CPE300: Digital System Architecture and Design

Fall 2011

MW 17:30-18:45 CBC C316

Simple RISC Computer

09122011

http://www.egr.unlv.edu/~b1morris/cpe300/

Outline

- Recap
- Addressing Modes
- Simple RISC Computer (SRC)
- Register Transfer Notation (RTN)

3,2,1,& 0 Address Instructions

- 3 address instruction
  - Specifies memory addresses for both operands and the result
  - R ← Op1 op Op2
- 2 address instruction
  - Overwrites one operand in memory with the result
  - Op2 ← Op1 op Op2
- 1 address instruction
  - Single accumulator register to hold one operand & the result (no address needed)
  - Acc ← Acc op Op1
- 0 address
  - Uses a CPU register stack to hold both operands and the result
  - TOS ← TOS op SOS (TOS is Top Of Stack, SOS is Second On Stack)

Example 2.1

- Evaluate `a = (b+c)*d-e`
- for 3- 2- 1- and 0-address machines

- What is size of program and amount of memory traffic in bytes?

Example 2.1

Instructions

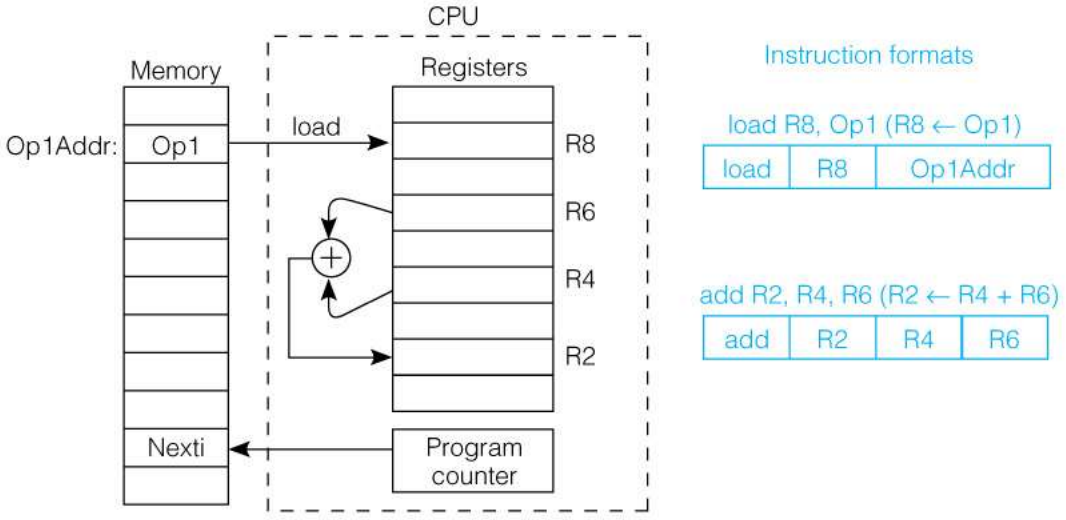
3-Address	2-Address	1-Address	0-Address
`add a, b, c`	`load a, b`	`lda b`	`push b`
`mult a, a, d`	`add a, c`	`add c`	`push c`
`sub a, a, e`	`mult a, d`	`mult d`	`add`
	`sub a, e`	`sub e`	`push d`
		`sta a`	`mult`
			`push e`
			`sub`
			`pop a`

Bytes size

	3-Address	2-Address	1-Address	0-Address
Instruction	30	28	20	23
Memory	27	33	15	15
Total	57	61	35	38

Fig. 2.8 General Register Machines

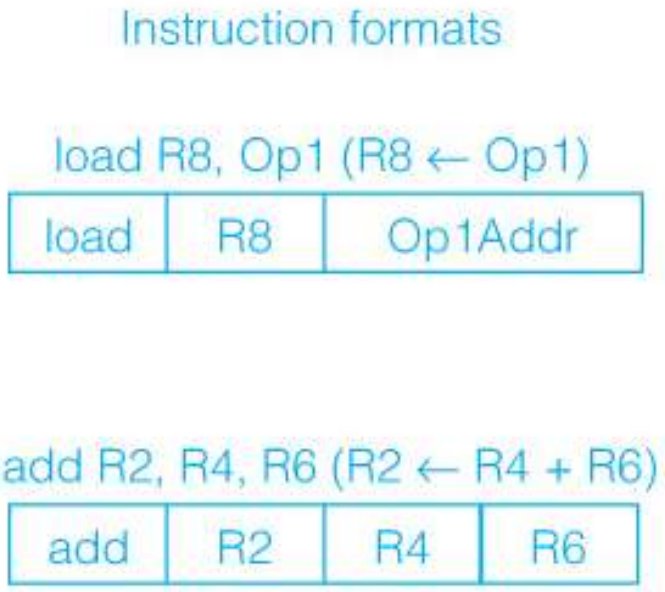
- Most common choice for general purpose computers
- Registers specified by "small" address (3 to 6 bits for 8 to 64 registers)
  - Close to CPU for speed and reuse for complex operations



1-1/2 Address Instructions

- "Small" register address = half address

- 1-1/2 addresses
  - Load/store have one long & one short address
  - 2-operand arithmetic instruction has 3 half addresses



Real Machines

- General registers offer greatest flexibility
  - Possible because of low price of memory
- Most real machines have a mixture of 3, 2, 1, 0, 1-1/2 address instructions
  - A distinction can be made on whether arithmetic instructions use data from memory
- Load-store machine
  - Registers used for operands and results of ALU instructions
  - Only load and store instructions reference memory
- Other machines have a mix of register-memory and memory-memory instructions

Instructions/Register Trade-Offs

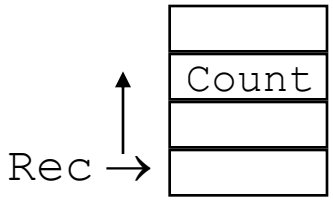
- 3-address machines have shortest code but large number of bits per instruction
- 0-address machines have longest code but small number of bits per instruction
  - Still require 1-address (push, pop) instructions
- General register machines use short internal register addresses in place of long memory addresses
- Load-store machines only allow memory addresses in data movement instructions (load, store)
- Register access is much faster than memory access
- Short instructions are faster

Addressing Modes

- 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
    - Record (struct) components:
      - variable base(full address) + const.(small)
    - Array components:
      - const. base(full address) + index var.(small)
- Possible to store constant values without 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`.



