CEG2136
Computer Architecture I
Basic Computer Organization

Voicu Groza
SITE Hall, Room 5017
562 5800 ext. 2159
Groza@SITE.uOttawa.ca

uOttawa
L'Université canadienne
Canada's university

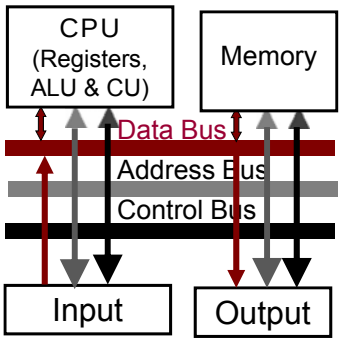
Université d'Ottawa | University of Ottawa

www.uOttawa.ca

Outline

- Instruction Codes
- Addressing Modes
- The Datapath of the Basic Computer
- Basic Computer Instructions
  - Register - Reference Instructions
  - Memory-Reference Instructions
  - Input-Output and Interrupt
- The Control Unit of the Basic Computer

Basic Operation of a Computer



1. The computer *accepts* information in the form of programs and data through an input unit and *stores* it in memory
2. The information stored in memory is *fetched*, under program control, and *processed* in an ALU
3. The processed information *leaves* the computer through an output unit
4. All activities inside the computer are *directed* by the control unit

Instruction Codes

- An **instruction code** is a group of bits (binary code) that instruct the computer to perform a sequence of micro-operations.
- Instruction codes together with data (operands) are stored in the computer memory.
- The control unit then interprets the binary code of the instruction and proceeds to execute it by deploying a sequence of micro-operations.
- An instruction code is usually divided in two parts: an **operation code (opcode)** and an **address** code.
- The operation code defines the operation to be performed: addition, subtraction, logic AND, etc.
- The address code usually (but not always) specifies the address of the operand.

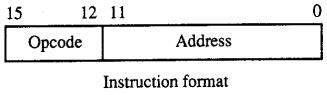
Instruction Codes

- The control unit receives the instruction code from the memory, interprets the operation code, and then issues a sequence of control signals to initiate the necessary sequence of micro-operations to be performed on the specified operands.
- An instruction code must specify not only the operation (defined by the operation code), but also the registers or memory words where the operands are to be found, as well as the register or memory word where the result is to be stored.
- A memory word can be specified within the instruction code by specifying its address in memory.
- A processor register can be specified within the instruction code by assigning k bits that identify one of the 2^k possible processor registers available in the system.
- In this chapter, we will discuss the basic organization, functionality, and design of a small-scale computer system.

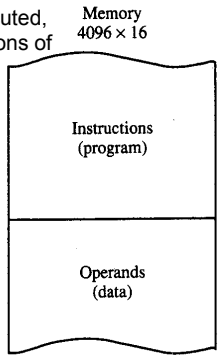
5

Stored Program Organization

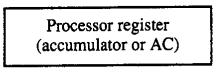
- A 4096 × 16 memory unit is used ⇒ 16-bit words.
- The memory is divided into two parts:
  - A program part, storing the program (set of instructions) to be executed,
  - A data part, storing the data (operands) to be used for the instructions of the program.



Instruction format

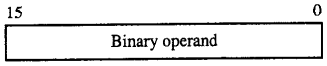


- Each **instruction** in the program part of the memory is divided in two parts:
  - The first part is composed of 4 bits (bits 12 to 15) and it specifies the operation code (**opcode**) of one of a maximum of $2^4 = 16$ possible operations that can be performed.
  - The second part is composed of 12 bits (bits 0 to 11) and it specifies the address of the operand in memory.
- The 12 bits in the address part of the instruction code can specify a maximum of $2^{12} = 4096$ different word addresses in the memory, which is the size of the memory used in our case.

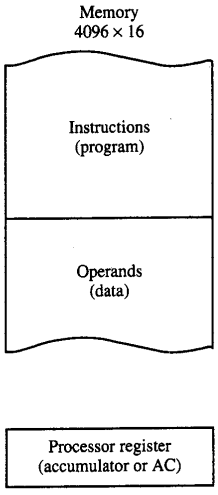


Stored Program Operation

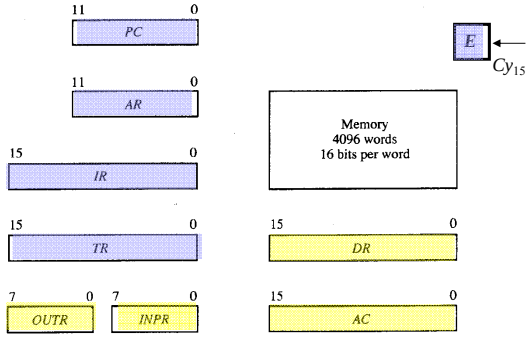
- Each word in the data part of the memory holds a 16-bit operand.
- The accumulator *AC* is a processor register that is usually used to store one operand of the operation to be performed.



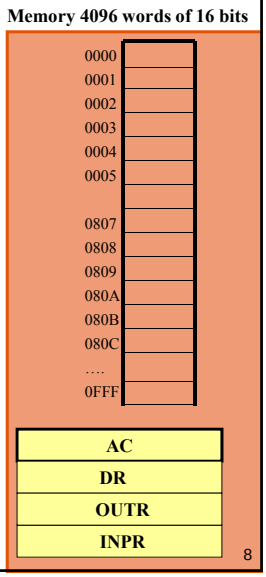
- A program instruction is executed in the following sequential order:
  1. The control unit reads the 16-bit instruction code from the program portion of the memory.
  2. The 12-bit address part of the instruction code is then used to read the 16-bit operand from the data portion of the memory.
  3. The 4-bit operation code is then used to perform the desired operation on the operand just read.



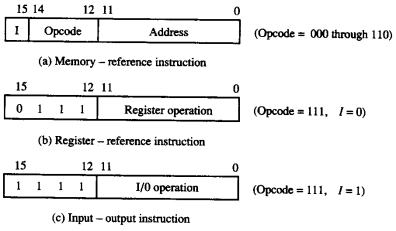
Basic Computer Registers & Memory



Register symbol	Number of bits	Register name	Function
DR	16	Data register	Holds memory operand
AC	16	Accumulator	Processor register
IR	16	Instruction register	Holds instruction code
TR	16	Temporary register	Holds temporary data
AR	12	Address register	Holds address for memory
PC	12	Program counter	Holds address of instruction
INPR	8	Input register	Holds input character
OUTR	8	Output register	Holds output character



Basic Computer Instruction List (Hex)



Symbol	Hexadecimal code $I = 0$	Hexadecimal code $I = 1$	Description
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load memory word to AC
STA	3xxx	Bxxx	Store content of AC in memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero
CLA	7800		Clear AC
CLE	7400		Clear E
CMA	7200		Complement AC
CME	7100		Complement E
CIR	7080		Circulate right AC and E
CIL	7040		Circulate left AC and E
INC	7020		Increment AC
SPA	7010		Skip next instruction if AC positive
SNA	7008		Skip next instruction if AC negative
SZA	7004		Skip next instruction if AC zero
SZE	7002		Skip next instruction if E is 0
HLT	7001		Halt computer
INP	F800		Input character to AC
OUT	F400		Output character from AC
SKI	F200		Skip on input flag
SKO	F100		Skip on output flag
ION	F080		Interrupt on
IOF	F040		Interrupt off

Addressing Modes

- Register Addressing
  - Operands reference internal registers
  - Also called *inherent addressing* (Motorola)
- Immediate Addressing
  - Operand is constant and is contained in the instruction word *immediately* following the opcode
- Direct Addressing
  - The Operand is to be fetched from the memory location whose address is given by the 12 bits following the opcode in the instruction word
- Indirect Addressing
  - Instruction specifies where the **address of the operand** is located

NOT IMPLEMENTED HERE:

- Indexed addressing
  - Finds the data address using an index (an offset)
- Relative Addressing
  - Add offset to current value of the PC

Notation used in Comments

- Register Name: Indicates a register and its contents
  - Example: AC refers to the contents of accumulator AC
- → (->) Right arrow indicates data transfer operation (<-> indicates an exchange of data)
  - Example: AC -> DR indicates the contents of AC are transferred (copied) to DR
- M[…] Contents of a memory location
  - Example: M[1234H] -> AC indicates that the contents of memory location HEX1234 are transferred to AC
- M[M[…]] Indirect addressing – inner parentheses specifies a memory address that contains the data address
  - Example: AC -> M[M[1234H]] indicates that the contents of AC are transferred to a memory location whose address is found in memory location with address 1234H (STA @1234H)

Inherent Addressing (Register Addressing)

- All data for instructions are in the CPU
- Instructions with inherent addressing
  - Are among the fastest to execute
  - Coded with the least amount of bits
- Also called Register Addressing or Implied Addressing

CLA ; 0 -> AC

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Opcode				Register Address											
CLA =7800$_H$	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0

INC ; AC <- AC+1

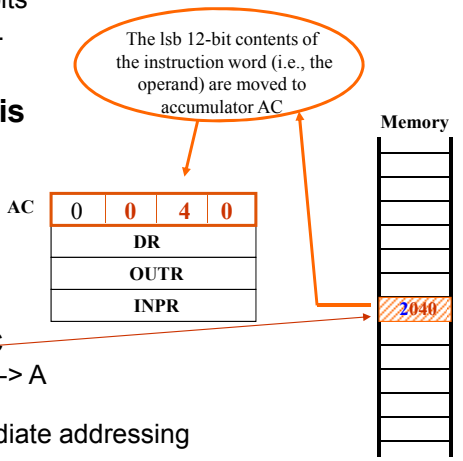
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Opcode				Register Address											
INC=7020$_H$	0	1	1	1	0	0	0	0	0	0	1	0	0	0	0	0

Immediate Addressing

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Opcode				Operand Value											
ADD=$1adr	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0

An argument is contained in the bits immediately following the opcode.

It is not implemented in this BASIC COMPUTER !!!



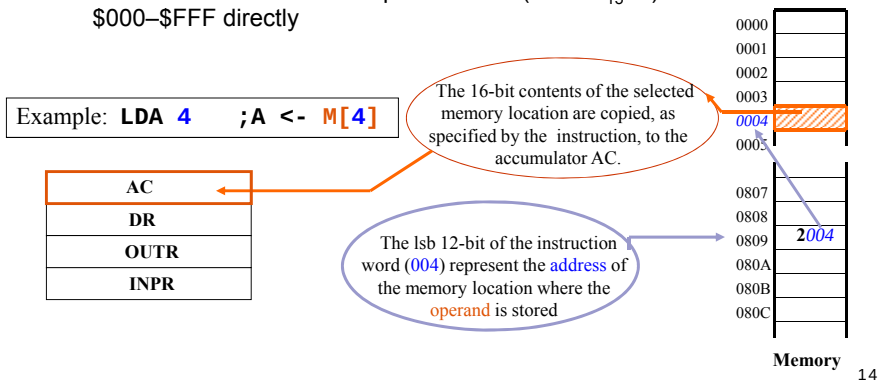
Immediate Addressing Examples

```
LDA   #64     ; Decimal 64 -> AC
LDA   #64H    ; Hexadecimal 64 -> A
```

- The # symbol indicates immediate addressing
- It is easy to forget the #

Direct Addressing

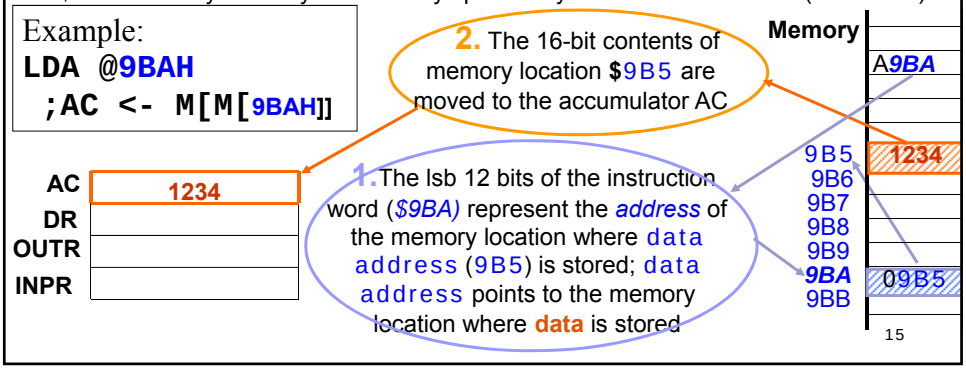
- Instruction contains the data address
  - Single-level addressing
  - Operand's address is given by the low-order 12 bits following the opcode.
  - Instructions encoded with opcodes 0 - 6 (msb IR_{15}=0) access addresses $000–$FFF directly



Indirect Addressing

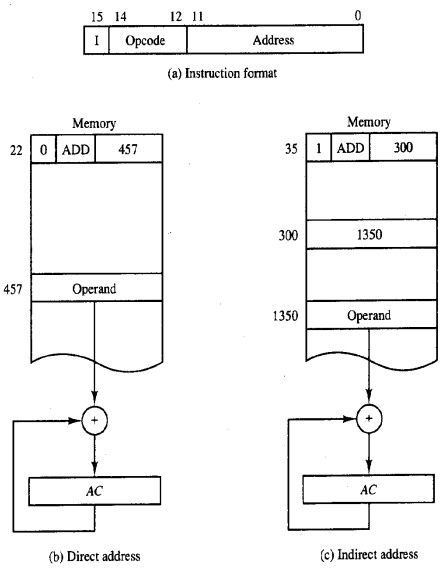
- Also called *pointer addressing*
- Instruction contains address of memory location holding the data address (*pointer*)
- Two level addressing mechanism
  - First level provided by instruction gives *address of memory containing an address*
  - Second level is the address that specifies where the data is located
  - Instructions encoded with opcodes 8 - E (msb IR_{15}=1) access addresses $000–$FFF indirectly. Actually the memory space they can address is 64kW (kilo words)

Example:
LDA @9BAH
;AC <- M[M[9BAH]]



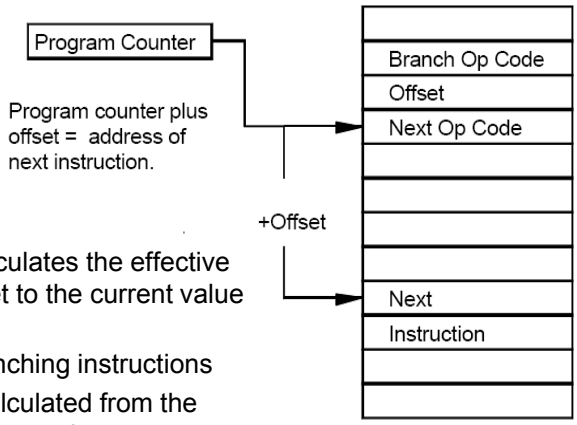
Direct VS Indirect Address

The instruction code is composed of a 3-bit opcode, a 12-bit address, and an indirect address mode bit *I* (*I* = 0 for a direct address, and *I* = 1 for an indirect one).
Fig. (b) shows a direct address mode (*I* = 0) with an opcode of an ADD operation. The 12-bit address code holds the binary equivalent of 457. ⇒ The operand is found in address 457 of the memory.
The operand is fetched and added to the number stored in the accumulator *AC*.
Fig. (c) shows an indirect address mode (*I* = 1) with an opcode of an ADD operation. The 12-bit address code holds the binary equivalent of 300. ⇒ The address of the operand is fetched from address 300, which contains the binary equivalent of 1350.
The operand is then fetched from address 1350 and added to the number stored in the accumulator *AC* .



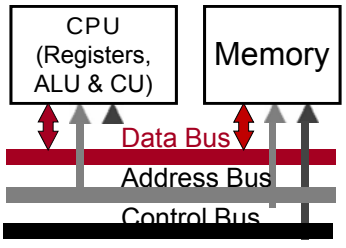
Relative Addressing

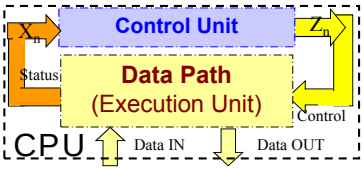
- The address of the operand is called the **effective address**.
- For instance, the effective address in the previous Fig. (b) is 457, whereas it is 1350 in Fig. (c).



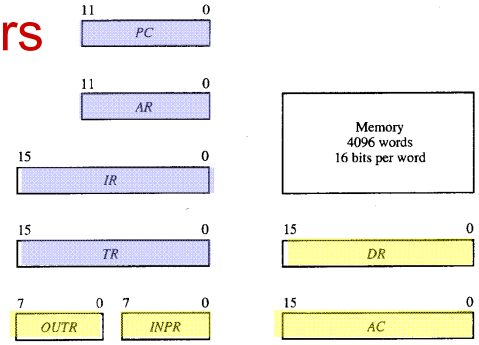
- Relative addressing calculates the effective address by adding offset to the current value of the PC
  - Used mostly for branching instructions
  - Normally offset is calculated from the address of the next op-code
- Not implemented in our BASIC Computer

Basic Computer





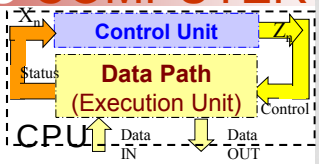
Computer Registers



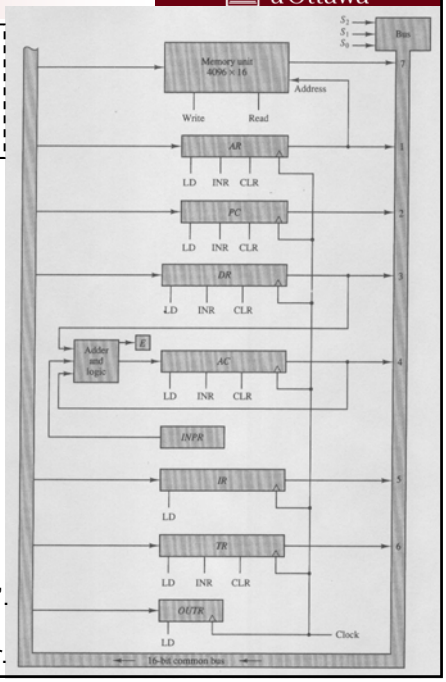
- The memory unit has a capacity of 4096 words with 16 bits each.
- As shown in Figure 29(a), each 16-bit instruction code is composed of 12 address bits, a 3-bit opcode, and one bit to specify a direct or indirect address.
- The data register (*DR*) holds the operar read from the memory.
- The accumulator (*AC*) is a general purpose processing register.
- The instruction register (*IR*) holds the instruction read from the memory.
- The temporary register (*TR*) holds temporary data during processing
- The address register (*AR*) is a 12-bit register that holds the address of the word to be accessed in the memory.
- The program counter (*PC*) is a 12-bit register that holds the address of the next instruction to be fetched from the memory after the current instruction is executed.
- *INPR* is an input register that holds an 8-bit code of a character read from an input device. *OUTR* is an output register that holds an 8-bit code of a character to be transferred to an output device.

