CENTRAL PROCESSING UNIT

- Introduction
- General Register Organization
- Stack Organization
- Instruction Formats
- Addressing Modes
- Data Transfer and Manipulation
- Program Control
- Reduced Instruction Set Computer
MAJOR COMPONENTS OF CPU

- **Storage Components**
  - Registers
  - Flags

- **Execution (Processing) Components**
  - Arithmetic Logic Unit (ALU)
    - Arithmetic calculations, Logical computations, Shifts/Rotates

- **Transfer Components**
  - Bus

- **Control Components**
  - Control Unit
In Basic Computer, there is only one general purpose register, the Accumulator (AC).

In modern CPUs, there are many general purpose registers.

It is advantageous to have many registers:
- Transfer between registers within the processor are relatively fast.
- Going “off the processor” to access memory is much slower.

How many registers will be the best?
General Register Organization

General Register Organization

Input

Clock

Load (7 lines)

SEL A

MUX

MUX

3 x 8 decoder

SELD

ALU

Output

Output

A bus

B bus

OPR

Input
The control unit
Directs the information flow through ALU by
- Selecting various Components in the system
- Selecting the Function of ALU

Example: R1 ← R2 + R3
  [1] MUX A selector (SELA): BUS A ← R2
  [3] ALU operation selector (OPR): ALU to ADD

Control Word

Encoding of register selection fields

<table>
<thead>
<tr>
<th>Binary Code</th>
<th>SELA</th>
<th>SELB</th>
<th>SELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Input</td>
<td>Input</td>
<td>None</td>
</tr>
<tr>
<td>001</td>
<td>R1</td>
<td>R1</td>
<td>R1</td>
</tr>
<tr>
<td>010</td>
<td>R2</td>
<td>R2</td>
<td>R2</td>
</tr>
<tr>
<td>011</td>
<td>R3</td>
<td>R3</td>
<td>R3</td>
</tr>
<tr>
<td>100</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
</tr>
<tr>
<td>101</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
</tr>
<tr>
<td>110</td>
<td>R6</td>
<td>R6</td>
<td>R6</td>
</tr>
<tr>
<td>111</td>
<td>R7</td>
<td>R7</td>
<td>R7</td>
</tr>
</tbody>
</table>
ALU CONTROL

Encoding of ALU operations

<table>
<thead>
<tr>
<th>OPR</th>
<th>Select</th>
<th>Operation</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>00000</td>
<td>Transfer A</td>
<td>TSFA</td>
</tr>
<tr>
<td>00001</td>
<td>00000</td>
<td>Increment A</td>
<td>INCA</td>
</tr>
<tr>
<td>00010</td>
<td>00000</td>
<td>ADD A + B</td>
<td>ADD</td>
</tr>
<tr>
<td>00101</td>
<td>00000</td>
<td>Subtract A - B</td>
<td>SUB</td>
</tr>
<tr>
<td>00110</td>
<td>00000</td>
<td>Decrement A</td>
<td>DECA</td>
</tr>
<tr>
<td>01000</td>
<td>00000</td>
<td>AND A and B</td>
<td>AND</td>
</tr>
<tr>
<td>01010</td>
<td>00000</td>
<td>OR A and B</td>
<td>OR</td>
</tr>
<tr>
<td>01100</td>
<td>00000</td>
<td>XOR A and B</td>
<td>XOR</td>
</tr>
<tr>
<td>01110</td>
<td>00000</td>
<td>Complement A</td>
<td>COMA</td>
</tr>
<tr>
<td>10000</td>
<td>00000</td>
<td>Shift right A</td>
<td>SHRA</td>
</tr>
<tr>
<td>11000</td>
<td>00000</td>
<td>Shift left A</td>
<td>SHLA</td>
</tr>
</tbody>
</table>

Examples of ALU Microoperations

<table>
<thead>
<tr>
<th>Microoperation</th>
<th>Symbolic Designation</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1 ← R2 – R3</td>
<td>SELA R2 SELB R3 SELD R1 OPR SUB Control Word 010 011 001 00101</td>
</tr>
<tr>
<td>R4 ← R4 ∨ R5</td>
<td>SELA R4 SELB R5 SELD R4 OPR OR Control Word 100 101 100 01010</td>
</tr>
<tr>
<td>R6 ← R6 + 1</td>
<td>SELA R6 SELD R6 SELD R6 INCA Control Word 110 000 110 00001</td>
</tr>
<tr>
<td>R7 ← R1</td>
<td>SELA R1 SELD R1 SELD R7 TSFA Control Word 001 000 111 00000</td>
</tr>
<tr>
<td>Output ← R2</td>
<td>SELA R2 SELD None SELD TSFA Control Word 010 000 000 00000</td>
</tr>
<tr>
<td>Output ← Input</td>
<td>SELA Input SELD None SELD TSFA Control Word 000 000 000 00000</td>
</tr>
<tr>
<td>R4 ← shl R4</td>
<td>SELA R4 SELD R4 SELD SHLA Control Word 100 000 100 11000</td>
</tr>
<tr>
<td>R5 ← 0</td>
<td>SELA R5 SELD R5 SELD XOR Control Word 101 101 101 01100</td>
</tr>
</tbody>
</table>
REGISTER STACK ORGANIZATION

Stack
- Very useful feature for nested subroutines, nested interrupt services
- Also efficient for arithmetic expression evaluation
- Storage which can be accessed in LIFO
- Pointer: SP
- Only PUSH and POP operations are applicable

Register Stack

Push, Pop operations

/* Initially, SP = 0, EMPTY = 1, FULL = 0 */

**PUSH**
- \( SP \leftarrow SP + 1 \)
- \( M[SP] \leftarrow DR \)
- If \((SP = 0)\) then \((FULL \leftarrow 1)\)
- \( EMPTY \leftarrow 0 \)

**POP**
- \( DR \leftarrow M[SP] \)
- \( SP \leftarrow SP - 1 \)
- If \((SP = 0)\) then \((EMPTY \leftarrow 1)\)
- \( FULL \leftarrow 0 \)
- A portion of memory is used as a stack with a processor register as a stack pointer

- PUSH:  \( SP \leftarrow SP - 1 \)
  \( M[SP] \leftarrow DR \)