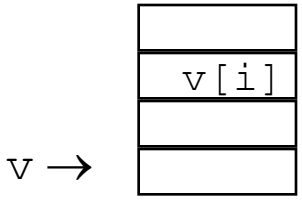


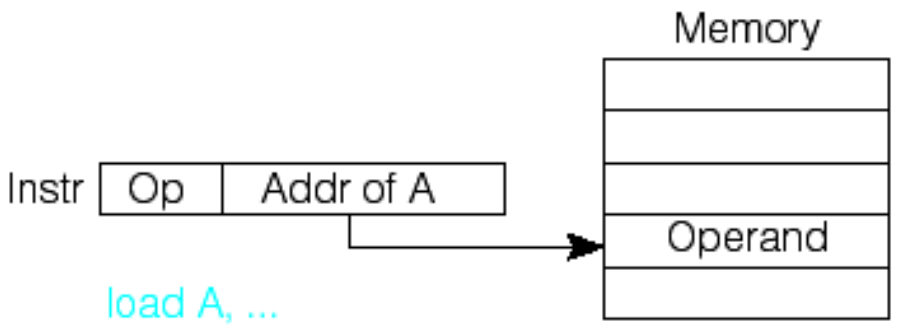
Fig 2.9 Common Addressing Modes a-d

(a) Immediate addressing:

instruction contains the operand

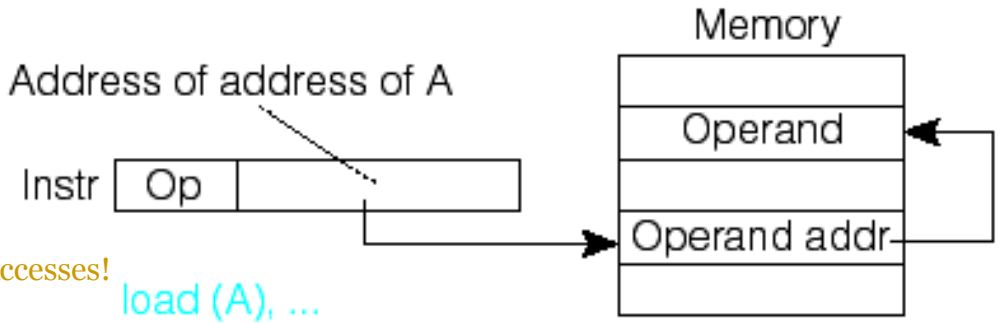
(b) Direct addressing:

instruction contains address of operand



(c) Indirect addressing:

instruction contains address of address of operand



Two Memory Accesses!

(d) Register direct addressing:

register contains operand

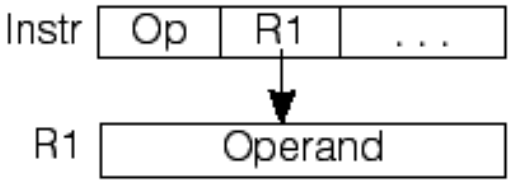
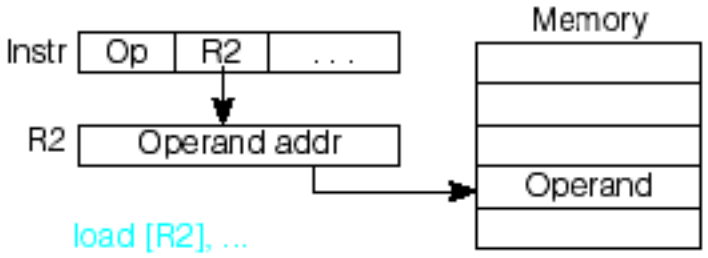


Fig 2.9 Common Addressing Modes e-g

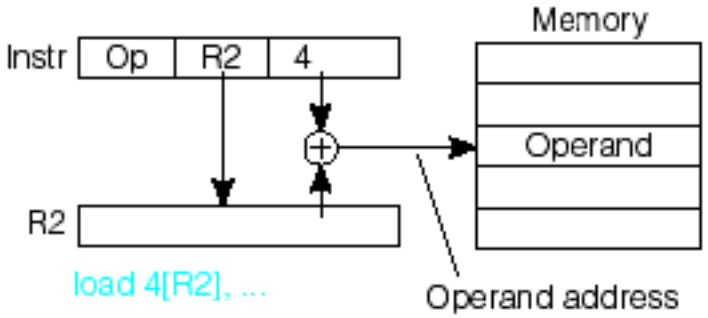
(e) Register indirect addressing:

register contains address of operand



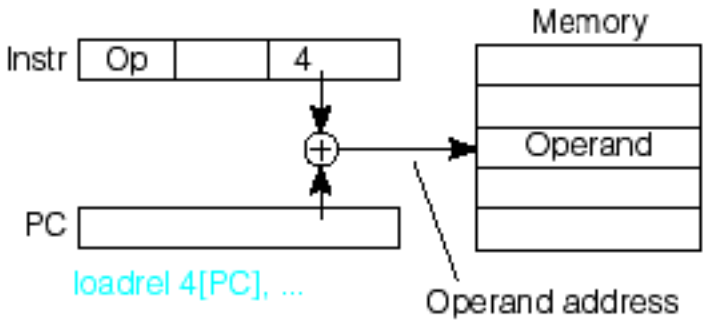
(f) Displacement (based or indexed) addressing:

