Chapter 2: Machines, Machine Languages, and Digital Logic

Instruction sets, SRC, RTN, and the mapping of register transfers to digital logic circuits
Chapter 2 Topics

- 2.1 Classification of Computers and Instructions
- 2.1 Different ISA styles: 3, 2, 1, & 0 Address Operations
- 2.3 Informal Description of the Simple RISC Computer, SRC
- 2.4 Formal Description of SRC using Register Transfer Notation (RTN)
- 2.5 RTN Description of Addressing Modes
- 2.6 Register Transfers and Logic Circuits: from Behavior to Hardware
What are the components of an ISA?

- Sometimes known as *The Programmers 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
  - Defines 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
Fig. 2.1 Programmer’s Models of Various Machines

We saw in Chap. 1 a variation in number and type of storage cells.
What Must an Instruction Specify?

- Which operation to perform: add r0, r1, r3
  - 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 br endloop
  - 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?
Instructions Can Be Divided into 3 Classes

- Data movement instructions
  - Move data from a memory location or register to another memory location or register without changing its form
  - Load—source is memory and destination is register
  - Store—source is register and destination is memory
- Arithmetic and logic (ALU) instructions
  - Changes the form of one or more operands to produce a result stored in another location
  - Add, Sub, Shift, etc.
- Branch instructions (control flow instructions)
  - Any instruction that alters the normal flow of control from executing the next instruction in sequence
  - Br Loc, Brz Loc2,—unconditional or conditional branches
## 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 mem. Loc. A to loc. B</td>
<td>VAX 11</td>
</tr>
<tr>
<td>lwzR3, A</td>
<td>Move 32 bits from mem. Loc. A to reg. R3</td>
<td>PPC601</td>
</tr>
<tr>
<td>li $3, 455</td>
<td>Load the 32-bit integer 455 into Reg. 3</td>
<td>R3000</td>
</tr>
<tr>
<td>mov R4, dout</td>
<td>Move 16 bits from R4 to port dout</td>
<td>DEC DP11</td>
</tr>
<tr>
<td>IN A1, KBD</td>
<td>Load a byte from port KBD to Accum.</td>
<td>Pentium</td>
</tr>
<tr>
<td>LEA.L (A0), A2</td>
<td>Load address pointed to by A0 into A2</td>
<td>M68000</td>
</tr>
</tbody>
</table>

- Lots of variation, even with one instruction type
### Tbl 2.2  Examples of ALU (Arithmetic and Logic Unit) 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 locations A and B and store in C</td>
<td>VAX 11</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 reg 1 with 255 into reg 2</td>
<td>R3000</td>
</tr>
<tr>
<td>DEC R2</td>
<td>Decrement the 16 bit value stored in reg R2</td>
<td>DEC DP11</td>
</tr>
<tr>
<td>SHL AX, 4</td>
<td>Shift the 16-bit value in AX left by 4 bits</td>
<td>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>JCXZ 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 Instrs.

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

Processor State

| Program Counter | | Condition Codes |
|-----------------|-----------------|
|                 | CNVZ             |
| 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
L1 ... ; action if false
```
CPU Registers may have a “personality”

- Architecture classes are often based on how where the operands and result are located and how they are specified by the instruction.
- They can be in CPU registers or main memory
3, 2, 1, & 0 Address ISAs

- The classification is based on arithmetic instructions that have two operands and one result.
- The key issue is “how many of these are specified by memory addresses, as opposed to being specified implicitly.”
- A 3 address instruction specifies memory addresses for both operands and the result $R \leftarrow Op1 \; op \; Op2$
- A 2 address instruction overwrites one operand in memory with the result $Op2 \leftarrow Op1 \; op \; Op2$
- A 1 address instruction has a register, called the accumulator register to hold one operand & the result (no addr. needed) $Acc \leftarrow Acc \; op \; Op1$
- A 0 address instruction uses a CPU register stack to hold both operands and the result $TOS \leftarrow TOS \; op \; SOS$ where $TOS$ is Top Of Stack, $SOS$ is Second On Stack)
- The 4-address instruction, hardly ever seen, also allows the address of the next instruction to specified explicitly.
Fig. 2.2 The 4 Address Instruction

- Explicit addresses for operands, result & next instruction
- Example assumes 24-bit addresses
  - Discuss: size of instruction in bytes

```
Memory

Op1Addr: Op1
Op2Addr: Op2
ResAddr: Res

CPU


NextiAddr: Nexti

Instruction format

<table>
<thead>
<tr>
<th>Bits</th>
<th>8</th>
<th>24</th>
<th>24</th>
<th>24</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>ResAddr</td>
<td>Op1Addr</td>
<td>Op2Addr</td>
<td>NextiAddr</td>
<td></td>
</tr>
<tr>
<td>Which operation</td>
<td>Where to put result</td>
<td>Where to find operands</td>
<td>Where to find next instruction</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Fig 2.3 The 3 Address Instruction