19

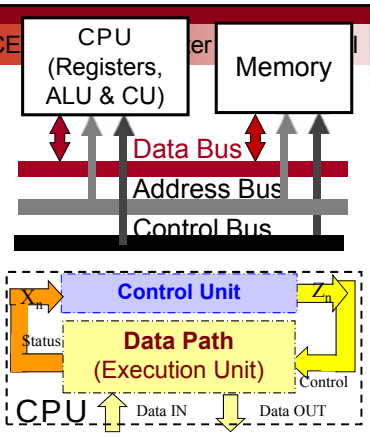
BASIC COMPUTER Data Path



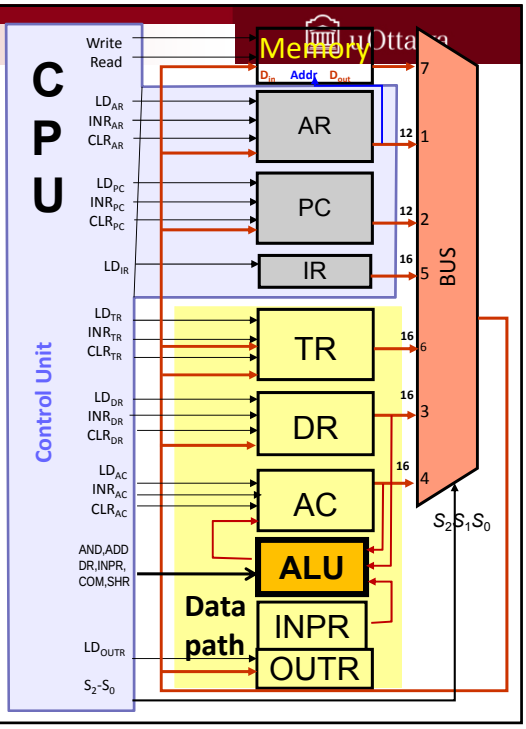
- The common bus lines are connected to the data input lines of each register as well as the data inputs of the memory.
- The particular register whose *LD* (load) input bit is enabled, receives the data from the bus during the next clock pulse.
- The content of the bus is loaded into the memory when its *Write* control bit is activated.
- The memory places one of its 16-bit words on the bus when its *Read* control bit is activated **and** $S_2S_1S_0 = 111$.
- Two of the registers connected to the bus have 12 bits only (*AR* and *PC*).
- When the content of *AR* or *PC* is applied to the 16-bit bus, the bus' 4 msb's are reset to '0'.
- When *AR* or *PC* loads the content of the bus, only the 12 lsb's are transferred to the register.



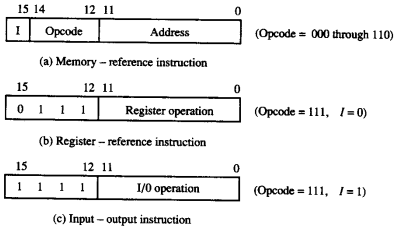
BASIC COMPUTER



- The basic computer has 8 registers, a memory unit, and a control unit, interconnected via a common bus.
- The bus selection bits S_2, S_1, and S_0, determine the output of the specific register or memory to be placed on the bus lines at any given time.
- The number along each BUS input in next figure shows the decimal equivalent of the required binary selection.



Basic Computer Instructions



There are 3 instruction formats.
The opcode is composed of 3 bits ($I_{14}I_{13}I_{12}$)
000-110 = operations to be performed on *AC*

- the rest of the12-bit (I_{11} - I_0) specify an address in memory:
- I_{15} = 0 => instruction with direct address
- I_{15} = 1 => instruction with indirect address

111 & I_{15} = 0 => a register reference instr.
111 & I_{15} = 1 => an input-output instruction. the other 12 bits (I_{11} - I_0) are used to specify the type of operation to be performed

Symbol	I=0	I= 1	Description
MRI			
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load memory word to AC
STA	3xxx	Bxxx	Store content of AC in memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero
RRI			
CLA	7800		Clear AC
CLE	7400		Clear E
CMA	7200		Complement AC
CME	7100		Complement E
CIR	7080		Circulate right AC and E
CIL	7040		Circulate left AC and E
INC	7020		Increment AC
SPA	7010		Skip next instruction if AC positive
SNA	7008		Skip next instruction if AC negative
SZA	7004		Skip next instruction if AC zero
SZE	7002		Skip next instruction if E is 0
HLT	7001		Halt computer
IOI			
INP		F800	Input character to AC
UT		F400	Output character from AC
SKI		F200	Skip on input flag
SKO		F100	Skip on output flag
ION		F080	Interrupt on
IOP		F040	Interrupt off

Basic Computer Instruction List (Binary)

D_7	Instr. Dec	I_{14}	I_{13}	I_{12}	I_{11}	I_{10}	I_9	I_8	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	I_{15}=0 MRI Direct Addr.	I_{15}=1 MRI Indirect Addr.
		op	code															
D_7=0 mem	D_0	0	0	0	AR_{11}	AR_{10}	AR_9	AR_8	AR_7	AR_6	AR_5	AR_4	AR_3	AR_2	AR_1	AR_0	**AND**=$0adr	AND_i =$8(adr)
	D_1	0	0	1	AR_{11}	AR_{10}	AR_9	AR_8	AR_7	AR_6	AR_5	AR_4	AR_3	AR_2	AR_1	AR_0	**ADD**=$1adr	ADD_i =$9(adr)
	D_2	0	1	0	AR_{11}	AR_{10}	AR_9	AR_8	AR_7	AR_6	AR_5	AR_4	AR_3	AR_2	AR_1	AR_0	**LDA** =$2adr	LDA_i =$A(adr)
	D_3	0	1	1	AR_{11}	AR_{10}	AR_9	AR_8	AR_7	AR_6	AR_5	AR_4	AR_3	AR_2	AR_1	AR_0	**STA** =$3adr	STA_i =$B(adr)
	D_4	1	0	0	AR_{11}	AR_{10}	AR_9	AR_8	AR_7	AR_6	AR_5	AR_4	AR_3	AR_2	AR_1	AR_0	**BUN** =$4adr	BUN_i =$C(adr)
	D_5	1	0	1	AR_{11}	AR_{10}	AR_9	AR_8	AR_7	AR_6	AR_5	AR_4	AR_3	AR_2	AR_1	AR_0	**BSA** =$5adr	BSA_i =$D(adr)
	D_6	1	1	0	AR_{11}	AR_{10}	AR_9	AR_8	AR_7	AR_6	AR_5	AR_4	AR_3	AR_2	AR_1	AR_0	**ISZ** =$6adr	ISZ_i =$E(adr)
																	RRI	**IOI**
D_7=1 reg	D_7	1	1	1	**1**	0	0	0	0	0	0	0	0	0	0	0	**CLA** =$7800	INP =$F800
	D_7	1	1	1	0	**1**	0	0	0	0	0	0	0	0	0	0	**CLE** =$7400	OUT=$F400
	D_7	1	1	1	0	0	**1**	0	0	0	0	0	0	0	0	0	**CMA** =$7200	SKI=$F200
	D_7	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	**CME** =$7100	SKO =$F100
	D_7	1	1	1	0	0	0	0	**1**	0	0	0	0	0	0	0	**CIR** =$7080	ION =$F080
	D_7	1	1	1	0	0	0	0	0	**1**	0	0	0	0	0	0	**CIL** =$7040	IOF =$F040
	D_7	1	1	1	0	0	0	0	0	0	**1**	0	0	0	0	0	**INC** =$7020	n/a
	D_7	1	1	1	0	0	0	0	0	0	0	**1**	0	0	0	0	**SPA** =$7010	n/a
	D_7	1	1	1	0	0	0	0	0	0	0	0	**1**	0	0	0	**SNA** =$7008	n/a
	D_7	1	1	1	0	0	0	0	0	0	0	0	0	**1**	0	0	**SZA** =$7004	n/a
	D_7	1	1	1	0	0	0	0	0	0	0	0	0	0	**1**	0	**SZE** =$7002	n/a
	D_7	1	1	1	0	0	0	0	0	0	0	0	0	0	0	**1**	**HLT** =$7001	n/a

Binary encoding — **One-hot** (*bit-per-state*) encoding

adr = 12 bit address
(adr) = address of the op address

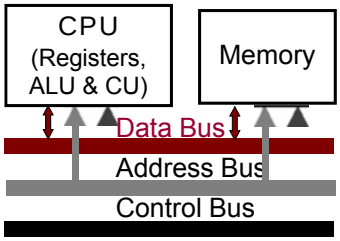
23

Instruction Set Completeness

- A computer instruction set is complete if it can be used to evaluate any function that is known to be computable.
- To be complete, the instruction set must contain enough instructions in each of the following categories:
  1. Arithmetic, logic, and shift instructions.
  2. Instructions for moving information to and from memory and processor registers.
  3. Program control instructions together with instructions that check status conditions.
  4. Input and output instructions
- Program control instructions, such as branch instructions, are used to change the sequence in which the program is executed.
- Input and output instructions are needed for communication between the computer and the user (outside world).
- In this context, the instruction list of the Basic Computer is complete as it provides enough instructions in each category.

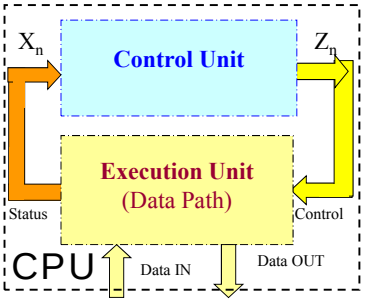
24

Top-Down Design



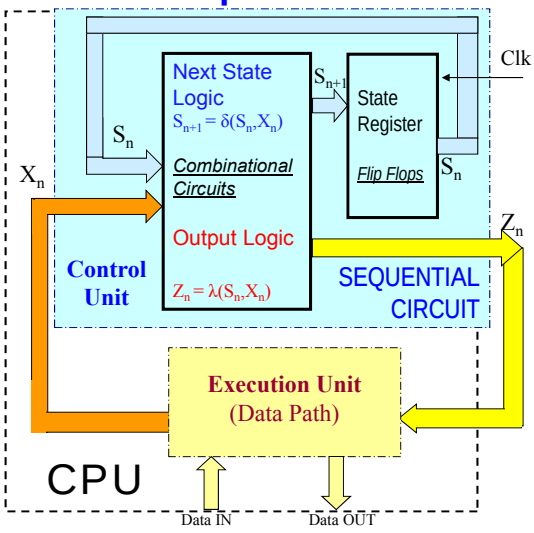
Central Processing Unit

CPU Block Diagram



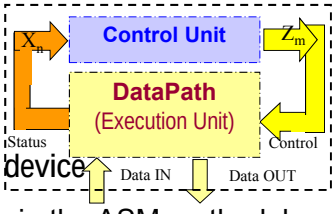
CPU **CONTROL UNIT**

as a Sequential Circuit



uOttawa

Algorithmic State Machine (ASM)



- The ASM method is utilized to solve a given problem, assuming that the target is a digital electronic **device,** consisting in a **datapath** and a **control unit**.
- The following are the five major steps in the ASM methodology:

ASM # 1. Using *pseudocode* describe the algorithm to be executed by the device

ASM # 2. Convert the pseudocode into an *ASM flowchart* (with ***RTL***)

ASM # 3. Design the *datapath* based on the ASM flowchart

ASM # 4. Create a *detailed ASM chart* (equivalent to FSM) with *control signals* that have to be generated by the *control unit* to direct the *datapath*

ASM # 5. Design the *control logic* based on the detailed ASM chart

- If followed correctly, the ASM method produces a hardware design in a systematic and logical manner
  - ☐ Very robust and easily modified
  - ☐ Refined through pseudocode iterations

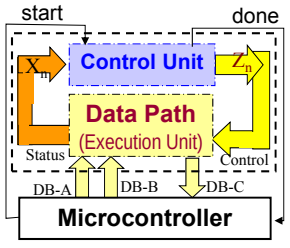
ASM Flowchart Rules (ASM#2)

- There are some rules that have to be abided by during ASM design
  1. State boxes should contain only RTL statements, control signals in parentheses, and state-transfer statements in parentheses
  2. All operations within a state box should be concurrently executable in one clock cycle
  3. If the operations in two consecutive state boxes can be executed in the same clock cycle, then the two state boxes can be combined into one state box
  4. Condition boxes should contain only simple queries that can be evaluated using purely combinational logic
  5. A new register must be assigned for each unique name in the set of RTN statements
  6. For each register-transfer statement, there must be a path between the source and the destination registers (if a transformation takes place during the transfer, then a combinational device, such as an adder or ALU, must be inserted into the path between the source and destination registers)
  7. If there are several paths leading into a combinational device or register, a multiplexer or tri-state buffer must be used
  8. For each simple binary query in a condition box, a combinational device or status signal must be used
  9. Finally, control signal inputs must be attached to each register and multiplexer so that register transfers can be precisely controlled

ASM Design Example

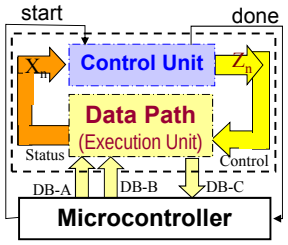
- The datapath of a dedicated coprocessor has three 16-bit registers *AR, BR,* and *CR*.
- When the coprocessor is triggered by a signal *start*, it will perform the following operations:
  - Transfer two 16-bit signed numbers (provided by the microcontroller over two data buses *DB-A* and *DB-B* in 2's-complement representation) to *AR* and *BR*.
  - If the number in *AR* is negative, divide the number in *AR* by 2 and transfer the result to register *CR*.
  - If the number in *AR* is positive but nonzero, multiply the number in *BR* by 2 and transfer the result to register *CR*.
  - If the number in *AR* is zero, clear register *CR* to 0.
  - Signalize the end of the processing by setting a flag *done*.

#1 Block Diagram

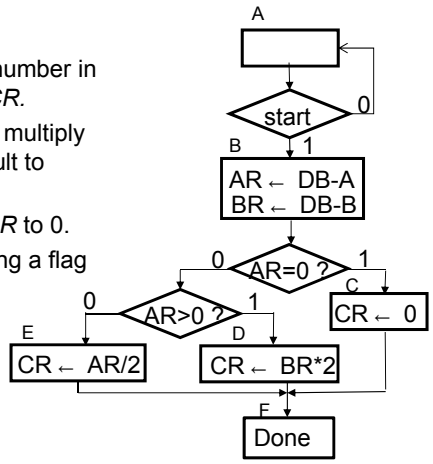


#1 Block Diagram

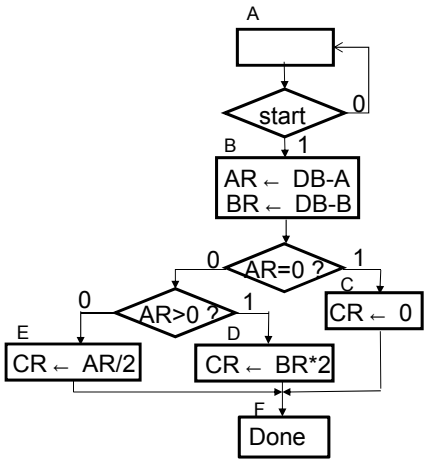
- Transfer two 16-bit signed numbers (provided by the microcontroller over two data buses *DB-A* and *DB-B* in 2's-complement representation) to *AR* & *BR*.
- If the number in *AR* is negative, divide the number in *AR* by 2 and transfer the result to register *CR*.
- If the number in *AR* is positive but nonzero, multiply the number in *BR* by 2 and transfer the result to register *CR*.
- If the number in *AR* is zero, clear register *CR* to 0.
- Signalize the end of the processing by setting a flag *done*.



