Agenda for Today & Next Few Lectures

- The von Neumann model
- LC-3: An example of von Neumann machine
- LC-3 and MIPS Instruction Set Architectures
- LC-3 and MIPS assembly and programming
- Introduction to microarchitecture and single-cycle microarchitecture
- Multi-cycle microarchitecture
What Have We Been Learning?

- Basic elements of a computer & the von Neumann model
  - LC-3: An example von Neumann machine

- Instruction Set Architectures: LC-3 and MIPS
  - Operate instructions
  - Data movement instructions
  - Control instructions

- Instruction formats

- Addressing modes
Readings

**Last week**
- Von Neumann Model, ISA, LC-3, and MIPS
  - P&P, Chapters 4, 5 (we will follow these today & tomorrow)
  - H&H, Chapter 6 (until 6.5)
  - P&P, Appendices A and C (ISA and microarchitecture of LC-3)
  - H&H, Appendix B (MIPS instructions)
- Programming
  - P&P, Chapter 6 (we will follow this tomorrow)
- **Recommended**: H&H Chapter 5, especially 5.1, 5.2, 5.4, 5.5

**This week**
- Introduction to microarchitecture and single-cycle microarchitecture
  - H&H, Chapter 7.1-7.3
  - P&P, Appendices A and C
- **Multi-cycle microarchitecture**
  - H&H, Chapter 7.4
  - P&P, Appendices A and C
Recall: LC-3: A von Neumann Machine

- Program Counter
- Control signals
- Data
- Finite State Machine (for Generating Control Signals)
- Instruction Register
- Memory Data Register
- Memory Address Register

**Diagram Components:**
- ALU: 2 inputs, 1 output
- 8 General Purpose Registers (GPR)
- ALU operation
- GateALU
- Keyboard KBDR (data), KBSR (status)
- Monitor DDR (data), DSR (status)

**Figure 4.3** The LC-3 as an example of the von Neumann model
Recall: Instruction Types

- There are **three main types of instructions**

- **Operate instructions**
  - Execute operations in the ALU

- **Data movement instructions**
  - Read from or write to memory

- **Control flow instructions**
  - Change the sequence of execution

- Let us start with some example instructions
Instruction (Processing) Cycle
Recall: The Instruction (Processing) Cycle

- FETCH
- DECODE
- EVALUATE ADDRESS
- FETCH OPERANDS
- EXECUTE
- STORE RESULT
Recall: Control of the Instruction Cycle

State 1
- The FSM asserts GatePC and LD.MAR
- It selects input (+1) in PCMUX and asserts LD.PC

State 2
- MDR is loaded with the instruction

State 3
- The FSM asserts GateMDR and LD.IR

State 4
- The FSM goes to next state depending on opcode

State 63
- JMP loads register into PC

Full state diagram in Patt&Pattel, Appendix C
Recall: Full State Machine for LC-3b

Figure C.2: A state machine for the LC-3b

NOTES
B=off6: Base + SEXT[off6]
PC=off9: PC + SEXT[off9]
*OP2 may be SR or SEXT[imm5]
** [15:8] or [7:0] depending on MAR[6]
LC-3 and MIPS
Instruction Set Architectures
Instructions (Opcodes)
Data Types
Addressing Modes
Recall: Addressing Modes

An addressing mode is a mechanism for specifying where an operand is located.

There are five addressing modes in LC-3:
- **Immediate or literal** (constant)
  - The operand is in some bits of the instruction
- **Register**
  - The operand is in one of R0 to R7 registers
- Three memory addressing modes
  - **PC-relative**
  - **Indirect**
  - **Base+offset**

MIPS has *pseudo-direct addressing* (for *j* and *jal*), additionally, but does **not** have indirect addressing.
Recall: Many Tradeoffs in ISA Design...

- Execution model – sequencing model and processing style
- Instruction length
- Instruction format
- Instruction types
- Instruction complexity vs. simplicity
- Data types
- Number of registers
- Addressing mode types
- Memory organization (address space, addressability, endianness, ...)
- Memory access restrictions and permissions
- Support for multiple instructions to execute in parallel?
- ...

...
Operate Instructions
Data Movement Instructions and Addressing Modes
Data Movement Instructions

- In LC-3, there are seven data movement instructions
  - LD, LDR, LDI, LEA, ST, STR, STI

- Format of load and store instructions
  - Opcode (bits [15:12])
  - DR or SR (bits [11:9])
  - Address generation bits (bits [8:0])
  - Four ways to interpret bits, called **addressing modes**
    - PC-Relative Mode
    - Indirect Mode
    - Base+Offset Mode
    - Immediate Mode

- In MIPS, there are only **Base+offset** and **Immediate modes** for load and store instructions
### PC-Relative Addressing Mode

- **LD (Load) and ST (Store)**

  ![Addressing Mode Table]

  - **OP** = opcode
    - E.g., LD = 0010
    - E.g., ST = 0011
  
  - **DR** = destination register in LD
  
  - **SR** = source register in ST

  - **LD:** \( DR \leftarrow \text{Memory}[PC^\dagger + \text{sign-extend}(\text{PCoffset9})] \)

  - **ST:** \( \text{Memory}[PC^\dagger + \text{sign-extend}(\text{PCoffset9})] \leftarrow SR \)