- POP:  \( DR \leftarrow M[SP] \)
  \( SP \leftarrow SP + 1 \)

- Most computers do not provide hardware to check stack overflow (full stack) or underflow (empty stack) → must be done in software
REVERSE POLISH NOTATION

• Arithmetic Expressions: A + B
  A + B  Infix notation
  + A B  Prefix or Polish notation
  A B +  Postfix or reverse Polish notation

  - The reverse Polish notation is very suitable for stack manipulation

• Evaluation of Arithmetic Expressions
  Any arithmetic expression can be expressed in parenthesis-free Polish notation, including reverse Polish notation

(3 * 4) + (5 * 6)  ⇒  3 4 * 5 6 * +
• In general, most processors are organized in one of 3 ways

  – Single register (Accumulator) organization
    » Basic Computer is a good example
    » Accumulator is the only general purpose register

  – General register organization
    » Used by most modern computer processors
    » Any of the registers can be used as the source or destination for computer operations

  – Stack organization
    » All operations are done using the hardware stack
    » For example, an OR instruction will pop the two top elements from the stack, do a logical OR on them, and push the result on the stack
INSTRUCTION FORMAT

• Instruction Fields
  OP-code field - specifies the operation to be performed
  Address field - designates memory address(es) or a processor register(s)
  Mode field - determines how the address field is to be interpreted (to
                get effective address or the operand)

• The number of address fields in the instruction format
  depends on the internal organization of CPU

• The three most common CPU organizations:
  Single accumulator organization:
    ADD  X                /* AC ← AC + M[X] */
  General register organization:
    ADD  R1, R2, R3       /* R1 ← R2 + R3 */
    ADD  R1, R2           /* R1 ← R1 + R2 */
    MOV  R1, R2           /* R1 ← R2 */
    ADD  R1, X            /* R1 ← R1 + M[X] */
  Stack organization:
    PUSH X                /* TOS ← M[X] */
    ADD
THREE, AND TWO-ADDRESS INSTRUCTIONS

• Three-Address Instructions

Program to evaluate \( X = (A + B) \times (C + D) \):

- ADD \( R1, A, B \) /* \( R1 \leftarrow M[A] + M[B] \) */
- ADD \( R2, C, D \) /* \( R2 \leftarrow M[C] + M[D] \) */
- MUL \( X, R1, R2 \) /* \( M[X] \leftarrow R1 \times R2 \) */

- Results in short programs
- Instruction becomes long (many bits)

• Two-Address Instructions

Program to evaluate \( X = (A + B) \times (C + D) \):

- MOV \( R1, A \) /* \( R1 \leftarrow M[A] \) */
- ADD \( R1, B \) /* \( R1 \leftarrow R1 + M[A] \) */
- MOV \( R2, C \) /* \( R2 \leftarrow M[C] \) */
- ADD \( R2, D \) /* \( R2 \leftarrow R2 + M[D] \) */
- MUL \( R1, R2 \) /* \( R1 \leftarrow R1 \times R2 \) */
- MOV \( X, R1 \) /* \( M[X] \leftarrow R1 \) */
ONE, AND ZERO-ADDRESS INSTRUCTIONS

• One-Address Instructions
  - Use an implied AC register for all data manipulation
  - Program to evaluate $X = (A + B) \times (C + D)$:

    LOAD A /* AC ← M[A] */
    ADD B /* AC ← AC + M[B] */
    STORE T /* M[T] ← AC */
    LOAD C /* AC ← M[C] */
    ADD D /* AC ← AC + M[D] */
    MUL T /* AC ← AC \times M[T] */
    STORE X /* M[X] ← AC */

• Zero-Address Instructions
  - Can be found in a stack-organized computer
  - Program to evaluate $X = (A + B) \times (C + D)$:

    PUSH A /* TOS ← A */
    PUSH B /* TOS ← B */
    ADD /* TOS ← (A + B) */
    PUSH C /* TOS ← C */
    PUSH D /* TOS ← D */
    ADD /* TOS ← (C + D) */
    MUL /* TOS ← (C + D) \times (A + B) */
    POP X /* M[X] ← TOS */
Addressing Modes

• Addressing Modes

* Specifies a rule for interpreting or modifying the address field of the instruction (before the operand is actually referenced)

* Variety of addressing modes

  - to give programming flexibility to the user
  - to use the bits in the address field of the instruction efficiently
TYPES OF ADDRESSING MODES

• Implied Mode
  Address of the operands are specified implicitly in the definition of the instruction
  - No need to specify address in the instruction
  - EA = AC, or EA = Stack[SP]
  - Examples from Basic Computer
    CLA, CME, INP

• Immediate Mode
  Instead of specifying the address of the operand, operand itself is specified
  - No need to specify address in the instruction
  - However, operand itself needs to be specified
  - Sometimes, require more bits than the address
  - Fast to acquire an operand
TYPES OF ADDRESSING MODES

• Register Mode
  Address specified in the instruction is the register address
  - Designated operand need to be in a register
  - Shorter address than the memory address
  - Saving address field in the instruction
  - Faster to acquire an operand than the memory addressing
  - EA = IR(R) (IR(R): Register field of IR)

• Register Indirect Mode
  Instruction specifies a register which contains the memory address of the operand
  - Saving instruction bits since register address is shorter than the memory address
  - Slower to acquire an operand than both the register addressing or memory addressing
  - EA = [IR(R)] ([x]: Content of x)

• Autoincrement or Autodecrement Mode
  - When the address in the register is used to access memory, the value in the register is incremented or decremented by 1 automatically
TYPES OF ADDRESSING MODES

• Direct Address Mode
  Instruction specifies the memory address which can be used directly to access the memory
  - Faster than the other memory addressing modes
  - Too many bits are needed to specify the address for a large physical memory space
  - $EA = IR(addr)$ ($IR(addr)$: address field of IR)

• Indirect Addressing Mode
  The address field of an instruction specifies the address of a memory location that contains the address of the operand
  - When the abbreviated address is used large physical memory can be addressed with a relatively small number of bits
  - Slow to acquire an operand because of an additional memory access
  - $EA = M[IR(address)]$
TYPES OF ADDRESSING MODES