- Address of next instruction kept in processor state register—the PC (Except for explicit Branches/Jumps)
- Rest of addresses in instruction
  - Discuss: savings in instruction word size

<table>
<thead>
<tr>
<th>Instruction format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits: 8 24 24 24 24</td>
</tr>
<tr>
<td>add</td>
</tr>
<tr>
<td>Which operation</td>
</tr>
</tbody>
</table>
Fig. 2.4 The 2 Address Instruction

- Result overwrites Operand 2
- Needs only 2 addresses in instruction but less choice in placing data
**Fig. 2.5 1 Address Instructions**

### Need instructions to load and store operands:
- **LDA OpAddr**
- **STA OpAddr**

### Special CPU register, the accumulator, supplies 1 operand and stores result

### One memory address used for other operand

---

- The accumulator is a special CPU register that supplies one operand and stores the result.
- One memory address is used for the other operand.
- Instructions to load and store operands include:
  - **LDA OpAddr**
  - **STA OpAddr**

---

- **Instruction format**
  - Bits: 8 24
  - **add Op1Addr**
  - The accumulator performs the operation: `add Op1 (Acc ← Acc + Op1)`.
  - Where to find operand2, and where to put result:
    - `Op1Addr`:
    - `NextiAddr`:
    - `Nexti`: 
    - `Program counter`: 24
    - `Accumulator`: 
      - `add Op1`: `Acc ← Acc + Op1`
**Fig. 2.6 The 0 Address Instruction**

- Uses a push down stack in CPU
- Arithmetic uses stack for both operands and the result
- Computer must have a 1 address instruction to push and pop operands to and from the stack

**Instruction formats**

- **push Op1** (TOS ← Op1)
  - Bits: 8
  - Format: push Op1Addr
  - Operation: push
  - Result: Op1Addr

- **add** (TOS ← TOS + SOS)
  - Bits: 8
  - Format: add
  - Operation: add
  - Which operation: add

Where to find next instruction

Where to find operands, and where to put result (on the stack)
Example 2.1 Expression evaluation for 3-0 address instructions.

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

<table>
<thead>
<tr>
<th>3-address</th>
<th>2-address</th>
<th>1-address</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>add a, b, c</td>
<td>load a, b</td>
<td>load b</td>
<td>push b</td>
</tr>
<tr>
<td>mpy a, a, d</td>
<td>add a, c</td>
<td>add c</td>
<td>push c</td>
</tr>
<tr>
<td>sub a, a, e</td>
<td>mpy a, d</td>
<td>mpy d</td>
<td>add</td>
</tr>
<tr>
<td></td>
<td>sub a, e</td>
<td>sub e</td>
<td>push d</td>
</tr>
<tr>
<td></td>
<td></td>
<td>store a</td>
<td>mpy</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>push e</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>sub</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>pop a</td>
</tr>
</tbody>
</table>

- # of instructions & # of addresses both vary
- Discuss as examples: size of code in each case
Fig. 2.7 General Register Machines (1 \(1/2\) Address Machine)

- It is the most common choice in today’s general purpose computers
- Which register is specified by small “address” (3 to 6 bits for 8 to 64 registers)
- Load and store have one long & one short address: 1 1/2 addresses
- Arith. instruction has 3 “half” addresses
Real Machines are Not So Simple

- Most real machines have a mixture of 3, 2, 1, 0, 1 ½ 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 (1 or 1 ½ address machine) and memory-memory (2 or 3 address machine) instructions.
Trade-Offs

- The 3-address machines have the shortest code sequences, but large number of bits per instruction.
- The 0-address machines have the longest code sequences, but small number of bits per instruction.
- Even in 0-register machines, you need 1-address instructions.
- General register machines use short addresses in place of the long memory addresses to address internal registers.
- Load-store machines only include memory addresses in the data movement instructions.
- Register access is much faster than memory access.
- Short instructions are faster!
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 vars.
  - 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) + const.(small)
    - Array components:
      - const. base(full address) + index var.(small)
  - Possible to store constant values w/o using another memory cell by storing them with 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.
Fig 2.8 Common Addressing Modes

a) Immediate Addressing
   (Instruction contains the operand.)

Instr | Op'n | 3
LOAD #3, ...

b) Direct Addressing
   (Instruction contains address of operand)

Instr | Op'n | Addr of A
LOAD A, ...

---

c) Indirect Addressing
   (Instruction contains address of address of operand)

Instr | Op'n |
LOAD (A), ...
Address of address of A

---

d) Register Indirect Addressing
   (Register contains address of operand)

Instr | Op'n | R2 | ...
LOAD [R2], ...