  \(^\dagger\) This is the incremented PC
LD in LC-3

- **LD assembly and machine code**

**LC-3 assembly**

LD R2, 0x1AF

**Field Values**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>PCoffset9</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>0x1AF</td>
</tr>
</tbody>
</table>

**Machine Code**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>PCoffset9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0</td>
<td>0 1 0</td>
<td>1 1 0 1 0 1 1 1 1</td>
</tr>
</tbody>
</table>

The memory address is **only +255 to -256 locations away of the LD or ST instruction**

**Limitation:** The PC-relative addressing mode cannot address far away from the instruction
Indirect Addressing Mode

- **LDI (Load Indirect) and STI (Store Indirect)**

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
<td>DR/SR</td>
<td>PCoffset9</td>
</tr>
</tbody>
</table>

- **OP = opcode**
  - E.g., LDI = 1010
  - E.g., STI = 1011

- **DR = destination register in LDI**
- **SR = source register in STI**

- **LDI:** $\text{DR} \leftarrow \text{Memory[Memory[PC}^\dagger + \text{sign-extend(PCoffset9)]]}$

- **STI:** $\text{Memory[Memory[PC}^\dagger + \text{sign-extend(PCoffset9)]]} \leftarrow \text{SR}$

$^\dagger$ This is the incremented PC
Now the address of the operand can be anywhere in the memory
Base+Offset Addressing Mode

- **LDR (Load Register) and STR (Store Register)**

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>8 7 6</th>
<th>5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
<td>DR/SR</td>
<td>BaseR</td>
<td>offset6</td>
</tr>
<tr>
<td>4 bits</td>
<td>3 bits</td>
<td>3 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

- **OP = opcode**
  - E.g., LDR = 0110
  - E.g., STR = 0111

- **DR = destination register in LDR**
- **SR = source register in STR**

- **LDR:** $DR \leftarrow \text{Memory}[\text{BaseR} + \text{sign-extend}(\text{offset6})]$  

- **STR:** $\text{Memory}[\text{BaseR} + \text{sign-extend}(\text{offset6})] \leftarrow \text{SR}$
LDR in LC-3

- LDR assembly and machine code

LC-3 assembly

LDR R1, R2, 0x1D

Field Values

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>BaseR</th>
<th>offset6</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
<td>0x1D</td>
</tr>
</tbody>
</table>

Machine Code

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>BaseR</th>
<th>offset6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0110</td>
<td>001</td>
<td>010</td>
<td>011101</td>
</tr>
</tbody>
</table>

Again, the address of the operand can be anywhere in the memory.
Address Calculation in LC-3 Data Path

Figure 5.18 The data path of the LC-3
In MIPS, **lw** and **sw** use base+offset mode (or **base addressing mode**)

**High-level code**

\[ A[2] = a; \]

**MIPS assembly**

\[ sw \quad $s3, \quad 8($s0) \]

\[ Memory[\$s0 + 8] \leftarrow \$s3 \]

**Field Values**

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>43</td>
<td>16</td>
<td>19</td>
<td>8</td>
</tr>
</tbody>
</table>

**imm** is the 16-bit offset, which is **sign-extended to 32 bits**
### High-level code

\[
a = A[0];
\]
\[
c = a + b - 5;
\]
\[
B[0] = c;
\]

### MIPS registers

\[
A = \$s0
\]
\[
b = \$s2
\]
\[
B = \$s1
\]

### LC-3 registers

\[
A = R0
\]
\[
b = R2
\]
\[
B = R1
\]

### MIPS assembly

\[
lw \quad \$t0, 0(\$s0)
\]
\[
add \quad \$t1, \$t0, \$s2
\]
\[
addi \quad \$t2, \$t1, -5
\]
\[
sw \quad \$t2, 0(\$s1)
\]

### LC-3 assembly

\[
LDR \quad R5, R0, #0
\]
\[
ADD \quad R6, R5, R2
\]
\[
ADD \quad R7, R6, #-5
\]
\[
STR \quad R7, R1, #0
\]
Immediate Addressing Mode (in LC-3)

- **LEA** (Load Effective Address)

  - **OP** = 1110

  - **DR** = destination register

  - **LEA**: DR ← PC$^*$ + sign-extend(PCoffset9)

  $^*$This is the incremented PC

What is the **difference from PC-Relative** addressing mode?

**Answer**: Instructions with **PC-Relative** mode load from memory, but **LEA** does not → Hence the name **Load Effective Address**

$^*$This is the incremented PC
LEA in LC-3

- LEA assembly and machine code

LC-3 assembly

LEA R5, #-3

Field Values

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>PCoffset9</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>5</td>
<td>0x1FD</td>
</tr>
</tbody>
</table>

Machine Code

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>PCoffset9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110</td>
<td>101</td>
<td>1111111101</td>
</tr>
</tbody>
</table>

Figure 5.9: Data path relevant to the execution of LEA R5, #-3

Note that the Base+offset addressing mode also allows the address of the operand to be anywhere in the computer's memory.

5.3.4 Immediate Mode

The fourth and last addressing mode used by the data movement instructions is the immediate (or, literal) addressing mode. It is used only with the load effective address (LEA) instruction.

LEA (opcode = 1110) loads the register specified by bits [11:9] of the instruction with the value formed by adding the incremented program counter to the sign-extended bits [8:0] of the instruction. The immediate addressing mode is so named because the operand to be loaded into the destination register is obtained immediately, that is, without requiring any access of memory.