address of operand = register + constant



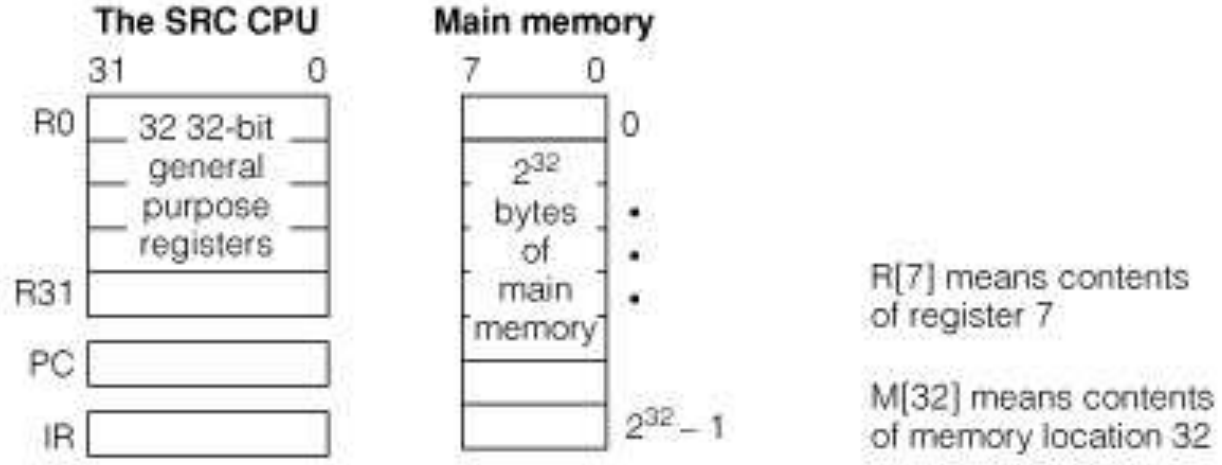
(g) Relative addressing:

address of operand = PC + constant



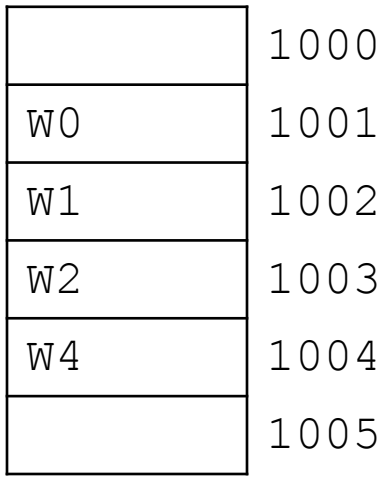
Simple RISC Computer (SRC)

- 32 general purpose registers (32 bits wide)
- 32 bit program counter (PC) and instruction register (IR)
- 2^{32} bytes of memory address space
- Use C-style array referencing for addresses



SRC Memory

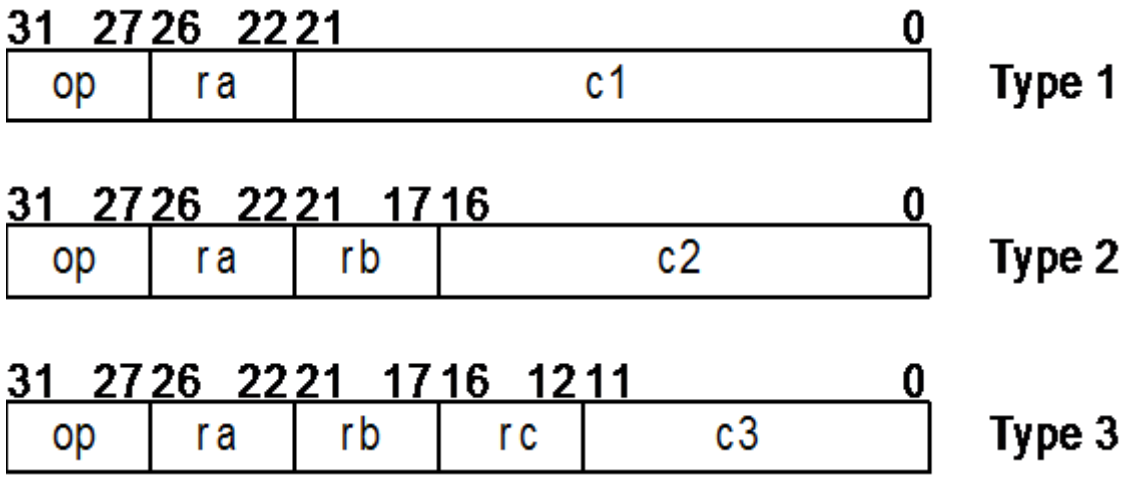
- 2^{32} bytes of memory address space
- Access is 32 bit words
  - 4 bytes make up word, requires 4 addresses
  - Lower address contains most significant bits (msb) – highest least significant bits (lsb)