---

e) Displacement (Based) (Indexed) Addressing
   (address of operand = register + constant)

Instr | Op'n | R2 | 4
LOAD 4[R2], ...
Operand Addr.

---

f) Relative Addressing
   (Address of operand = PC+constant)

Instr | Op'n | 4
LOADRel 4[PC], ...
Operand Addr.
Example Computer, SRC
Simple RISC Computer

- 32 general purpose registers of 32 bits
- 32 bit program counter, PC and instruction reg., IR
- $2^{32}$ bytes of memory address space

The SRC CPU

<table>
<thead>
<tr>
<th></th>
<th>The SRC CPU</th>
<th>Main memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>32 32-bit</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>general</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>purpose</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>registers</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>$2^{32}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bytes of</td>
</tr>
<tr>
<td></td>
<td></td>
<td>main</td>
</tr>
<tr>
<td></td>
<td></td>
<td>memory</td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R0</td>
<td>32-bit</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>general</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>purpose</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>registers</td>
<td></td>
</tr>
<tr>
<td>R31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

R[7] means contents of register 7
M[32] means contents of memory location 32
SRC Characteristics

- Load-store design: only way to access memory is through load and store instructions
- Only a few addressing modes are supported
- ALU Instructions are 3-register type
- Branch instructions can branch unconditionally or conditionally on whether the value in a specified register is = 0, <> 0, >= 0, or < 0.
- Branch-and-link instructions are similar, but leave the value of current PC in any register, useful for subroutine return.
- All instructions are 32-bits (1-word) long.
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

<table>
<thead>
<tr>
<th>Type 1</th>
<th>Type 2</th>
<th>Type 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>ra</td>
<td>c1</td>
</tr>
<tr>
<td>op</td>
<td>ra</td>
<td>rb</td>
</tr>
<tr>
<td>op</td>
<td>ra</td>
<td>rb</td>
</tr>
</tbody>
</table>

- Details of formats:
### Instruction formats

#### 1. Id, st, la, addi, andi, ori

<table>
<thead>
<tr>
<th>Op</th>
<th>ra</th>
<th>rb</th>
<th>rc</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>27</td>
<td>26</td>
<td>22</td>
<td>21 17 16</td>
</tr>
</tbody>
</table>

**Example**
- `Id r3, A` (R[3] = M[A])

#### 2. Idr, str, lar

<table>
<thead>
<tr>
<th>Op</th>
<th>ra</th>
<th>rc</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>27</td>
<td>26</td>
<td>22</td>
</tr>
</tbody>
</table>

**Example**
- `Idr r5, 8` (R[5] = M[PC + 8])
- `lar r6, 45` (R[6] = PC + 45)

#### 3. neg, not

<table>
<thead>
<tr>
<th>Op</th>
<th>ra</th>
<th>rc</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>27</td>
<td>26</td>
<td>22</td>
</tr>
</tbody>
</table>

**Example**
- `neg r7, r9` (R[7] = – R[9])

#### 4. br

<table>
<thead>
<tr>
<th>Op</th>
<th>ra</th>
<th>rb</th>
<th>rc</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>27</td>
<td>26</td>
<td>22</td>
<td>17 16 12 11</td>
</tr>
</tbody>
</table>

**Example**
- `brz r4, r0` (branch to R[4] if R[0] == 0)

#### 5. brl

<table>
<thead>
<tr>
<th>Op</th>
<th>ra</th>
<th>rb</th>
<th>rc</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>27</td>
<td>26</td>
<td>22</td>
<td>17 16 12 11</td>
</tr>
</tbody>
</table>

**Example**
- `brlnz r6, r4, r0` (R[6] = PC; branch to R[4] if R[0] ≠ 0)

#### 6. add, sub, and, or

<table>
<thead>
<tr>
<th>Op</th>
<th>ra</th>
<th>rb</th>
<th>rc</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>27</td>
<td>26</td>
<td>22</td>
<td>17 16 12 11</td>
</tr>
</tbody>
</table>

**Example**
- `add r0, r2, r4` (R[0] = R[2] + R[4])

#### 7a. shr, shra, shl, shic

<table>
<thead>
<tr>
<th>Op</th>
<th>ra</th>
<th>rb</th>
<th>rc</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>27</td>
<td>26</td>
<td>22</td>
<td>17</td>
</tr>
</tbody>
</table>

**Example**
- `shr r0, r1, #4` (R[0] = R[1] shifted right by 4 bits)

#### 7b. shl, shic

<table>
<thead>
<tr>
<th>Op</th>
<th>ra</th>
<th>rb</th>
<th>rc</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>27</td>
<td>26</td>
<td>22</td>
<td>17 16 12</td>
</tr>
</tbody>
</table>