The LEA instruction is useful to initialize a register with an address that is very close to the address of the instruction doing the initializing. If memory location x4018 contains the instruction LEA R5, #−3, and the PC contains x4018, R5 will contain x4016 after the instruction at x4018 is executed.

Figure 5.9 shows the relevant parts of the data path required to execute the LEA instruction. Note that no access to memory is required to obtain the value to be loaded.
Address Calculation in LC-3 Data Path

Figure 5.18 The data path of the LC-3
Immediate Addressing Mode in MIPS

- In MIPS, **lui** (load upper immediate) loads a 16-bit immediate into the upper half of a register and sets the lower half to 0

- It is used to assign 32-bit constants to a register

High-level code

```c
a = 0x6d5e4f3c;
```

MIPS assembly

```assembly
# $s0 = a
lui $s0, 0x6d5e
ori $s0, 0x4f3c
```
What is the final value of R3?

P&P, Chapter 5.3.5

<table>
<thead>
<tr>
<th>Address</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x30F6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>x30F7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>x30F8</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>x30F9</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>x30FA</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>x30FB</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>x30FC</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

R1 <- PC-3
R2 <- R1+14
M[x30F4] <- R2
R2 <- 0
R2 <- R2+5
M[R1+14] <- R2
R3 <- M[M[x30F4] ]
## LC-ISA Instruction Encodings

