ENIAC (U.Penn., 1946)
First Electronic General-Purpose Computer

• Blazingly fast (multiply in 2.8ms!)
  – 10 decimal digits x 10 decimal digits
• But needed 2–3 days to setup new program, as programmed with patch cords and switches

Big Idea: 
Stored-Program Computer

– Instructions are represented as bit patterns - can think of these as numbers
– Therefore, entire programs can be stored in memory to be read or written just like data
– Can reprogram quickly (seconds), don’t have to rewrite computer (days)
– Known as the “von Neumann” computers after widely distributed tech report on EDVAC project
  • Wrote-up discussions of Eckert and Mauchly
  • Anticipated earlier by Turing and Zuse

EDSAC (Cambridge, 1949)
First General Stored-Program Computer

• Programs held as numbers in memory
• 35 bit binary 2’s complement words

Consequence #1:
Everything Has a Memory Address

• Since all instructions and data are stored in memory, everything has a memory address: instructions, data words
  – Both branches and jumps use these
• C pointers are just memory addresses: they can point to anything in memory
  – Unconstrained use of addresses can lead to nasty bugs; avoiding errors up to you in C; limited in Java by language design
• One register keeps address of instruction being executed: “Program Counter” (PC)
  – Basically a pointer to memory
  – Intel calls it Instruction Pointer (a better name)
Consequence #2: Binary Compatibility

- Programs are distributed in binary form
  - Programs bound to specific instruction set
  - Different version for phones and PCs
- New machines want to run old programs ("binaries") as well as programs compiled to new instructions
- Leads to "backward-compatible" instruction set evolving over time
- Selection of Intel 8088 in 1981 for 1st IBM PC is major reason latest PCs still use 80x86 instruction set; could still run program from 1981 PC today

Instructions as Numbers (1/2)

- Most data we work with is in words (32-bit chunks):
  - Each register is a word
  - lw and sw both access memory one word at a time
- So how do we represent instructions?
  - Remember: Computer only understands 1s and 0s, so assembler string "add x10, x11, x0" is meaningless to hardware
  - RISC-V seeks simplicity: since data is in words, make instructions be fixed-size 32-bit words also
    - Same 32-bit instructions used for RV32, RV64, RV128

Instructions as Numbers (2/2)

- One word is 32 bits, so divide instruction word into “fields”
- Each field tells processor something about instruction
- We could define different fields for each instruction, but RISC-V seeks simplicity, so define six basic types of instruction formats:
  - R-format for register-register arithmetic operations
  - I-format for register-immediate arithmetic operations and loads
  - S-format for stores
  - B-format for branches (minor variant of S-format, called SB before)
  - U-format for 20-bit upper immediate instructions
  - J-format for jumps (minor variant of U-format, called UJ before)

Summary of RISC-V Instruction Formats

<table>
<thead>
<tr>
<th>Field's bit positions</th>
<th>Number of bits in field</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs1</td>
<td>rs2</td>
</tr>
<tr>
<td>7</td>
<td>5 + 5 + 3 + 5 + 7 = 32</td>
</tr>
<tr>
<td>Examples</td>
<td></td>
</tr>
<tr>
<td>- opcode is a 7-bit field that lives in bits 6-0 of the instruction</td>
<td></td>
</tr>
<tr>
<td>- rs2 is a 5-bit field that lives in bits 24-20 of the instruction</td>
<td></td>
</tr>
</tbody>
</table>

R-Format Instruction Layout

<table>
<thead>
<tr>
<th>Field's bit positions</th>
<th>Name of field</th>
<th>Number of bits in field</th>
</tr>
</thead>
<tbody>
<tr>
<td>funct7</td>
<td>rs2</td>
<td>rs1</td>
</tr>
<tr>
<td>7</td>
<td>5 + 5 + 3 + 5 + 7 = 32</td>
<td></td>
</tr>
<tr>
<td>Examples</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- opcode: partially specifies what instruction it is</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Note: This field is equal to 0110011 two for all R-Format register arithmetic instructions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- funct7+funct3: combined with opcode, these two fields describe what operation to perform</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Question: Why aren’t opcode and funct7 and funct3 a single 17-bit field?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- We’ll answer this later</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
R-Format Instructions register specifiers

- **rs1**: (Source Register #1): specifies register containing first operand
- **rs2**: specifies second register operand
- **rd**: (Destination Register): specifies register which will receive result of computation
- Each register field holds a 5-bit unsigned integer (0-31) corresponding to a register number (x0-x31)