**Example**

#### 8. nop, stop

<table>
<thead>
<tr>
<th>Op</th>
<th>ra</th>
<th>rb</th>
<th>rc</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>27</td>
<td>26</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Example**
- `stop`
## Tbl 2.4 Example Load & Store Instructions: Memory Addressing

- Address can be constant, constant+register, or constant+PC
- Memory contents or address itself can be loaded

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op</th>
<th>ra</th>
<th>rb</th>
<th>c1</th>
<th>Meaning</th>
<th>Addressing Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld r1, 32</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>32</td>
<td>R[1] ← M[32]</td>
<td>Direct</td>
</tr>
<tr>
<td>ld r22, 24(r4)</td>
<td>1</td>
<td>22</td>
<td>4</td>
<td>24</td>
<td>R[22] ← M[24+R[4]]</td>
<td>Displacement</td>
</tr>
<tr>
<td>st r4, 0(r9)</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>0</td>
<td>M[R[9]] ← R[4]</td>
<td>Register indirect</td>
</tr>
<tr>
<td>la r7, 32</td>
<td>5</td>
<td>7</td>
<td>0</td>
<td>32</td>
<td>R[7] ← 32</td>
<td>Immediate</td>
</tr>
<tr>
<td>lar r3, 0</td>
<td>6</td>
<td>3</td>
<td>–</td>
<td>0</td>
<td>R[3] ← PC</td>
<td>Register (!)</td>
</tr>
</tbody>
</table>

(Note use of `la` to load a constant)
## 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 r2, r3</td>
<td>Not (r2 = r3')</td>
</tr>
<tr>
<td>add ra, rb, rc</td>
<td>add r2, r3, r4</td>
<td>2’s complement addition</td>
</tr>
<tr>
<td>sub ra, rb, rc</td>
<td>sub r3,r4,r5</td>
<td>2’s complement subtraction</td>
</tr>
<tr>
<td>and ra, rb, rc</td>
<td>and r1,r2,r3</td>
<td>Logical and</td>
</tr>
<tr>
<td>or ra, rb, rc</td>
<td>or r2, r5, r6</td>
<td>Logical or</td>
</tr>
<tr>
<td>addi ra, rb, c2</td>
<td>addi r1, r3, #1</td>
<td>Immediate 2’s complement add</td>
</tr>
<tr>
<td>andi ra, rb, c2</td>
<td>andi r1,r2,#128</td>
<td>Immediate logical and</td>
</tr>
<tr>
<td>ori ra, rb, c2</td>
<td>ori r3,r1,#7</td>
<td>Immediate logical or</td>
</tr>
</tbody>
</table>

- Immediate subtract not needed since constant in addi may be negative
### Branch Instruction Format

There are actually only two branch instructions:

- **br** rb, rc, c3<2..0> ; branch to R[rb] if R[rc] meets the condition defined by c3<2..0>
- **brl** ra, rb, rc, c3<2..0> ; R[ra] ← PC; branch as above

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

<table>
<thead>
<tr>
<th>lsbs</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, brlzr</td>
<td>brzr r2, r4, r5</td>
</tr>
<tr>
<td>011</td>
<td>if rc ≠ 0</td>
<td>brnz, brlnz</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>if rc ≥ 0</td>
<td>brpl, brlpl</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>if rc &lt; 0</td>
<td>brmi, brlmi</td>
<td></td>
</tr>
</tbody>
</table>

- Note that branch target address is always in register R[rb].
- It must be placed there explicitly by a previous instruction.
## Tbl. 2.6 Branch Instruction Examples