### Formats of the Entire LC-3 Instruction Set

*NOTE: * `+` indicates instructions that modify condition codes.

```plaintext
| Instruction | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-------------|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| ADD*        | 0001 | DR | SR1 | 00 | SR2 |
| ADD*        | 0001 | DR | SR1 | 1  | imm5|
| AND*        | 0101 | DR | SR1 | 00 | SR2 |
| AND*        | 0101 | DR | SR1 | 1  | imm5|
| BR          | 0000 | n  | z  | p  | PCoffset9 |
| JMP         | 1100 | 000 | BaseR | 000000 |
| JSR         | 0100 | 1  |     | PCoffset11 |
| JSRR        | 0100 | 0  | 00  | BaseR | 000000 |
| LD*         | 0010 | DR |     | PCoffset9 |
| LDI*        | 1010 | DR |     | PCoffset9 |
| LDR*        | 0110 | DR | BaseR | offset6 |
| LEA*        | 1110 | DR |     | PCoffset9 |
| NOT*        | 1001 | DR | SR  | 111111 |
| RET         | 1100 | 000 | 111 | 000000 |
| RTI         | 1000 |     |     | 0000000000 |
| ST          | 0011 | SR |     | PCoffset9 |
| STI         | 1011 | SR |     | PCoffset9 |
| STR         | 0111 | SR | BaseR | offset6 |
| TRAP        | 1111 | 0000 |     | trapvect8 |
| reserved    | 1101 |     |     |     |

Figure 5.3  Formats of the entire LC-3 instruction set. NOTE: *+* indicates instructions that modify condition codes.

Patt & Patel, Back cover inside
Addressing Example in LC-3

- What is the final value of R3?

The final value of R3 is 5

| Address | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---------|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| x30F6   | LEA| 1  | 1  | 0  | 0  | 0  | 1 | -3| 1 | 1 | 1 | 1 | 1 | 1 | 1 | R1 = PC – 3 = 0x30F7 – 3 = 0x30F4 |
|         | ADD| 0  | 0  | 1  | 0  | 1  | 0 | 0  | 0 | 0 | 1 | 4 | 1 | 1 | 1 | R2 = R1 + 14 = 0x30F4 + 14 = 0x3102 |
| x30F8   | ST | 0  | 1  | 1  | 0  | 1  | 0 | -5| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | M[PC – 5] = M[0x030F4] = 0x3102 |
| x30F9   | AND| 1  | 0  | 1  | 0  | 1  | 0 | 0  | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | R2 = 0 |
| x30FA   | ADD| 0  | 0  | 1  | 0  | 1  | 0 | 0  | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | R2 = R2 + 5 = 5 |
| x30FB   | STR| 1  | 1  | 1  | 0  | 1  | 0 | 0  | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | M[R1 + 14] = M[0x30F4 + 14] = M[0x3102] = 5 |
| x30FC   | LDI| 0  | 1  | 0  | 0  | 1  | 1 | -3| 1 | 1 | 1 | 1 | 1 | R3 = M[M[PC – 9]] = M[M[0x30FD – 9]] = M[M[0x30F4]] = M[0x3102] = 5 |
Control Flow Instructions
Control Flow Instructions

- Allow a program to execute out of sequence

- Conditional branches and unconditional jumps
  - Conditional branches are used to make decisions
    - E.g., if-else statement
  - In LC-3, three condition codes are used
  - Jumps are used to implement
    - Loops
    - Function calls
  - JMP in LC-3 and j in MIPS
    - We have already seen these
Conditional Control Flow (Conditional Branching)
Condition Codes in LC-3

- Each time one GPR (R0-R7) is written, three single-bit registers are updated.

- Each of these condition codes are either set (set to 1) or cleared (set to 0).
  - If the written value is negative
    - N is set, Z and P are cleared.
  - If the written value is zero
    - Z is set, N and P are cleared.
  - If the written value is positive
    - P is set, N and Z are cleared.

- x86 and SPARC are examples of ISAs that use condition codes.
Conditional Branches in LC-3

- **BRz (Branch if Zero)**

  - $\text{BRz \ PCoffset9}$
  - 0000 | n | z | p | PCoffset9
  - 4 bits | 9 bits

- $n, z, p =$ **which condition code is tested** (N, Z, and/or P)
  - n, z, p: instruction bits to identify the condition codes to be tested
  - N, Z, P: values of the corresponding condition codes

- PCoffset9 = **immediate or constant value**

- **if** $(n \text{ AND } N) \text{ OR } (p \text{ AND } P) \text{ OR } (z \text{ AND } Z))$
  - then $PC \leftarrow PC^\dagger + \text{sign-extend(PCoffset9)}$

- **Variations:** BRn, BRz, BRp, BRzp, BRnp, BRnz, BRnzp

$^\dagger$ This is the incremented PC
Conditional Branches in LC-3

- **BRz**

**BRz 0x0D9**

*What if $n = z = p = 1$?*  
(i.e., BRnzp)

*And what if $n = z = p = 0$?*

---

$n, z, p$ are the instruction bits to identify the condition codes to be tested.
Conditional Branches in MIPS

- **beq (Branch if Equal)**

\[
\text{beq } \$s0, \$s1, \text{offset}
\]

<table>
<thead>
<tr>
<th></th>
<th>rs</th>
<th>rt</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- 4 = opcode
- rs, rt = source registers
- offset = immediate or constant value
- if rs == rt
  - then PC ← PC\(^\dagger\) + sign-extend(offset) \(*\) 4
- Variations: beq, bne, blez, bgtz

\(^\dagger\) This is the incremented PC
Branch If Equal in MIPS and LC-3

- This is an example of **tradeoff** in the instruction set
  - The same functionality requires **more instructions** in LC-3
  - But, the **control logic** requires **more complexity** in MIPS
What We Learned

- Basic elements of a computer & the von Neumann model
  - LC-3: An example von Neumann machine

- Instruction Set Architectures: LC-3 and MIPS
  - Operate instructions
  - Data movement instructions
  - Control instructions

- Instruction formats

- Addressing modes

<table>
<thead>
<tr>
<th>Problem</th>
<th>Algorithm</th>
<th>Program/Language</th>
<th>System Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW/HW Interface</td>
<td>Micro architecture</td>
<td>Logic</td>
<td>Devices</td>
</tr>
<tr>
<td>Electrons</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
There Is A Lot More to Cover on ISAs
Many Different ISAs Over Decades

- x86
- PDP-x: Programmed Data Processor (PDP-11)
- VAX
- IBM 360
- CDC 6600
- SIMD ISAs: CRAY-1, Connection Machine
- VLIW ISAs: Multiflow, Cydrome, IA-64 (EPIC)
- PowerPC, POWER
- RISC ISAs: Alpha, MIPS, SPARC, ARM, RISC-V, ...

What are the fundamental differences?