#2 ASM Flowchart (High Level ***RTL***)

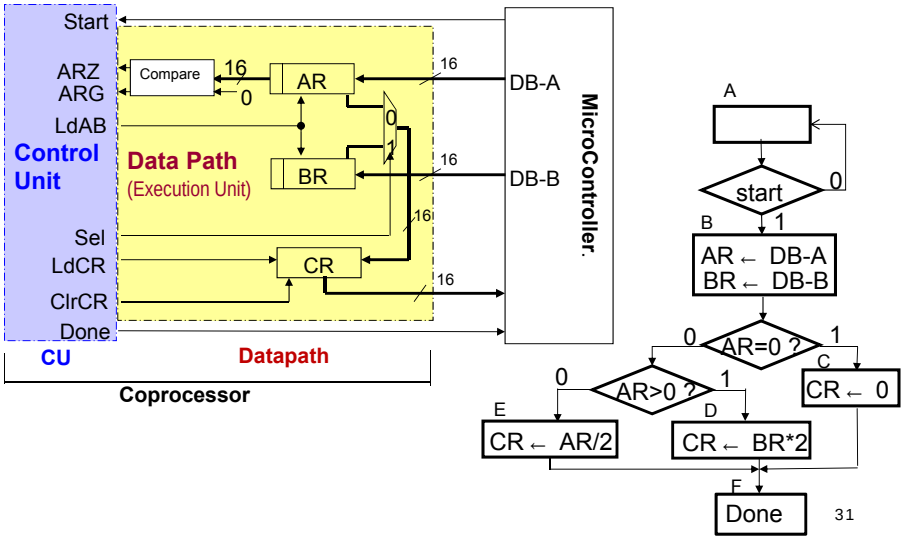


#2 ASM Flowchart (High Level ***RTL***)

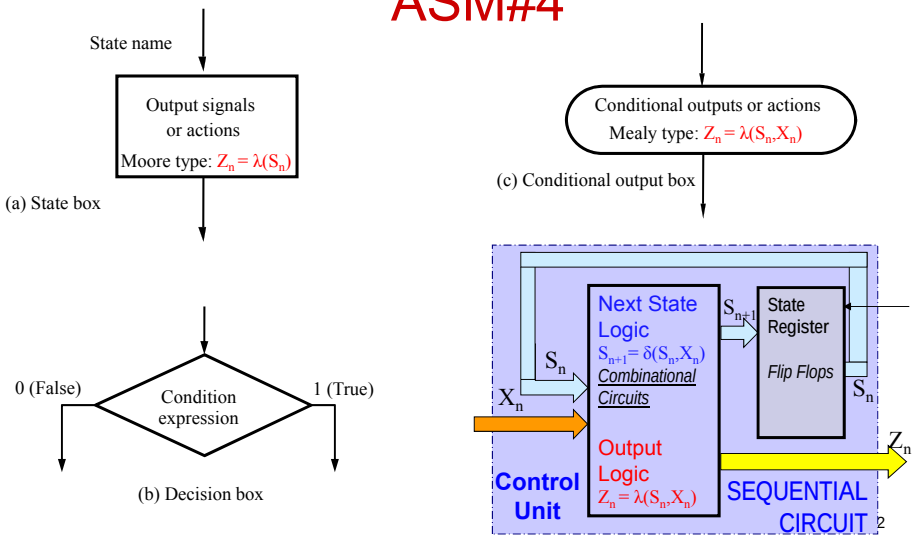


	$Z_n = \lambda(S_n)$	$S_{n+1} = \delta(S_n, X_n)$
A Start:		SC←B
B:	AR← DB-A, BR←DB-B	
B ARZ:		SC←C
B ARZ' ARG:		SC←D
B ARZ' ARG':		SC←E
C:	CR←0	SC←F
D:	CR←BR*2	SC←F
E:	CR←AR/2	SC←F
F:		SC←A

#3 ASM Datapath

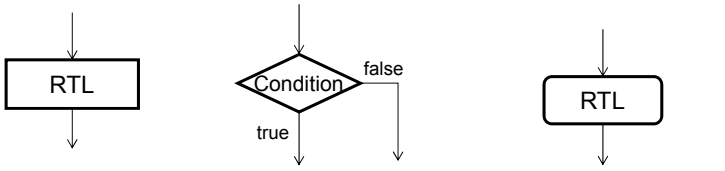


Symbols used in ASM detailed charts ASM#4



ASM Flowcharts Constructs (ASM#2)

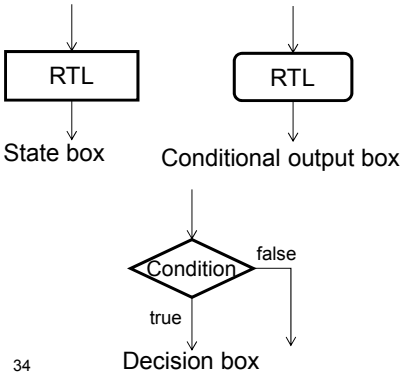
- Original ASM constructs consisted of **state** boxes, **condition** boxes and **conditional output** boxes
  - Conditional output boxes (Mealy FSM's) are difficult to represent in text or table ASM chart format, and allow conditional data transfers (RTL) … can be avoided
- ASM textual or tabular charts have replaced flowcharts because of their ease of modeling complexity and representing large synchronous sequential circuits



State box | Condition box | Conditional output box

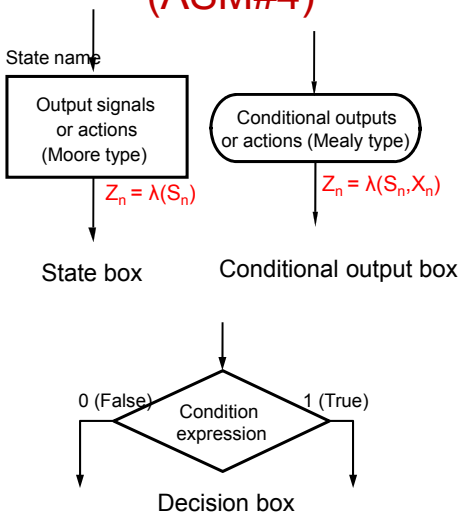
ASM Flowcharts (ASM#2)

ASM textual or tabular charts replace flowcharts because of their ease of modeling complexity and representing large synchronous sequential circuits.



State box | Conditional output box | Decision box

ASM Detailed Chart (ASM#4)



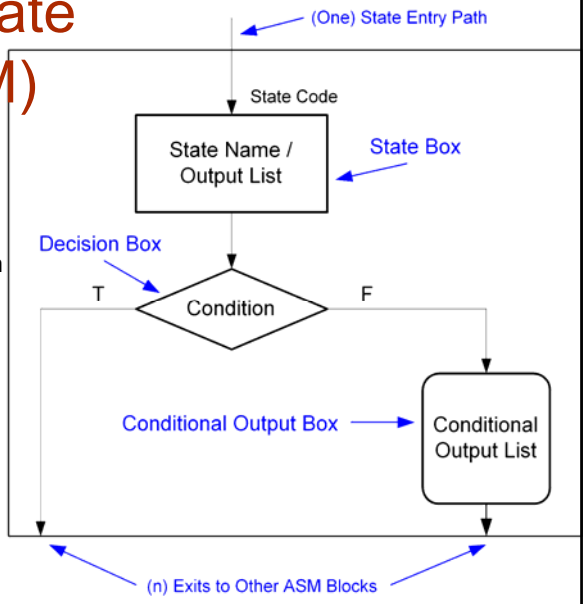
State box | Conditional output box | Decision box

Algorithmic State Machine (ASM)

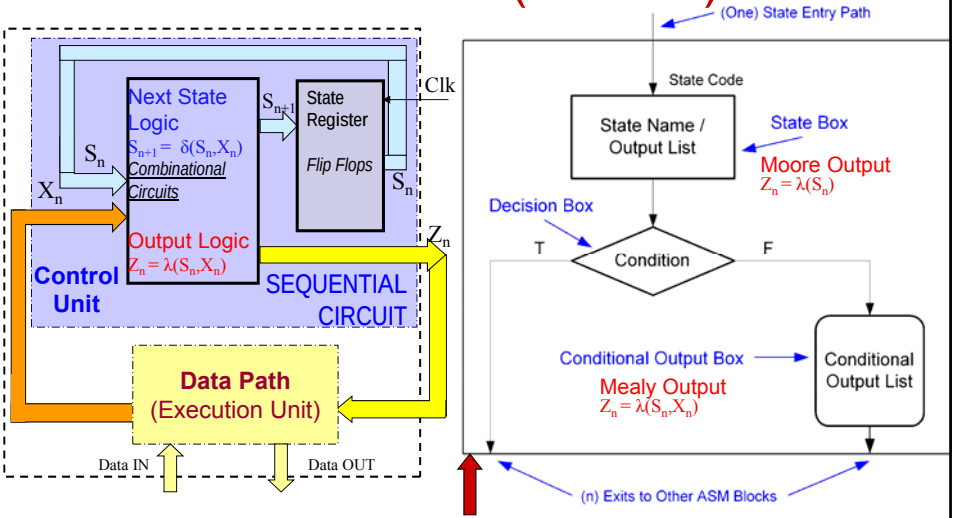
The following are the major steps in the ASM methodology:

- Create an algorithm, using *pseudocode*, to describe the desired operation of the device
- Convert the pseudocode into an *ASM chart* (the basic brick = **ASM block**)
- Design the *datapath* based on the ASM chart
- Create a *detailed ASM chart* based on the datapath
- Design the *control logic* based on the detailed ASM chart

Combination of **datapath** and **control logic** makes up the actual logic system that will solves the original problem.



ASM Detailed Chart (ASM#4)



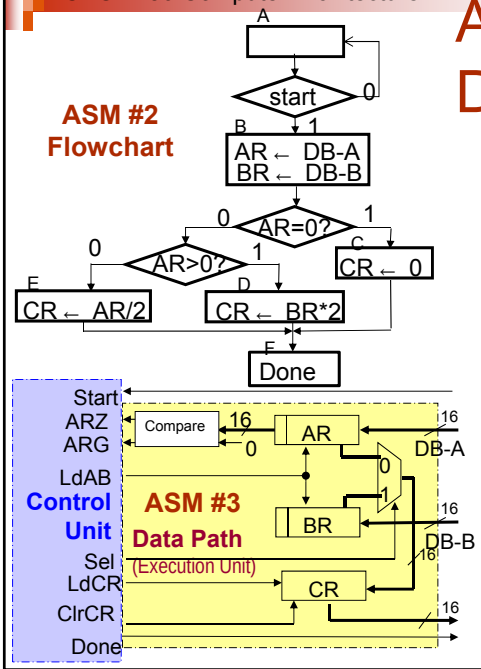
The basic brick = **ASM block**

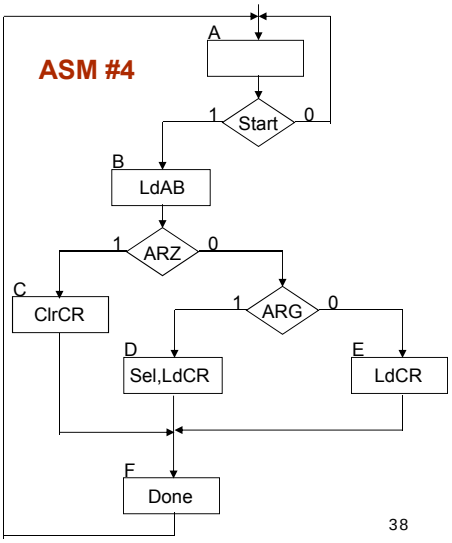
Combination of **datapath** and **control unit** makes up the actual logic system that will solve the original problem.

ASM Detailed Chart Rules (ASM#4)

- There are some rules that have to be abided by during detailed ASM chart design
  1. For each box in the ASM chart, there should be a corresponding box in the *detailed ASM chart*
  2. A state box should contain a comma-delimited list of entries of the form *csig* or *cond*: (*csig*,…,*csig*) where *csig* is a control signal or state transfer statement in parentheses and *cond* is a logical combination of status signals.
  3. For each RTL operation in a state box of an ASM chart, there should be a list of control signals that accomplishes the RTL operation in the corresponding state box of the detailed ASM chart. These control signals are generated by the *control unit* through its output function Z = λ(S,X) - as CU is a sequential circuit.
  4. All control signals are assumed to have the value '1' if specified and the value '0' if left unspecified
  5. For each condition box in the ASM chart, there should be a corresponding condition box in the detailed ASM chart
  6. Each condition box in the detailed ASM chart must contain a logical combination of status signals that implements the combinational logic query in the corresponding ASM chart condition box
- Rules 1-6 provide the mapping between the ASM chart and the detailed ASM chart
- Pay special attention to the state box and conditional box mappings!
- Control logic is the heart of the digital circuit, and is often the most complex and detailed block of the entire system, and thus must be designed **very carefully!**
- The detailed ASM chart should easily be converted to a control path, as long as the aforementioned rules are followed

ASM #4 Detailed chart



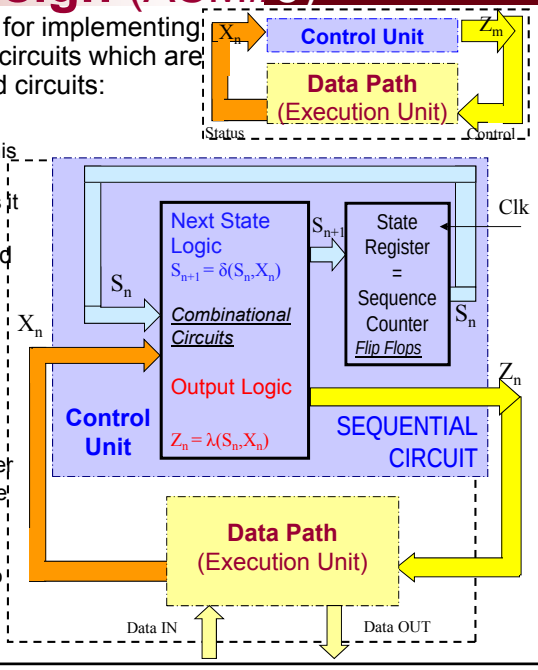


Control Unit Design (ASM#5)

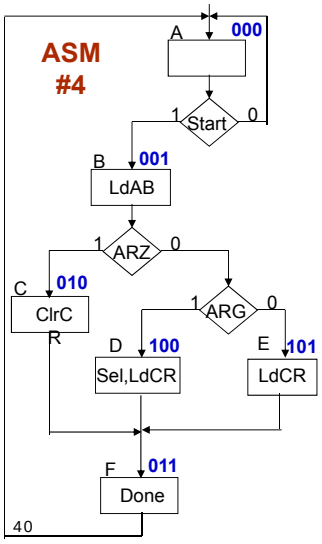
There are three major methods for implementing the State Register of sequential circuits which are used as control logic ASM-based circuits:

1. The **one-FF-per-state** method
   - Also called One-hot, we will see this method during this lecture
   - Most popular design technique, as it is simple and easy to model
2. The **Binary Encoded State** method
   - Uses state encoding to reduce the number of D flip-flops required in the state register
   - Control logic is spread out over several different digital devices
3. The **sequence-counter** method
   - Matches the cyclical nature of instruction execution on a computer
   - Control is simplified because of the cyclical state transition pattern

The **PLD-based** method
- Amalgamates all logic devices into one single VLSI chip
- Could contain both the datapath and the control logic!



ASM #5.3



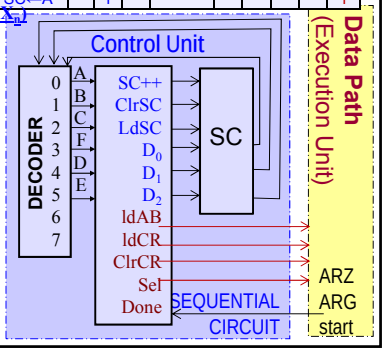
RTL			δ				λ				
	$Z_n = \lambda$	$S_{n+1} = \delta$	SC++	ClrSC	LdSC	SC_{n+1} $D_2D_1D_0$	LdAB	LdCR	ClrCR	Sel	Done
A Start:		SC←B	1								
B:	AR← DB-A, BR←DB-B						1				
B ARZ:		SC←C	1								
B·ARZ'·ARG:		SC←D			1	100					
B·ARZ'ARG':		SC←E			1	101					
C:	CR←0	SC←F	1						1		
D:	CR←BR*2	SC←F			1	011		1		1	
E:	CR←AR/2	SC←F			1	011		1			
F:		SC←A		1							1

Next State Logic $S_{n+1} = \delta(S_n, X_n)$

SC++ = A·start + B·ARZ+C
ClrSC = F
LdSC = B·ARZ'·ARG + B·ARZ'ARG'+D+E
D_0 = B·ARZ'ARG'+D+E
D_1 = D+E
D_2 = B·ARZ'·ARG + B·ARZ'ARG

Output Logic $Z_n = \lambda(S_n)$

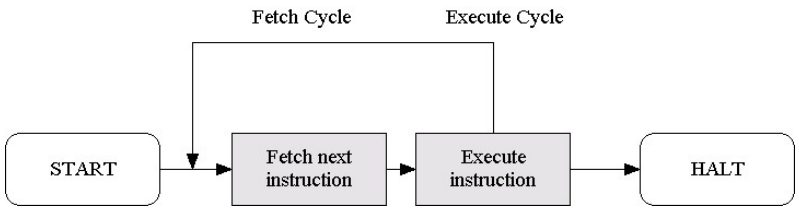
ldAB = B
ldCR = D+E
ClrCR = C
Sel = D
Done = F



Rules for a Safe Design

1. Keep the design simple and modular
   - Use an iterative procedure, if necessary, to build through refinement
   - Implement one module at a time, while unit testing each one
2. Develop good documentation during the design
3. Beware of clock skew
   - Use similar path lengths to all flip-flops
   - Do not use gated clocks!
   - Use all positive-edge-triggered or all negative-edge-triggered flip-flops
4. Be wary of asynchronous inputs to the circuit
   - Avoid them whenever possible!
   - Synchronize any ones that cannot be avoided
   - Use debounced switches to provide clean input signals
5. Beware of noise on power and signal lines
6. Avoid dependencies on minimum logic gate delays
7. Initialize all flip-flops to known values at the beginning

Basic Instruction Cycle

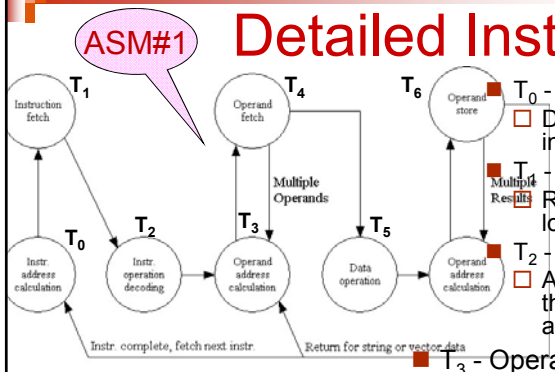


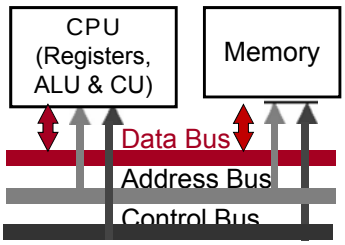
- Computer performs the instruction cycle forever! (or at least until it is turned off, faces an error or is instructed to do so)

Instruction Cycle

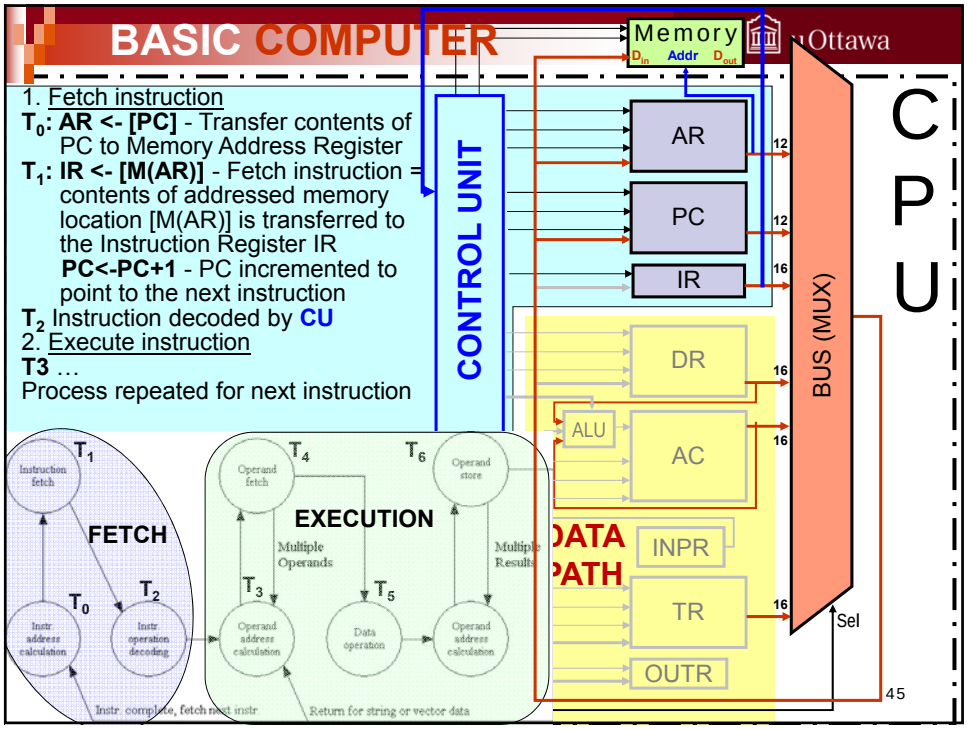
- A program residing in the memory of the computer consists of a sequence of instructions.
- Each instruction is executed in several steps:
  1. Fetch an instruction from memory.
  2. Decode the instruction.
  3. Read the effective address from memory if the instruction has an indirect address.
  4. Execute the instruction.
- Upon the completion of step 4, the control goes back to step 1 to start over the cycle for the next instruction.
- The process is repeated indefinitely unless a HALT instruction is encountered.