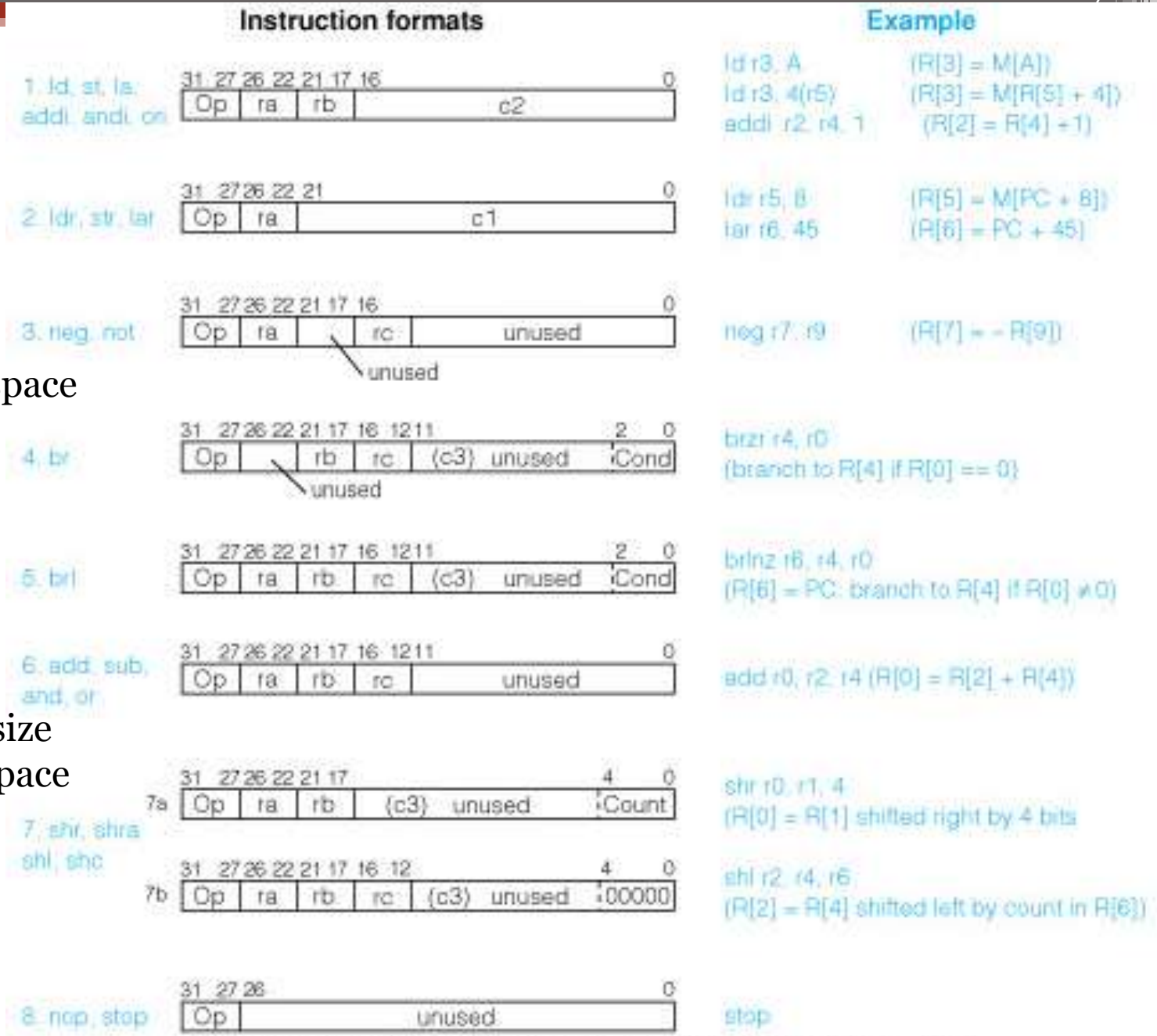


Bits	31	23	15	7	0
Address	1001	1002	1003	1004	
Value	W0	W1	W2	W3	

SRC Basic Instruction Formats

- There are three basic instruction format types
- The number of register specific fields and length of the constant field vary
- Other formats result from unused fields or parts





Notice the unused space

Trade-off between
- Fixed instruction size
- Wasted memory space

Ch3 -
single instruction
per clock cycle

SRC Characteristics

- (=) Load-store design - only memory access through load/store instructions
- (–) Operations on 32-bit words only (no byte or half-word operations)
- (=) Only a few addressing modes are supported
- (=) ALU instructions are 3-registertype
- (–) 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.
- (–) Can only branch to an address in a register, not to a direct address.
- (=) All instructions are 32-bits (1-word) long.

(=) – Similar to commercial RISC machines
(–) – Less powerful than commercial RISC machines

SRC Assembly Language

- Full Instruction listing available in Appendix B.5
- Form of line of SRC assembly code

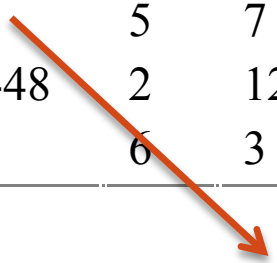
```
Label:      opcode      operands              ;comments
```

- `Label:` = assembly defined symbol
  - Could be constant, label, etc. – very useful but not always present
- `Opcode` = machine instruction or pseudo-op
- `Operands` = registers and constants
  - Comma separated
  - Values assumed to be decimal unless indicated (B, 0x)

SRC Load/Store Instructions

- Load/store design provides only access to memory
- Address can be constant, constant+register, or constant+PC
- Memory contents or address itself can be loaded

Instruction	op	ra	rb	c2	Meaning	Addressing Mode
ld r1, 32	1	1	0	32	R[1] ← M[32]	Direct
ld r22, 24(r4)	1	22	4	24	R[22] ← M[24+R[4]]	Displacement
st r4, 0(r9)	3	4	9	0	M[R[9]] ← R[4]	Register indirect
la r7, 32	5	7	0	32	R[7] ← 32	Immediate
ldr r12, -48	2	12	–	-48	R[12] ← M[PC -48]	Relative
lar r3, 0	6	3	–	0	R[3] ← PC	Register (!)



Note: use of la to load constant

SRC ALU Instructions

Format	Example	Meaning
neg ra, rc	neg r1, r2	;Negate (r1 = -r2)
not ra, rc	not r2, r3	;Not (r2 = r3´)
add ra, rb, rc	add r2, r3, r4	;2's complement addition
sub ra, rb, rc		;2's complement subtraction
and ra, rb, rc		;Logical and
or ra, rb, rc		;Logical or
addi ra, rb, c2	addi r1, r3, 1	;Immediate 2's complement add
andi ra, rb, c2		;Immediate logical and
ori ra, rb, c2		;Immediate logical or

- Note:
  - No multiply instruction (can be done based on addition)
  - Immediate subtract not needed since constant in addi may be negative (take care of sign bit)

SRC Branch Instruction

- Only 2 branch opcodes

```
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
```

- `c3<2..0>`, the 3 lsbs of c3, that define the branch condition

lsbs	condition	Assy language form	Example
000	never	brlnv	brlnv r6
001	always	br, brl	br r5, brl r5
010	if rc = 0	brzr, brlzr	brzr r2, r4
011	if rc ≠ 0	brnz, brlnz	
100	if rc ≥ 0	brpl, brlpl	
101	if rc < 0	brmi, brlmi	

- Note: branch target address is always in register `R[rb]`
  - Must be placed in register explicitly by a previous instruction

Branch Instruction Examples

Ass'y lang.	Example instr.	Meaning	op	ra	rb	rc	c3 ⟨2..0⟩	Branch Cond'n.
brlnv	brlnv r6	R[6] ← PC	9	6	—	—	000	never
br	br r4	PC ← R[4]	8	—	4	—	001	always
brl	brl r6,r4	R[6] ← PC; PC ← R[4]	9	6	4	—	001	always
brzr	brzr r5,r1	if (R[1]=0) PC ← R[5]	8	—	5	1	010	zero
brlzr	brlzr r7,r5,r1	R[7] ← PC;	9	7	5	1	010	zero
brnz	brnz r1, r0	if (R[0]≠0) PC← R[1]	8	—	1	0	011	nonzero
brlnz	brlnz r2,r1,r0	R[2] ← PC; if (R[0]≠0) PC← R[1]	9	2	1	0	011	nonzero
brpl	brpl r3, r2	if (R[2]□0) PC← R[3]	8	—	3	2	100	plus
brlpl	brlpl r4,r3,r2	R[4] ← PC; if (R[2]□0) PC← R[3]	9	4	3	2		plus
brmi	brmi r0, r1	if (R[1]<0) PC← R[0]	8	—	0	1	101	minus
brlmi	brlmi r3,r0,r1	R[3] ← PC; if (r1<0) PC← R[0]	9	3	0	1		minus