- E.g., how instructions are specified and what they do
- E.g., how complex are instructions, data types, addr. modes
Complex vs. Simple Instructions + Data Types

- **Complex instruction**: An instruction does a lot of work, e.g. many operations
  - Insert in a doubly linked list
  - Compute FFT
  - String copy
  - Matrix multiply
  - ...

- **Simple instruction**: An instruction does little work -- it is a primitive using which complex operations can be built
  - Add
  - XOR
  - Multiply
  - ...
Complex vs. Simple Instructions + Data Types

- **Advantages of Complex Instructions + Data Types**
  
  + Denser encoding $\rightarrow$ smaller code size $\rightarrow$ better memory utilization, saves off-chip bandwidth, better cache hit rate (better packing of instructions)
  
  + Simpler compiler: no need to optimize small instructions as much

- **Disadvantages of Complex Instructions + Data Types**
  
  - Larger chunks of work $\rightarrow$ compiler has less opportunity to optimize (limited in fine-grained optimizations it can do)
  
  - More complex hardware $\rightarrow$ translation from a high level to control signals and optimization needs to be done by hardware
Semantic Gap

- How close instructions & data types & addressing modes are to high-level language (HLL)

<table>
<thead>
<tr>
<th>ISA with</th>
<th>HLL</th>
<th>HW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complex Inst &amp; Data Types &amp; Addressing Modes</td>
<td>Small Semantic Gap</td>
<td>Control Signals</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ISA with</th>
<th>HLL</th>
<th>HW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Inst &amp; Data Types &amp; Addressing Modes</td>
<td>Large Semantic Gap</td>
<td>Control Signals</td>
</tr>
</tbody>
</table>

- Easier mapping of HLL to ISA
  - Less work for software designer
  - More work for hardware designer
  - Optimization burden on HW

- Harder mapping of HLL to ISA
  - More work for software designer
  - Less work for hardware designer
  - Optimization burden on SW
How to Change the Semantic Gap Tradeoffs

- Translate from one ISA into a different “implementation” ISA

![Diagram showing the relationship between HLL, ISA, and Implementation ISA for X86-64 and ARM v8.4.]

- X86-64: ISA with Complex Inst & Data Types & Addressing Modes
- ARM v8.4: Implementation ISA with Simple Inst & Data Types & Addressing Modes

Small Semantic Gap
Software or Hardware Translator
An Example: Rosetta 2 Binary Translator

Rosetta 2  [ edit ]

In 2020, Apple announced Rosetta 2 would be bundled with macOS Big Sur, to aid in the Mac transition to Apple silicon. The software permits many applications compiled exclusively for execution on x86-64-based processors to be translated for execution on Apple silicon.\[2\]\[8\]

In addition to the just-in-time (JIT) translation support, Rosetta 2 offers ahead-of-time compilation (AOT), with the x86-64 code fully translated, just once, when an application without a universal binary is installed on an Apple silicon Mac.\[9\]

Rosetta 2's performance has been praised greatly.\[10\]\[11\] In some benchmarks, x86-64-only programs performed better under Rosetta 2 on a Mac with an Apple M1 SOC than natively on a Mac with an Intel x86-64 processor. One of the key reasons why Rosetta 2 provides such high level of translation efficiency is the support of x86-64 memory ordering in Apple M1 SOC.\[12\]

Although Rosetta 2 works for most software, some software doesn't work at all\[13\] or is reported to be "sluggish".\[14\] A lot of software can be made compatible with the new Macs by the vendor recompiling the software, often a simple task; while for some software (such as software that includes assembly language code, or that generates machine code), the changes to make them work aren't simple and cannot be automated.

Similar to the first version, Rosetta 2 does not normally require user intervention. When a user attempts to launch an x86-64-only application for the first time, macOS prompts them to install Rosetta 2 if it is not already available. Subsequent launches of x86-64 programs will execute via translation automatically. An option also exists to force a universal binary to run as x86-64 code through Rosetta 2, even on an ARM-based machine.\[15\]

https://en.wikipedia.org/wiki/Rosetta_(software)#Rosetta_2
An Example: Rosetta 2 Binary Translator

Source: https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested
Another Example: Intel and AMD Processors

- **HLL**
- **Small Semantic Gap**
- **Hardware Translator**
- **Secret Micro-operations**
- **X86-64**
- **ISA with Complex Inst & Data Types & Addressing Modes**
- **Implementation ISA with Simple Inst & Data Types & Addressing Modes**
- **HW Control Signals**
Another Example: Intel and AMD Processors

Source: https://twitter.com/Locuza_/status/1454152714930331652
Another Example: Intel and AMD Processors

AMD Ryzen 5000, 2020

Core Count: 8 cores/16 threads

L1 Caches: 32 KB per core

L2 Caches: 512 KB per core

L3 Cache: 32 MB shared

Another Example: NVIDIA Denver

ARM ISA
ISA with Complex Inst & Data Types & Addressing Modes

Secret Micro-operations
HW Control Signals

HLL
Small Semantic Gap

Hardware Translator
Implementation ISA with Simple Inst & Data Types & Addressing Modes
Another Example: NVIDIA Denver

The Secret of Denver: Binary Translation & Code Optimization

As we alluded to earlier, NVIDIA's decision to forgo a traditional out-of-order design for Denver means that much of Denver's potential is contained in its software rather than its hardware. The underlying chip itself, though by no means simple, is at its core a very large in-order processor. So it falls to the software stack to make Denver sing.