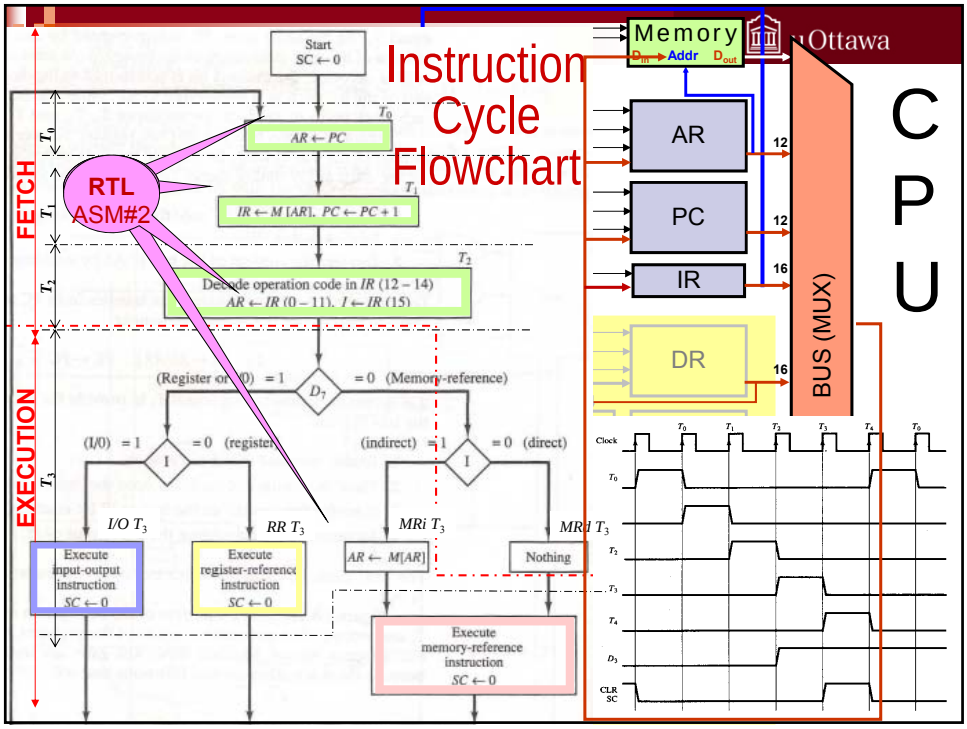
Detailed Instruction Cycle





- T_0 - Instruction address calculation
  - Determines the address of the next instruction to be executed (PC)
- T_1 - Instruction **fetch**
  - Reads the instruction from its memory location into the processor -IR
- T_2 - Instruction operation decoding
  - Analyzes the instruction to determine the type of operation to be performed and the operand(s) to be used
- T_3 - Operand address calculation
  - Determines the address of the operand (if needed) - RA
- T_4 - Instruction **execution** Operand fetch
  - Fetches the operand from memory or read it from I/O -> AC, e.g.
- T_5 – Instruction **execution** Data operation
  - Performs the operation indicated in the instruction (ALU)
- T_6 - Instruction **execution** Operand store
  - Write the results into memory or out to I/O


BASIC COMPUTER
Ottawa
1. Fetch instruction
T0: AR <- [PC] - Transfer contents of PC to Memory Address Register
T1: IR <- [M(AR)] - Fetch instruction = contents of addressed memory location [M(AR)] is transferred to the Instruction Register IR
PC<-PC+1 - PC incremented to point to the next instruction
T2 Instruction decoded by CU
2. Execute instruction
T3 …
Process repeated for next instruction
CONTROL UNIT
Memory
Din Addr Dout
AR
PC
IR
DR
ALU
AC
INPR
TR
OUTR
BUS (MUX)
Sel
CPU
FETCH
EXECUTION
Multiple Operands
Multiple Results
Instruction fetch
Instr address calculation
Instr operation decoding
Operand fetch
Operand address calculation
Data operation
Operand store
Instr. complete, fetch next instr.
Return for string or vector data
45

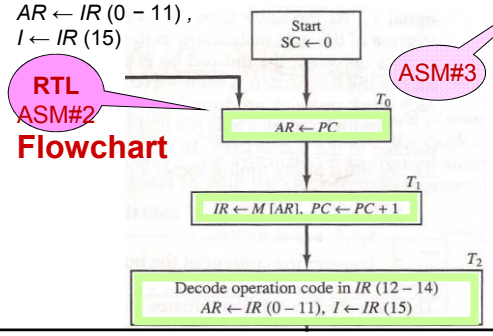

Instruction Cycle Flowchart
RTL ASM#2
Ottawa
Start SC ← 0
AR ← PC
IR ← M [AR], PC ← PC + 1
Decode operation code in IR (12 – 14)
AR ← IR (0 – 11), I ← IR (15)
= 0 (Memory-reference)
(I/O) = 1
= 0 (register)
(indirect) = 1
= 0 (direct)
I/O T3
RR T3
MRi T3
MRd T3
Execute input-output instruction SC ← 0
Execute register-reference instruction SC ← 0
AR ← M[AR]
Nothing
Execute memory-reference instruction SC ← 0
FETCH
EXECUTION
Memory
AR
PC
IR
DR
BUS (MUX)
CPU
Clock
CLR SC

Fetch&Decode

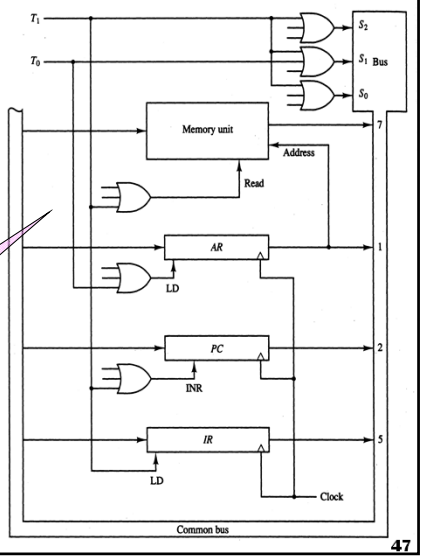
- When a program is executed, the program counter gets initially loaded with the address of the first instruction of the program, and *SC* is cleared making T_0=1
- After each clock pulse, *SC* is automatically incremented, so that the timing signals go through T_0, T_1, T_2, ….
- The micro-operations of the fetch and decode phases can be specified by the following RTL statements:

  T_0 : $AR \leftarrow PC$
  T_1 : $IR \leftarrow M[AR]$, $PC \leftarrow PC + 1$
  T_2 : $D0, \ldots, D7 \leftarrow$ Decode $IR(12-14)$, $AR \leftarrow IR(0-11)$, $I \leftarrow IR(15)$

RTL
ASM#2
Flowchart

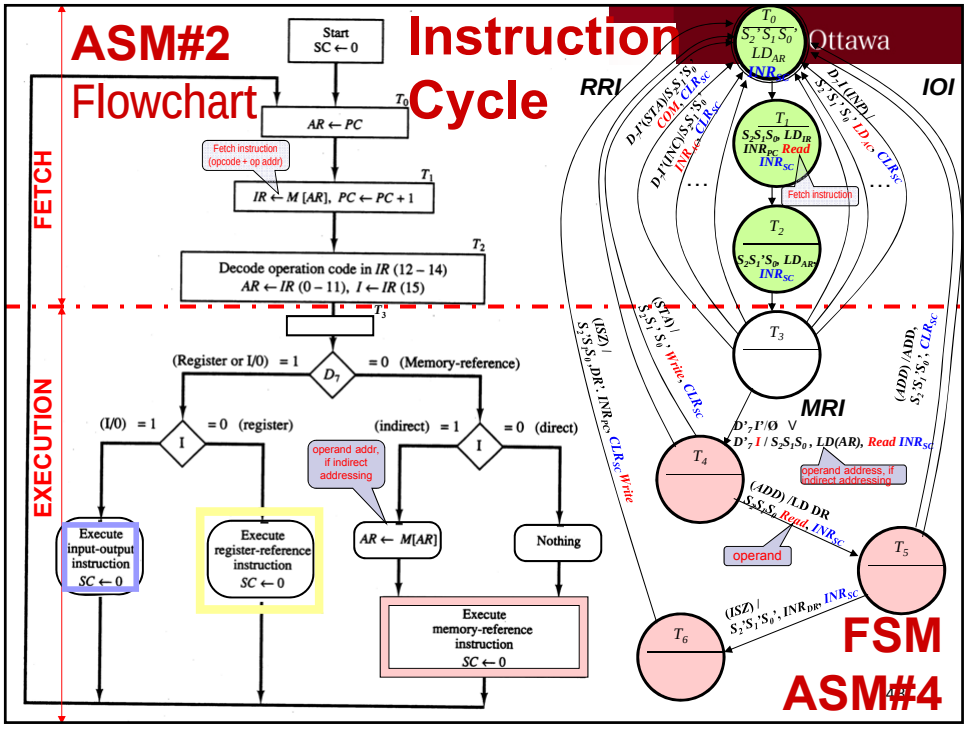


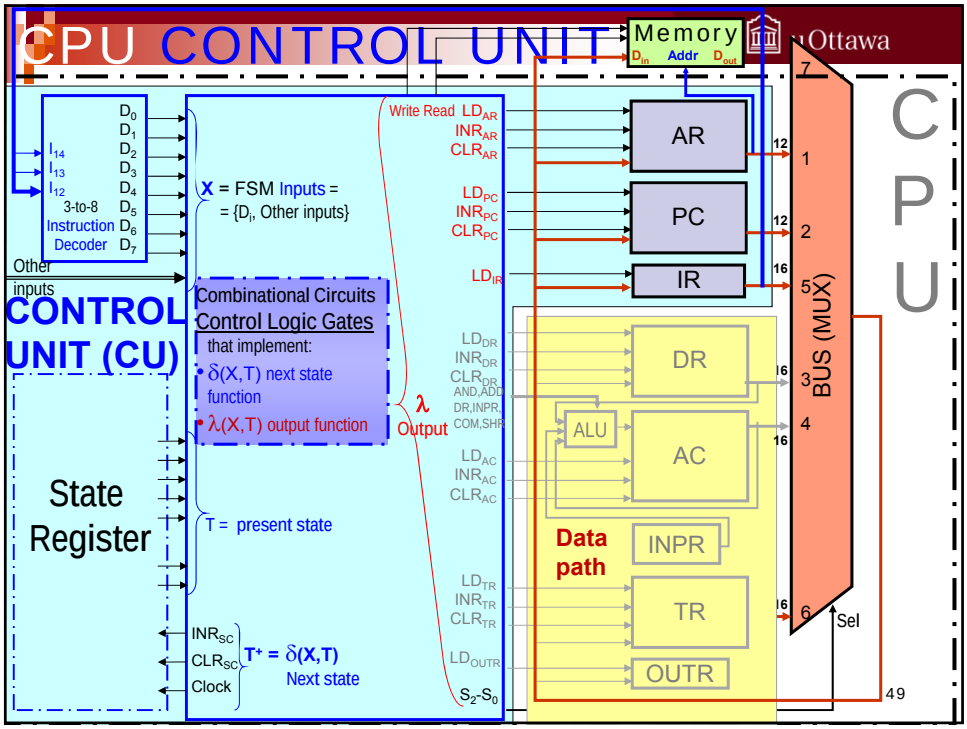
ASM#3

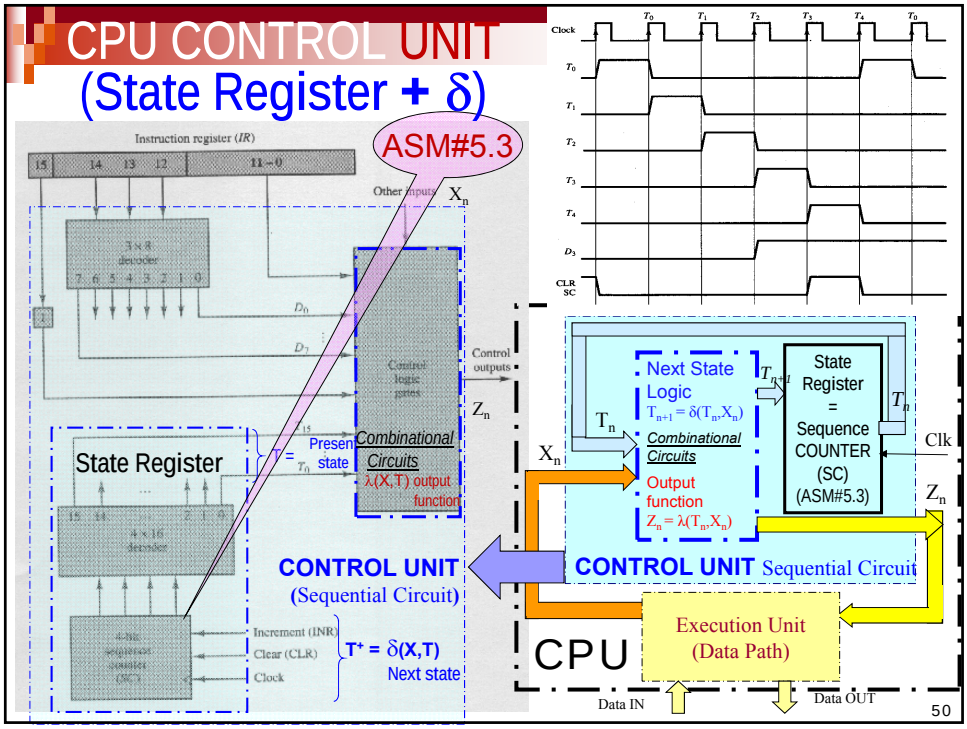


47

ASM#2 Flowchart — Instruction Cycle — FSM ASM#4

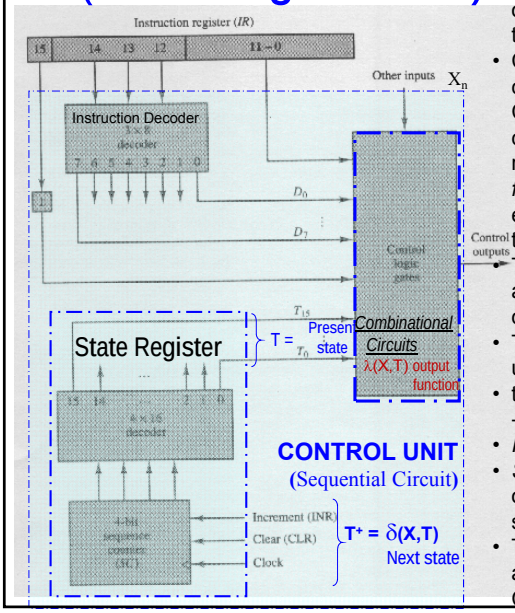



CPU CONTROL UNIT
Memory
Ottawa
CPU
AR
PC
IR
DR
ALU
AC
INPR
TR
OUTR
BUS (MUX)
Sel
Write Read
LD_AR INR_AR CLR_AR
LD_PC INR_PC CLR_PC
LD_IR
LD_DR INR_DR CLR_DR
LD_AC INR_AC CLR_AC
LD_TR INR_TR CLR_TR
LD_OUTR
S2-S0
Data path
3-to-8 Instruction Decoder
Other inputs
CONTROL UNIT (CU)
X = FSM Inputs = = {Di, Other inputs}
Combinational Circuits
Control Logic Gates
that implement:
δ(X,T) next state function
λ(X,T) output function
λ Output
State Register
T = present state
INR_SC
CLR_SC
Clock
T+ = δ(X,T)
Next state
49


CPU CONTROL UNIT
(State Register + δ)
ASM#5.3
Instruction register (IR)
Other inputs
Xn
Control outputs
Zn
State Register
Present state
Combinational Circuits
λ(X,T) output function
Increment (INR)
Clear (CLR)
Clock
T+ = δ(X,T)
Next state
CONTROL UNIT
(Sequential Circuit)
Next State Logic
Tn+1 = δ(Tn,Xn)
Combinational Circuits
Output function
Zn = λ(Tn,Xn)
State Register = Sequence COUNTER (SC) (ASM#5.3)
Clk
CONTROL UNIT Sequential Circuit
CPU
Execution Unit (Data Path)
Data IN
Data OUT
50

