W hat Are the Components of a n ISA?

- Sometimes known as The Programmer’s Model of the machine
- Storage cells
  - General and special purpose registers in the CPU
  - Many general purpose cells of same size in memory
- Storage associated with I/O devices
- The machine instruction set
  - The instruction set is the entire repertoire of machine operations
  - Makes use of storage cells, formats, and results of the fetch/execute cycle
  - i.e., register transfers
- The instruction format
  - Size and meaning of fields within the instruction
  - The nature of the fetch-execute cycle
  - Things that are done before the operation code is known

W hat Must an Instruction Specify?

- Which operation to perform
  - add r0, r1, r3
  - Ans: Op code: add, load, branch, etc.
- Where to find the operand or operands
  - add r0, r1, r3
  - In CPU registers, memory cells, I/O locations, or part of instruction
- Place to store result
  - add r0, r1, r3
  - Again CPU register or memory cell
- Location of next instruction
  - add r0, r1, r3
  - Almost always memory cell pointed to by program counter—PC
  - Sometimes there is no operand, or no result, or no next instruction. Can you think of examples?

Tbl 2.1 Examples of Data Movement Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
<th>Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV A, B</td>
<td>Move 16 bits from memory location A to location B</td>
<td>VAX11</td>
</tr>
<tr>
<td>LDA A, Addr</td>
<td>Load accumulator A with the byte at memory location Addr</td>
<td>M6800</td>
</tr>
<tr>
<td>lws R3, A</td>
<td>Move 32-bit data from memory location A to register R3</td>
<td>PPC601</td>
</tr>
<tr>
<td>l 0$3, 455</td>
<td>Load the 32-bit integer 455 into register 3</td>
<td>MIPS R3000</td>
</tr>
<tr>
<td>mov R4, dout</td>
<td>Move 16-bit data from #4 to output port dout</td>
<td>DEC PDP11</td>
</tr>
<tr>
<td>IN, AC, R80</td>
<td>Load a byte from input port X to Accumulator</td>
<td>Intel Pentium</td>
</tr>
<tr>
<td>LDA L (A0), A2</td>
<td>Load the address pointed to by XA into A2</td>
<td>M6800</td>
</tr>
</tbody>
</table>

- Lots of variation, even with one instruction type

Tbl 2.2 Examples of ALU Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
<th>Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULF A, B, C</td>
<td>multiply the 32-bit floating point values at mem loc rs and r3, store at C</td>
<td>VAX11</td>
</tr>
<tr>
<td>nabs r3, r1</td>
<td>Store abs value of r1 in r3</td>
<td>PPC601</td>
</tr>
<tr>
<td>ori $2, $1, 255</td>
<td>Store logical OR of $1 with 255 into reg $2</td>
<td>MIPS R3000</td>
</tr>
<tr>
<td>DEC R2</td>
<td>Decrement the 16-bit value stored in reg R2</td>
<td>DEC PDP11</td>
</tr>
<tr>
<td>SHL AX, 4</td>
<td>Shift the 16-bit value in reg AX left by 4 bit pos</td>
<td>Intel 8086</td>
</tr>
</tbody>
</table>

- Notice again the complete dissimilarity of both syntax and semantics.
### Tbl 2.3 Examples of Branch Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
<th>Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLSS A, Tgt</td>
<td>Branch to address Tgt if the least significant bit of mem loc'n A is set (i.e. = 1)</td>
<td>VAX11</td>
</tr>
<tr>
<td>bun r2</td>
<td>Branch to location in R2 if result of previous floating point computation was Not a Number (NAN)</td>
<td>PPC601</td>
</tr>
<tr>
<td>beq $2, $1, 32</td>
<td>Branch to location (PC + 4 + 32) if contents of $1 and $2 are equal</td>
<td>MIPS R3000</td>
</tr>
<tr>
<td>SOB R4, Loop</td>
<td>Decrement R4 and branch to Loop if R4 ≠ 0</td>
<td>DEC PDP11</td>
</tr>
<tr>
<td>JCNZ Addr</td>
<td>Jump to Addr if contents of register CX ≠ 0</td>
<td>Intel 8086</td>
</tr>
</tbody>
</table>

### CPU Registers Associated with Flow of Control—Branch Instructions

- Program counter usually locates next instruction
- Condition codes may control branch
- Branch targets may be separate registers

**Processor State**

<table>
<thead>
<tr>
<th>C</th>
<th>N</th>
<th>V</th>
<th>Z</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Program Counter</th>
<th>Condition Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Branch Targets**

### HLL Conditionals Implemented by Control Flow Change

- Conditions are computed by arithmetic instructions
- Program counter is changed to execute only instructions associated with true conditions

