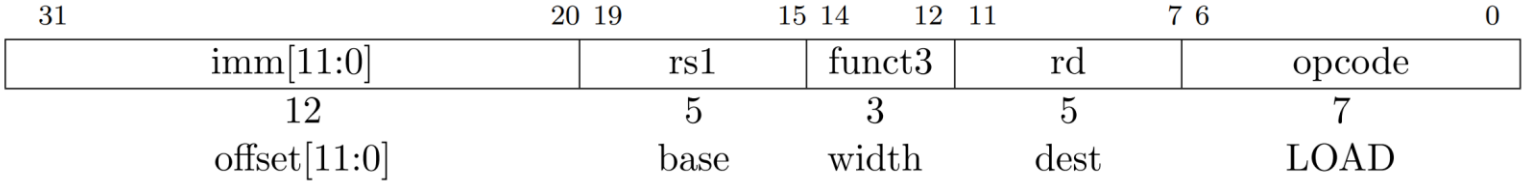
RISC-V Load and Store instructions

- **Q1 Solution: Using LUI (Load Upper Immediate) instruction**
  - Move upper 20-bits
  - 0x10010000
  - shifted right by 12 = 0x10010
  - Assembly: **lui x2, 0x10010**



RISC-V Load and Store instructions

- **Q2: Solution**
  - Opcode: 0000011
  - rd: The destination register, x5, in binary is 00101.
  - func3: 100 (for LBU)
  - rs1: The base register, x2, in binary is 00010.
  - imm[11:0]: The offset 0x2A in 12-bit binary is 000000101010.
  - Instruction code: 000000101010-00010-100-00101-0000011 (binary)
  - Instruction code: **0x02A14503 (hex)**



RISC-V Load and Store instructions

- **Exercise 2:**
  - Given a 32-bit RISC-V CPU, write assembly to load an unsigned 16-bit value located at address 0x00000005 into register x5.
  - Use x2 as your base address.
  - Assume that the following words are stored in memory addresses 0x4, 0x8, and 0xC, respectively 0x00800000, 0xC654FFFF, 0xFFFF7777.

RISC-V Load and Store instructions

- **Exercise2 Solution:**
  - 0x5 is not half-word-aligned, so you cannot perform the operation with a single load instruction
  - Final result in x5 is 0x00008000

```
# Assuming x2 needs to be set to the base address of 0x4
addi x2, x0, 0x4  # Load immediate value 0x4 into x2
lw x5, 0(x2)      # Load the word at address 0x4 into x5
srl x5, x5, 8     # Shift right by 8 bits
```

RISC-V Load and Store instructions

- **Problem:**
  - Given a 32-bit RISC-V CPU, consider you have a memory segment starting at address 0x10010000, and you need to store a half-word from register x10 into memory address 0x1001002C. Use x2 as your base address.
  - Q1: Write the assembly code for this load instruction
  - Q2: Write the instruction code for this load instruction

RISC-V Control and Status Instructions

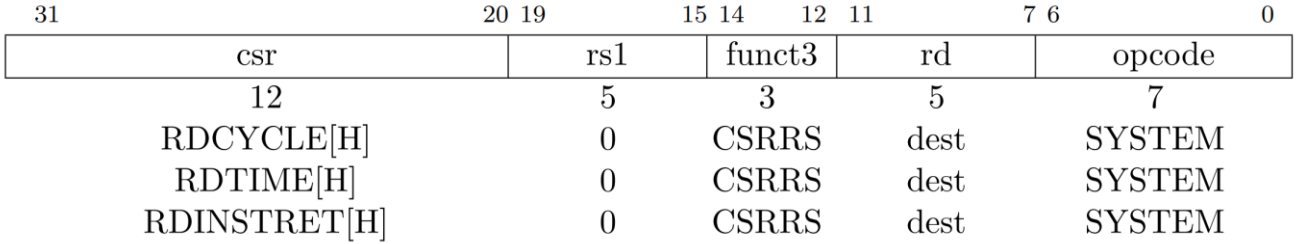
- User-level CSR Table
- Standard mapping
  - FPU flags
  - Cycle
  - Time
  - Instret
  - HPM Counters

Number	Privilege	Name	Description
Unprivileged Floating-Point CSRs			
0x001	URW	fflags	Floating-Point Accrued Exceptions.
0x002	URW	frm	Floating-Point Dynamic Rounding Mode.
0x003	URW	fcsr	Floating-Point Control and Status Register (frm + fflags).
Unprivileged Counter/Timers			
0xC00	URO	cycle	Cycle counter for RDCYCLE instruction.
0xC01	URO	time	Timer for RDTIME instruction.
0xC02	URO	instret	Instructions-retired counter for RDINSTRET instruction.
0xC03	URO	hpmcounter3	Performance-monitoring counter.
0xC04	URO	hpmcounter4	Performance-monitoring counter.
		⋮	
0xC1F	URO	hpmcounter31	Performance-monitoring counter.
0xC80	URO	cycleh	Upper 32 bits of cycle, RV32 only.
0xC81	URO	timeh	Upper 32 bits of time, RV32 only.
0xC82	URO	instreth	Upper 32 bits of instret, RV32 only.
0xC83	URO	hpmcounter3h	Upper 32 bits of hpmcounter3, RV32 only.
0xC84	URO	hpmcounter4h	Upper 32 bits of hpmcounter4, RV32 only.
		⋮	
0xC9F	URO	hpmcounter31h	Upper 32 bits of hpmcounter31, RV32 only.

Currently allocated RISC-V unprivileged CSR addresses.

RISC-V Control and Status Instructions

- Timers and counters
  - 64-bit registers
  - RD_CYCLE: total cycle time
  - RD_TIME: wall-clock time
  - RD_INSTRET: total instructions



31 20	19 15	14 12	11 7	6 0
csr	rs1	funct3	rd	opcode
12	5	3	5	7
RDCYCLE[H]	0	CSRRS	dest	SYSTEM
RDTIME[H]	0	CSRRS	dest	SYSTEM
RDINSTRET[H]	0	CSRRS	dest	SYSTEM

RISC-V Control and Status Instructions

- Timers and counters
  - How to read RD_CYCLE on 32-bit machine?
  - Why could this be challenging?
    - Counter 32-bit overflow
  - Solution
    - Read Low 32-bit
    - Read high 32-bit

RISC-V Control and Status Instructions

- **Assembly code**
  - csrrs x2, cycle, x0
  - csrrs x3, cycleh, x0
- **What timing issue may emerge from the above code?**
  - ***x2* and *x3* may not match**
  - **Hundreds of CPU cycles can happen between the two calls**
  - **Reading *cycleh* could happen after *cycle* 32-bit overflow**

RISC-V Control and Status Instructions

- **Solution**
  - *loop*: csrrs x3, cycleh, x0
  - csrrs x2, cycle, x0
  - csrrs x4, cycleh, x0
  - bne x3, x4, *loop*
- **How does this work?**
  - It takes a while for cycleh change
  - The loop handle the overflow cases

Q&A