Unconditional Branch Example

- C code
  - `goto Label3`

- SRC

```
            lar r0, Label3    ;load branch target address into register r0
            br r0             ;branch
            …
Label3      …                 ;branch address
```

Conditional Branch Example

- C definition

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

- SRC assembly

```
        .org  0
Cost:   .equ  125          ;define symbolic constant
        .org  1000         ;next word loaded at address 1000(10)

X:      .dw   1            ;reserve 1 word for variable X
        .org  5000         ;program will be loaded at 5000(10)

        lar   r0, Over     ;load address of false jump locations
        ld    r1, X        ;get value of X into r1
        brpl  r0, r1       ;branch to r0 if r1 >= 0
        neg   r1, r1       ;negate r1 value
Over:   …
```

Pseudo-Operations

- Not part of ISA but assembly specific
  - Known as assembler directives
  - No machine code generated – for use by assembler, linker, loader

- Pseudo-ops
  - `.org` = origin
  - `.equ` = equate
  - `.dx` = define (word, half-word, byte)

Synthetic Instructions

- Single instruction (not in machine language) that assembler accepts and converts to single instruction in machine language
  - `CLR R0` `andi r0, r0, 0`
  - `MOVE D0, D1` `or r1, r0, r0`
    - (Other instructions possible besides `and` and `or`)
- Only synthetic instructions in SRC are conditional branches
  - `brzr r1, r2` `br r1, r2, 010`

if R[2] = 0

Miscellaneous Instructions

- `nop` – no operation
  - Place holder or time waster
  - Essential for pipelined implementations
- `stop`
  - Halts program execution, sets Run to zeros
  - Useful for debugging purposes

Register Transfer Notation (RTN)

- Provides a formal means of describing machine structure and function
  - Mix natural language and mathematical expressions
- Does not replace hardware description languages.
  - Formal description and design of electronic circuits (digital logic) – operation, organization, etc.
- Abstract RTN
  - Describes what a machine does without the how
- Concrete RTN
  - Describe a particular hardware implementation (how it is done)
- Meta-language = language to describe machine language

RTN Symbol Definitions (Appendix B.4)

←	Register transfer: register on LHS stores value from RHS
[]	Word index: selects word or range from named memory
<>	Bit index: selects bit or bit range from named memory
n..m	Index range: from left index n to right index m; can be decreasing
→	If-then: true condition of left yields value and/or action on right
:=	Definition: text substitution with dummy variables
#	Concatenation: bits on right appended to bits on left
:	Parallel separator: actions or evaluations carried out simultaneously
;	Sequential separator: RHS evaluated and/or performed after LHS
@	Replication: LHS repetitions of RHS are concatenated
{}	Operation modifier: information about preceding operation, e.g., arithmetic type
()	Operation or value grouping
= ≠ < ≤ ≥ >	Comparison operators: produce binary logical values
+ - ÷ ×	Arithmetic operators
∧ ∨ ¬ ⊕ ≡	Logical operators: and, or, not, xor, equivalence

Specification Language Notes

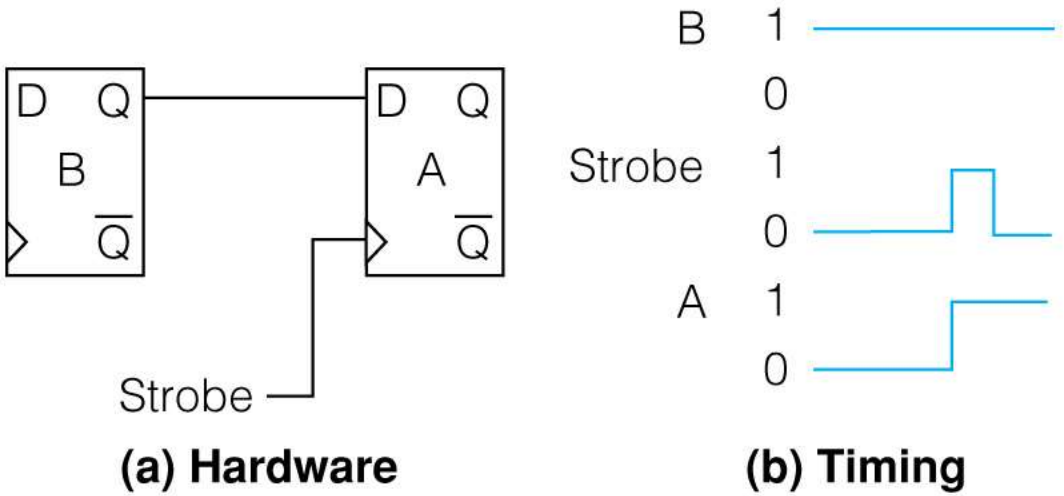
- They allow the description of *what* without having to specify *how*.
- They allow precise and unambiguous specifications, unlike natural language.
- They reduce errors:
  - errors due to misinterpretation of imprecise specifications written in natural language
  - errors due to confusion in design 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.