**C language**

```c
if NUM==5 then SET=7
```

**Assembly language**

```assembly
CMP.W #5, NUM    ; the comparison
BNE    L1        ; conditional branch
MOV.W #7, SET    ; action if true
L   L1            ; action if false
```

### CPU Registers May Have a “Personality”

- Architecture classes are often based on how the operands and result are located and how they are specified by the instruction.
- They can be in CPU registers or main memory:

**Stack**

- Arithmetic Registers
- Address Registers
- General Purpose Registers

**CPU Registers May Have a “Personality”**

- They can be in CPU registers or main memory:

**4-address instruction**

- Uses a CPU register stack to hold both operands and the result:
  - The 4-address instruction, hardly ever seen, also allows the address of the next instruction to be specified explicitly

---

**Fig 2.2 The 4-Address Machine and Instruction Format**

- Explicit addresses for operands, result, & next instruction
- Example assumes 24-bit addresses
- Discuss: size of instruction in bytes
### Fig 2.3 The 3-Address Machine and Instruction Format

- Address of next instruction kept in processor state register — the PC (except for explicit branches/jumps)
- Rest of addresses in instruction

### Memory

- **Op1Addr:**
- **Op2Addr:**
- **Op1**
- **Program counter**
- **ResAddr:**
- **NextiAddr:**

### Instruction format

- Bits: 8 24 24

### CPU

- **add:** Res, Op1, Op2 (Res ← Op2 + Op1)

### Example


### Example 2.1 Expression Evaluation for 3-, 2-, 1-, and 0-Address Machines

**Evaluate:** \( a = (b+c) \times d - e \)

- **3-address:**
  - \( \text{add } a, b, c \)
  - \( \text{load } a, b \)
  - \( \text{push } b \)
  - \( \text{add } a, c \)
  - \( \text{push } c \)
  - \( \text{sub } a, d \)
  - \( \text{push } d \)
  - \( \text{store } a \)
  - \( \text{push } e \)
  - \( \text{pop } a \)

- **2-address:**
  - \( \text{add } a, b \)
  - \( \text{load } b \)
  - \( \text{push } b \)
  - \( \text{add } a, c \)
  - \( \text{push } c \)
  - \( \text{sub } a, d \)
  - \( \text{push } d \)

- **1-address:**
  - \( \text{add } a, b \)
  - \( \text{load } b \)
  - \( \text{push } b \)

- **0-address:**
  - \( \text{push } b \)
  - \( \text{load } b \)
  - \( \text{store } a \)

**Discussion:** Savings in instruction word size

### Fig 2.4 The 2-Address Machine and Instruction Format

- Result overwrites Operand 2
- Needs only 2 addresses in instruction but less choice in placing data

### Memory

- **Op1Addr:**
- **Op2Addr:**
- **Op1**
- **Program counter**
- **Op2, Res**
- **NextiNextiAddr:**

### Instruction format

- Bits: 8 24 24

### CPU

- **add Op2, Op1 (Op2 ← Op2 + Op1)**

### Example 2.1 Expression Evaluation for 3-, 2-, 1-, and 0-Address Machines

**Evaluate:** \( a = (b+c) \times d - e \)

- **3-address:**
  - \( \text{add } a, b, c \)
  - \( \text{load } a, b \)
  - \( \text{push } b \)
  - \( \text{add } a, c \)
  - \( \text{push } c \)
  - \( \text{sub } a, d \)
  - \( \text{push } d \)
  - \( \text{store } a \)
  - \( \text{push } e \)
  - \( \text{pop } a \)

- **2-address:**
  - \( \text{add } a, b \)
  - \( \text{load } b \)
  - \( \text{push } b \)
  - \( \text{add } a, c \)
  - \( \text{push } c \)
  - \( \text{sub } a, d \)
  - \( \text{push } d \)

- **1-address:**
  - \( \text{add } a, b \)
  - \( \text{load } b \)
  - \( \text{push } b \)

- **0-address:**
  - \( \text{push } b \)
  - \( \text{load } b \)
  - \( \text{store } a \)

**Discussion:** Savings in instruction word size
Real Machines Are Not So Simple

- Most real machines have a mixture of 3, 2, 1, 0, and 1½ address instructions.
- A distinction can be made on whether arithmetic instructions use data from memory.
- If ALU instructions only use registers for operands and result, machine type is load-store.
- Only load and store instructions reference memory.
- Other machines have a mix of register-memory and memory-memory instructions.