<table>
<thead>
<tr>
<th>Ass’y lang.</th>
<th>Example instr.</th>
<th>Meaning</th>
<th>op</th>
<th>ra</th>
<th>rb</th>
<th>rc</th>
<th>Branch Cond’n.</th>
</tr>
</thead>
<tbody>
<tr>
<td>brlnv</td>
<td>brlnv r6</td>
<td>R[6] ← PC</td>
<td>9</td>
<td>6</td>
<td>—</td>
<td>—</td>
<td>000 never</td>
</tr>
<tr>
<td>br</td>
<td>br r4</td>
<td>PC ← R[4]</td>
<td>8</td>
<td>—</td>
<td>4</td>
<td>—</td>
<td>001 always</td>
</tr>
<tr>
<td>brzr</td>
<td>brzr r5,r1</td>
<td>if (R[1]=0) PC ← R[5]</td>
<td>8</td>
<td>—</td>
<td>5</td>
<td>1</td>
<td>010 zero</td>
</tr>
<tr>
<td>brlzr</td>
<td>brlzr r7,r5,r1</td>
<td>R[7] ← PC; if (R[1]=0) PC ← R[5]</td>
<td>9</td>
<td>7</td>
<td>5</td>
<td>1</td>
<td>010 zero</td>
</tr>
<tr>
<td>brnz</td>
<td>brnz r1, r0</td>
<td>if (R[0]≠0) PC ← R[1]</td>
<td>8</td>
<td>—</td>
<td>1</td>
<td>0</td>
<td>011 nonzero</td>
</tr>
<tr>
<td>brlnz</td>
<td>brlnz r2,r1,r0</td>
<td>R[2] ← PC; if (R[0]≠0) PC ← R[1]</td>
<td>9</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>011 nonzero</td>
</tr>
<tr>
<td>brpl</td>
<td>brpl r3, r2</td>
<td>if (R[2]≥0) PC ← R[3]</td>
<td>8</td>
<td>—</td>
<td>3</td>
<td>2</td>
<td>100 plus</td>
</tr>
<tr>
<td>brmi</td>
<td>brmi r0, r1</td>
<td>if (R[1]&lt;0) PC ← R[0]</td>
<td>8</td>
<td>—</td>
<td>0</td>
<td>1</td>
<td>101 minus</td>
</tr>
<tr>
<td>brlmi</td>
<td>brlmi r3,r0,r1</td>
<td>R[3] ← PC; if (r1&lt;0) PC ← R[0]</td>
<td>9</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>minus</td>
</tr>
</tbody>
</table>
Branch Instructions—Example

C: goto Label3

SRC:

lar r0, Label3 ; put branch target address into tgt reg.

br r0 ; and branch

Label3 • • •
Example of conditional branch

in C: \[ \text{if } (X<0) \text{ then } X = -X; \]

in SRC:

\[
\begin{align*}
&.org \ 1000 &; \text{next word will be loaded at address} \\
&1000_{10} \\
&X: \ .dw \ 1 &; \text{reserve 1 word for variable } X \\
&.org \ 5000 &; \text{program will be loaded at location} \\
&5000_{10} \\
&\text{lar } r0, \text{ Over} &; \text{load address of “false” jump location} \\
&\text{ld } r1, X &; \text{load value of } X \text{ into } r1 \\
&\text{brpl } r0, r1 &; \text{branch to Else if } r1 \geq 0 \\
&\text{neg } r1, r1 &; \text{negate value} \\
&\text{Over: \ \cdots} &; \text{continue}
\end{align*}
\]
SRC Simulator

- There is a Java-based assembler/simulator for the SRC that is available from the class web site.
- Run the simulator with the command:
  ```
  java -classpath SRCTools.jar.zip SRCTools.SRCSim
  ```
- This works on both UNIX and your PC.
  - On CEC’s UNIX machines, do a “pkgadd mgc” and then
    ```
    java -classpath $CLASS/ee362/src/SRCTools_jar.zip SRCTools.SRCSim
    ```
  - On your own PC, follow the instructions on the class web page.
Example SRC Simulation

Let’s simulate the conditional branch.

- Use *edit* to open an editor window.
- Enter your code
- Click on *assemble* to assemble your code. If all goes well you will have no errors. If you have errors, fix them before proceeding.
- Once you have no errors, click on *bin->sim* to load your binary file into the simulator.
- Run your simulation by *stepping* through your code one line at a time, or click *run* to run until done.
- Click *print* to print out the state of the simulator.
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)
RTN Notation (Cont’d.)

- At first you may find this “meta description” confusing, because it is a language that is used to describe a language.
- You will find that developing a familiarity with RTN will aid greatly in your understanding of new machine design concepts.
- We will describe RTN by using it to describe SRC.
Some RTN Features—Using RTN to describe a machine’s static 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.
Using RTN to describe Dynamic Properties

Dynamic Properties

• Conditional expressions:

\[(\text{op}=12) \rightarrow R[ra] \leftarrow R[rb] + R[rc]: \; \text{; defines the add instruction}\]

“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

Strt: start signal

R[0..31]<31..0>: general purpose registers
RTN Register Declarations

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

\[ R[0..31] \langle 31..0 \rangle : \]

- Name of registers
- Register # in square brackets
- .. specifies a range of indices
- msb #
- lsb#
- Bit # in angle brackets
- Colon separates statements with no ordering

Memory Declaration: RTN Naming Operator

- Defining names with formal parameters is a powerful formatting tool
- Used here to define word memory (big endian)

Main memory state

\[ \text{Mem}[0..2^{32} - 1]<7..0>: \quad 2^{32} \text{ addressable bytes of memory} \]

\[ \text{M}[x]<31..0]:= \text{Mem}[x]\#\text{Mem}[x+1]\#\text{Mem}[x+2]\#\text{Mem}[x+3]: \]

- Dummy parameter
- Naming operator
- Concatenation operator
- All bits in register if no bit index given
RTN Instruction Formatting Uses Renaming of IR Bits