• Relative Addressing Modes
  The Address fields of an instruction specifies the part of the address (abbreviated address) which can be used along with a designated register to calculate the address of the operand
  - Address field of the instruction is short
  - Large physical memory can be accessed with a small number of address bits
  - \( EA = f(IR(\text{address}), R) \), \( R \) is sometimes implied

3 different Relative Addressing Modes depending on \( R \);
  * **PC Relative Addressing Mode** \( (R = PC) \)
    - \( EA = PC + IR(\text{address}) \)
  * **Indexed Addressing Mode** \( (R = IX, \text{ where } IX: \text{ Index Register}) \)
    - \( EA = IX + IR(\text{address}) \)
  * **Base Register Addressing Mode**
    - \( (R = BAR, \text{ where } BAR: \text{ Base Address Register}) \)
    - \( EA = BAR + IR(\text{address}) \)
**ADDRESSING MODES - EXAMPLES**

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Effective Address</th>
<th>Content of AC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct address</td>
<td>500</td>
<td>/* AC ← (500) */ 800</td>
</tr>
<tr>
<td>Immediate operand</td>
<td>-</td>
<td>/* AC ← 500 */ 500</td>
</tr>
<tr>
<td>Indirect address</td>
<td>800</td>
<td>/* AC ← (500) */ 300</td>
</tr>
<tr>
<td>Relative address</td>
<td>702</td>
<td>/* AC ← (PC+500) */ 325</td>
</tr>
<tr>
<td>Indexed address</td>
<td>600</td>
<td>/* AC ← (RX+500) */ 900</td>
</tr>
<tr>
<td>Register</td>
<td>-</td>
<td>/* AC ← R1 */ 400</td>
</tr>
<tr>
<td>Register indirect</td>
<td>400</td>
<td>/* AC ← (R1) */ 700</td>
</tr>
<tr>
<td>Autoincrement</td>
<td>400</td>
<td>/* AC ← (R1)+ */ 700</td>
</tr>
<tr>
<td>Autodecrement</td>
<td>399</td>
<td>/* AC ← -(R) */ 450</td>
</tr>
</tbody>
</table>

**Addressing**

- **PC = 200**
- **R1 = 400**
- **XR = 100**

**Memory**

- **Address 200**: Load to AC Mode
- **Address 201**: Address = 500
- **Address 202**: Next instruction
- **Address 399**: 450
- **Address 400**: 700
- **Address 500**: 800
- **Address 600**: 900
- **Address 702**: 325
- **Address 800**: 300
DATA TRANSFER INSTRUCTIONS

• Typical Data Transfer Instructions

<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Assembly Convention</th>
<th>Register Transfer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>LD</td>
<td></td>
<td>AC ← M[ADR]</td>
</tr>
<tr>
<td>Store</td>
<td>ST</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move</td>
<td>MOV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exchange</td>
<td>XCH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input</td>
<td>IN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output</td>
<td>OUT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Push</td>
<td>PUSH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pop</td>
<td>POP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Data Transfer Instructions with Different Addressing Modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Assembly Convention</th>
<th>Register Transfer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct address</td>
<td>LD ADR</td>
<td>AC ← M[ADR]</td>
</tr>
<tr>
<td>Indirect address</td>
<td>LD @ADR</td>
<td>AC ← M[M[ADR]]</td>
</tr>
<tr>
<td>Relative address</td>
<td>LD $ADR</td>
<td>AC ← M[PC + ADR]</td>
</tr>
<tr>
<td>Immediate operand</td>
<td>LD #NBR</td>
<td>AC ← NBR</td>
</tr>
<tr>
<td>Index addressing</td>
<td>LD ADR(X)</td>
<td>AC ← M[ADR + XR]</td>
</tr>
<tr>
<td>Register</td>
<td>LD R1</td>
<td>AC ← R1</td>
</tr>
<tr>
<td>Register indirect</td>
<td>LD (R1)</td>
<td>AC ← M[R1]</td>
</tr>
<tr>
<td>Autoincrement</td>
<td>LD (R1)+</td>
<td>AC ← M[R1], R1 ← R1 + 1</td>
</tr>
<tr>
<td>Autodecrement</td>
<td>LD -(R1)</td>
<td>R1 ← R1 - 1, AC ← M[R1]</td>
</tr>
</tbody>
</table>
DATA MANIPULATION INSTRUCTIONS

• Three Basic Types: Arithmetic instructions
  Logical and bit manipulation instructions
  Shift instructions

• Arithmetic Instructions

<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increment</td>
<td>INC</td>
</tr>
<tr>
<td>Decrement</td>
<td>DEC</td>
</tr>
<tr>
<td>Add</td>
<td>ADD</td>
</tr>
<tr>
<td>Subtract</td>
<td>SUB</td>
</tr>
<tr>
<td>Multiply</td>
<td>MUL</td>
</tr>
<tr>
<td>Divide</td>
<td>DIV</td>
</tr>
<tr>
<td>Add with Carry</td>
<td>ADDC</td>
</tr>
<tr>
<td>Subtract with Borrow</td>
<td>SUBB</td>
</tr>
<tr>
<td>Negate(2’s Complement)</td>
<td>NEG</td>
</tr>
</tbody>
</table>

• Logical and Bit Manipulation Instructions

• Shift Instructions

<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical shift right</td>
<td>SHR</td>
</tr>
<tr>
<td>Logical shift left</td>
<td>SHL</td>
</tr>
<tr>
<td>Arithmetic shift right</td>
<td>SHRA</td>
</tr>
<tr>
<td>Arithmetic shift left</td>
<td>SHLA</td>
</tr>
<tr>
<td>Rotate right</td>
<td>ROR</td>
</tr>
<tr>
<td>Rotate left</td>
<td>ROL</td>
</tr>
<tr>
<td>Rotate right thru carry</td>
<td>RORC</td>
</tr>
<tr>
<td>Rotate left thru carry</td>
<td>ROLC</td>
</tr>
</tbody>
</table>
FLAG, PROCESSOR STATUS WORD

- In Basic Computer, the processor had several (status) flags – 1 bit value that indicated various information about the processor’s state – E, FGI, FGO, I, IEN, R
- In some processors, flags like these are often combined into a register – the processor status register (PSR); sometimes called a processor status word (PSW)
- Common flags in PSW are
  - C (Carry): Set to 1 if the carry out of the ALU is 1
  - S (Sign): The MSB bit of the ALU’s output
  - Z (Zero): Set to 1 if the ALU’s output is all 0’s
  - V (Overflow): Set to 1 if there is an overflow
Program Control