Addressing Modes

- An addressing mode is hardware support for a useful way of determining a memory address.
- Different addressing modes solve different HLL problems.
- Some addresses may be known at compile time, e.g., global variables.
- Others may not be known until run time, e.g., pointers.
- Addresses may have to be computed. Examples include:
  - Record (struct) components: variable base (full address) + constant (small).
  - Array components: constant base (full address) + index variable (small).
- Possible to store constant values w/o using another memory cell by storing them w/ or adjacent to the instruction itself.

HLL Examples of Structured Addresses

- C language: rec -> count
  - rec is a pointer to a record: full address variable
  - count is a field name: fixed byte offset, say 24
- C language: v[i]
  - v is fixed base address of array: full address constant
  - i is name of variable index: no larger than array size
- Variables must be contained in registers or memory cells.
- Small constants can be contained in the instruction.
- Result: need for "address arithmetic."
  - E.g., Address of Rec -> Count is address of Rec + offset of count.

Example: Computer, SRC

- 32 general purpose registers of 32 bits.
- 32-bit program counter, PC, and instruction register, IR.
- 2³² bytes of memory address space.
- The SRC CPU
<table>
<thead>
<tr>
<th>General</th>
<th>Purpose</th>
<th>Memory</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 32-bit</td>
<td>2³² bytes</td>
<td>2³² bits</td>
<td>M(32) means contents of memory location 32</td>
</tr>
<tr>
<td>R[7] means contents of register 7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
- Main memory
<table>
<thead>
<tr>
<th>PC</th>
<th>IR</th>
<th>R0</th>
<th>...</th>
<th>R31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>0</td>
</tr>
</tbody>
</table>
**SRC Basic Instruction Formats**

- There are three basic instruction format types.
- The number of register specifier fields and length of the constant field vary.
- Other formats result from unused fields or parts.
- Details of formats on next slide.

```
<table>
<thead>
<tr>
<th>Type</th>
<th>Format</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>31 27 26 22 11 0</td>
<td>R[3] ← brlnz r2, r1, r0</td>
</tr>
<tr>
<td>2</td>
<td>31 27 22 11 17 16 0</td>
<td>brlpl r4, r3, r2</td>
</tr>
<tr>
<td>3</td>
<td>31 27 22 11 16 12 11 0</td>
<td>brlmi r3, r0, r1</td>
</tr>
</tbody>
</table>
```

**Assy language form**

```
R[6] ← 1001 ← — brlzr r7, r5, r1
```

**plus ≠**

```
```

**condition**

```
R[2] ← R[0] ← 011 if rc |
```

There are actually only two branch instructions:

- br
- brl

**Fig 2.9 (Partial)**

**Total of 7 Detailed Formats**

**Assembly Language Forms of Arithmetic and Logic Instructions**

```
<table>
<thead>
<tr>
<th>Format</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>neg ra, rc</td>
<td>neg r1, r2</td>
<td>;Negate (r1 = ~r2)</td>
</tr>
<tr>
<td>not ra, rc</td>
<td>not r1, r3</td>
<td>;Not (r2 = ~r3 )</td>
</tr>
</tbody>
</table>
```

```
add ra, rb, rc | add r2, r3, r4 | ;2’s complement addition |
```

```
sub ra, rb, rc | sub r2, r3, r4 | ;2’s complement subtraction |
```

```
and ra, rb, rc | and r2, r3, r4 | ;Logical and |
```

```
or ra, rb, rc | or r2, r3, r4 | ;Logical or |
```

```
addi ra, rb, c2 | addi r1, r3, #1 | ;Immediate 2’s complement add |
```

```
andi ra, rb, c2 | andi r1, r3, #1 | ;Immediate logical and |
```

```
or ra, rb, c2 | or r1, r2, c2 | ;Immediate logical or |
```

- Immediate subtract not needed since constant in addi may be negative.