Accomplishing this task is NVIDIA's dynamic code optimizer (DCO). The purpose of the DCO is to accomplish two tasks: to translate ARM code to Denver's native format, and to optimize this code to make it run better on Denver. With no out-of-order hardware on Denver, it is the DCO's task to find instruction level parallelism within a thread to fill Denver's many execution units, and to reorder instructions around potential stalls, something that is no simple task.

https://www.anandtech.com/show/8701/the-google-nexus-9-review/4
https://www.toradex.com/computer-on-modules/apalis-arm-family/nvidia-tegra-k1
Transmeta: x86 to VLIW Translation

Figure 5. The Code Morphing software mediates between x86 software and the Crusoe processor.


https://www.wikiwand.com/en/Transmeta_Efficeon
ISA-level Tradeoffs: Number of Registers

- **Affects:**
  - Number of bits used for encoding register address
  - Number of values kept in fast storage (register file)
  - \((uarch)\) Size, access time, power consumption of register file

- **Large number of registers:**
  - \(+\) Enables better register allocation (and optimizations) by compiler \(\rightarrow\) fewer saves/restores
    - -- Larger instruction size
    - -- Larger register file size
There Is A Lot More to Cover on ISAs
There Is A Lot More to Cover on ISAs

ISA-level Tradeoffs: Number of Registers

- Affects:
  - Number of bits used for encoding register address
  - Number of values kept in fast storage (register file)
  - (uarch) Size, access time, power consumption of register file

- Large number of registers:
  + Enables better register allocation (and optimizations) by compiler → fewer saves/restores
    -- Larger instruction size
    -- Larger register file size

https://www.youtube.com/onurmutlulectures
Detailed Lectures on ISAs & ISA Tradeoffs

- **Computer Architecture, Spring 2015, Lecture 3**
  - ISA Tradeoffs (CMU, Spring 2015)
  - [https://www.youtube.com/watch?v=QKdiZSfwg-q&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=3](https://www.youtube.com/watch?v=QKdiZSfwg-q&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=3)

- **Computer Architecture, Spring 2015, Lecture 4**
  - ISA Tradeoffs & MIPS ISA (CMU, Spring 2015)
  - [https://www.youtube.com/watch?v=RBgeCCW5Hjs&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=4](https://www.youtube.com/watch?v=RBgeCCW5Hjs&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=4)

- **Computer Architecture, Spring 2015, Lecture 2**
  - Fundamental Concepts and ISA (CMU, Spring 2015)
  - [https://www.youtube.com/watch?v=NpC39uS4K4o&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=2](https://www.youtube.com/watch?v=NpC39uS4K4o&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=2)

[https://www.youtube.com/onurmutuluncles](https://www.youtube.com/onurmutuluncles)
ISA Design and Tradeoffs: More Critical Thinking
The Von Neumann Model/Architecture

Stored program

Sequential instruction processing
The von Neumann Model/Architecture

- Von Neumann model is also called *stored program computer* (instructions in memory). It has two key properties:

  - **Stored program**
    - Instructions stored in a linear memory array
    - Memory is unified between instructions and data
      - The interpretation of a stored value depends on the control signals

  - **Sequential instruction processing**
Whether a value fetched from memory is interpreted as an instruction depends on when that value is fetched in the instruction processing cycle.
The von Neumann Model/Architecture

- Von Neumann model is also called *stored program computer* (instructions in memory). It has two key properties:

- **Stored program**
  - Instructions stored in a linear memory array
  - *Memory is unified* between instructions and data
    - The interpretation of a stored value depends on the control signals

- **Sequential instruction processing**
  - One instruction processed (fetched, executed, completed) at a time
  - *Program counter (instruction pointer)* identifies the current instruction
  - *Program counter is advanced sequentially* except for control transfer instructions

When is a value interpreted as an instruction?
The von Neumann Model/Architecture

- **Recommended reading**
  - Burks, Goldstein, von Neumann, “Preliminary discussion of the logical design of an electronic computing instrument,” 1946.

- **Important reading**
  - Patt and Patel book, Chapter 4, “The von Neumann Model”

- **Stored program**

- **Sequential instruction processing**
The Von Neumann Model (of a Computer)

INPUT
- Keyboard,
- Mouse,
- Disk...

OUTPUT
- Monitor,
- Printer,
- Disk...

MEMORY
- Mem Addr Reg
- Mem Data Reg

PROCESSING UNIT
- ALU
- TEMP

CONTROL UNIT
- PC or IP
- Inst Register
The Von Neumann Model (of a Computer)

- Q: Is this the only way that a computer can process computer programs?
  - A: No.
  - Qualified Answer: No. But, it has been the dominant way
    - i.e., the dominant paradigm for computing
    - for N decades

Let's examine a completely different model for processing computer programs.
The Dataflow Execution Model of a Computer
The Dataflow Model (of a Computer)

- **Von Neumann model:** An instruction is fetched and executed in control flow order
  - As specified by the program counter (instruction pointer)
  - Sequential unless explicit control flow instruction

- **Dataflow model:** An instruction is fetched and executed in data flow order
  - i.e., when its operands are ready
  - i.e., there is no program counter (instruction pointer)
  - Instruction ordering specified by data flow dependence
    - Each instruction specifies “who” should receive the result
    - An instruction can “fire” whenever all operands are received
  - Potentially many instructions can execute at the same time
    - Inherently more parallel
Von Neumann vs. Dataflow

Consider a Von Neumann program

- What is the significance of the program order?
- What is the significance of the storage locations?