CPU CONTROL UNIT (State Register + δ)

uOttawa



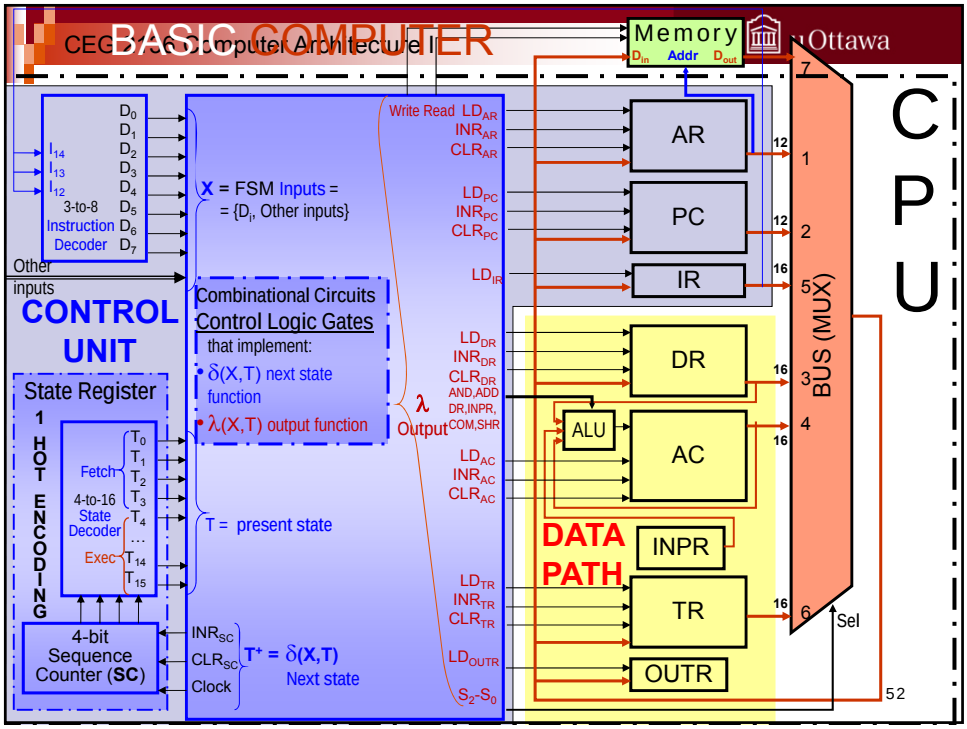
- The control unit (CU) is a *programmable sequential circuit*, whose transition (δ) and output (λ) functions changes in accord with the current instruction I_{14}-I_0.
- CU consists of a State Register (sequence counter SC + 4x16 decoder) & Combinational Circuits that compute the *output function* (λ - control signals for the multiplexers and registers in the system) & the *transition function* (δ - CLR clears SC at the end of the execution of every instruction to return SC to the initial state T_0, for the next instruction).
- The current instruction is read from memory and it is stored in the instruction register *IR* during its execution.
- The 3-bit opcode $IR_{14}IR_{13}IR_{12}$ is decoded using a 3 x 8 decoder (*Instruction Decoder*);
- the outputs of the *Instruction Decoder* are D_0 - D_7, each corresponding to one operation.
- IR_{15} is transferred to a flip-flop *I*.
- *SC* can count in binary from 0 to 15. The outputs of *SC* are decoded into 16 timing signals T_0-T_{15}
- The 12 least significant bits of *IR*, *I*, D_0-D_7, and T_0 - T_{15}, are all passed as inputs to the Control Logic Gates.

51

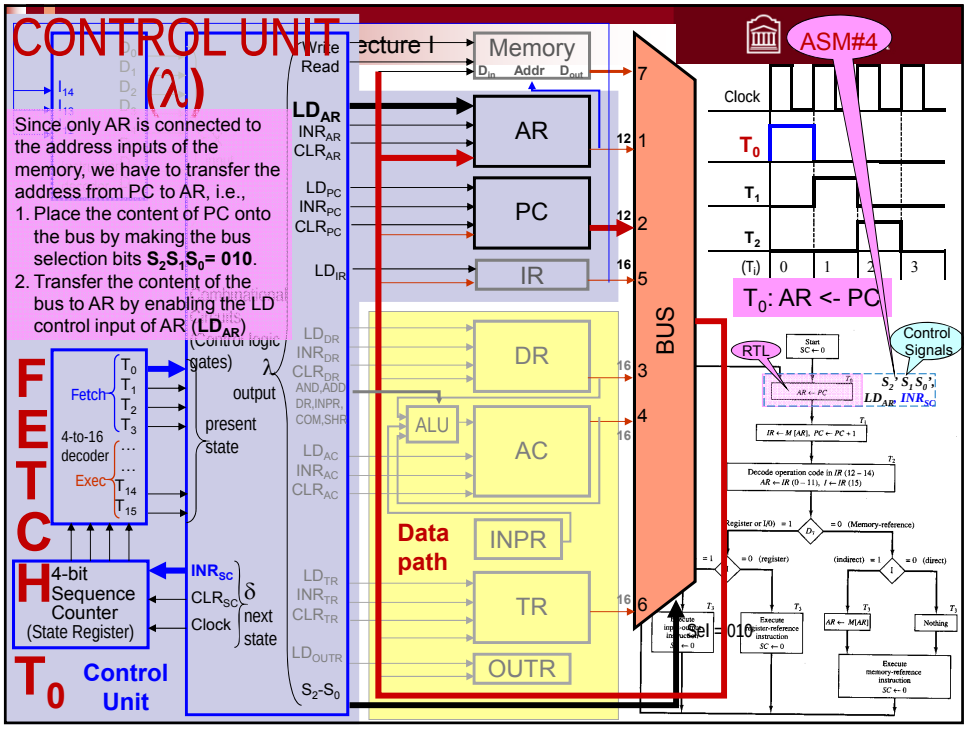
BASIC COMPUTER

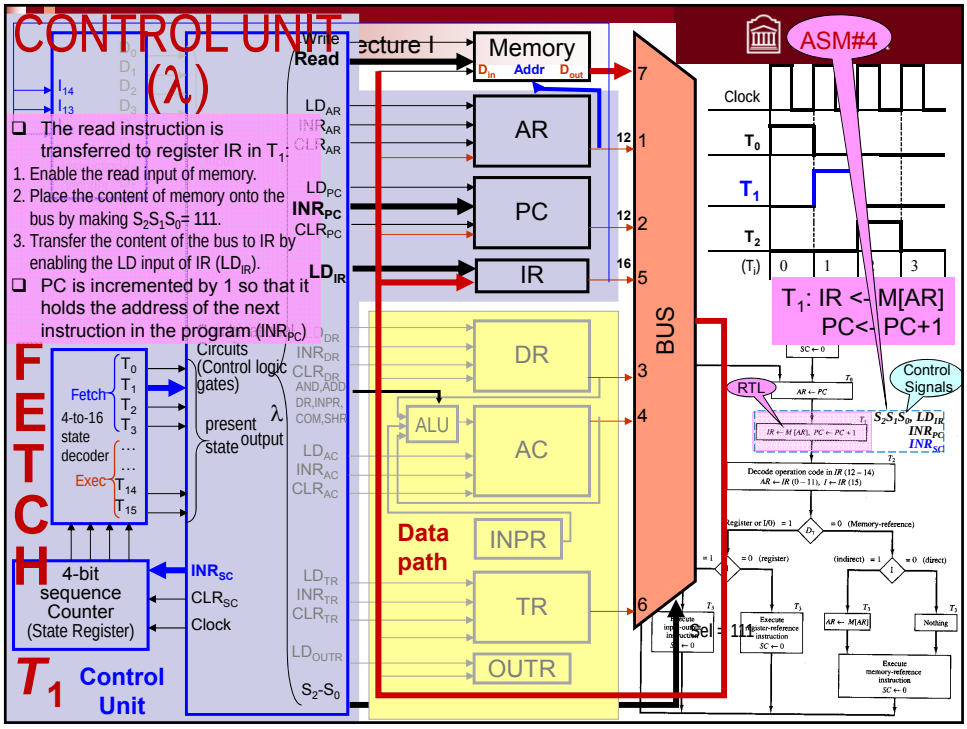
CEG 3566 Computer Architecture I

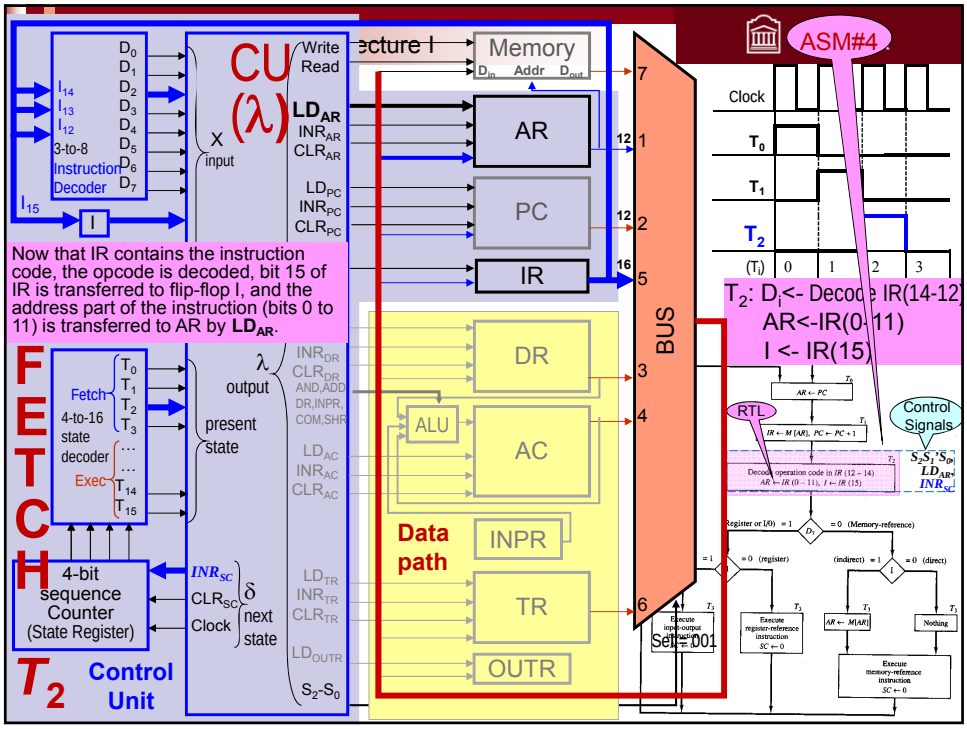
uOttawa



52

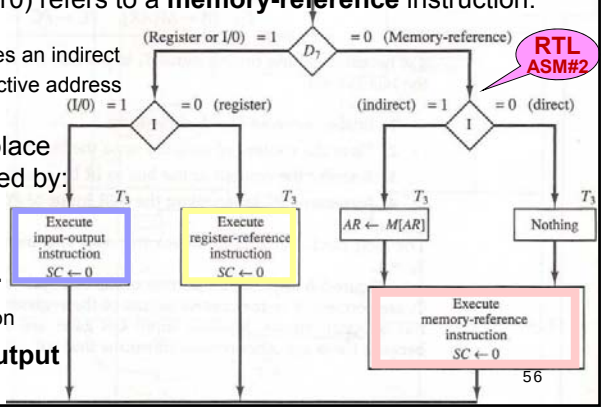

CONTROL UNIT
(λ)
Since only AR is connected to the address inputs of the memory, we have to transfer the address from PC to AR, i.e.,
1. Place the content of PC onto the bus by making the bus selection bits S2S1S0= 010.
2. Transfer the content of the bus to AR by enabling the LD control input of AR (LDAR)
FETCH
Fetch
4-to-16 decoder
Exec
4-bit Sequence Counter (State Register)
T0 Control Unit
(Control logic gates)
λ
output
present state
δ
next state
INRSC
CLRSC
Clock
Write
Read
LDAR
INRAR
CLRAR
LDPC
INRPC
CLRPC
LDIR
LDDR
INRDR
CLRDR
AND,ADD
DR,INPR,
COM,SHR
LDAC
INRAC
CLRAC
LDTR
INRTR
CLRTR
LDOUTR
S2-S0
Memory
Addr
AR
PC
IR
DR
ALU
AC
INPR
TR
OUTR
Data path
BUS
Sel = 010
ASM#4
Clock
T0
T1
T2
T0: AR <- PC
RTL
Control Signals


CONTROL UNIT
(λ)
The read instruction is transferred to register IR in T1:
1. Enable the read input of memory.
2. Place the content of memory onto the bus by making S2S1S0= 111.
3. Transfer the content of the bus to IR by enabling the LD input of IR (LDIR).
PC is incremented by 1 so that it holds the address of the next instruction in the program (INRPC)
FETCH
Fetch
4-to-16 state decoder
Exec
4-bit sequence Counter (State Register)
T1 Control Unit
Circuits (Control logic gates)
λ
output
present state
δ
next state
INRSC
CLRSC
Clock
Write
Read
LDAR
INRAR
CLRAR
LDPC
INRPC
CLRPC
LDIR
LDDR
INRDR
CLRDR
AND,ADD
DR,INPR,
COM,SHR
LDAC
INRAC
CLRAC
LDTR
INRTR
CLRTR
LDOUTR
S2-S0
Memory
Addr
AR
PC
IR
DR
ALU
AC
INPR
TR
OUTR
Data path
BUS
Sel = 111
ASM#4
Clock
T0
T1
T2
T1: IR <- M[AR]
PC<- PC+1
RTL
Control Signals



Determine the Type of Instruction

- The instruction fetching and decoding take place at times T_0 to T_2.
- During T_3, CU determines the type of instruction just read from the memory.
- The following segment of the ASM flowchart for instruction cycle presents an initial configuration of the instruction cycle.
- D_7 = 0 (opcode is 000 to 110) refers to a **memory-reference** instruction.
  - If I = 1, then the instruction carries an indirect address and the operand's effective address is read by transferring it to AR.
- All four operations taking place at time T_3 can be symbolized by:
  - $D'_7\, I\, T_3$: $AR \leftarrow M[AR]$
  - $D'_7\, I'\, T_3$: Nothing
  - $D_7\, I'\, T_3$: Execute a **register-reference** instruction
  - $D_7\, I\, T_3$: Execute an **input-output** instruction



Recall:
Basic Computer Instruction List (Binary)

D_7	Instr. Decod	I_{14}	I_{13}	I_{12}	I_{11}	I_{10}	I_9	I_8	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	I_{15}=0 MRI Direct Addr.	I_{15}=1 MRI Indirect Addr.
		op	code															
D_7=0 mem	D_0	0	0	0	AR_{11}	AR_{10}	AR_9	AR_8	AR_7	AR_6	AR_5	AR_4	AR_3	AR_2	AR_1	AR_0	**AND**=$0adr	AND_i =$8(adr)
	D_1	0	0	1	AR_{11}	AR_{10}	AR_9	AR_8	AR_7	AR_6	AR_5	AR_4	AR_3	AR_2	AR_1	AR_0	**ADD**=$1adr	ADD_i =$9(adr)
	D_2	0	1	0	AR_{11}	AR_{10}	AR_9	AR_8	AR_7	AR_6	AR_5	AR_4	AR_3	AR_2	AR_1	AR_0	**LDA** =$2adr	LDA_i =$A(adr)
	D_3	0	1	1	AR_{11}	AR_{10}	AR_9	AR_8	AR_7	AR_6	AR_5	AR_4	AR_3	AR_2	AR_1	AR_0	**STA** =$3adr	STA_i =$B(adr)
	D_4	1	0	0	AR_{11}	AR_{10}	AR_9	AR_8	AR_7	AR_6	AR_5	AR_4	AR_3	AR_2	AR_1	AR_0	**BUN** =$4adr	BUN_i =$C(adr)
	D_5	1	0	1	AR_{11}	AR_{10}	AR_9	AR_8	AR_7	AR_6	AR_5	AR_4	AR_3	AR_2	AR_1	AR_0	**BSA** =$5adr	BSA_i =$D(adr)
	D_6	1	1	0	AR_{11}	AR_{10}	AR_9	AR_8	AR_7	AR_6	AR_5	AR_4	AR_3	AR_2	AR_1	AR_0	**ISZ** =$6adr	ISZ_i =$E(adr)
																	RRI	**IOI**
D_7=1 reg	D_7	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	**CLA** =$7800	INP =$F800
	D_7	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	**CLE** =$7400	OUT=$F400
	D_7	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	**CMA** =$7200	SKI=$F200
	D_7	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	**CME** =$7100	SKO =$F100
	D_7	1	1	1	0	0	0	0	1	0	0	0	0	0	0	0	**CIR** =$7080	ION =$F080
	D_7	1	1	1	0	0	0	0	0	1	0	0	0	0	0	0	**CIL** =$7040	IOF =$F040
	D_7	1	1	1	0	0	0	0	0	0	1	0	0	0	0	0	**INC** =$7020	n/a
	D_7	1	1	1	0	0	0	0	0	0	0	1	0	0	0	0	**SPA** =$7010	n/a
	D_7	1	1	1	0	0	0	0	0	0	0	0	1	0	0	0	**SNA** =$7008	n/a
	D_7	1	1	1	0	0	0	0	0	0	0	0	0	1	0	0	**SZA** =$7004	n/a
	D_7	1	1	1	0	0	0	0	0	0	0	0	0	0	1	0	**SZE** =$7002	n/a
	D_7	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	**HLT** =$7001	n/a
		Binary encoding			**One-hot** (*bit-per-state*) encoding												adr = 12 bit address (adr) = address of the op address	

Instruction Cycle (EXEC)
Register-Reference Instructions (RRI)