**Tbl 2.6 Forms and Formats of the br and brl Instructions**

```
<table>
<thead>
<tr>
<th>Ans.’s lang.</th>
<th>Example instr.</th>
<th>Meaning</th>
<th>op ra rb rc c3</th>
<th>Branch Cond'n</th>
</tr>
</thead>
<tbody>
<tr>
<td>br</td>
<td>br r4</td>
<td>PC ← R[4]</td>
<td>8 — 4 — 001 always</td>
<td></td>
</tr>
<tr>
<td>brl</td>
<td>brl r8, r4</td>
<td>R[6] ← PC</td>
<td>9 6 4 — 001 always</td>
<td></td>
</tr>
<tr>
<td>brl</td>
<td>brl r5</td>
<td>PC ← R[4]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>brzr</td>
<td>brzr r5, r1</td>
<td>(R[1] = 0)</td>
<td>8 — 5 1 010 zero</td>
<td></td>
</tr>
<tr>
<td>brln</td>
<td>brln r7, r5, r1</td>
<td>R[7] ← PC</td>
<td>9 7 5 1 010 zero</td>
<td></td>
</tr>
<tr>
<td>bmnz</td>
<td>bmnz r1, r6</td>
<td>(R[6] = 0)</td>
<td>8 — 1 0 011 nonzero</td>
<td></td>
</tr>
<tr>
<td>bmnz</td>
<td>bmnz r1, r6</td>
<td>(R[6] = 0)</td>
<td>8 — 1 0 011 nonzero</td>
<td></td>
</tr>
<tr>
<td>brpl</td>
<td>brpl r13, r2</td>
<td>(R[2] = 0)</td>
<td>8 — 3 2 100 plus</td>
<td></td>
</tr>
<tr>
<td>brpl</td>
<td>brpl r14, r3, r2</td>
<td>(R[3] = 0)</td>
<td>8 — 3 2 100 plus</td>
<td></td>
</tr>
<tr>
<td>bmnz</td>
<td>brmnz r0, r1</td>
<td>(R[1] = 0)</td>
<td>8 — 0 1 101 minus</td>
<td></td>
</tr>
<tr>
<td>bmnz</td>
<td>brmnz r13, r0, r1</td>
<td>(R[1] = 0)</td>
<td>8 — 0 1 101 minus</td>
<td></td>
</tr>
</tbody>
</table>
```

**Branch Instruction Format**

- There are actually only two branch instructions:
  - br
  - brl

- It is c3<2..0>, the 3lsbs of c3, that governs what the branch condition is:

```
<table>
<thead>
<tr>
<th>label</th>
<th>condition</th>
<th>Assy language form</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>never</td>
<td>brlnv</td>
<td>brlnv r6</td>
</tr>
<tr>
<td>001</td>
<td>always</td>
<td>br, brl</td>
<td>br r5, brl r5</td>
</tr>
<tr>
<td>010</td>
<td>if rc = 0</td>
<td>brzr, brzr</td>
<td>brzr r2, r4, r5</td>
</tr>
<tr>
<td>011</td>
<td>if rc = 0</td>
<td>brnz, brnz</td>
<td>brnz r2, r4, r5</td>
</tr>
<tr>
<td>100</td>
<td>if rc = 0</td>
<td>brij, brij</td>
<td>brij r2, r4, r5</td>
</tr>
</tbody>
</table>
```

- Note that branch target address is always in register R[b].
- It must be placed there explicitly to a previous instruction.
### Branch Instructions—Example

C: goto Label3

**SRC:**

```plaintext
lar r0, Label3 ; put branch target address into tgt reg.
br r0 ; and branch
```

### Example of Conditional Branch

**in C:**

```plaintext
#define Cost 125
if (X<0) then X = -X;
```

**in SRC:**

```plaintext
Cost .equ 125 ;define symbolic constant
.org 1000 ;next word will be loaded at address 1000
X: .dw 1 ;reserve 1 word for variable X
.org 5000 ;program will be loaded at location 5000
5000: lar r0, Over ;load address of "false" jump location
ld r1, X ;load value of X into r1
brpl r0, r1 ;branch to Else if r1>0
neg r1, r1 ;negate value
Over: ;continue
```

### RTN (Register Transfer Notation)

- Provides a formal means of describing machine structure and function
- Is at the “just right” level for machine descriptions
- Does not replace hardware description languages
- Can be used to describe what a machine does (an abstract RTN) without describing how the machine does it
- Can also be used to describe a particular hardware implementation (a concrete RTN)

### Using RTN to Describe Dynamic Properties

- **Static Properties**
  - Specifying registers
    - IR[31..0] specifies a register named “IR” having 32 bits numbered 31 to 0
    - “Naming” using the := naming operator:
      - op[4..0] := IR[31..27] specifies that the 5 msbs of IR be called op, with bits 4.0
      - Notice that this does not create a new register, it just generates another name, or “alias,” for an already existing register or part of a register

- **Dynamic Properties**
  - Conditional expressions:
    - (op=12) → R[ra] ← R(rb) + R(rc); : defines the add instruction
    - “if” condition “then” RTN Assignment Operator

This fragment of RTN describes the SRC add instruction. It says, “when the op field of IR = 12, then store in the register specified by the ra field, the result of adding the register specified by the rb field to the register specified by the rc field.”
### Using RTN to Describe the SRC (Static) Processor State

**Processor state**
- **PC[31..0]**: program counter (memory addr. of next inst.)
- **IR[31..0]**: instruction register
- **Run**: one bit run/halt indicator
- **Srt**: start signal
- **R[0..31]**: general purpose registers