In-Line Sequencing (Next instruction is fetched from the next adjacent location in the memory)

Address from other source; Current Instruction, Stack, etc; Branch, Conditional Branch, Subroutine, etc

• Program Control Instructions

<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch</td>
<td>BR</td>
</tr>
<tr>
<td>Jump</td>
<td>JMP</td>
</tr>
<tr>
<td>Skip</td>
<td>SKP</td>
</tr>
<tr>
<td>Call</td>
<td>CALL</td>
</tr>
<tr>
<td>Return</td>
<td>RTN</td>
</tr>
<tr>
<td>Compare(by – )</td>
<td>CMP</td>
</tr>
<tr>
<td>Test(by AND)</td>
<td>TST</td>
</tr>
</tbody>
</table>

* CMP and TST instructions do not retain their results of operations (– and AND, respectively). They only set or clear certain Flags.
## CONDITIONAL BRANCH INSTRUCTIONS

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Branch condition</th>
<th>Tested condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>BZ</td>
<td>Branch if zero</td>
<td>Z = 1</td>
</tr>
<tr>
<td>BNZ</td>
<td>Branch if not zero</td>
<td>Z = 0</td>
</tr>
<tr>
<td>BC</td>
<td>Branch if carry</td>
<td>C = 1</td>
</tr>
<tr>
<td>BNC</td>
<td>Branch if no carry</td>
<td>C = 0</td>
</tr>
<tr>
<td>BP</td>
<td>Branch if plus</td>
<td>S = 0</td>
</tr>
<tr>
<td>BM</td>
<td>Branch if minus</td>
<td>S = 1</td>
</tr>
<tr>
<td>BV</td>
<td>Branch if overflow</td>
<td>V = 1</td>
</tr>
<tr>
<td>BNV</td>
<td>Branch if no overflow</td>
<td>V = 0</td>
</tr>
</tbody>
</table>

### Unsigned compare conditions (A - B)
- BHI: Branch if higher  
  \[ A > B \]
- BHE: Branch if higher or equal  
  \[ A \geq B \]
- BLO: Branch if lower  
  \[ A < B \]
- BLOE: Branch if lower or equal  
  \[ A \leq B \]
- BE: Branch if equal  
  \[ A = B \]
- BNE: Branch if not equal  
  \[ A \neq B \]

### Signed compare conditions (A - B)
- BGT: Branch if greater than  
  \[ A > B \]
- BGE: Branch if greater or equal  
  \[ A \geq B \]
- BLT: Branch if less than  
  \[ A < B \]
- BLE: Branch if less or equal  
  \[ A \leq B \]
- BE: Branch if equal  
  \[ A = B \]
- BNE: Branch if not equal  
  \[ A \neq B \]
**SUBROUTINE CALL AND RETURN**

- **Subroutine Call**
  - Call subroutine
  - Jump to subroutine
  - Branch to subroutine
  - Branch and save return address

- **Two Most Important Operations are Implied;**
  - *Branch to the beginning of the Subroutine*
    - Same as the Branch or Conditional Branch
  - *Save the Return Address to get the address of the location in the Calling Program upon exit from the Subroutine

- **Locations for storing Return Address**
  - Fixed Location in the subroutine (Memory)
  - Fixed Location in memory
  - In a processor Register
  - In memory *stack*
    - most efficient way

**CALL**

- \( SP \leftarrow SP - 1 \)
- \( M[SP] \leftarrow PC \)
- \( PC \leftarrow EA \)

**RTN**

- \( PC \leftarrow M[SP] \)
- \( SP \leftarrow SP + 1 \)
Types of Interrupts

External interrupts
- External Interrupts initiated from the outside of CPU and Memory
  - I/O Device → Data transfer request or Data transfer complete
  - Timing Device → Timeout
  - Power Failure
  - Operator

Internal interrupts (traps)
- Internal Interrupts are caused by the currently running program
  - Register, Stack Overflow
  - Divide by zero
  - OP-code Violation
  - Protection Violation

Software Interrupts
- Both External and Internal Interrupts are initiated by the computer HW.
- Software Interrupts are initiated by the executing an instruction.
  - Supervisor Call → Switching from a user mode to the supervisor mode
    → Allows to execute a certain class of operations
      which are not allowed in the user mode
Interrupt Procedure and Subroutine Call

- The interrupt is usually initiated by an internal or an external signal rather than from the execution of an instruction (except for the software interrupt)

- The address of the interrupt service program is determined by the hardware rather than from the address field of an instruction

- An interrupt procedure usually stores all the information necessary to define the state of CPU rather than storing only the PC.

  The state of the CPU is determined from;
  - Content of the PC
  - Content of all processor registers
  - Content of status bits
  - Many ways of saving the CPU state depending on the CPU architectures
RISC: Historical Background

IBM System/360, 1964
- The real beginning of modern computer architecture
- Distinction between *Architecture* and *Implementation*
- Architecture: The abstract structure of a computer seen by an assembly-language programmer

- Continuing growth in semiconductor memory and microprogramming
  ⇒ A much richer and complicated instruction sets
  ⇒ CISC(Complex Instruction Set Computer)
ARGUMENTS ADVANCED AT THAT TIME

- Richer instruction sets would simplify compilers
- Richer instruction sets would alleviate the software crisis
  - move as much functions to the hardware as possible
- Richer instruction sets would improve *architecture quality*
• Large microprograms would add little or nothing to the cost of the machine
  ← Rapid growth of memory technology
  ⇒ Large General Purpose Instruction Set

• Microprogram is much faster than the machine instructions
  ← Microprogram memory is much faster than main memory
  ⇒ Moving the software functions into microprogram for the high performance machines

• Execution speed is proportional to the program size
  ← Architectural techniques that led to small program
  ⇒ High performance instruction set

• Number of registers in CPU has limitations
  ← Very costly
  ⇒ Difficult to utilize them efficiently
COMPARISONS OF EXECUTION MODELS