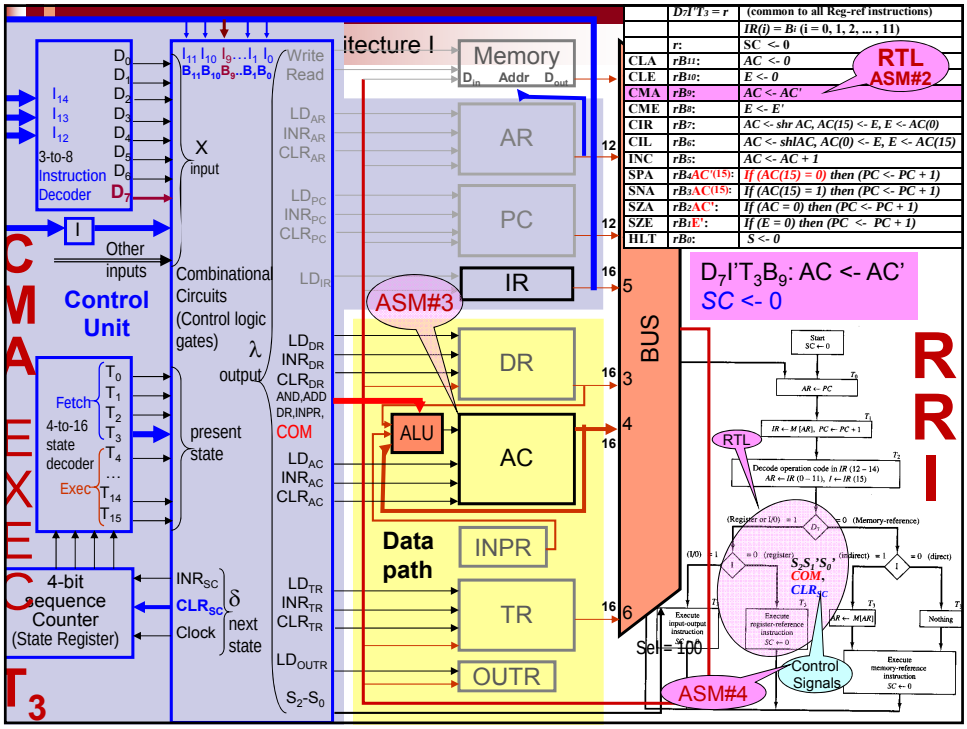
From the Instruction List: $I_{14}I_{13}I_{12} = 111 \Rightarrow D_7 = 1$
- $I_{15} = I = 0$

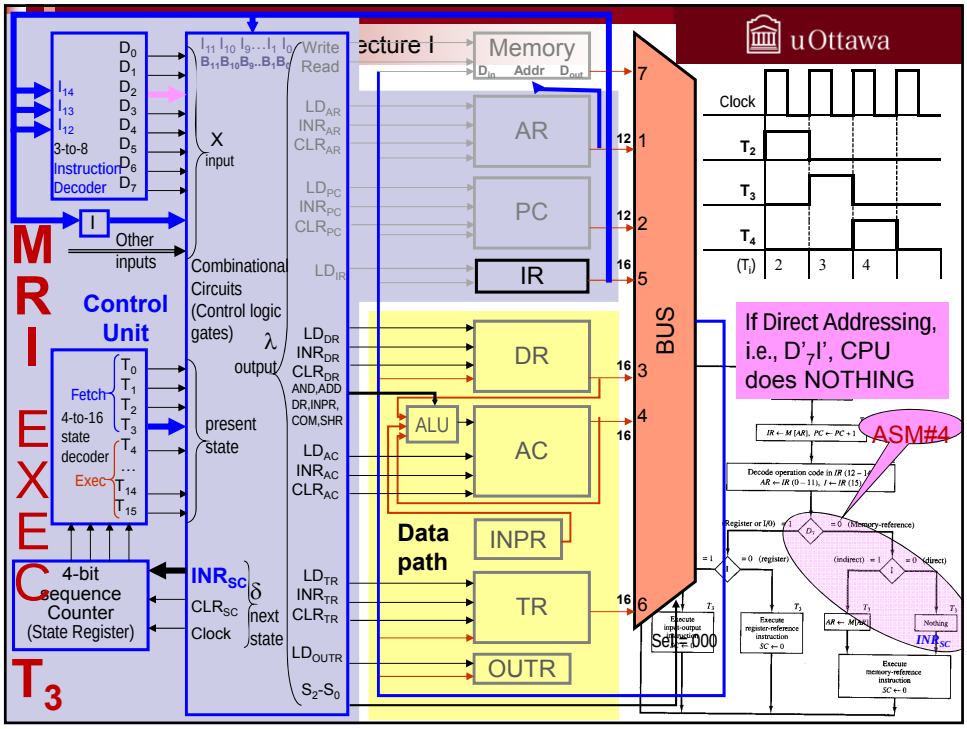
RTL ASM#2

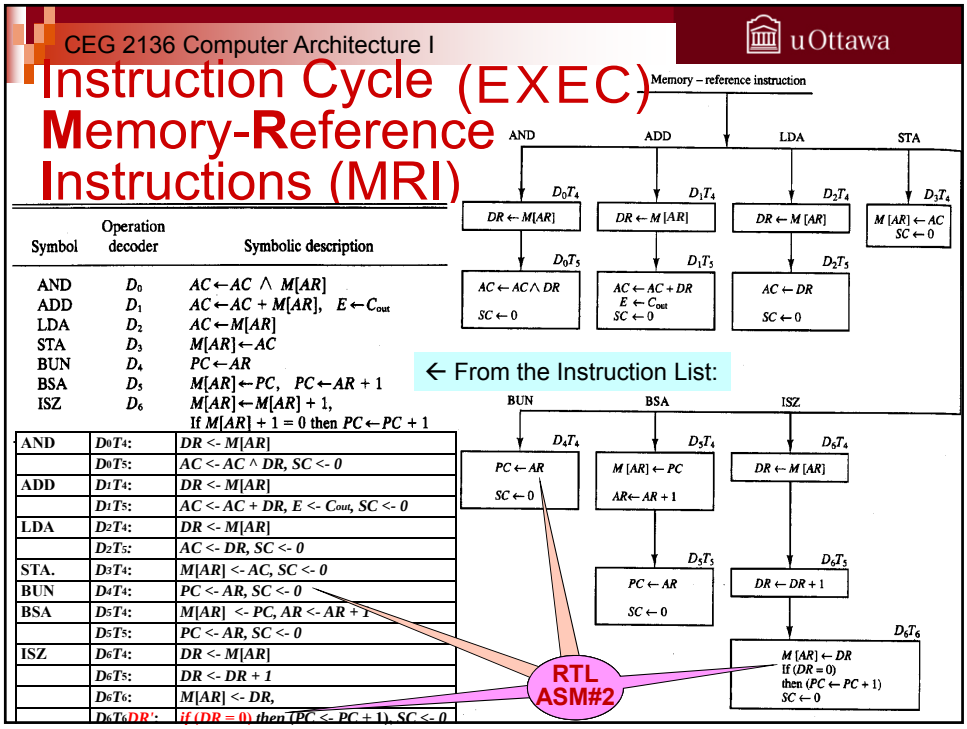
$D_7I'T_3 = r$ (common to all register-reference instructions)
$IR(i) = B_i$ [bit in $IR(0\text{–}11)$ that specifies the operation]

	r:	$SC \leftarrow 0$	Clear SC
CLA	rB_{11}:	$AC \leftarrow 0$	Clear AC
CLE	rB_{10}:	$E \leftarrow 0$	Clear E
CMA	rB_9:	$AC \leftarrow \overline{AC}$	Complement AC
CME	rB_8:	$E \leftarrow \overline{E}$	Complement E
CIR	rB_7:	$AC \leftarrow$ shr AC, $AC(15) \leftarrow E$, $E \leftarrow AC(0)$	Circulate right
CIL	rB_6:	$AC \leftarrow$ shl AC, $AC(0) \leftarrow E$, $E \leftarrow AC(15)$	Circulate left
INC	rB_5:	$AC \leftarrow AC + 1$	Increment AC
SPA	rB_4:	If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$	Skip if positive
SNA	rB_3:	If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$	Skip if negative
SZA	rB_2:	If $(AC = 0)$ then $PC \leftarrow PC + 1)$	Skip if AC zero
SZE	rB_1:	If $(E = 0)$ then $(PC \leftarrow PC + 1)$	Skip if E zero
HLT	rB_0:	$S \leftarrow 0$ (S is a start–stop flip-flop)	Halt computer

Every **R**egister-**R**eference **I**nstruction is executed during T_3


itecture I
Memory
AR
PC
IR
DR
AC
ALU
INPR
TR
OUTR
BUS
Control Unit
Combinational Circuits (Control logic gates)
Data path
4-bit Sequence Counter (State Register)
3-to-8 Instruction Decoder
4-to-16 state decoder
Fetch
Exec
Other inputs
input
output
present state
next state
Write
Read
Clock
RTL
ASM#2
ASM#3
ASM#4
Control Signals
Sel = 100
D7I'T3B9: AC <- AC'
SC <- 0
RRI
CEG 2136 Computer Architecture I
uOttawa
Instruction Cycle (EXEC)
Memory-Reference Instructions (MRI)
From the Instruction List:
I14I13I12 = [000 – 110] => Di = 1, i = {0 .. 6}
I15 = I = 0 => direct addressing
I15 = I = 1 => indirect addressing
Symbol
Operation decoder
Symbolic description
AND D0 AC ← AC ∧ M[AR]
ADD D1 AC ← AC + M[AR], E ← Cout
LDA D2 AC ← M[AR]
STA D3 M[AR] ← AC
BUN D4 PC ← AR
BSA D5 M[AR] ← PC, PC ← AR + 1
ISZ D6 M[AR] ← M[AR] + 1,
If M[AR] + 1 = 0 then PC ← PC + 1
60



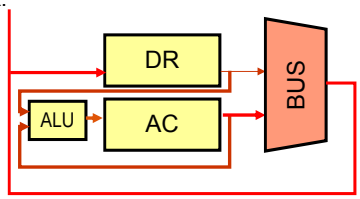


Symbol	Operation decoder	Symbolic description
AND	D_0	$AC \leftarrow AC \wedge M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR],\ E \leftarrow C_{out}$
LDA	D_2	$AC \leftarrow M[AR]$
STA	D_3	$M[AR] \leftarrow AC$
BUN	D_4	$PC \leftarrow AR$
BSA	D_5	$M[AR] \leftarrow PC,\ PC \leftarrow AR + 1$
ISZ	D_6	$M[AR] \leftarrow M[AR] + 1$, If $M[AR] + 1 = 0$ then $PC \leftarrow PC + 1$

AND	D_0T_4:	$DR \leftarrow M[AR]$
	D_0T_5:	$AC \leftarrow AC \wedge DR,\ SC \leftarrow 0$
ADD	D_1T_4:	$DR \leftarrow M[AR]$
	D_1T_5:	$AC \leftarrow AC + DR,\ E \leftarrow C_{out},\ SC \leftarrow 0$
LDA	D_2T_4:	$DR \leftarrow M[AR]$
	D_2T_5:	$AC \leftarrow DR,\ SC \leftarrow 0$
STA.	D_3T_4:	$M[AR] \leftarrow AC,\ SC \leftarrow 0$
BUN	D_4T_4:	$PC \leftarrow AR,\ SC \leftarrow 0$
BSA	D_5T_4:	$M[AR] \leftarrow PC,\ AR \leftarrow AR + 1$
	D_5T_5:	$PC \leftarrow AR,\ SC \leftarrow 0$
ISZ	D_6T_4:	$DR \leftarrow M[AR]$
	D_6T_5:	$DR \leftarrow DR + 1$
	D_6T_6:	$M[AR] \leftarrow DR$,
	D_6T_6DR':	if $(DR = 0)$ then $(PC \leftarrow PC + 1),\ SC \leftarrow 0$

Note on RTL

- The RTL statements should be read as having the present values on the right side of "=" , while the left side refers to the next state of that register; the transfer (register load) takes place on the rising edge of the clock.



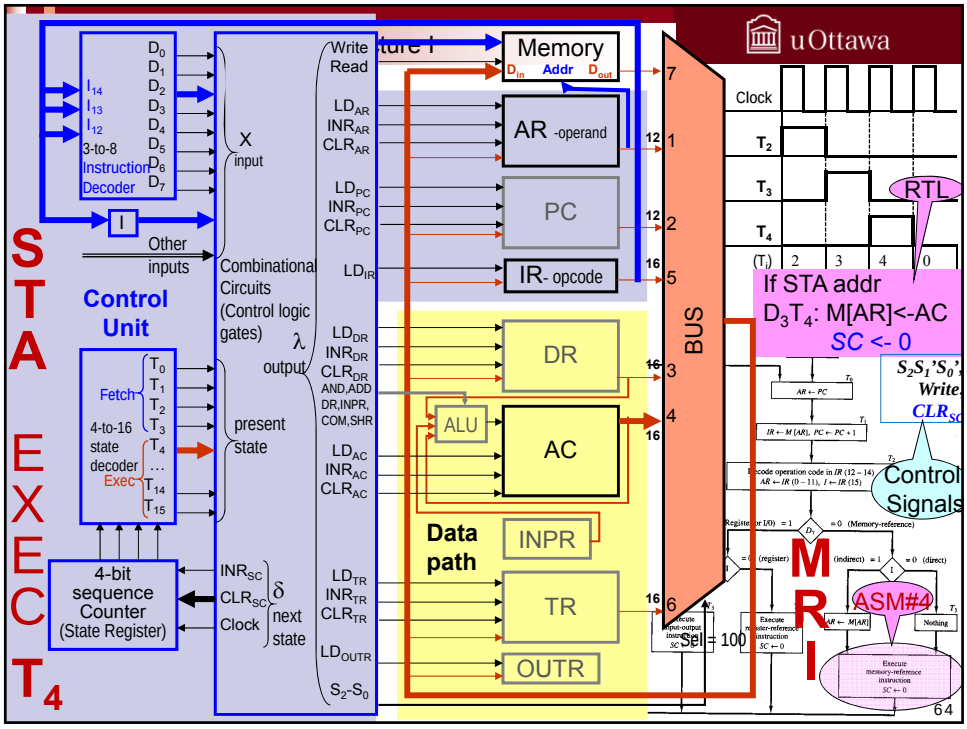
- For example:

  D_1T_5: DR<- AC, AC<- AC + DR

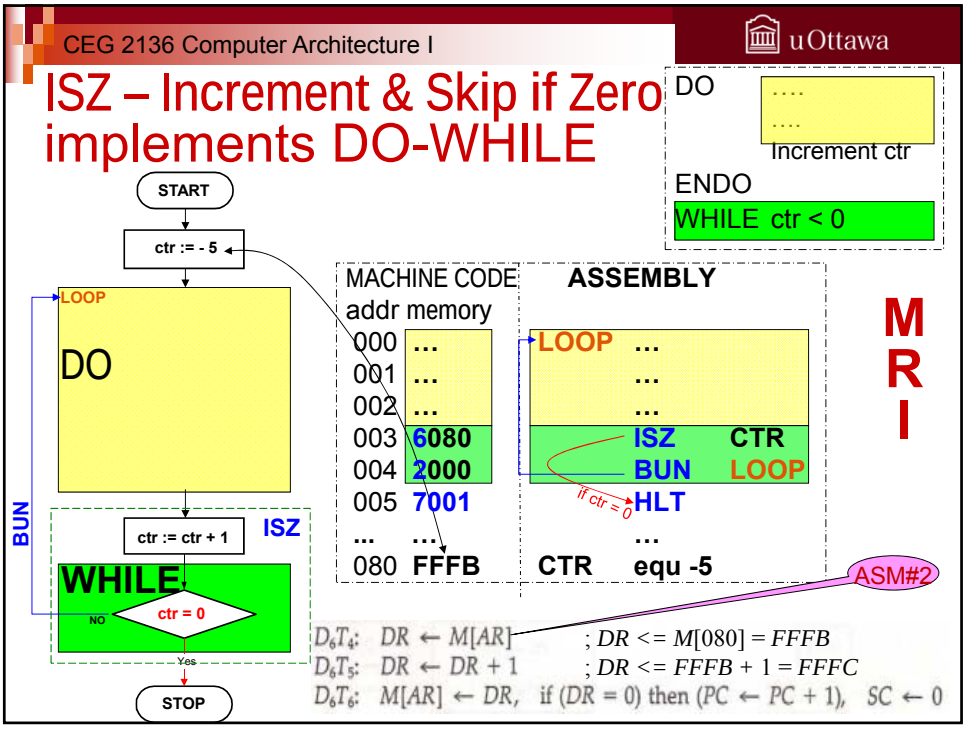
  reads:

  D_1T_5: DR(n+1)<- AC(n)
  AC(n+1)<- AC(n) + DR(n)

  Even the second RTL makes sense since **AC(n)+DR(n)** is the ALU's output which is to be loaded into AC on the rising edge of the clock. After loading the newer value into AC, there will be a delay until the ALU output will change, since ALU needs some time to "figure out" what is the newer result. As such, by the time ALU output will change, the rising edge will be gone and nothing will be allowed into AC until the next rising edge!



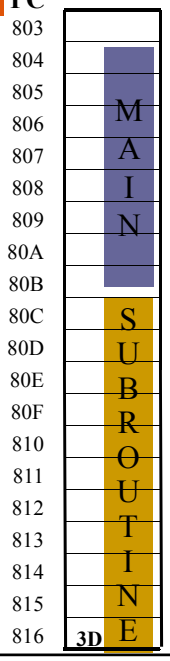
ISZ – Increment & Skip if Zero implements DO-WHILE



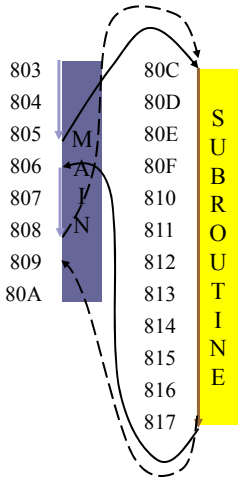
How …

1. … DOES A SUBROUTINE WORK?
2. … TO DEFINE IT?
3. … TO USE IT?

HOW DOES IT WORK?