### RTN Register Declarations

- General register specifications show some features of the notation
- Describes a set of 32 32-bit registers with names R[0] to R[31]

<table>
<thead>
<tr>
<th>Name of registers</th>
<th>Register # in square brackets</th>
<th>... specifies a range of indices</th>
<th>msb #</th>
<th>lsb #</th>
<th>Bit # in angle brackets</th>
</tr>
</thead>
<tbody>
<tr>
<td>R[0..31]</td>
<td>31..0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### RTN Instruction Formatting Uses Renaming of IR Bits

**Instruction formats**
- **op[4..0]** := IR[31..27]: operation code field
- **ra[4..0]** := IR[26..22]: target register field
- **rb[4..0]** := IR[21..17]: operand, address index, or branch target register
- **rc[4..0]** := IR[16..12]: second operand, conditional test, or shift count register
- **c1[21..0]** := IR[21..0]: long displacement field
- **c2[16..0]** := IR[16..0]: short displacement or immediate field
- **c3[11..0]** := IR[11..0]: count or modifier field

### Specifying Dynamic Properties of SRC: RTN Gives Specifics of Address Calculation

**Effective address calculations (occur at runtime):**
- **disp[31..0]** := ((rb=0) \(\rightarrow\) c2[16..0] (sign extend); displacement (rb=0) \(\rightarrow\) R[rb] + c2[16..0] (sign extend, Z's comp.)): address
- **rel[31..0]** := PC[31..0] + c1[21..0] (sign extend, Z's comp.): relative address

- Renaming defines displacement and relative addresses
- New RTN notation is used

<table>
<thead>
<tr>
<th>Condition expression</th>
<th>Renaming defines displacement and relative addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>If condition then</td>
<td>New RTN notation is used</td>
</tr>
</tbody>
</table>

### Detailed Questions Answered by the RTN for Addresses

- What set of memory cells can be addressed by direct addressing (displacement with rb=0)
  - If c2[16]=0 (positive displacement) absolute addresses range from 00000000H to 0000FFFFH
  - If c2[16]=1 (negative displacement) absolute addresses range from FFFF0000H to FFFFFFFFH
- What range of memory addresses can be specified by a relative address
  - The largest positive value of |C1[21..0]| is 2^21-1 and its most negative value is \(-2^{21}\), so addresses up to \(2^{21}-1\) forward and \(-2^{21}\) backward from the current PC value can be specified
- Note the difference between rb and R[rb]
Instruction Interpretation: RTN

Description of Fetch-Execute

- Need to describe actions (not just declarations)
- Some new notation

Logical NOT
Logical AND

instruction_interpretation := (Run × Strt → Run ← 1:
Run → (IR ← M[PC]: PC ← PC + 4; instruction_execution));

Register transfer
Separates statements that occur in sequence

RTN Sequence and Clocking

- In general, RTN statements separated by : take place during the same clock pulse
- Statements separated by ; take place on successive clock pulses
- This is not entirely accurate since some things written with one RTN statement can take several clocks to perform
- More precise difference between : and ;
  - The order of execution of statements separated by : does not matter
  - If statements are separated by ; the one on the left must be complete before the one on the right starts

More About Instruction Interpretation RTN

- In the expression IR ← M[PC]: PC ← PC + 4; which value of PC applies to M[PC]?
- The rule in RTN is that all right hand sides of ‘;’-separated RTs are evaluated before any LHS is changed
  - In logic design, this corresponds to “master-slave” operation of flip-flops
- We see what happens when Run is true and when Run is false but Strt is true. What about the case of Run and Strt both false?
  - Since no action is specified for this case, the RTN implicitly says that no action occurs in this case

Individual Instructions

- instruction_interpretation contained a forward reference to instruction_execution
- instruction_execution is a long list of conditional operations
- The condition is that the op code specifies a given instruction
- The operation describes what that instruction does
- Note that the operations of the instruction are done after (;) the instruction is put into IR and the PC has been advanced to the next instruction

RTN Instruction Execution for Load and Store Instructions

instruction_execution := {
id (:= op = 1) → R[ra] ← M[disp]: load register
ldr (:= op = 2) → R[ra] ← M[ref]: load register relative
str (:= op = 3) → M[disp] ← R[ra]: store register relative
la (:= op = 5) → R[ra] ← disp: load displacement address
lar (:= op = 6) → R[ra] ← rel: load relative address

- The in-line definition (= op = 1) saves writing a separate definition id := op = 1 for the Id mnemonic
- The previous definitions of disp and rel are needed to understand all the details