A ← B + C  Data: 32-bit

- **Register-to-register**

<table>
<thead>
<tr>
<th>8</th>
<th>4</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>rB</td>
<td>B</td>
</tr>
<tr>
<td>Load</td>
<td>rC</td>
<td>C</td>
</tr>
<tr>
<td>Add</td>
<td>rA</td>
<td>rB rC</td>
</tr>
<tr>
<td>Store</td>
<td>rA</td>
<td>A</td>
</tr>
</tbody>
</table>

  I = 104b; D = 96b; M = 200b

- **Memory-to-register**

<table>
<thead>
<tr>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>B</td>
</tr>
<tr>
<td>Add</td>
<td>C</td>
</tr>
<tr>
<td>Store</td>
<td>A</td>
</tr>
</tbody>
</table>

  I = 72b; D = 96b; M = 168b

- **Memory-to-memory**

<table>
<thead>
<tr>
<th>8</th>
<th>16</th>
<th>16</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>B</td>
<td>C</td>
<td>A</td>
</tr>
</tbody>
</table>

  I = 56b; D = 96b; M = 152b
### FOUR MODERN ARCHITECTURES IN 70’s

<table>
<thead>
<tr>
<th></th>
<th>IBM 370/168</th>
<th>DEC VAX-11/780</th>
<th>Xerox Dorado</th>
<th>Intel iAPX-432</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Year</strong></td>
<td>1973</td>
<td>1978</td>
<td>1978</td>
<td>1982</td>
</tr>
<tr>
<td><strong># of instrs.</strong></td>
<td>208</td>
<td>303</td>
<td>270</td>
<td>222</td>
</tr>
<tr>
<td><strong>Control mem. size</strong></td>
<td>420 Kb</td>
<td>480 Kb</td>
<td>136 Kb</td>
<td>420 Kb</td>
</tr>
<tr>
<td><strong>Instr. size (bits)</strong></td>
<td>16-48</td>
<td>16-456</td>
<td>8-24</td>
<td>6-321</td>
</tr>
<tr>
<td><strong>Technology</strong></td>
<td>ECL MSI</td>
<td>TTL MSI</td>
<td>ECL MSI</td>
<td>NMOS VLSI</td>
</tr>
<tr>
<td><strong>Execution model</strong></td>
<td>reg-mem</td>
<td>reg-mem</td>
<td>stack</td>
<td>stack</td>
</tr>
<tr>
<td></td>
<td>mem-mem</td>
<td>mem-mem</td>
<td></td>
<td>mem-mem</td>
</tr>
<tr>
<td></td>
<td>reg-reg</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Cache size</strong></td>
<td>64 Kb</td>
<td>64 Kb</td>
<td>64 Kb</td>
<td>64 Kb</td>
</tr>
</tbody>
</table>
COMPLEX INSTRUCTION SET COMPUTER

• These computers with many instructions and addressing modes came to be known as Complex Instruction Set Computers (CISC)

• One goal for CISC machines was to have a machine language instruction to match each high-level language statement type
VARIABLE LENGTH INSTRUCTIONS

• The large number of instructions and addressing modes led CISC machines to have variable length instruction formats
• The large number of instructions means a greater number of bits to specify them
• In order to manage this large number of opcodes efficiently, they were encoded with different lengths:
  – More frequently used instructions were encoded using short opcodes.
  – Less frequently used ones were assigned longer opcodes.
• Also, multiple operand instructions could specify different addressing modes for each operand
  – For example,
    » Operand 1 could be a directly addressed register,
    » Operand 2 could be an indirectly addressed memory location,
    » Operand 3 (the destination) could be an indirectly addressed register.
• All of this led to the need to have different length instructions in different situations, depending on the opcode and operands used
VARIABLE LENGTH INSTRUCTIONS

• For example, an instruction that only specifies register operands may only be two bytes in length
  – One byte to specify the instruction and addressing mode
  – One byte to specify the source and destination registers.

• An instruction that specifies memory addresses for operands may need five bytes
  – One byte to specify the instruction and addressing mode
  – Two bytes to specify each memory address
» Maybe more if there’s a large amount of memory.

• Variable length instructions greatly complicate the fetch and decode problem for a processor

• The circuitry to recognize the various instructions and to properly fetch the required number of bytes for operands is very complex
Another characteristic of CISC computers is that they have instructions that act directly on memory addresses.

For example,

\[ \text{ADD } L1, L2, L3 \]

that takes the contents of \( M[L1] \) adds it to the contents of \( M[L2] \) and stores the result in location \( M[L3] \).

An instruction like this takes three memory access cycles to execute.

That makes for a potentially very long instruction execution cycle.

The problems with CISC computers are

- The complexity of the design may slow down the processor,
- The complexity of the design may result in costly errors in the processor design and implementation,
- Many of the instructions and addressing modes are used rarely, if ever.
SUMMARY: CRITICISMS ON CISC

High Performance General Purpose Instructions

- Complex Instruction
  → Format, Length, Addressing Modes
  → Complicated instruction cycle control due to the complex decoding HW and decoding process

- Multiple memory cycle instructions
  → Operations on memory data
  → Multiple memory accesses/instruction

- Microprogrammed control is necessity
  → Microprogram control storage takes substantial portion of CPU chip area
  → Semantic Gap is large between machine instruction and microinstruction

- General purpose instruction set includes all the features required by individually different applications
  → When any one application is running, all the features required by the other applications are extra burden to the application
In the late ‘70s and early ‘80s there was a reaction to the shortcomings of the CISC style of processors. Reduced Instruction Set Computers (RISC) were proposed as an alternative. The underlying idea behind RISC processors is to simplify the instruction set and reduce instruction execution time.

RISC processors often feature:

- Few instructions
- Few addressing modes
- Only load and store instructions access memory
- All other operations are done using on-processor registers
- Fixed length instructions
- Single cycle execution of instructions
- The control unit is hardwired, not microprogrammed
Since all but the load and store instructions use only registers for operands, only a few addressing modes are needed.

By having all instructions the same length, reading them in is easy and fast.

The fetch and decode stages are simple, looking much more like Mano’s Basic Computer than a CISC machine.

The instruction and address formats are designed to be easy to decode.

Unlike the variable length CISC instructions, the opcode and register fields of RISC instructions can be decoded simultaneously.

The control logic of a RISC processor is designed to be simple and fast.

The control logic is simple because of the small number of instructions and the simple addressing modes.