Instruction formats

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>op&lt;4..0&gt; := IR&lt;31..27&gt;:</td>
<td>operation code field</td>
</tr>
<tr>
<td>ra&lt;4..0&gt; := IR&lt;26..22&gt;:</td>
<td>target register field</td>
</tr>
<tr>
<td>rb&lt;4..0&gt; := IR&lt;21..17&gt;:</td>
<td>operand, address index, or branch target register</td>
</tr>
<tr>
<td>rc&lt;4..0&gt; := IR&lt;16..12&gt;:</td>
<td>second operand, conditional test, or shift count register</td>
</tr>
<tr>
<td>c1&lt;21..0&gt; := IR&lt;21..0&gt;:</td>
<td>long displacement field</td>
</tr>
<tr>
<td>c2&lt;16..0&gt; := IR&lt;16..0&gt;:</td>
<td>short displacement or immediate field</td>
</tr>
<tr>
<td>c3&lt;11..0&gt; := IR&lt;11..0&gt;:</td>
<td>count or modifier field</td>
</tr>
</tbody>
</table>
Specifying dynamic properties of SRC: RTN Gives Specifics of Address Calculation

Effective address calculations (occur at runtime):

\[
\text{disp}^{31..0} := \begin{cases} 
(rb=0) & \rightarrow \ c2^{16..0} \{\text{sign extend}\}: \quad \text{displacement} \\
(rb \neq 0) & \rightarrow \ R[rb] + c2^{16..0} \{\text{sign extend, 2's comp.}\} : \quad \text{address} 
\end{cases}
\]

\[
\text{rel}^{31..0} := \text{PC}^{31..0} + c1^{21..0} \{\text{sign extend, 2's comp.}\} : \quad \text{relative address}
\]

- Renaming defines displacement and relative addr.
- New RTN notation is used
  - condition \(\rightarrow\) expression means \textit{if condition then} expression
  - modifiers in \{\ } describe type of arithmetic or how short numbers are extended to longer ones
  - arithmetic operators (+ - * / etc.) can be used in expressions
- Register R[0] cannot be added to a displacement
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

\[
\text{instruction_interpretation} := ( \\
\text{Run} \land \text{Strt} \rightarrow \text{Run} \leftarrow 1; \\
\text{Run} \rightarrow (\text{IR} \leftarrow \text{M}[\text{PC}]: \text{PC} \leftarrow \text{PC} + 4; \text{instruction_execution}) );
\]

- Logical NOT
- Logical AND

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

• Incomplete Specification:
  • 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 inst.
  - 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 inst.
RTN Instruction Execution for Load and Store Instructions

instruction_execution := ( 
    ld (:= op= 1) → R[ra] ← M[disp]: load register  
    ldr (:= op= 2) → R[ra] ← M[rel]: load register relative  
    st (:= op= 3) → M[disp] ← R[ra]: store register  
    str (:= op= 4) → M[rel] ← 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 ld := op=1 for the ld mnemonic
• The previous definitions of disp and rel are needed to understand all the details
SRC RTN—The Main Loop

ii := instruction_interpretation:
    ie := instruction_execution:

ii := ( ¬¬¬¬ ¬Run∧Strt → Run ← 1:
        Run → (IR ← M[PC]: PC ← PC + 4;
        ie) );

ie := ( ld (:= op= 1) → R[ra] ← M[disp]:
        ldr (:= op= 2) → R[ra] ← M[rel]:
        . . .
        stop (:= op= 31) → Run ← 0:
    ); ii

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

\[ \text{ld (:= op= 1) } \rightarrow \text{R[ra]} \leftarrow \text{M[disp]}: \]

\[ \text{disp}^{31..0} := ((\text{rb}=0) \rightarrow \text{c2}^{16..0} \{\text{sign extend}\}:
\]
\[ (\text{rb}\neq0) \rightarrow \text{R[rb]} + \text{c2}^{16..0} \{\text{sign extend, 2'}s \text{ comp.}\}) \]:

\[ \text{ld (:= op= 1) } \rightarrow \text{R[ra]} \leftarrow \text{M[}
\]
\[ ((\text{rb}=0) \rightarrow \text{c2}^{16..0} \{\text{sign extend}\}:
\]
\[ (\text{rb}\neq0) \rightarrow \text{R[rb]} + \text{c2}^{16..0} \{\text{sign extend, 2'}s \text{ comp.}\}) \):

\[ \]

- An example:
  - If \( \text{IR} = 00001 00101 00011 000000000000001011 \)
  - then \( \text{ld } \rightarrow \text{R[5]} \leftarrow \text{M[ R[3] + 11 ]}: \)
RTN Descriptions of SRC Branch Instructions