```c
v = a + b;
w = b * 2;
x = v - w
y = v + w
z = x * y
```

**Sequential**

- `a, b` are the only inputs
- `z` is the only output

Which model is more natural to you as a programmer?
More on Dataflow

- In a dataflow machine, a program consists of dataflow nodes
  - A dataflow node fires (fetched and executed) when all its inputs are ready
    - i.e. when all inputs have tokens

- Dataflow node and its ISA representation
Example Dataflow Nodes

*Conditional

*Relational

*Barrier Synch
A Simple Example Dataflow Program

What is the value of OUT?

N is a non-negative integer
ISA-level Tradeoff: Program Counter

- **Do we want a Program Counter (PC or IP) in the ISA?**
  - **Yes:** Control-driven, sequential execution
    - An instruction is executed when the PC points to it
    - PC automatically changes sequentially (except for control flow instructions) → **sequential**
  - **No:** Data-driven, parallel execution
    - An instruction is executed when all its operand values are available → **dataflow**

- **Tradeoffs:** MANY high-level ones
  - Ease of programming (for average programmers)?
  - Ease of compilation?
  - Performance: Extraction of parallelism?
  - Hardware complexity?
ISA vs. Microarchitecture Level Tradeoff

- A similar tradeoff (control vs. data-driven execution) can be made at the microarchitecture level

- ISA: Specifies how the programmer sees the instructions to be executed
  - Programmer sees a sequential, control-flow execution order vs.
  - Programmer sees a dataflow execution order

- Microarchitecture: How the underlying implementation actually executes instructions
  - Microarchitecture can execute instructions in any order as long as it obeys the semantics specified by the ISA when making the instruction results visible to software
  - Programmer should see the order specified by the ISA
Let’s Get Back to the von Neumann Model

- But, if you want to learn more about dataflow...


- A later lecture

- If you are really impatient:
  - http://www.youtube.com/watch?v=D2uue7izU2c
The von Neumann Model

- All major *instruction set architectures* today use this model
  - x86, ARM, MIPS, SPARC, Alpha, POWER, RISC-V, ...

- Underneath (at the microarchitecture level), the execution model of almost all *implementations (or, microarchitectures)* is very different
  - Pipelined instruction execution: *Intel 80486 uarch*
  - Multiple instructions at a time: *Intel Pentium uarch*
  - Out-of-order execution: *Intel Pentium Pro uarch*
  - Separate instruction and data caches

- But, what happens underneath that is *not consistent* with the von Neumann model is *not exposed* to software
  - Difference between ISA and microarchitecture
What is Computer Architecture?

- **ISA+implementation definition:** The science and art of designing, selecting, and interconnecting hardware components and designing the hardware/software interface to create a computing system that meets functional, performance, energy consumption, cost, and other specific goals.

- **Traditional (ISA-only) definition:** “The term *architecture* is used here to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior *as distinct from* the organization of the dataflow and controls, the logic design, and the physical implementation.”

  *Gene Amdahl,* IBM Journal of R&D, April 1964
ISA vs. Microarchitecture

- **ISA**
  - Agreed upon interface between software and hardware
    - SW/compiler assumes, HW promises
  - What the software writer needs to know to write and debug system/user programs

- **Microarchitecture**
  - Specific implementation of an ISA
  - Not visible to the software

- **Microprocessor**
  - **ISA, uarch, circuits**
  - “Architecture” = ISA + microarchitecture
Microarchitecture

- A specific **implementation** of the ISA

- How do we implement the ISA?
  - We will discuss this for many lectures

- There can be many implementations of the same ISA
  - **MIPS** R2000, R3000, R4000, R6000, R8000, R10000, ...
  - **x86**: Intel 80486, Pentium, Pentium Pro, Pentium 4, Kaby Lake, Coffee Lake, Comet Lake, Ice Lake, Golden Cove, Sapphire Rapids, ..., AMD K5, K7, K9, Bulldozer, BobCat, Ryzen X, ...
  - **POWER** 4, 5, 6, 7, 8, 9, 10 (IBM), ..., **PowerPC** 604, 605, 620, ...
  - **ARM** Cortex-M*, ARM Cortex-A*, NVIDIA Denver, Apple A*, M1, ...
  - **Alpha** 21064, 21164, 21264, 21364, ...
  - **RISC-V** ...
  - ...
ISA vs. Microarchitecture

- What is part of ISA vs. Uarch?
  - Gas pedal: interface for “acceleration”
  - Internals of the engine: implement “acceleration”

- Implementation (uarch) can be various as long as it satisfies the specification (ISA)
  - Add instruction vs. Adder implementation
    - Bit serial, ripple carry, carry lookahead adders are all part of microarchitecture (see H&H Chapter 5.2.1)
  - x86 ISA has many implementations:
    - Intel 80486, Pentium, Pentium Pro, Pentium 4, Kaby Lake, Coffee Lake, Comet Lake, Ice Lake, Golden Cover, Sapphire Rapids, ..., AMD K5, K7, K9, Bulldozer, BobCat, Ryzen X, ...

- Microarchitecture usually changes faster than ISA
  - Few ISAs (x86, ARM, SPARC, MIPS, Alpha, RISC-V) but many uarchs
  - Why?

https://www.vox.com/2015/7/1/8877583/two-foot-driving-pedal-error
ISA: What Does It Specify?

- Instructions
  - Opcodes, Addressing Modes, Data Types
  - Instruction Types and Formats
  - Registers, Condition Codes

- Memory
  - Address space, Addressability, Alignment
  - Virtual memory management

- Call, Interrupt/Exception Handling

- Access Control, Priority/Privilege

- I/O: memory-mapped vs. instructions

- Task/thread Management

- Power & Thermal Management

- Multithreading & Multiprocessor support

- …
ISAs Keep Getting Extended

Intel® 64 and IA-32 Architectures Software Developer’s Manual

Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4

NOTE: This document contains all four volumes of the Intel 64 and IA-32 Architectures Software Developer’s Manual: Basic Architecture, Order Number 253665; Instruction Set Reference A-Z, Order Number 325383; System Programming Guide, Order Number 325384; Model-Specific Registers, Order Number 335592. Refer to all four volumes when evaluating your design needs.

ISAs Keep Getting Extended

Arm® A64 Instruction Set
for A-profile architecture

https://developer.arm.com/documentation/ddi0602/latest/
# ISA Manuals: Some Good Bedtime Reading

## Combined Volume Set of Intel® 64 and IA-32 Architectures Software Developer’s Manuals

<table>
<thead>
<tr>
<th>Document</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel® 64 and IA-32 Architectures Software Developer’s Manual</td>
<td>This document contains the following:</td>
</tr>
<tr>
<td>Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4</td>
<td><strong>Volume 1</strong>: Describes the architecture and programming environment of processors supporting IA-32 and Intel® 64 architectures.</td>
</tr>
<tr>
<td></td>
<td><strong>Volume 2</strong>: Includes the full instruction set reference, A-Z. Describes the format of the instruction and provides reference pages for instructions.</td>
</tr>
<tr>
<td></td>
<td><strong>Volume 3</strong>: Includes the full system programming guide, parts 1, 2, 3, and 4. Describes the operating-system support environment of Intel® 64 and IA-32 architectures, including: memory management, protection, task management, interrupt and exception handling, multi-processor support, thermal and power management features, debugging, performance monitoring, system management mode, virtual machine extensions (VMX) instructions, Intel® Virtualization Technology (Intel® VT), and Intel® Software Guard Extensions (Intel® SGX). Note: Performance monitoring events can be found here: <a href="https://perfmon-events.intel.com/">https://perfmon-events.intel.com/</a></td>
</tr>
<tr>
<td></td>
<td><strong>Volume 4</strong>: Describes the model-specific registers of processors supporting IA-32 and Intel® 64 architectures.</td>
</tr>
<tr>
<td>Intel® 64 and IA-32 Architectures Software Developer’s Manual Documentation Changes</td>
<td>Describes bug fixes made to the Intel® 64 and IA-32 architectures software developer's manual between versions.</td>
</tr>
<tr>
<td></td>
<td>NOTE: This change document applies to all Intel® 64 and IA-32 architectures software developer's manual sets (combined volume set, 4 volume set, and 10 volume set).</td>
</tr>
</tbody>
</table>

Specifications

The RISC-V instruction set architecture (ISA) and related specifications are developed, ratified and maintained by RISC-V International contributing members within the RISC-V International Technical Working Groups. Work on the specification is performed on GitHub, and the GitHub issue mechanism can be used to provide input into the specification.

If you would like more information on becoming a member, please see the membership page.

ISA Specification
The specifications shown below represent the current, ratified releases. Work is being done on GitHub.

- Volume 1, Unprivileged Spec v. 20191213 [PDF]
- Volume 2, Privileged Spec v. 20211203 [PDF]
- Recently ratified, but not yet integrated, extension specifications

Debug Specification
This is the currently ratified specification:

- External Debug Support v. 0.13.2 [PDF] [GitHub]

This is the current stable draft:

- External Debug Support v. 1.0.0-STABLE [PDF]

Trace Specification
The processor trace specification was approved on March 20, 2020.

- Trace Specification v. 1.0 [PDF] [GitHub]

Compatibility Test Framework
The RISC-V Architectural Compatibility Test Framework Version 2 is now available. This framework compares arbitrary models against a reference signature, and currently covers RV[32|64]MC unprivileged specifications only. Tests for the not-yet-ratified Crypto Scalar extension and RV32EMC extensions are also available.

Work on Version 3.0 framework (RISCOF) is

https://riscv.org/technical/specifications/
Lecture 9c: Assembly Programming

A Program for Adding Integers in LC-3

We use conditional branch instructions to create a loop

<table>
<thead>
<tr>
<th>Address</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3000</td>
<td>LEA</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x000F</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x3001</td>
<td>AND</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x3002</td>
<td>AND</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x3003</td>
<td>ADD</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x3004</td>
<td>BR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x3005</td>
<td>LDR</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x3006</td>
<td>ADD</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x3007</td>
<td>ADD</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x3008</td>
<td>ADD</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x3009</td>
<td>BR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>p</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

R1 = PC + 0x000F = 3100 // load address
R3 = 0 // reset register
R2 = 0 // reset register
R2 = R2 + 12 // initialize counter
BRz (PC + 1 + 5) = BRz 0x300A // check condition
R4 = M[R1 + 0] // load value
R3 = R3 + R4 // accumulate
R1 = R1 + 1 // increment address
R2 = R2 - 1 // decrement counter
BRnzp (PC - 6) = BRnzp 0x3004 // jump

Digital Design & Computer Architecture - Lecture 10b: Assembly Programming (Spring 2022)

2K views 11 months ago  Livestream - Digital Design and Computer Architecture - ETH Zürich (Spring 2022)
Digital Design and Computer Architecture, ETH Zürich, Spring 2022 (https://safari.ethz.ch/digitaltechnik..)

https://www.youtube.com/watch?v=oYaxD1j6MOU
Lecture 9c: Assembly Programming

Function Calls: Code Example

High-level code