The control logic is hardwired, rather than microprogrammed, because hardwired control is faster.
ARCHITECTURAL METRIC

A ← B + C
B ← A + C
D ← D - B

• Register-to-register (Reuse of operands)

<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>4</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>rB</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>Load</td>
<td>rC</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>rA</td>
<td>rB</td>
<td>rC</td>
</tr>
<tr>
<td>Store</td>
<td>rA</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>rB</td>
<td>rA</td>
<td>rC</td>
</tr>
<tr>
<td>Store</td>
<td>rB</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>Load</td>
<td>rD</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>Sub</td>
<td>rD</td>
<td>rD</td>
<td>rB</td>
</tr>
<tr>
<td>Store</td>
<td>rD</td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

I = 228b
D = 192b
M = 420b

• Register-to-register (Compiler allocates operands in registers)

<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>4</th>
<th>4</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>rA</td>
<td>rB</td>
<td>rC</td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>rB</td>
<td>rA</td>
<td>rC</td>
<td></td>
</tr>
<tr>
<td>Sub</td>
<td>rD</td>
<td>rD</td>
<td>rB</td>
<td></td>
</tr>
</tbody>
</table>

I = 60b
D = 0b
M = 60b

• Memory-to-memory

<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>16</th>
<th>16</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>B</td>
<td>C</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>A</td>
<td>C</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>Sub</td>
<td>B</td>
<td>D</td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

I = 168b
D = 288b
M = 456b
# CHARACTERISTICS OF INITIAL RISC MACHINES

<table>
<thead>
<tr>
<th></th>
<th>IBM 801</th>
<th>RISC I</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Year</strong></td>
<td>1980</td>
<td>1982</td>
<td>1983</td>
</tr>
<tr>
<td><strong>Number of instructions</strong></td>
<td>120</td>
<td>39</td>
<td>55</td>
</tr>
<tr>
<td><strong>Control memory size</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Instruction size (bits)</strong></td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td><strong>Technology</strong></td>
<td>ECL MSI reg-reg</td>
<td>NMOS VLSI reg-reg</td>
<td>NMOS VLSI reg-reg</td>
</tr>
<tr>
<td><strong>Execution model</strong></td>
<td>reg-reg</td>
<td>reg-reg</td>
<td>reg-reg</td>
</tr>
</tbody>
</table>
COMPARISON OF INSTRUCTION SEQUENCE

RISC 1

\[ A \leftarrow B + C \]
\[ A \leftarrow A + 1 \]
\[ D \leftarrow D - B \]

<table>
<thead>
<tr>
<th>OP</th>
<th>DEST</th>
<th>SOUR1</th>
<th>SOUR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>rA</td>
<td>rB</td>
<td>rC</td>
</tr>
<tr>
<td>ADD</td>
<td>rA</td>
<td>rA</td>
<td>immediate</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>operand 1</td>
</tr>
<tr>
<td>SUB</td>
<td>rD</td>
<td>rD</td>
<td>rB</td>
</tr>
</tbody>
</table>

VAX

<table>
<thead>
<tr>
<th>OP</th>
<th>DEST</th>
<th>SOUR1</th>
<th>SOUR2</th>
<th>SOUR3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>rA</td>
<td>B</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>INC</td>
<td>A</td>
<td>SUB</td>
<td>register operand B</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INC</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

432

<table>
<thead>
<tr>
<th>OP</th>
<th>DEST</th>
<th>SOUR1</th>
<th>SOUR2</th>
<th>SOUR3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>A</td>
<td>A</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>INC</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>INC</td>
<td>A</td>
<td>C</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>SUB</td>
<td>D</td>
<td>SUB</td>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>

3 operands in memory

4 operands in memory
• By simplifying the instructions and addressing modes, there is space available on the chip or board of a RISC CPU for more circuits than with a CISC processor
• This extra capacity is used to
  – Pipeline instruction execution to speed up instruction execution
  – Add a large number of registers to the CPU
• A very important feature of many RISC processors is the ability to execute an instruction each clock cycle.
• This may seem nonsensical, since it takes at least once clock cycle each to fetch, decode and execute an instruction.
• It is however possible, because of a technique known as pipelining.
  – We’ll study this in detail later.
• Pipelining is the use of the processor to work on different phases of multiple instructions in parallel.
PIPELINING

• For instance, at one time, a pipelined processor may be
  – Executing instruction $i_t$
  – Decoding instruction $i_{t+1}$
  – Fetching instruction $i_{t+2}$ from memory

• So, if we’re running three instructions at once, and it takes an average instruction three cycles to run, the CPU is executing an average of an instruction a clock cycle

• As we’ll see when we cover it in depth, there are complications
  – For example, what happens to the pipeline when the processor branches

• However, pipelined execution is an integral part of all modern processors, and plays an important role
REGISTERS

• By having a large number of general purpose registers, a processor can minimize the number of times it needs to access memory to load or store a value

• This results in a significant speed up, since memory accesses are much slower than register accesses

• Register accesses are fast, since they just use the bus on the CPU itself, and any transfer can be done in one clock cycle

• To go off-processor to memory requires using the much slower memory (or system) bus

• It may take many clock cycles to read or write to memory across the memory bus
  – The memory bus hardware is usually slower than the processor
  – There may even be competition for access to the memory bus by other devices in the computer (e.g. disk drives)

• So, for this reason alone, a RISC processor may have an advantage over a comparable CISC processor, since it only needs to access memory
  – for its instructions, and
  – occasionally to load or store a memory value
<Weighted Relative Dynamic Frequency of HLL Operations>

<table>
<thead>
<tr>
<th></th>
<th>Dynamic Occurrence</th>
<th>Machine-Instruction Weighted</th>
<th>Memory Reference Weighted</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pascal</td>
<td>C</td>
<td>Pascal</td>
</tr>
<tr>
<td>ASSIGN</td>
<td>45</td>
<td>38</td>
<td>13</td>
</tr>
<tr>
<td>LOOP</td>
<td>5</td>
<td>3</td>
<td>42</td>
</tr>
<tr>
<td>CALL</td>
<td>15</td>
<td>12</td>
<td>31</td>
</tr>
<tr>
<td>IF</td>
<td>29</td>
<td>43</td>
<td>11</td>
</tr>
<tr>
<td>GOTO</td>
<td>6</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Other</td>
<td>6</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