R-Format Example

- **RISC-V Assembly Instruction**: `add x18, x19, x10`

<table>
<thead>
<tr>
<th>31</th>
<th>25</th>
<th>20</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01010</td>
<td>10011</td>
<td>000</td>
<td>10010</td>
<td>0110011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>rs2=10</td>
<td>rs1=19</td>
<td>ADD</td>
<td>rd=18</td>
<td>Reg-Reg OP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

I-Format Instructions

- What about instructions with immediates?
  - 5-bit field only represents numbers up to the value 31: immediates may be much larger than this
  - Ideally, RISC-V would have only one instruction format (for simplicity): unfortunately, we need to compromise
- Define new instruction format that is mostly consistent with R-format
  - Notice if instruction has immediate, then uses at most 2 registers (one source, one destination)

I-Format Example

- **RISC-V Assembly Instruction**: `addi x15, x1, -50`

<table>
<thead>
<tr>
<th>31</th>
<th>25</th>
<th>20</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>111111001110</td>
<td>00001</td>
<td>000</td>
<td>01111</td>
<td>0010011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imm=-50</td>
<td>rs1=1</td>
<td>ADD</td>
<td>rd=15</td>
<td>OP-Imm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
All RV32 I-format Arithmetic Instructions

| imm[11:0] | rs1 | 000 | rd | 0000011 | ADDI
| imm[11:0] | rs1 | 010 | rd | 0000011 | SLTI
| imm[11:0] | rs1 | 011 | rd | 0000011 | SELTI
| imm[11:0] | rs1 | 100 | rd | 0000011 | XORI
| imm[11:0] | rs1 | 110 | rd | 0000011 | ORI
| imm[11:0] | rs1 | 111 | rd | 0000011 | ANDI
| 000000 | shamt | rs1 | 001 | rd | 0000011 | SLLI
| 000000 | shamt | rs1 | 101 | rd | 0000011 | SRLI
| 000000 | shamt | rs1 | 101 | rd | 0000011 | SRAI

One of the higher-order immediate bits is used to distinguish “shift right logical” (SRLI) from “shift right arithmetic” (SRAI).

“Shift-by-immediate” instructions only use lower 5 bits of the immediate value for shift amount (can only shift by 0-31 bit positions).

Administrivia

- HW0 Grades were released
  - If you have an issue with them, please fill out this form by tonight: https://goo.gl/forms/QH44Iw746pcCxx4ly2
- HW1 Part 1 Due next Monday, Part 2 Due Friday Sept 22
  - Homework-oriented office hours next Monday (check the website)
  - Homework Party on next Wednesday (6:30-10pm in 293 Cory)
  - Autograded results to be released every noon starting this Friday
- Midterm #1 in 1.5 weeks: September 26!
  - Two sided 8.5” x 11” cheat sheet + RISC-V Green Card that we give you
  - DSP students: please make sure we know about your special accommodations (contact Steven Ho the head TA if you haven’t yet)

Load Instructions are also I-Type

<table>
<thead>
<tr>
<th>31</th>
<th>20 19</th>
<th>15 14</th>
<th>12 11</th>
<th>7 6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>offset[11:0]</td>
<td>base</td>
<td>width</td>
<td>dest</td>
<td>LOAD</td>
<td></td>
</tr>
</tbody>
</table>

- The 12-bit signed immediate is added to the base address in register rs1 to form the memory address
  - This is very similar to the add-immediate operation but used to create address not to create final result
- The value loaded from memory is stored in register rd

I-Format Load Example

- RISC-V Assembly Instruction:
  \( \text{lw } x14, \ 8(x2) \)

<table>
<thead>
<tr>
<th>31</th>
<th>20 19</th>
<th>15 14</th>
<th>12 11</th>
<th>7 6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>000000001000</td>
<td>00010</td>
<td>010</td>
<td>01110</td>
<td>0000011</td>
<td></td>
</tr>
</tbody>
</table>

imm=+8 rs1=2 LW rd=14 LOAD

All RV32 Load Instructions

<table>
<thead>
<tr>
<th>31</th>
<th>20 19</th>
<th>15 14</th>
<th>12 11</th>
<th>7 6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>000000001000</td>
<td>00010</td>
<td>010</td>
<td>01110</td>
<td>0000011</td>
<td></td>
</tr>
</tbody>
</table>

LBU is “load unsigned byte”
- LH is “load halfword”, which loads 16 bits (2 bytes) and sign-extends to fill destination 32-bit register
- LHU is “load unsigned halfword”, which zero-extends 16 bits to fill destination 32-bit register
- There is no LWU in RV32, because there is no sign/zero extension needed when copying 32 bits from a memory location into a 32-bit register

funct3 field encodes size and signedness of load data

Break!
S-Format Used for Stores

<table>
<thead>
<tr>
<th>31</th>
<th>25 24</th>
<th>20 19</th>
<th>15 14</th>
<th>12 11</th>
<th>7 6</th>
<th>0</th>
</tr>
</thead>
</table>