Logic Circuits in ISA

- Logic circuits
  - Gates (AND, OR, NOT) for Boolean expressions
  - Flip-flops for state variables
- Computer design
  - Circuit components support data transmission and storage as well

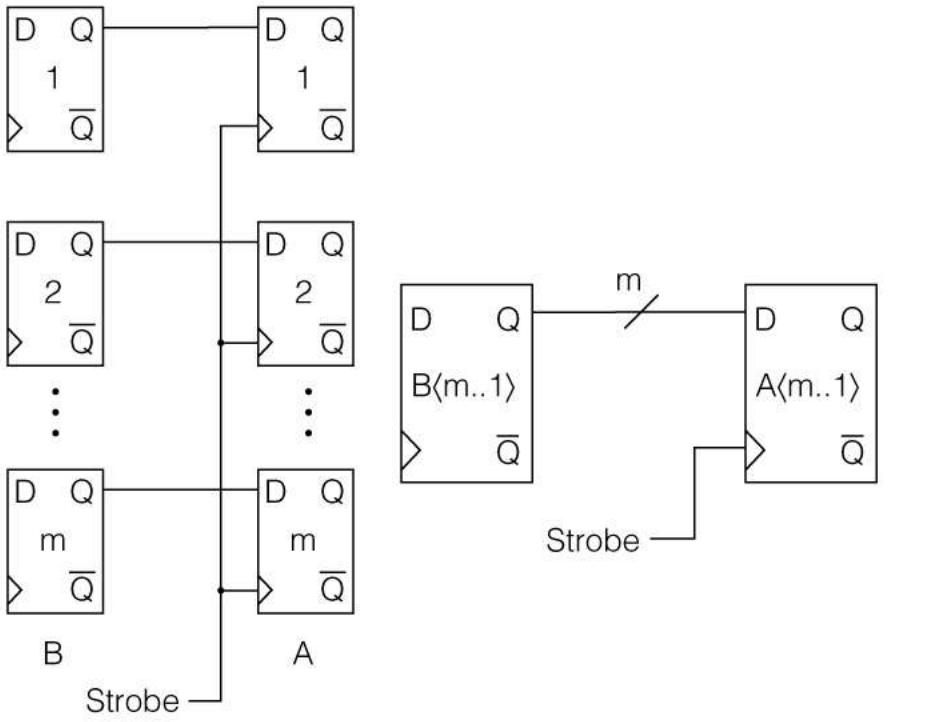
Logic Circuits for Register Transfer

- RTN statement `A` ← `B`



Multi-Bit Register Transfer

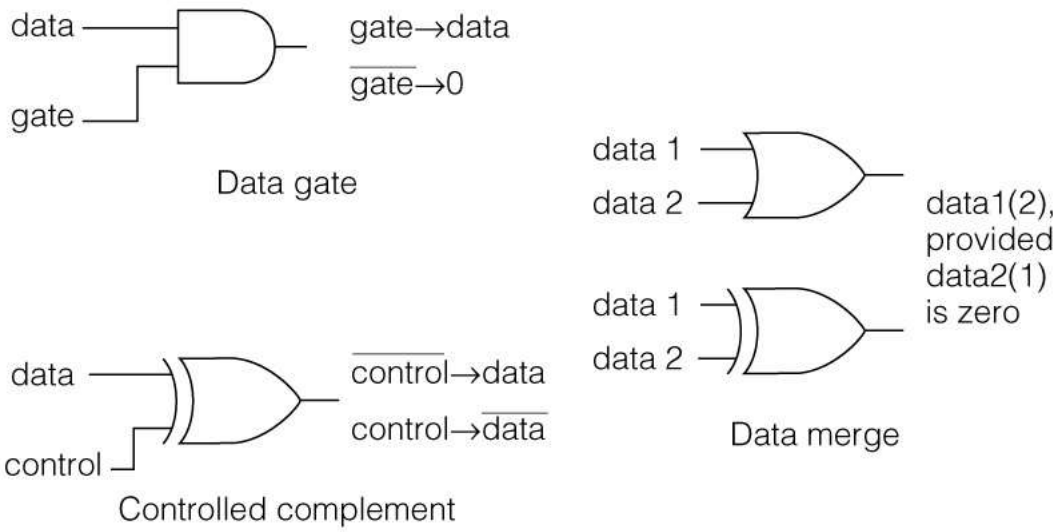
- Implementing `A<m..1>` ← `B<m..1>`



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

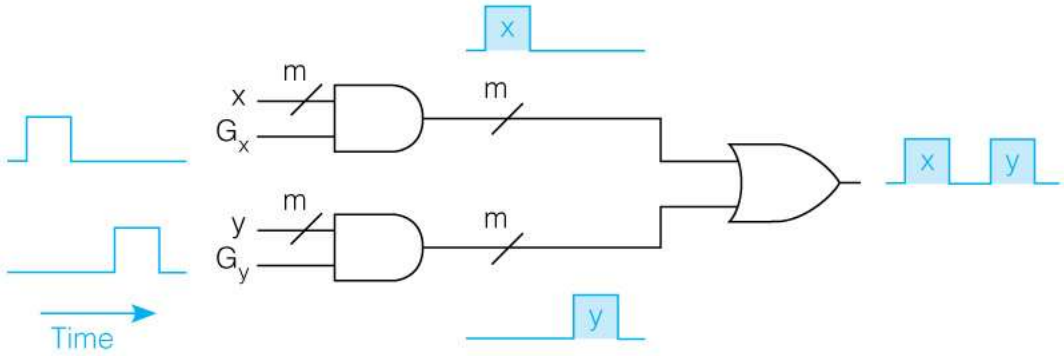
Logic Gates and Data Transmission

- Logic gates can control transmission of data

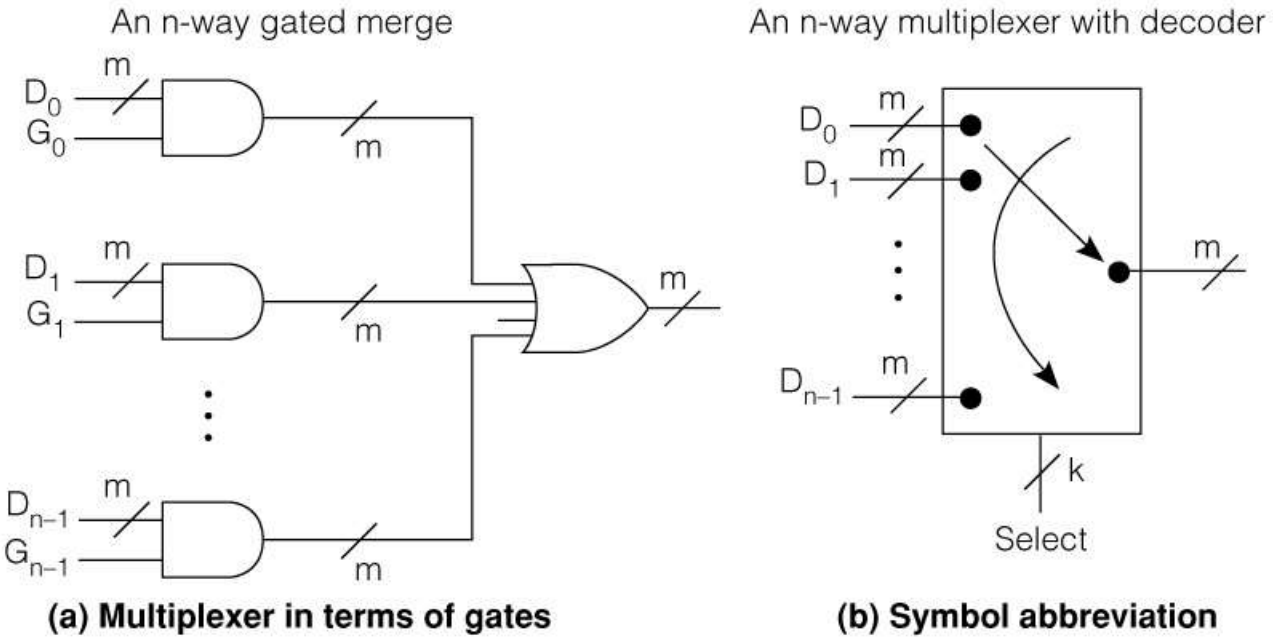


2-Way Multiplexer

- Data from multiple sources can be selected for transmission



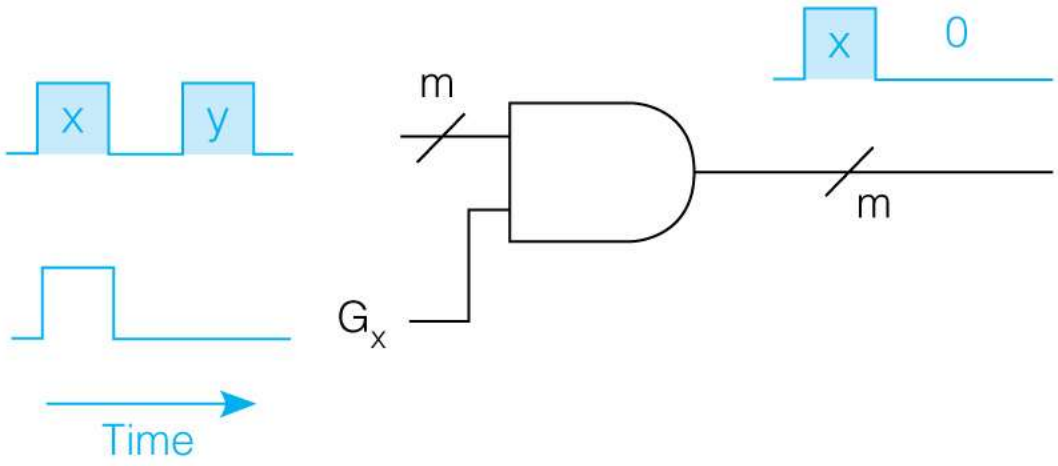
m-Bit Multiplexer