⇒ The procedure (function) call/return is the most time-consuming operations in typical HLL programs
CALL-RETURN BEHAVIOR

Call-return behavior as a function of nesting depth and time

[Graph showing call-return behavior over time and nesting depth, with a window depth of 5 indicated.]
• Observations
  - Weighted Dynamic Frequency of HLL Operations
    ⇒ Procedure call/return is the most time consuming operations

  - Locality of Procedure Nesting
    ⇒ The depth of procedure activation fluctuates
      within a relatively narrow range

  - A typical procedure employs only a few passed
    parameters and local variables

• Solution
  - Use multiple small sets of registers (windows),
    each assigned to a different procedure

  - A procedure call automatically switches the CPU to use a different
    window of registers, rather than saving registers in memory

  - Windows for adjacent procedures are overlapped
    to allow parameter passing
CIRCULAR OVERLAPPED REGISTER WINDOWS
OVERLAPPED REGISTER WINDOWS
OVERLAPPED REGISTER WINDOWS

• There are three classes of registers:
  – Global Registers
    » Available to all functions
  – Window local registers
    » Variables local to the function
  – Window shared registers
    » Permit data to be shared without actually needing to copy it

• Only one register window is active at a time
  – The active register window is indicated by a pointer

• When a function is called, a new register window is activated
  – This is done by incrementing the pointer

• When a function calls a new function, the high numbered registers of the calling function window are shared with the called function as the low numbered registers in its register window

• This way the caller’s high and the called function’s low registers overlap and can be used to pass parameters and results
• In addition to the overlapped register windows, the processor has some number of registers, $G$, that are global registers
  – This is, all functions can access the global registers.

• The advantage of overlapped register windows is that the processor does not have to push registers on a stack to save values and to pass parameters when there is a function call
  – Conversely, pop the stack on a function return

• This saves
  – Accesses to memory to access the stack.
  – The cost of copying the register contents at all

• And, since function calls and returns are so common, this results in a significant savings relative to a stack-based approach
BERKELEY RISC I

- 32-bit integrated circuit CPU
- 32-bit address, 8-, 16-, 32-bit data
- 32-bit instruction format
- total 31 instructions
- three addressing modes:
  - register; immediate; PC relative addressing
- 138 registers
  - 10 global registers
  - 8 windows of 32 registers each

Berkeley RISC I Instruction Formats

<table>
<thead>
<tr>
<th>Regsiter mode: (S2 specifies a register)</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 24 23 19 18 14 13 12 5 4 0</td>
</tr>
<tr>
<td>Opcode</td>
</tr>
<tr>
<td>8 5 5 1 8 5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register-immediate mode (S2 specifies an operand)</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 24 23 19 18 14 13 12 0</td>
</tr>
<tr>
<td>Opcode</td>
</tr>
<tr>
<td>8 5 5 1 13</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PC relative mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 24 23 19 18 0</td>
</tr>
<tr>
<td>Opcode</td>
</tr>
<tr>
<td>8 5 19</td>
</tr>
</tbody>
</table>
BERKELEY RISC I

• Register 0 was hard-wired to a value of 0.
• There are eight memory access instructions
  – Five load-from-memory instructions
  – Three store-to-memory instructions.

• The load instructions:
  LDL load long
  LDSU load short unsigned
  LDSS load short signed
  LDBU load byte unsigned
  LDBS load byte signed
  – Where long is 32 bits, short is 16 bits and a byte is 8 bits

• The store instructions:
  STL store long
  STS store short
  STB store byte
Berkeley RISC I

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDL</td>
<td>( R_d \leftarrow M[(R_s) + S_2] ) load long</td>
</tr>
<tr>
<td>LDSU</td>
<td>( R_d \leftarrow M[(R_s) + S_2] ) load short unsigned</td>
</tr>
<tr>
<td>LDSS</td>
<td>( R_d \leftarrow M[(R_s) + S_2] ) load short signed</td>
</tr>
<tr>
<td>LDBU</td>
<td>( R_d \leftarrow M[(R_s) + S_2] ) load byte unsigned</td>
</tr>
<tr>
<td>LDBS</td>
<td>( R_d \leftarrow M[(R_s) + S_2] ) load byte signed</td>
</tr>
<tr>
<td>STL</td>
<td>( M[(R_s) + S_2] \leftarrow R_d ) store long</td>
</tr>
<tr>
<td>STS</td>
<td>( M[(R_s) + S_2] \leftarrow R_d ) store short</td>
</tr>
<tr>
<td>STB</td>
<td>( M[(R_s) + S_2] \leftarrow R_d ) store byte</td>
</tr>
</tbody>
</table>

- Here the difference between the lengths is
  - A long is simply loaded, since it is the same size as the register (32 bits).
  - A short or a byte can be loaded into a register
    » Unsigned - in which case the upper bits of the register are loaded with 0’s.
    » Signed - in which case the upper bits of the register are loaded with the sign bit of the short/byte loaded.
## INSTRUCTION SET OF BERKELEY RISC I

### Data manipulation instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operands</th>
<th>Register Transfer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Rs,S2,Rd</td>
<td>Rd ← Rs + S2</td>
<td>Integer add</td>
</tr>
<tr>
<td>ADDC</td>
<td>Rs,S2,Rd</td>
<td>Rd ← Rs + S2 + carry</td>
<td>Add with carry</td>
</tr>
<tr>
<td>SUB</td>
<td>Rs,S2,Rd</td>
<td>Rd ← Rs - S2</td>
<td>Integer subtract</td>
</tr>
<tr>
<td>SUBC</td>
<td>Rs,S2,Rd</td>
<td>Rd ← Rs - S2 - carry</td>
<td>Subtract with carry</td>
</tr>
<tr>
<td>SUBR</td>
<td>Rs,S2,Rd</td>
<td>Rd ← S2 - Rs</td>
<td>Subtract reverse</td>
</tr>
<tr>
<td>SUBCR</td>
<td>Rs,S2,Rd</td>
<td>Rd ← S2 - Rs - carry</td>
<td>Subtract with carry</td>
</tr>
<tr>
<td>AND</td>
<td>Rs,S2,Rd</td>
<td>Rd ← Rs ∧ S2</td>
<td>AND</td>
</tr>
<tr>
<td>OR</td>
<td>Rs,S2,Rd</td>
<td>Rd ← Rs ∨ S2</td>
<td>OR</td>
</tr>
<tr>
<td>XOR</td>
<td>Rs,S2,Rd</td>
<td>Rd ← Rs ⊕ S2</td>
<td>Exclusive-OR</td>
</tr>
<tr>
<td>SLL</td>
<td>Rs,S2,Rd</td>
<td>Rd ← Rs shifted by S2</td>
<td>Shift-left</td>
</tr>
<tr>
<td>SRL</td>
<td>Rs,S2,Rd</td>
<td>Rd ← Rs shifted by S2</td>
<td>Shift-right logical</td>
</tr>
<tr>
<td>SRA</td>
<td>Rs,S2,Rd</td>
<td>Rd ← Rs shifted by S2</td>
<td>Shift-right arithmetic</td>
</tr>
</tbody>
</table>