- Store needs to read two registers, rs1 for base memory address, and rs2 for data to be stored, as well as need immediate offset!
- Can’t have both rs2 and immediate in same place as other instructions!
- Note that stores don’t write a value to the register file, no rd!
- RISC-V design decision is move low 5 bits of immediate to where rd field was in other instructions – keep rs1/rs2 fields in same place
  - register names more critical than immediate bits in hardware design

S-Format Example

- RISC-V Assembly Instruction: `sw x14, 8(x2)`

\[
\begin{array}{cccccc}
\text{imm[11:5]} & \text{rs2} & \text{rs1} & \text{funct3} & \text{imm[4:0]} & \text{opcode} \\
0 & 5 & 5 & 3 & 3 & 7 \\
\text{offset[11:5]} & \text{src} & \text{base} & \text{width} & \text{offset[4:0]} & \text{STORE} \\
000000 & 01110 & 00010 & 010 & 01000 & 0100011 \\
\end{array}
\]

0000000 01000

All RV32 Store Instructions

- Risc-V Conditional Branches
  - E.g., `BEQ x1, x2, Label`
  - Branches read two registers but don’t write a register (similar to stores)
  - How to encode label, i.e., where to branch to?

Branching Instruction Usage

- Branches typically used for loops (if-else, while, for)
  - Loops are generally small (< 50 instructions)
  - Function calls and unconditional jumps handled with jump instructions (j-Format)
- Recall: Instructions stored in a localized area of memory (Code/Text)
  - Largest branch distance limited by size of code
  - Address of current instruction stored in the program counter (PC)

PC-Relative Addressing

- PC-Relative Addressing: Use the immediate field as a two’s-complement offset to PC
  - Branches generally change the PC by a small amount
  - Can specify ±211 addresses from the PC
  - Why not use byte address offset from PC?
Scaling Branch Offset

- One idea: To improve the reach of a single branch instruction, multiply the offset by four bytes before adding to PC
- This would allow one branch instruction to reach $\pm 2^{11} \times 32$-bit instructions either side of PC
  - Four times greater reach than using byte offset

Branch Calculation

- If we don’t take the branch:
  \[ \text{PC} = \text{PC} + 4 \] (i.e., next instruction)
- If we do take the branch:
  \[ \text{PC} = \text{PC} + \text{immediate} \times 4 \]

Observations:
- immediate is number of instructions to jump (remember, specifies words) either forward (+) or backwards (−)

RISC-V Feature, n×16-bit instructions

- Extensions to RISC-V base ISA support 16-bit compressed instructions and also variable-length instructions that are multiples of 16-bits in length
- To enable this, RISC-V scales the branch offset by 2 bytes even when there are no 16-bit instructions
- Reduces branch reach by half and means that 3/4 of possible targets will be errors on RISC-V processors that only support 32-bit instructions (as used in this class)
- RISC-V conditional branches can only reach $\pm 2^{10} \times 32$-bit instructions either side of PC

RISC-V B-Format for Branches

- B-format is mostly same as S-Format, with two register sources (rs1/rs2) and a 12-bit immediate
- But now immediate represents values -4096 to +4094 in 2-byte increments
- The 12 immediate bits encode even 13-bit signed byte offsets (lowest bit of offset is always zero, so no need to store it)

Branch Example, determine offset

- RISC-V Code:
  
  Loop:  
  beq x19, x10, End
  add x18, x18, x10
  addi x19, x19, -1
  j Loop
  End:  # target instruction

- Branch offset = $4 \times 32$-bit instructions = 16 bytes
- (Branch with offset of 0, branches to itself)

Branch Example, encode offset

- RISC-V Code:
  
  Loop:  
  beq x19, x10, End
  add x18, x18, x10
  addi x19, x19, -1
  j Loop
  End:  # target instruction

- Offset = 16 bytes = 8x2

<table>
<thead>
<tr>
<th>imm</th>
<th>rs2=10</th>
<th>rs1=19</th>
<th>BEQ</th>
<th>imm</th>
<th>BRANCH</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>10101</td>
<td>10011</td>
<td>000</td>
<td>????</td>
<td>110011</td>
</tr>
</tbody>
</table>

imm  rs2=10  rs1=19  BEQ  imm  BRANCH
RISC-V Immediate Encoding

Instruction Encodings, inst[31:0]