- Multiplexer gate signals Gi may be produced by a binary to one-out-of n decoder
  - How many gates with how many inputs?
  - What is relationship between k and n?

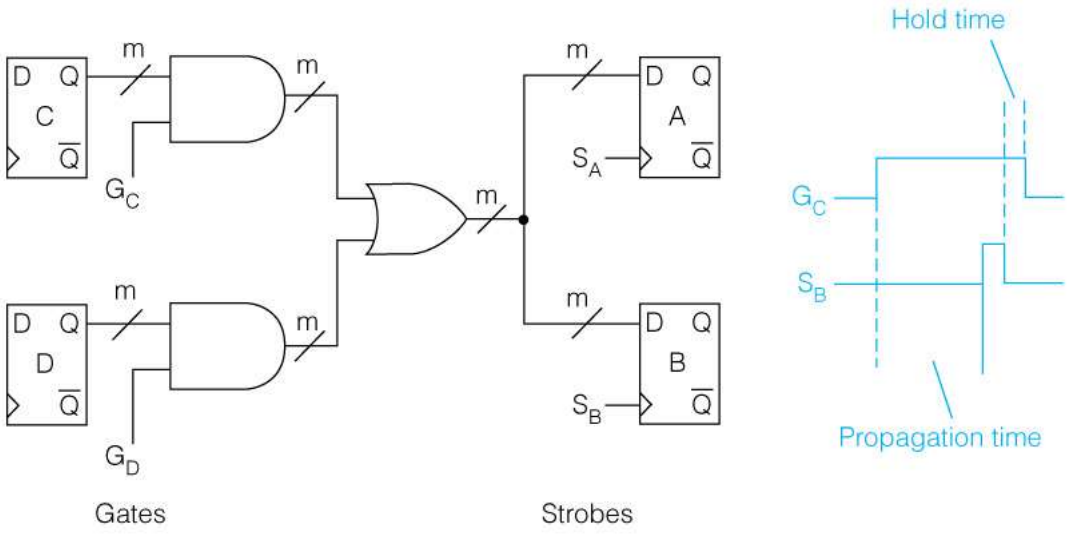
Separating Merged Data

- Merged data can be separated by gating at appropriate time
  - Can be strobed into a flip-flop when valid



Multiplexed Transfers using Gates and Strobes

- Selected gate and strobe determine which Register is transferred to where.
  - A←C, and B←C can occur together, but not A←C, and B←D

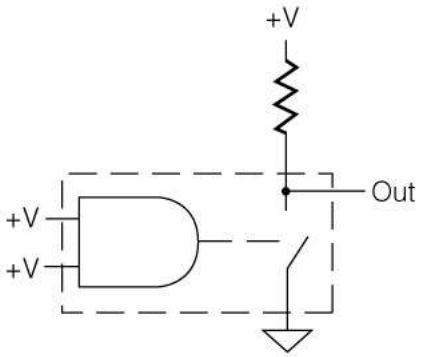


Open-Collector Bus

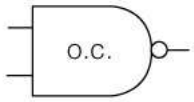
- Bus is a shared datapath (as in previous slides)
- Multiplexer is difficult to wire
  - Or-gate has large number of inputs (m x #gated inputs)
- Open-collector NAND gate to the rescue