SRC RTN—the Main Loop

ii := instruction_interpretation:
iel := instruction_execution:

ii := (¬Run × Strt → Run ← 1:
Run → (IR ← M[PC]: PC ← PC + 4; iel));
iel := {
id (:= op = 1) → R[ra] ← M[disp]:
ldr (:= op = 2) → R[ra] ← M[ref]:

- Big switch statement
- on the opcode
  - stop (:= op = 31) → Run ← 0;
};
i

Thus ii and iel invoke each other, as coroutines.
Use of RTN Definitions: Text Substitution Semantics

\[ \text{id (:= op=1)} \rightarrow \text{R[ra]} \leftarrow \text{M[disp]}; \]
\[ \text{disp/31..0} = ((\text{rb}=0) \rightarrow \text{c2/16..0} \text{ (sign extend)}; \]
\[ (\text{rb}=0) \rightarrow \text{R[rb] + c2/16..0} \text{ (sign extend, 2's comp.) ;} \]
\[ \text{id (:= op=1)} \rightarrow \text{R[ra]} \leftarrow \text{M[disp]}; \]
\[ (\text{rb}=0) \rightarrow \text{c2/16..0} \text{ (sign extend)}; \]
\[ (\text{rb}=0) \rightarrow \text{R[rb] + c2/16..0} \text{ (sign extend, 2's comp.) ;} \]

- An example:
  - If IR = 00001 0101 00011 00000000001011
  - then id \( \rightarrow \text{R[3]} \leftarrow \text{M[0]} \text{[3] + 11} \)

RTN Descriptions of SRC Branch Instructions

- Branch condition determined by 3 labs of instruction
- Link register (R[ra]) set to point to next instruction

\[ \text{cond := ( c2/2..0=0} \rightarrow \text{0; \ never} \]
\[ c2/2..0=1 \rightarrow \text{1; \ always} \]
\[ c2/2..0=2 \rightarrow \text{T[c2/3..0}; \ \text{if register is nonzero} \]
\[ c2/2..0=3 \rightarrow \text{T[c2/3..0}; \ \text{if positive or zero} \]
\[ c2/2..0=5 \rightarrow \text{T[c2/3..0}; \ \text{if negative} \]
\[ \text{br (= op= 8)} \rightarrow \{ \text{cond} \rightarrow \text{PC }\leftarrow \text{R[rb]} \}; \ \text{conditional branch} \]
\[ \text{brl (= op= 9)} \rightarrow \{ \text{cond} \rightarrow \text{PC }\leftarrow \text{R[rb]} \}; \ \text{branch and link} \]

RTN for Arithmetic and Logic

\[ \text{add (:= op=12)} \rightarrow \text{R[ra]} \leftarrow \text{R[rb]} + \text{R[rc]}; \]
\[ \text{addi (:= op=13)} \rightarrow \text{R[ra]} \leftarrow \text{R[rb]} + \text{c2/16..0} \text{ (2's comp. sign ext.)}; \]
\[ \text{sub (:= op=14)} \rightarrow \text{R[ra]} \leftarrow \text{R[rb]} - \text{R[rc]}; \]
\[ \text{neg (:= op=15)} \rightarrow \text{R[ra]} \leftarrow \text{~R[rc]}; \]
\[ \text{and (:= op=20)} \rightarrow \text{R[ra]} \leftarrow \text{R[rb]} \land \text{R[rc]}; \]
\[ \text{or (:= op=22)} \rightarrow \text{R[ra]} \leftarrow \text{R[rb]} \lor \text{R[rc]}; \]
\[ \text{orl (:= op=23)} \rightarrow \text{R[ra]} \leftarrow \text{R[rb]} \land \text{c2/16..0} \text{ (sign extend)}; \]
\[ \text{not (:= op=24)} \rightarrow \text{R[ra]} \leftarrow \text{~R[rc]}; \]
- Logical operators: and, or, and not

RTN for Shift Instructions

- Count may be 5 labs of a register or the instruction
- Notation: @ - replication, # - concatenation

\[ \text{n := ( (} \text{c3/4..0=0}) \rightarrow \text{R[rc]} \text{4..0}; \]
\[ \text{shl (:= op=28)} \rightarrow \text{R[ra]} \text{31..0} \leftarrow (\text{n @ 0}) \land \text{R[ra]} \text{31..0}; \]
\[ \text{shra (:= op=27)} \rightarrow \text{R[ra]} \text{31..0} \leftarrow (\text{n @ 0}) \land \text{R[rb]} \text{31..0}; \]
\[ \text{shr (:= op=26)} \rightarrow \text{R[ra]} \text{31..0} \leftarrow (\text{n @ 0}) \land \text{R[rb]} \text{31..0}; \]
\[ \text{shc (:= op=29)} \rightarrow \text{R[ra]} \text{31..0} \leftarrow (\text{n @ 0}) \land \text{R[rb]} \text{31..0}; \]