- Immediate Encodings, inst[31:0]
  - imm[31:12] rs2 rs1 imm[11:0] opcode | B-type
  - imm[12] imm[11:0] rs1 rs2 funct3 funct7 | I-type

32-bit immediates produced, imm[11:0]

38

Branch Example, complete encoding

beq x19, x10, offset = 16 bytes

13-bit immediate, imm[12:0], with value 16

imm[0] discarded, always zero

All RISC-V Branch Instructions

- beq x19, x10, far
- bne x10, x0, next

Questions on PC-addressing

- Does the value in branch immediate field change if we move the code?
  - If moving individual lines of code, then yes
  - If moving all of code, then no (because PC-relative offsets)
- What do we do if destination is > 2^10 instructions away from branch?
  - Other instructions save us

Break!
U-Format for “Upper Immediate” instructions

<table>
<thead>
<tr>
<th>31</th>
<th>r/m[31:12]</th>
<th>rd</th>
<th>(opcode)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>5</td>
<td>7</td>
<td>LUI</td>
</tr>
<tr>
<td>U-immediate[31:12]</td>
<td>dest</td>
<td>UIM</td>
<td></td>
</tr>
</tbody>
</table>

- Has 20-bit immediate in upper 20 bits of 32-bit instruction word
- One destination register, rd
- Used for two instructions
  - LUI – Load Upper Immediate
  - AUIPC – Add Upper Immediate to PC

LUI to create long immediates

- LUI writes the upper 20 bits of the destination with the immediate value, and clears the lower 12 bits.
- Together with an ADDI to set low 12 bits, can create any 32-bit value in a register using two instructions (LUI/ADDI).

LUI x10, 0x87654 # x10 = 0x87654000
ADDI x10, x10, 0x321 # x10 = 0x87654321

One Corner Case

How to set 0xDEADBEEF?
LUI x10, 0xDEADB # x10 = 0xDEADB000
ADDI x10, x10, 0xEEF # x10 = 0xDEADBEF

ADDI 12-bit immediate is always sign-extended, if top bit is set, will subtract -1 from upper 20 bits

Solution

How to set 0xDEADBEEF?
LUI x10, 0xDEADC # x10 = 0xDEADC000
ADDI x10, x10, 0xEEF # x10 = 0xDEADBEEF

Pre-increment value placed in upper 20 bits, if sign bit will be set on immediate in lower 12 bits.

Assembler pseudo-op handles all of this:
li x10, 0xDEADBEEF # Creates two instructions

AUIPC

- Adds upper immediate value to PC and places result in destination register
- Used for PC-relative addressing
- Label: AUIPC x10, 0 # Puts address of label in x10

J-Format for Jump Instructions

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|  1 |  10 |  1 |  8 |  5 |  7 | offset[20:1] | dest | JAL |

- JAL saves PC+4 in register rd (the return address)
  - Assembler "j" jump is pseudo-instruction, uses JAL but sets rd=x0 to discard return address.
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within ±2^19 locations, 2 bytes apart
  - ±2^18 32-bit instructions
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost
### Uses of JAL

# j pseudo-instruction

\[ j \text{ Label} = \text{jal} \ x0, \text{ Label} \] # Discard return address

# Call function within 2^{18} instructions of PC

\[ \text{jal} \ \text{ra}, \ \text{FuncName} \]

### JALR Instruction (I-Format)

| 31 | 29 | 20 | 19 | 18 | 17 | 16 | 14 | 13 | 11 | 10 | 9 | 8 | 7 | 6 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|

- JALR rd, rs, immediate
  - Writes PC+4 to rd (return address)
  - Sets PC = rs + immediate
  - Uses same immediates as arithmetic and loads
    * no multiplication by 2 bytes

### Uses of JALR

# ret and jr pseudo-instructions

\[ \text{ret} = \text{jr} \ \text{ra} = \text{jalr} \ x0, \ ra, \ 0 \]

# Call function at any 32-bit absolute address

\[ \text{lui} \ x1, <\text{hi20bits}> \]

\[ \text{jalr} \ \text{ra}, \ x1, <\text{lo12bits}> \]

# Jump PC-relative with 32-bit offset

\[ \text{auipc} \ x1, <\text{hi20bits}> \]

\[ \text{jalr} \ x0, x1, <\text{lo12bits}> \]

### Summary of RISC-V Instruction Formats

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 0 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| funct7 | n2 | n1 | funct3 | rd | opc0 | R-type |
| imm[11:0] | n1 | funct3 | rd | opc0 | I-type |
| imm[31:15] | n2 | n1 | funct3 | imm[4:0] | opc0 | J-type |

### Complete RV32I ISA

Not in CS61C