Inputs		Output	
0v	0v	Open	(Out = +V)
0v	+V	Open	(Out = +V)
+V	0v	Open	(Out = +V)
+V	+V	Closed	(Out = 0v)

(a) Open-collector NAND truth table



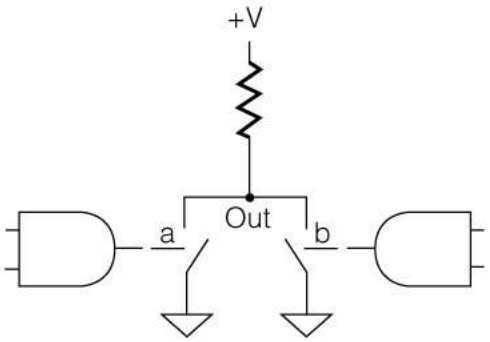
(b) Open-collector NAND



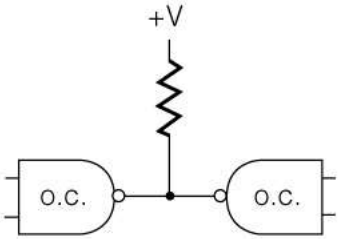
(c) Symbol

Wired AND Connection

- Connect outputs of 2 OC NAND gates
  - Only get high value when both gates are open



(a) Wired AND connection



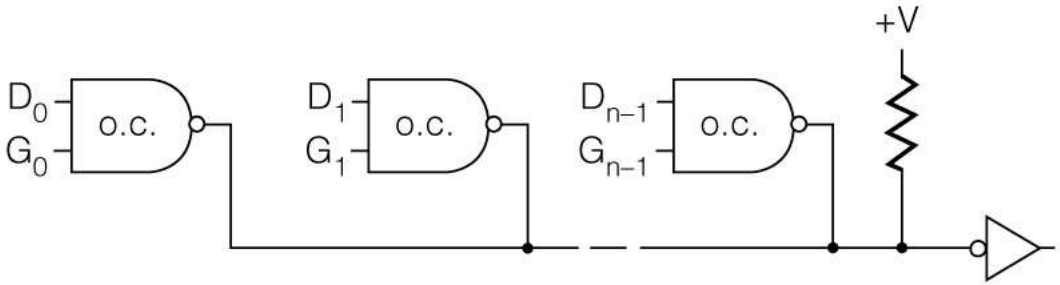
(b) With symbols

Switch		Wired AND output
a	b	
Closed(0)	Closed(0)	0v (0)
Closed(0)	Open (1)	0v (0)
Open (1)	Closed(0)	0v (0)
Open (1)	Open (1)	+V (1)

(c) Truth table

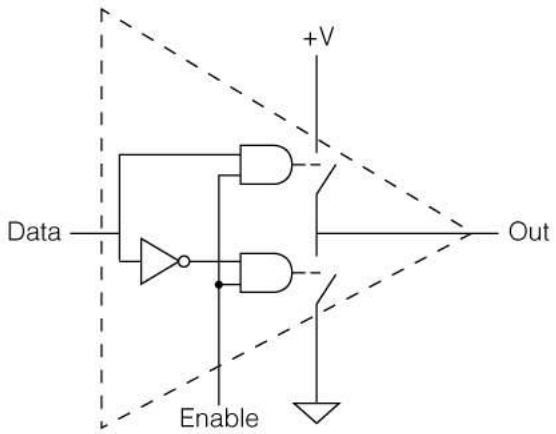
Wired-OR Bus

- Convert AND to OR using DeMorgan's Law
- Single pull-up resistor for whole bus
- OR distributed over the entire connection

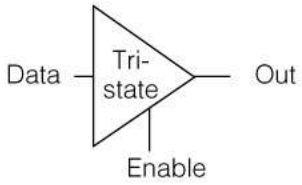


Tri-State Gate

- Controlled gating
  - Only one gate active at a time
  - Undefined output when not active



(a) Tri-state gate structure

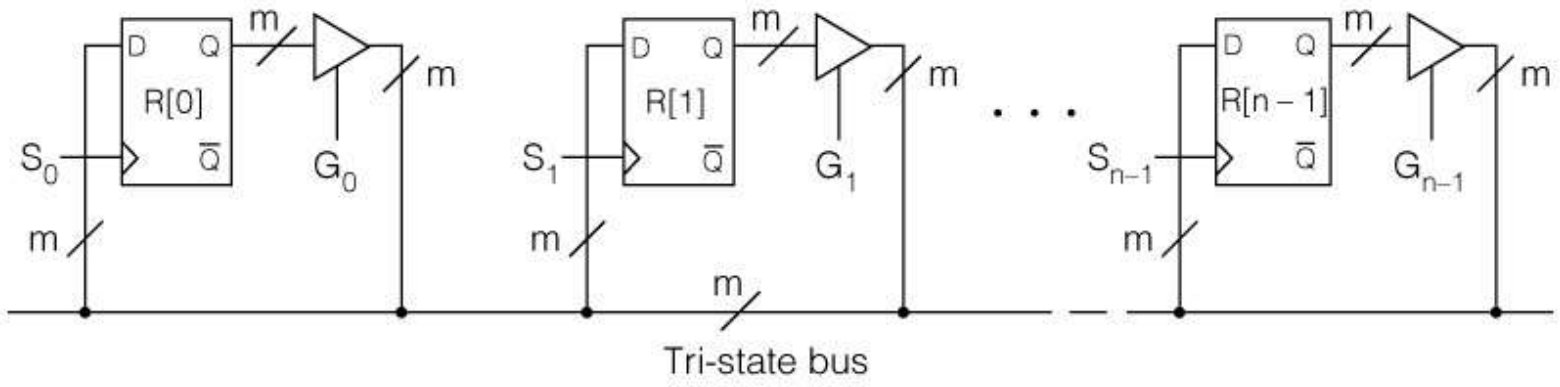


(b) Tri-state gate symbol

Enable	Data	Output
0	0	Hi-Z
0	1	Hi-Z
1	0	0
1	1	1

(c) Tri-state gate truth table

Tri-State Bus



- Can make any register transfer `R[i]` ← `R[j]`
- Only single gate may be active at a time
  - $G_i \neq G_j$

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