Example of Replication and Concatenation in Shift

- Arithmetic shift right by 13 concatenates 13 copies of the sign bit with the upper 19 bits of the operand

\[ \text{shra r1, r2, 13}; \]
\[ \text{R[2] = \begin{array}{c}
1001 1111 0110 1110 1100 0001 0110 \\
13@R[2](31) \# R[2](31..13)
\end{array}} \]

Assembly Language for Shift

- Form of assembly language instruction tells whether to set c3=0

\[ \text{shr ra, rb, rc}; \ \text{Shift rb right into ra by 5 labs of rc} \]
\[ \text{shr ra, rb, count}; \ \text{Shift rb right into ra by 5 labs of inst} \]
\[ \text{shl ra, rb, rc}; \ \text{Shift rb left into ra by 5 labs of rc} \]
\[ \text{shl ra, rb, count}; \ \text{Shift rb left into ra by 5 labs of inst} \]
\[ \text{shc ra, rb, rc}; \ \text{Shift rb circ. into ra by 5 labs of rc} \]
\[ \text{shc ra, rb, count}; \ \text{Shift rb circ. into ra by 5 labs of inst} \]
End of RTN Definition of instruction_execution

nop (:= op= 0) ;
stop (:= op= 31) ; Run := 0;

instruction_interpretation.

- We will find special use for nop in pipelining
- The machine waits for Strt after executing stop
- The long conditional statement defining instruction_execution ends with a direction to go repeat instruction_interpretation, which will fetch and execute the next instruction (if Run still = 1)

nop (:= op= 0) ® No operation
stop (:= op= 31) ® Run = 0: Stop instruction

End of instruction_execution

Confused about RTN and SRC?

- SRC is a Machine Language
  - It can be interpreted by either hardware or software simulator.
- RTN is a Specification Language
  - Specification languages are languages that are used to specify other languages or systems—a metalanguage.
  - Other examples: LEX, YACC, VHDL, Verilog

Figure 2.10 may help clear this up...

Fig 2.10 The Relationship of RTN to SRC

SRC specification written in RTN

RTN compiler
Generated processor

SRC program and data
SRC interpreter or simulator

Data output

A Note About Specification Languages

- They allow the description of what without having to specify how.
- They allow precise and unambiguous specifications, unlike natural language.
- They reduce errors:
  - Errors due to misinterpretation of imprecise specifications written in natural language.
  - Errors due to confusion in design and implementation—"human error."
- Now the designer must debug the specification!
  - Specifications can be automatically checked and processed by tools.
- An RTN specification could be input to a simulator generator that would produce a simulator for the specified machine.
- An RTN specification could be input to a compiler generator that would generate a compiler for the language, whose output could be run on the simulator.

Fig 2.11 Register Transfers Hardware and Timing for a Single-Bit Register Transfer: A ← B

- Implementing the RTN statement A ← B

Implement the RTN statement A ← B

(a) Hardware

(b) Timing

Addressing Modes Described in RTN (Not SRC)

<table>
<thead>
<tr>
<th>Mode name</th>
<th>Assembler</th>
<th>RTN meaning</th>
<th>Use</th>
<th>Target register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Ra</td>
<td>R[i] − R[a]</td>
<td>Tmp. Var.</td>
<td></td>
</tr>
<tr>
<td>Register indirect</td>
<td>(Ra)</td>
<td>R[i] − M[R[a]]</td>
<td>Pointer</td>
<td></td>
</tr>
<tr>
<td>Immediate</td>
<td>X</td>
<td>X</td>
<td>Constant</td>
<td></td>
</tr>
<tr>
<td>Indirect</td>
<td>(X)</td>
<td>R[i] − M[R[X]]</td>
<td>Pointer Var.</td>
<td></td>
</tr>
<tr>
<td>indexed, based, or displacement</td>
<td>X(Ra)</td>
<td>R[i] − M[X + R[a]]</td>
<td>Arrays, structs</td>
<td></td>
</tr>
<tr>
<td>Relative</td>
<td>X(PC)</td>
<td>R[i] − M[X + PC]</td>
<td>Val stored w pgm</td>
<td></td>
</tr>
<tr>
<td>Autoncrement</td>
<td>(Ra)+</td>
<td>R[i] − M[R[a]; R[a] ← R[a] + 1</td>
<td>Sequential</td>
<td></td>
</tr>
<tr>
<td>Autodecrement</td>
<td>-(Ra)</td>
<td>R[a] ← R[a]− 1</td>
<td>R[i] ← M[R[a]]</td>
<td>access.</td>
</tr>
</tbody>
</table>
Fig 2.12 Multiple Bit Register Transfer: \( A(m..1) \leftarrow B(m..1) \)