### Data transfer instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operands</th>
<th>Register Transfer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDL</td>
<td>(Rs)S2,Rd</td>
<td>Rd ← M[Rs + S2]</td>
<td>Load long</td>
</tr>
<tr>
<td>LDSU</td>
<td>(Rs)S2,Rd</td>
<td>Rd ← M[Rs + S2]</td>
<td>Load short unsigned</td>
</tr>
<tr>
<td>LDSS</td>
<td>(Rs)S2,Rd</td>
<td>Rd ← M[Rs + S2]</td>
<td>Load short signed</td>
</tr>
<tr>
<td>LDBU</td>
<td>(Rs)S2,Rd</td>
<td>Rd ← M[Rs + S2]</td>
<td>Load byte unsigned</td>
</tr>
<tr>
<td>LDBS</td>
<td>(Rs)S2,Rd</td>
<td>Rd ← M[Rs + S2]</td>
<td>Load byte signed</td>
</tr>
<tr>
<td>LDHI</td>
<td>Rd,Y</td>
<td>Rd ← Y</td>
<td>Load immediate high</td>
</tr>
<tr>
<td>STL</td>
<td>Rd,(Rs)S2</td>
<td>M[Rs + S2] ← Rd</td>
<td>Store long</td>
</tr>
<tr>
<td>STS</td>
<td>Rd,(Rs)S2</td>
<td>M[Rs + S2] ← Rd</td>
<td>Store short</td>
</tr>
<tr>
<td>STB</td>
<td>Rd,(Rs)S2</td>
<td>M[Rs + S2] ← Rd</td>
<td>Store byte</td>
</tr>
<tr>
<td>GETPSW</td>
<td>Rd</td>
<td>Rd ← PSW</td>
<td>Load status word</td>
</tr>
<tr>
<td>PUTPSW</td>
<td>Rd</td>
<td>PSW ← Rd</td>
<td>Set status word</td>
</tr>
</tbody>
</table>
### Program control instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operands</th>
<th>Register Transfer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMP</td>
<td>COND,S2(Rs)</td>
<td>PC ← Rs + S2</td>
<td>Conditional jump</td>
</tr>
<tr>
<td>JMPR</td>
<td>COND,Y</td>
<td>PC ← PC + Y</td>
<td>Jump relative</td>
</tr>
<tr>
<td>CALL</td>
<td>Rd,S2(Rs)</td>
<td>Rd ← PC, PC ← Rs + S2</td>
<td>Call subroutine and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CWP ← CWP - 1</td>
<td>change window</td>
</tr>
<tr>
<td>CALLR</td>
<td>Rd,Y</td>
<td>Rd ← PC, PC ← PC + Y</td>
<td>Call relative and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CWP ← CWP - 1</td>
<td>change window</td>
</tr>
<tr>
<td>RET</td>
<td>Rd,S2</td>
<td>PC ← Rd + S2</td>
<td>Return and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CWP ← CWP + 1</td>
<td>change window</td>
</tr>
<tr>
<td>CALLINT</td>
<td>Rd</td>
<td>Rd ← PC,CWP ← CWP - 1</td>
<td>Call an interrupt pr.</td>
</tr>
<tr>
<td>RETINT</td>
<td>Rd,S2</td>
<td>PC ← Rd + S2</td>
<td>Return from</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CWP ← CWP + 1</td>
<td>interrupt pr.</td>
</tr>
<tr>
<td>GTLPC</td>
<td>Rd</td>
<td>Rd ← PC</td>
<td>Get last PC</td>
</tr>
</tbody>
</table>
CHARACTERISTICS OF RISC

• RISC Characteristics
  - Relatively few instructions
  - Relatively few addressing modes
  - Memory access limited to load and store instructions
  - All operations done within the registers of the CPU
  - Fixed-length, easily decoded instruction format
  - Single-cycle instruction format
  - Hardwired rather than microprogrammed control

• Advantages of RISC
  - VLSI Realization
  - Computing Speed
  - Design Costs and Reliability
  - High Level Language Support
ADVANTAGES OF RISC

• VLSI Realization
  Control area is considerably reduced
  ⇒ RISC chips allow a large number of registers on the chip
  - Enhancement of performance and HLL support
  - Higher regularization factor and lower VLSI design cost

  The GaAs VLSI chip realization is possible

• Computing Speed
  - Simpler, smaller control unit ⇒ faster
  - Simpler instruction set; addressing modes; instruction format
    ⇒ faster decoding
  - Register operation ⇒ faster than memory operation
  - Register window ⇒ enhances the overall speed of execution
  - Identical instruction length, One cycle instruction execution
    ⇒ suitable for pipelining ⇒ faster

Example:
  RISC I: 6%
  RISC II: 10%
  MC68020: 68%
  general CISCs: ~50%
ADVANTAGES OF RISC

• Design Costs and Reliability
  - Shorter time to design
    ⇒ reduction in the overall design cost and
    reduces the problem that the end product will
    be obsolete by the time the design is completed
  - Simpler, smaller control unit
    ⇒ higher reliability
  - Simple instruction format (of fixed length)
    ⇒ ease of virtual memory management

• High Level Language Support
  - A single choice of instruction
    ⇒ shorter, simpler compiler
  - A large number of CPU registers
    ⇒ more efficient code
  - Register window
    ⇒ Direct support of HLL
  - Reduced burden on compiler writer