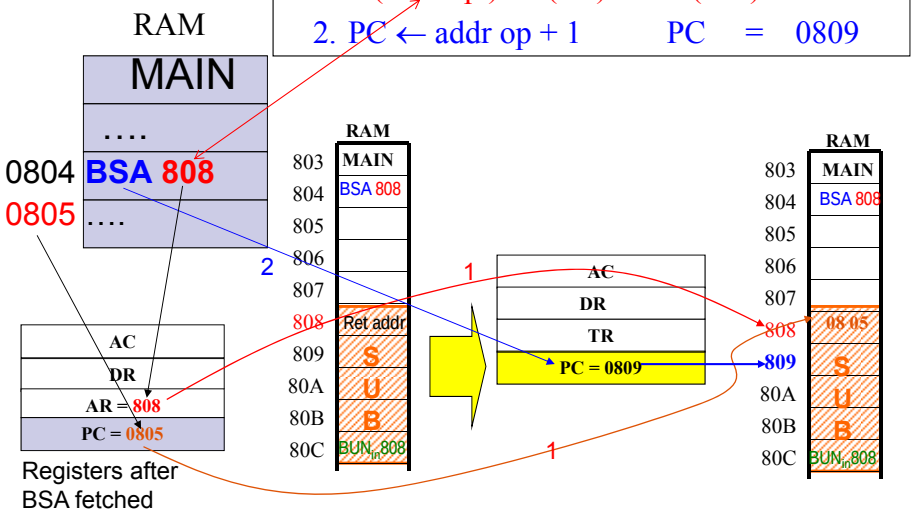


- A subroutine is a program module that is independent of the main program
- To use it, the main program transfers control to the subroutine
- The subroutine performs its function and then returns control to the main program

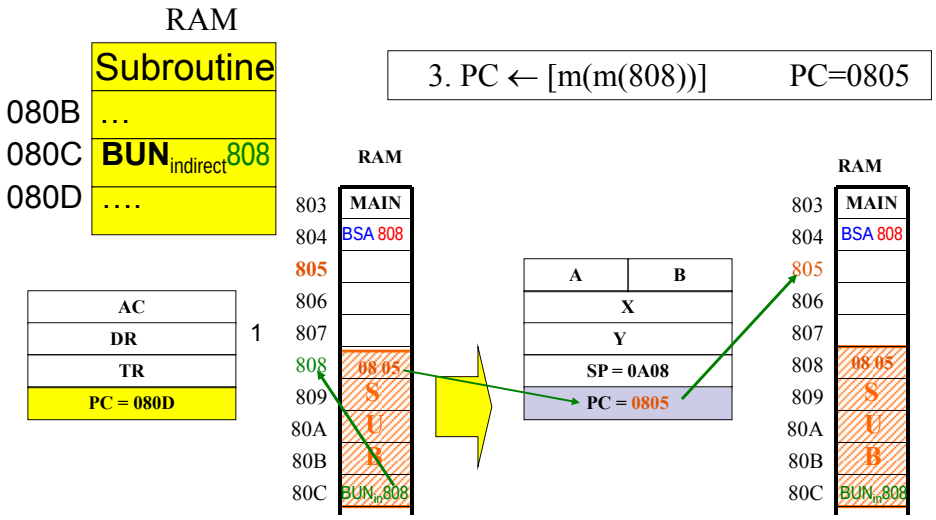


MAIN **CALLS** SUBROUTINE

1. m(addr op) ← (PC) m(808) = 0805
2. PC ← addr op + 1 PC = 0809



SUBROUTINE **RETURNS** TO MAIN



BSA: Branch and Save Return Address

The BSA instruction

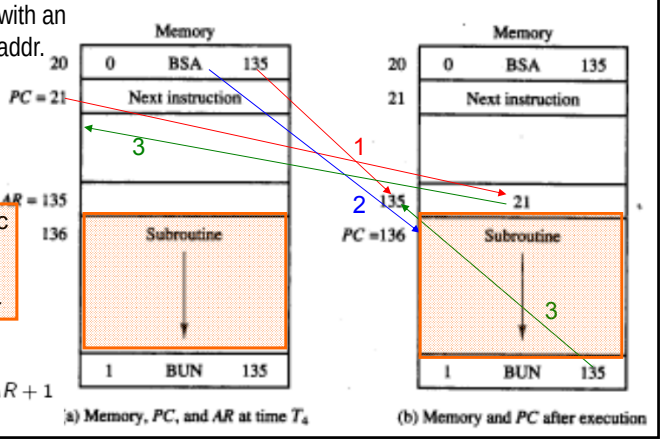
1. ***stores the return address*** (i.e., the address of the instruction to be run after the subroutine is executed, 21 in our example; this address is already prepared in PC) into the memory location specified by its effective address (135).
2. **branches to the first instruction of the *subroutine*** which is stored in the memory at the next address (136) after the stored return address.

3. The subroutine has to end with an indirect BUN to the return addr.

20	BSA 135
21	next instruction
22	……
….	…….
135	stored ret. addr (21)
136	Subroutine 1st instruc
137	…..
….	…..
159	Subroutine last instr
160	$BUN_{indirect}$ 135

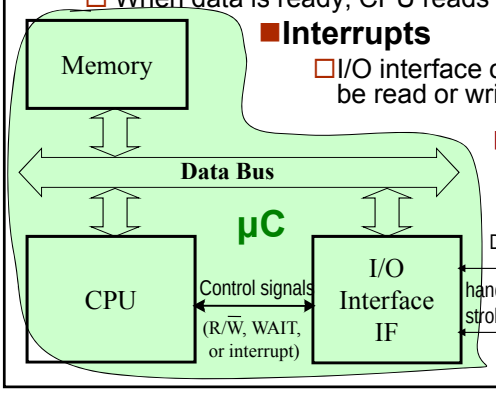
D_5T_4 : $M[AR] \leftarrow PC$, $AR \leftarrow AR + 1$

D_5T_5 : $PC \leftarrow AR$, $SC \leftarrow 0$

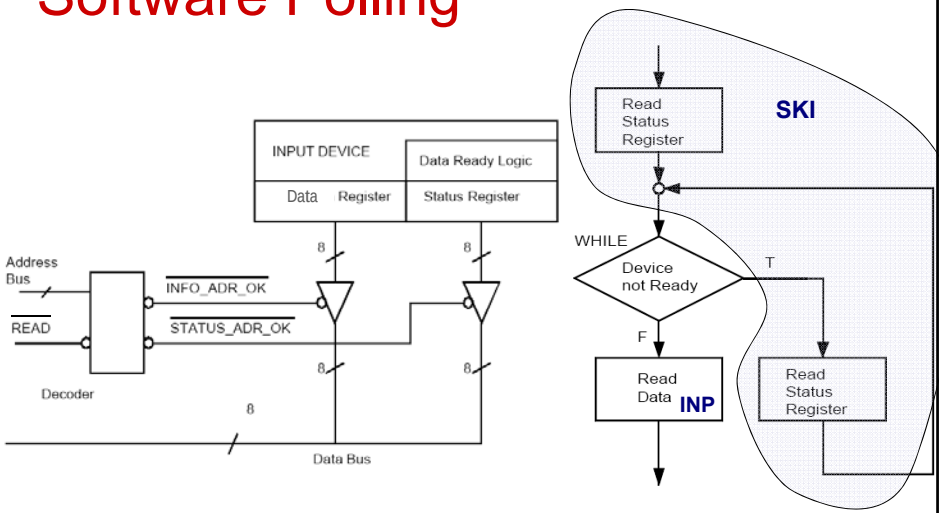


CPU & I/O Interface

- Hardware provides an **interface** to the I/O device. Contains:
  - data register (for data transfer, here: INPR and OUTR)
  - status register (flags=peripheral ready, here: FGI & FGO)
  - control register (here: R, IEN)
- **SW Polling** (Programmed Control)
  - CPU monitors the status register
  - When data is ready, CPU reads or writes from/to the data register
- **Interrupts**
  - I/O interface can interrupt the CPU when data can be read or written (will study later)
- **DMA (direct-memory access):**
  - the IF takes over the control of the system's Data, Address & Control Buses and transfers sequences of data direct to the memory.



Software Polling



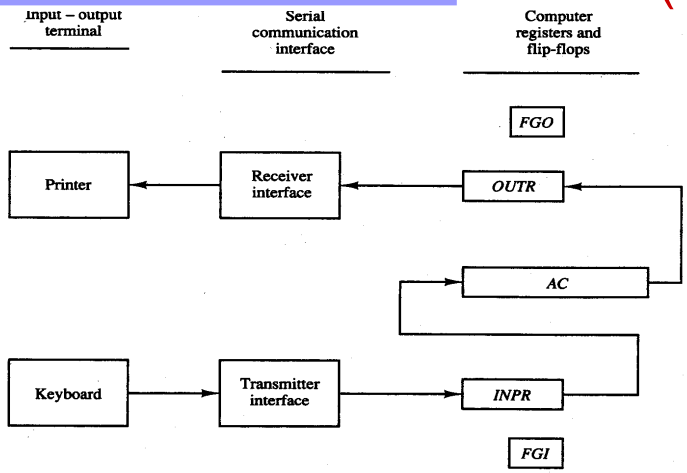
uOttawa

Basic Computer I/O Instructions (IOI)

$D_7IT_3 = p$ (common to all input-output instructions)
$IR(i) = B_i$ [bit in $IR(6\text{–}11)$ that specifies the instruction]

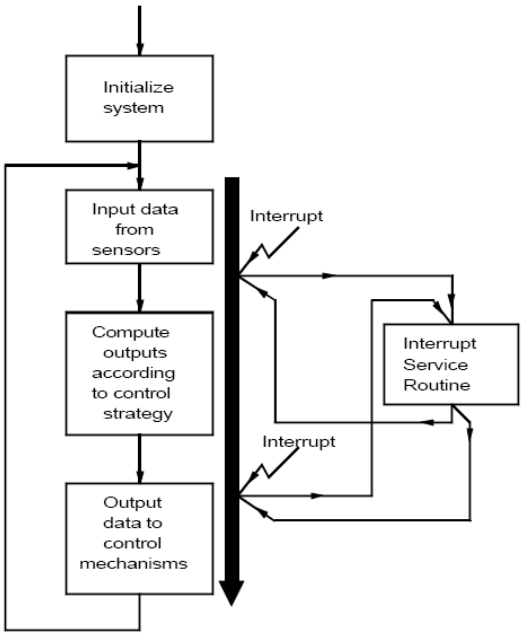
	p:	$SC \leftarrow 0$	Clear SC
INP	pB_{11}:	$AC(0\text{–}7) \leftarrow INPR$, $FGI \leftarrow 0$	Input character
OUT	pB_{10}:	$OUTR \leftarrow AC(0\text{–}7)$, $FGO \leftarrow 0$	Output character
SKI	pB_9:	If ($FGI = 1$) then ($PC \leftarrow PC + 1$)	Skip on input flag
SKO	pB_8:	If ($FGO = 1$) then ($PC \leftarrow PC + 1$)	Skip on output flag
ION	pB_7:	$IEN \leftarrow 1$	Interrupt enable on
IOF	pB_6:	$IEN \leftarrow 0$	Interrupt enable off

RTL



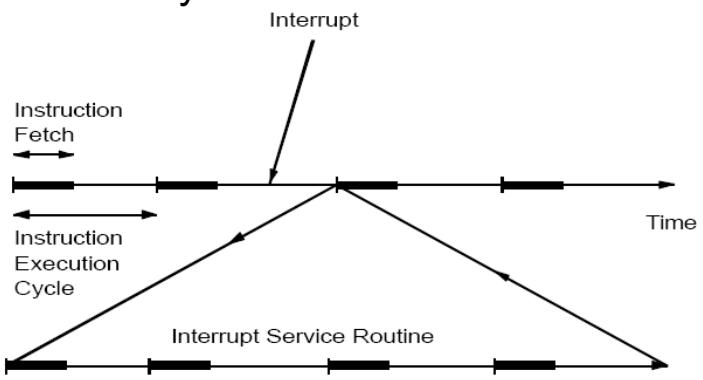
uOttawa

Interrupt Fundamental Concepts



Internal Asynchronous CPU Timing

- Interrupts can occur at any time during an instruction cycle



75

Internal CPU Interrupt Hardware

- Use a flip-flop (**R**) to catch IRQ
  - Multiple device interrupt lines can be wire-ORed together
  - CPU must wait until CPU can interrupt the process
- Can enable/disable interrupt using enable-disable flip-flop (IEN)
  - When interrupt is acknowledged, interrupts are disabled
- The source of interrupt is signalized by FGI (input flag) or by FGO (output flag)
  -> the Interrupt Service Routine (ISR) has to check which I/O device generated the IRQ
- At the end of ISR, $BUN_{ind}0$ is executed to return to the interrupted program

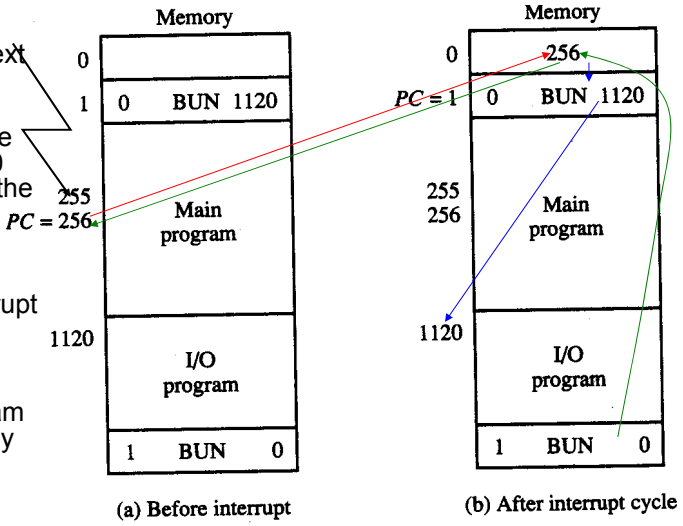
76

Interrupt System Specification

- Allow for asynchronous events to occur and be recognized.
- Wait for the current instruction to finish before servicing an interrupt.
- Service interrupt with a sub-routine and return to interrupted code.
- Enabling and disabling interrupts
- Multiple (here 2) sources of interrupts
  - Simultaneous interrupts.

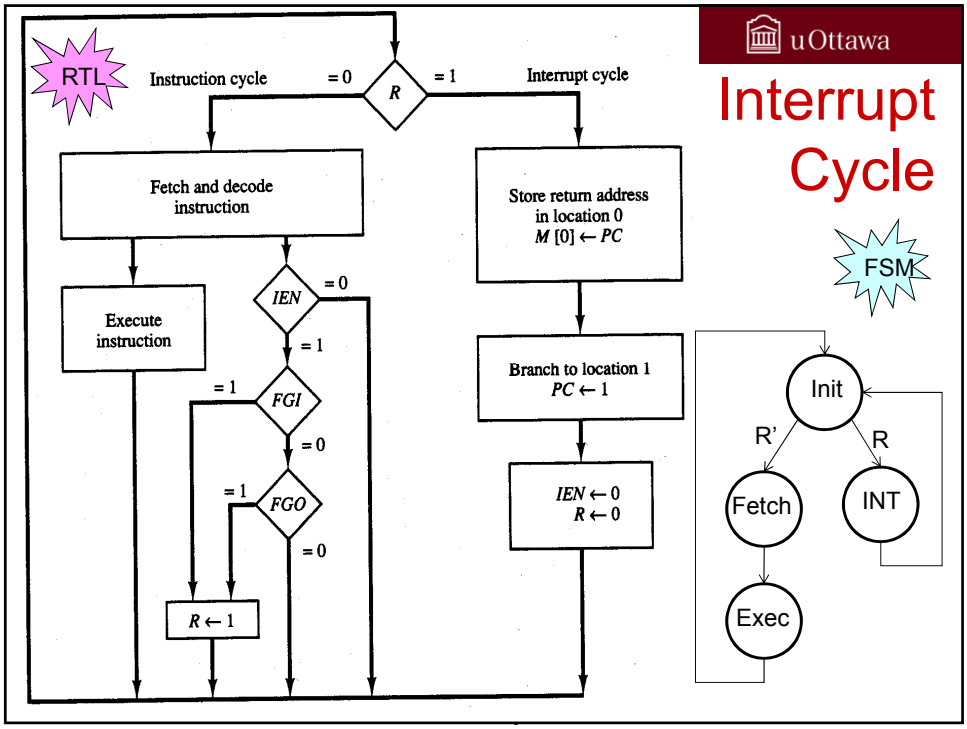
Interrupt Service Cycle (INT)

1. Save the PC (address of the next instruction) at the address 0
2. BUN directly to the ISR address 1120 corresponding to the interrupt
3. Disable interrupts (IEN<-0, R<-0)
4. Execute the Interrupt Service Routine (ISR)
5. Resume the interrupted program with BUN indirectly to address 0