- Branch condition determined by 3 lsbs of inst.
- Link register (R[ra]) set to point to next inst.

\[
\text{cond} := (\ c3\langle2..0\rangle=0 \rightarrow 0: \quad \text{never} \\
\quad \quad c3\langle2..0\rangle=1 \rightarrow 1: \quad \text{always} \\
\quad \quad c3\langle2..0\rangle=2 \rightarrow R[rc]=0: \quad \text{if register is zero} \\
\quad \quad c3\langle2..0\rangle=3 \rightarrow R[rc] \neq 0: \quad \text{if register is nonzero} \\
\quad \quad c3\langle2..0\rangle=4 \rightarrow R[rc]\langle31\rangle=0: \quad \text{if positive or zero} \\
\quad \quad c3\langle2..0\rangle=5 \rightarrow R[rc]\langle31\rangle=1 ): \quad \text{if negative}
\]

br (:= op= 8) → (cond → PC ← R[rb]): conditional branch
brl (:= op= 9) → (R[ra] ← PC:
    cond → (PC ← R[rb])): branch and link
RTN for Arithmetic and Logic

add (:= op=12) → R[ra] ← R[rb] + R[rc]:
addi (:= op=13) → R[ra] ← R[rb] + c2〈16..0〉 {2's comp. sign ext.}:
sub (:= op=14) → R[ra] ← R[rb] - R[rc]:
neg (:= op=15) → R[ra] ← -R[rc]:
and (:= op=20) → R[ra] ← R[rb] ∧ R[rc]:
andi (:= op=21) → R[ra] ← R[rb] ∧ c2〈16..0〉 {sign extend}:
or (:= op=22) → R[ra] ← R[rb] ∨ R[rc]:
ori (:= op=23) → R[ra] ← R[rb] ∨ c2〈16..0〉 {sign extend}:
not (:= op=24) → R[ra] ← ¬R[rc]:
  • Logical operators: and ∧ or ∨ and not ¬
RTN for Shift Instructions

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

\[
\begin{align*}
n & := ( (c3\langle 4..0 \rangle = 0) \rightarrow R[ra]\langle 4..0 \rangle; \\
& \quad (c3\langle 4..0 \rangle \neq 0) \rightarrow c3\langle 4..0 \rangle ); \\
shr & := op=26 \rightarrow R[ra]\langle 31..0 \rangle \leftarrow (n \@ 0) \# R[rb]\langle 31..n \rangle; \\
shra & := op=27 \rightarrow R[ra]\langle 31..0 \rangle \leftarrow (n \@ R[rb]\langle 31 \rangle) \# R[rb]\langle 31..n \rangle; \\
shl & := op=28 \rightarrow R[ra]\langle 31..0 \rangle \leftarrow R[rb]\langle 31-n..0 \rangle \# (n \@ 0); \\
shc & := op=29 \rightarrow R[ra]\langle 31..0 \rangle \leftarrow R[rb]\langle 31-n..0 \rangle \# R[rb]\langle 31..32-n \rangle; \\
\end{align*}
\]
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

```
shra r1, r2, 13
```

```
R[2]= 1001 0111 1110 1010 1110 1100 0001 0110
```

```
R[1]= 1111 1111 1111 1 100 1011 1111 0101 0111
```
Assembly Language for Shift

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

shr ra, rb, rc ;Shift rb right into ra by 5 lsbs of rc
shr ra, rb, count ;Shift rb right into ra by 5 lsbs of inst
shra ra, rb, rc ;Ashift rb right into ra by 5 lsbs of rc
shra ra, rb, count ;Ashift rb right into ra by 5 lsbs of inst
shl ra, rb, rc ;Shift rb left into ra by 5 lsbs of rc
shl ra, rb, count ;Shift rb left into ra by 5 lsbs of inst
shc ra, rb, rc ;Shift rb circ. into ra by 5 lsbs of rc
shc ra, rb, count ;Shift rb circ. into ra by 5 lsbs of inst
End of RTN Definition of instruction_execution

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

}; End of instruction_execution

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)
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...
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.
## Addressing Modes Described in RTN (Not SRC)