(a) Individual flip-flops  (b) Abbreviated notation

Fig 2.13 Data Transmission View of Logic Gates

- Logic gates can be used to control the transmission of data:
  - Data gate
  - Control gate
  - Data merge
  - Controlled complement

Fig 2.14 Two-Way Gated Merge, or Multiplexer

- Data from multiple sources can be selected for transmission

Fig 2.15 Basic Multiplexer and Symbol Abbreviation

- Multiplexer gate signals \( G_i \) may be produced by a binary to one-out-of-\( n \) decoder

Fig 2.16 Separating Merged Data

- Merged data can be separated by gating at the right time
- It can also be strobed into a flip-flop when valid

Fig 2.17 Multiplexed Register Transfers Using Gates and Strobes

- Selected gate and strobe determine which RT
- \( A \leftarrow C \) and \( B \leftarrow C \) can occur together, but not \( A \leftarrow C \) and \( B \leftarrow D \)
Fig 2.18 Open-Collector NAND Gate Output Circuit

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0V, 0V</td>
<td>Open</td>
</tr>
<tr>
<td>+V, 0V</td>
<td>Open</td>
</tr>
<tr>
<td>+V, +V</td>
<td>+V</td>
</tr>
</tbody>
</table>

(a) Open-collector NAND truth table

Fig 2.19 Wired AND Connection of Open-Collector Gates

(a) Wired AND connection

<table>
<thead>
<tr>
<th>Switch</th>
<th>Wired AND output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open (a)</td>
<td>Open (a)</td>
</tr>
<tr>
<td>Open (b)</td>
<td>Open (b)</td>
</tr>
</tbody>
</table>

(b) Symbol

Fig 2.20 Open-Collector Wired OR Bus

- DeMorgan's OR by NOT of AND of NOTS
- Pull-up resistor removed from each gate - open collector
- One pull-up resistor for whole bus
- Forms an OR distributed over the connection

Fig 2.21 Tri-State Gate Internal Structure and Symbol

(a) Tri-state gate structure

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

(c) Tri-state gate truth table

Fig 2.22 Registers Connected by a Tri-State Bus

- Can make any register transfer \( R[i] \rightarrow R[j] \)
- Can't have \( G_i = 1 \) for \( i \neq j \)
- Violating this constraint gives low resistance path from power supply to ground—with predictable results!

Fig 2.23 Registers and Arithmetic Units Connected by One Bus

Example:
Abstract RTN
\[ R[3] \leftarrow R[1] \cdot R[2]; \]

Concrete RTN
\[ Y = R[2]; \]
\[ Z = R[1] \cdot Y; \]
\[ R[3] = Z; \]

Control Sequence
\[ R[2] \rightarrow Y; \]
\[ R[1] \rightarrow Z; \]
\[ Z \rightarrow R[3]; \]

Notice that what could be described in one step in the abstract RTN took three steps on this particular hardware.
RTs Possible with the One-Bus Structure

- $R[i]$ or $Y$ can get the contents of anything but $Y$
- Since result different from operand, it cannot go on the bus that is carrying the operand
- Arithmetic units thus have result registers
- Only one of two operands can be on the bus at a time, so adder has register for one operand
- $R[i] \leftarrow R[i] + R[k]$ is performed in 3 steps: $Y \leftarrow R[k]$; $Z \leftarrow R[i] + Y$; $R[i] \leftarrow Z$
- $R[i] \leftarrow R[i] + R[k]$ is high level RTN description
- $Y \leftarrow R[k]$; $Z \leftarrow R[i] + Y$; $R[i] \leftarrow Z$ is concrete RTN
- Map to control sequence is: $R[2]_{out}$, $Y_{in}$; $R[1]_{out}$, $Z_{in}$; $Z_{out}$, $R[3]_{in}$

From Abstract RTN to Concrete RTN to Control Sequences

- The ability to begin with an abstract description, then describe a hardware design and resulting concrete RTN and control sequence is powerful.
- We shall use this method in Chapter 4 to develop various hardware designs for SRC.

Chapter 2 Summary

- Classes of computer ISAs
- Memory addressing modes
- SRC: a complete example ISA
- RTN as a description method for ISAs
- RTN description of addressing modes
- Implementation of RTN operations with digital logic circuits
- Gates, strobes, and multiplexers