```c
int main()
{
    int y;
    ...
    // 4 arguments
    y = diffofsums(2, 3, 4, 5);
    ...
}
int diffofsums(int f, int g, int h, int i)
{
    int result;
    result = (f + g) - (h + i);
    // return value
    return result;
}
```

MIPS assembly

```mips
# $s0 = y
main:
    ...
    addi $a0, $0, 2      # argument 0 = 2
    addi $a1, $0, 3      # argument 1 = 3
    addi $a2, $0, 4      # argument 2 = 4
    addi $a3, $0, 5      # argument 3 = 5
    jal diffofsums       # call procedure
    add $s0, $v0, $0     # y = returned value
    ...
# $s0 = result
diffofsums:
    add $t0, $a0, $a1    # $t0 = f + g
    add $t1, $a2, $a3    # $t1 = h + i
    sub $s0, $t0, $t1    # result=(f + g) - (h + i)
    add $v0, $s0, $0     # put return value in $v0
    jr $ra               # return to caller
```

Return address

Argument values

Return value

Digital Design & Computer Architecture - Lecture 10b: Assembly Programming (Spring 2022)

2,069 views  Premiered Mar 28, 2022  Livestream - Digital Design and Computer Architecture - ETH Zürich (Spring 2022)
Digital Design and Computer Architecture, ETH Zürich, Spring 2022 (https://safari.ethz.ch/digitaltechnik...)

https://www.youtube.com/watch?v=oYaxD1j6MOU