<table>
<thead>
<tr>
<th>Mode name</th>
<th>Assembler</th>
<th>RTN meaning</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Ra</td>
<td>$R[t] \leftarrow R[a]$</td>
<td>Tmp. Var.</td>
</tr>
<tr>
<td>Register indirect</td>
<td>(Ra)</td>
<td>$R[t] \leftarrow M[R[a]]$</td>
<td>Pointer</td>
</tr>
<tr>
<td>Immediate</td>
<td>#X</td>
<td>$R[t] \leftarrow X$</td>
<td>Constant</td>
</tr>
<tr>
<td>Direct, absolute</td>
<td>X</td>
<td>$R[t] \leftarrow M[X]$</td>
<td>Global Var.</td>
</tr>
<tr>
<td>Indirect</td>
<td>(X)</td>
<td>$R[t] \leftarrow M[ M[X] ]$</td>
<td>Pointer Var.</td>
</tr>
<tr>
<td>Indexed, based, or displacement</td>
<td>X(Ra)</td>
<td>$R[t] \leftarrow M[X + R[a]]$</td>
<td>Arrays, structs</td>
</tr>
<tr>
<td>Relative</td>
<td>X(PC)</td>
<td>$R[t] \leftarrow M[X + PC]$</td>
<td>Vals stored w pgm</td>
</tr>
<tr>
<td>Autoincrement</td>
<td>(Ra)+</td>
<td>$R[t] \leftarrow M[R[a]]$; R[a] $\leftarrow R[a] + 1$</td>
<td>Sequential</td>
</tr>
<tr>
<td>Autodecrement</td>
<td>- (Ra)</td>
<td>$R[a] \leftarrow R[a] - 1$; $R[t] \leftarrow M[R[a]]$</td>
<td>access.</td>
</tr>
</tbody>
</table>

Target register
Fig. 2.11 Register transfers can be mapped to Digital Logic Circuits.

- Implementing the RTN statement \( A \leftarrow B \)
Fig. 2.12 Multiple Bit Register Transfer

- Implementing $A_{m..1} \leftarrow B_{m..1}$

Abbreviated notation
Fig. 2.13 Data Transmission View of Logic Gates

- Logic gates can be used to control the transmission of data:

![Data gate diagram]

Data gate

![Controlled complement diagram]

Controlled complement

![Data merge diagram]

Data merge

\[ \text{data} \rightarrow \text{gate} \rightarrow \text{data} \]

\[ \text{gate} \rightarrow 0 \]

\[ \text{control} \rightarrow \text{data} \]

\[ \text{control} \rightarrow \text{data} \]

\[ \text{data1(2), provided data2(1) is zero} \]
Fig. 2.14 Multiplexer as a 2 Way Gated Merge

- Data from multiple sources can be selected for transmission
Fig. 2.15  m-bit Multiplexer and Symbol

An n-way gated merge

(a) Multiplexer in terms of gates

(b) 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←C, and B←C can occur together, but not A←C, and B←D
Fig. 2.18 Open-Collector NAND Gate Output Circuit

(a) Open-collector NAND truth table

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Output</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0v 0v</td>
<td>Open</td>
<td>(Out = +V)</td>
</tr>
<tr>
<td>0v +V</td>
<td>Open</td>
<td>(Out = +V)</td>
</tr>
<tr>
<td>+V 0v</td>
<td>Open</td>
<td>(Out = +V)</td>
</tr>
<tr>
<td>+V +V</td>
<td>Closed</td>
<td>(Out = 0v)</td>
</tr>
</tbody>
</table>

(b) Open-collector NAND

(c) Symbol
Fig. 2.19 Wired AND Connection of Open-Collector Gates

(a) Wired AND connection

(b) With symbols

(c) Truth table

<table>
<thead>
<tr>
<th>Switch</th>
<th>Wired AND output</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>Closed (0)</td>
<td>Closed (0)</td>
</tr>
<tr>
<td>Closed (0)</td>
<td>Open (1)</td>
</tr>
<tr>
<td>Open (1)</td>
<td>Closed (0)</td>
</tr>
<tr>
<td>Open (1)</td>
<td>Open (1)</td>
</tr>
</tbody>
</table>
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

(b) Tri-state gate symbol

(c) Tri-state gate truth table

<table>
<thead>
<tr>
<th>Enable</th>
<th>Data</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Hi-Z</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Hi-Z</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Fig. 2.22 Registers Connected by a Tri-state Bus

- Can make any register transfer $R[i] \leftarrow R[j]$
- Can’t have $G_i = G_j = 1$ 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 Connected by One Bus

Example
Abstract RTN

Concrete RTN
\( Y \leftarrow R[2] \);
\( Z \leftarrow R[1]+Y \);
\( R[3] \leftarrow Z \);

Control Sequence
\( R[2]_{\text{out}}, Y_{\text{in}} \);
\( R[1]_{\text{out}}, Z_{\text{in}} \);
\( Z_{\text{out}}, R[3]_{\text{in}} \);

Notice that what could be described in one step in the abstract RTN took three steps on this particular hardware
RT’s 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] ← R[j] + R[k] is performed in 3 steps: Y ← R[k]; Z ← R[j] + Y; R[i] ← Z;
- R[i] ← R[j] + R[k] is high level RTN description
- Y ← R[k]; Z ← R[j] + Y; R[i] ← 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, enables, and multiplexers