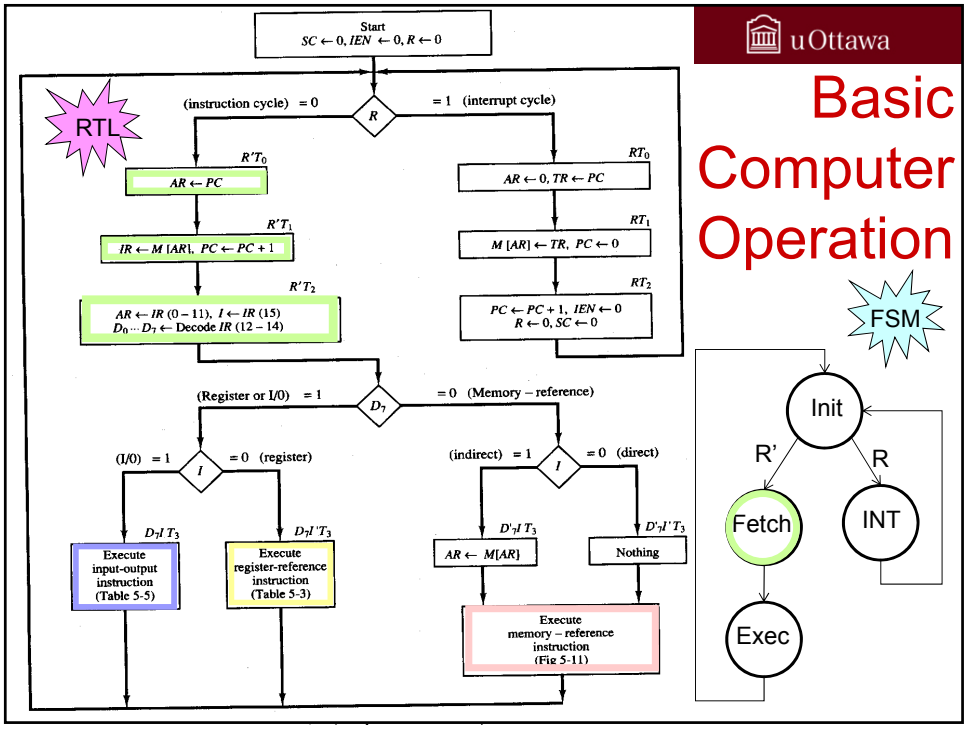
uOttawa

Interrupt Cycle

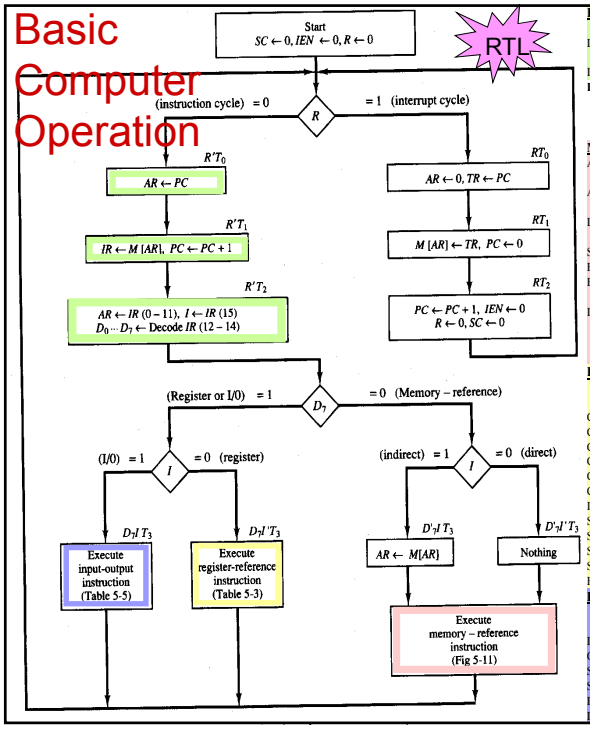


uOttawa

Basic Computer Operation

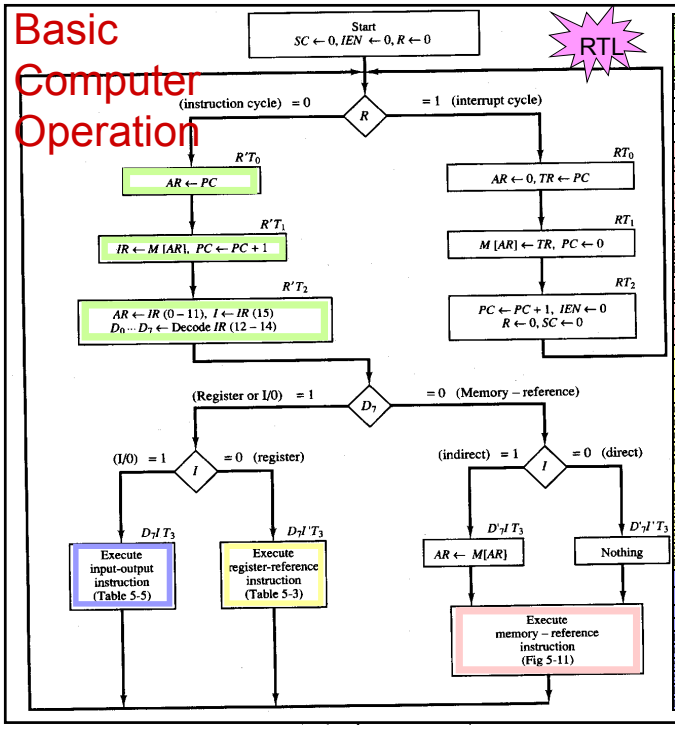


Basic Computer Operation

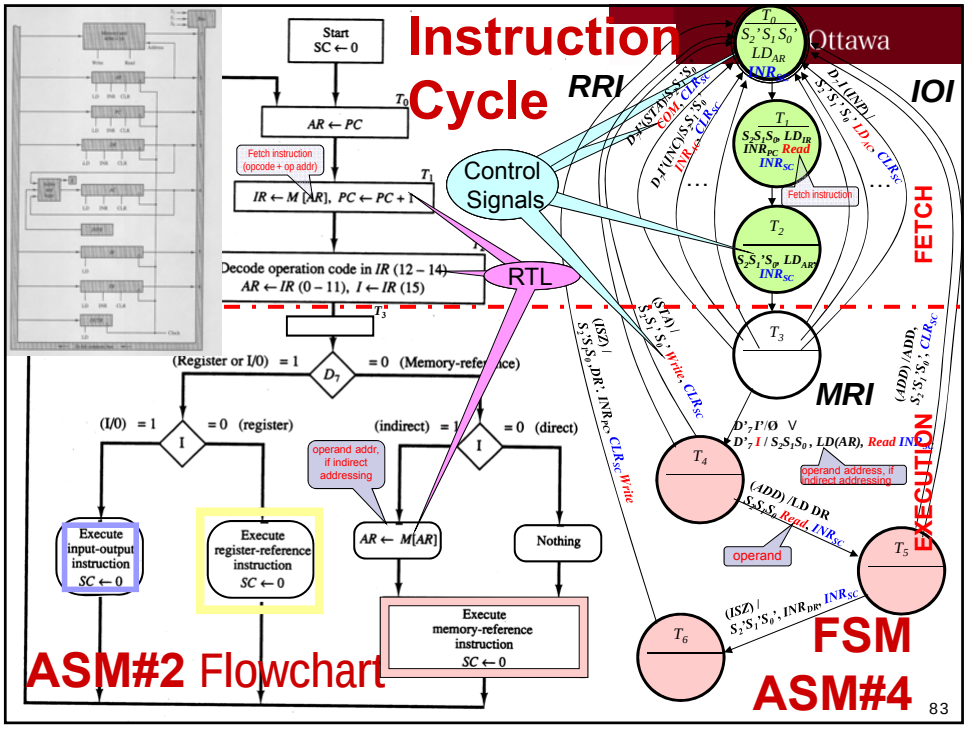


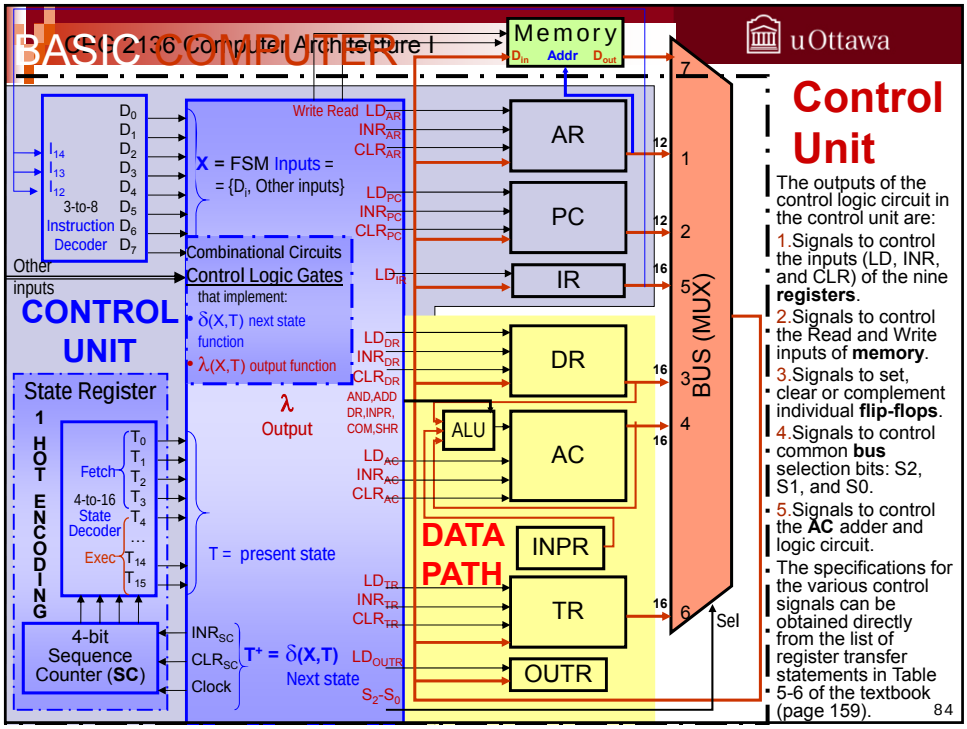
Fetch	$R'T_0$:	$AR \leftarrow PC$
	$R'T_1$:	$IR \leftarrow M[AR]$, $PC \leftarrow PC + 1$
Decode	$R'T_2$:	$D_0, \ldots, D_7 \leftarrow$ Decode $IR(12\text{-}14)$, $AR \leftarrow IR(0\text{-}11)$, $I \leftarrow IR(15)$
Indirect	D'_7IT_3:	$AR \leftarrow M[AR]$
Interrupt:	$T_0T_1T_2(IEN)(FGI + FGO)$:	$R \leftarrow 1$
	RT_0:	$AR \leftarrow 0$, $TR \leftarrow PC$
	RT_1:	$M[AR] \leftarrow TR$, $PC \leftarrow 0$
	RT_2:	$PC \leftarrow PC+1$, $IEN \leftarrow 0$, $R \leftarrow 0$, $SC \leftarrow 0$
Memory-reference:		
AND	D_0T_4:	$DR \leftarrow M[AR]$
	D_0T_5:	$AC \leftarrow AC \wedge DR$, $SC \leftarrow 0$
ADD	D_1T_4:	$DR \leftarrow M[AR]$
	D_1T_5:	$AC \leftarrow AC + DR$, $E \leftarrow C_{out}$, $SC \leftarrow 0$
LDA	D_2T_4:	$DR \leftarrow M[AR]$
	D_2T_5:	$AC \leftarrow DR$, $SC \leftarrow 0$
STA.	D_3T_4:	$M[AR] \leftarrow AC$, $SC \leftarrow 0$
BUN	D_4T_4:	$PC \leftarrow AR$, $SC \leftarrow 0$
BSA	D_5T_4:	$M[AR] \leftarrow PC$, $AR \leftarrow AR + 1$
	D_5T_5:	$PC \leftarrow AR$, $SC \leftarrow 0$
ISZ	D_6T_4:	$DR \leftarrow M[AR]$
	D_6T_5:	$DR \leftarrow DR + 1$
	D_6T_6:	$M[AR] \leftarrow DR$,
	D_6T_6DR':	*if (DR = 0) then* $(PC \leftarrow PC + 1)$, $SC \leftarrow 0$
Reg-ref	$D_7IT_3 = r$	(common to all reg.-ref. instr.)
		$IR(i) = B_i$ (i = 0, 1, 2, ... , 11)
	r:	$SC \leftarrow 0$
CLA	rB_{11}:	$AC \leftarrow 0$
CLE	rB_{10}:	$E \leftarrow 0$
CMA	rB_9:	$AC \leftarrow AC'$
CME	rB_8:	$E \leftarrow E'$
CIR	rB_7:	$AC \leftarrow shr\ AC$, $AC(15) \leftarrow E$, $E \leftarrow AC(0)$
CIL	rB_6:	$AC \leftarrow shl AC$, $AC(0) \leftarrow E$, $E \leftarrow AC(15)$
INC	rB_5:	$AC \leftarrow AC + 1$
SPA	$rB_4AC(15)$:	*If (AC(15) = 0) then* $(PC \leftarrow PC + 1)$
SNA	$rB_3AC(15)$:	*If (AC(15) = 1) then* $(PC \leftarrow PC + 1)$
SZA	rB_2AC':	*If (AC = 0) then* $(PC \leftarrow PC + 1)$
SZE	rB_1E':	*If (E = 0) then* $(PC \leftarrow PC + 1)$
HLT	rB_0:	$S \leftarrow 0$
In-out	$D_7IT_3 = p$	(common to all input-output instructions)
	$IR(i) = B_i$	(i = 6, 7, 8, 9, 10, 11)
	p:	$SC \leftarrow 0$
INP	pB_{11}:	$AC(0\text{-}7) \leftarrow INPR$, $FGI \leftarrow 0$
OUT	pB_{10}:	$OUTR \leftarrow AC(0\text{-}7)$, $FGO \leftarrow 0$
SKI	pB_9FGI:	*If (FGI = 1) then* $(PC \leftarrow PC + 1)$
SKO	pB_8FGO:	*If (FGO = 1) then* $(PC \leftarrow PC + 1)$
ION	pB_7:	$IEN \leftarrow 1$
IOF	pB_6:	$IEN \leftarrow 0$

Basic Computer Operation



Fetch	$R'T_0$:	$AR \leftarrow PC$
	$R'T_1$:	$IR \leftarrow M[AR]$, $PC \leftarrow PC + 1$
Decode	$R'T_2$:	$D_0, \ldots, D_7 \leftarrow$ Decode $IR(12\text{-}14)$, $AR \leftarrow IR(0\text{-}11)$, $I \leftarrow IR(15)$
Indirect	D'_7IT_3:	$AR \leftarrow M[AR]$
Interrupt	$T_0T_1T_2$ IEN $(FGI+FGO)$:	$R \leftarrow 1$
	RT_0:	$AR \leftarrow 0$, $TR \leftarrow PC$
	RT_1:	$M[AR] \leftarrow TR$, $PC \leftarrow 0$
	RT_2:	$PC \leftarrow PC+1$, $IEN \leftarrow 0$, $R \leftarrow 0$, $SC \leftarrow 0$
Execute Memory-reference Instructions (MRI)		
AND	D_0T_4:	$DR \leftarrow M[AR]$
	D_0T_5:	$AC \leftarrow AC \wedge DR$, $SC \leftarrow 0$
ADD	D_1T_4:	$DR \leftarrow M[AR]$
	D_1T_5:	$AC \leftarrow AC + DR$, $E \leftarrow C_{out}$, $SC \leftarrow 0$
LDA	D_2T_4:	$DR \leftarrow M[AR]$
	D_2T_5:	$AC \leftarrow DR$, $SC \leftarrow 0$
STA.	D_3T_4:	$M[AR] \leftarrow AC$, $SC \leftarrow 0$
BUN	D_4T_4:	$PC \leftarrow AR$, $SC \leftarrow 0$
BSA	D_5T_4:	$M[AR] \leftarrow PC$, $AR \leftarrow AR + 1$
	D_5T_5:	$PC \leftarrow AR$, $SC \leftarrow 0$
ISZ	D_6T_4:	$DR \leftarrow M[AR]$
	D_6T_5:	$DR \leftarrow DR + 1$
	D_6T_6:	$M[AR] \leftarrow DR$,
	D_6T_6DR':	*if (DR = 0) then* $(PC \leftarrow PC + 1)$, $SC \leftarrow 0$
Execute Register-reference Instructions (RRI)		
	$D_7IT_3 = r$	(common to all Register-ref. instructions)
		$IR(i) = B_i$ (i = 0, 1, 2, ... , 11)
	r:	$SC \leftarrow 0$
CLA	rB_{11}:	$AC \leftarrow 0$
CLE	rB_{10}:	$E \leftarrow 0$
CMA	rB_9:	$AC \leftarrow AC'$
CME	rB_8:	$E \leftarrow E'$
CIR	rB_7:	$AC \leftarrow shr\ AC$, $AC(15) \leftarrow E$, $E \leftarrow AC(0)$
CIL	rB_6:	$AC \leftarrow shl AC$, $AC(0) \leftarrow E$, $E \leftarrow AC(15)$
INC	rB_5:	$AC \leftarrow AC + 1$
SPA	$rB_4AC(15)$:	*If (AC(15) = 0) then* $(PC \leftarrow PC + 1)$
SNA	$rB_3AC(15)$:	*If (AC(15) = 1) then* $(PC \leftarrow PC + 1)$
SZA	rB_2AC':	*If (AC = 0) then* $(PC \leftarrow PC + 1)$
SZE	rB_1E':	*If (E = 0) then* $(PC \leftarrow PC + 1)$
HLT	rB_0:	$S \leftarrow 0$
Execute Input-output (IOI)		
	$D_7IT_3 = p$	(common to all input-output instructions)
	$IR(i) = B_i$	(i = 6, 7, 8, 9, 10, 11)
	p:	$SC \leftarrow 0$
INP	pB_{11}:	$AC(0\text{-}7) \leftarrow INPR$, $FGI \leftarrow 0$
OUT	pB_{10}:	$OUTR \leftarrow AC(0\text{-}7)$, $FGO \leftarrow 0$
SKI	pB_9FGI:	*If (FGI = 1) then* $(PC \leftarrow PC + 1)$
SKO	pB_8FGO:	*If (FGO = 1) then* $(PC \leftarrow PC + 1)$
ION	pB_7:	$IEN \leftarrow 1$
IOF	pB_6:	$IEN \leftarrow 0$


Instruction Cycle
Ottawa
RRI
IOI
Control Signals
RTL
FETCH
MRI
EXECUTION
Start SC ← 0
AR ← PC
Fetch instruction (opcode + op addr)
IR ← M [AR], PC ← PC + 1
Decode operation code in IR (12 – 14)
AR ← IR (0 – 11), I ← IR (15)
(Register or I/0) = 1
= 0 (Memory-reference)
(I/0) = 1
= 0 (register)
(indirect) = 1
= 0 (direct)
operand addr. if indirect addressing
Execute input-output instruction SC ← 0
Execute register-reference instruction SC ← 0
AR ← M[AR]
Nothing
Execute memory-reference instruction SC ← 0
operand address, if indirect addressing
operand
Fetch instruction
ASM#2 Flowchart
FSM ASM#4
83


BASIC COMPUTER
CSG 2136 Computer Architecture I
uOttawa
Memory
Control Unit
The outputs of the control logic circuit in the control unit are:
1.Signals to control the inputs (LD, INR, and CLR) of the nine registers.
2.Signals to control the Read and Write inputs of memory.
3.Signals to set, clear or complement individual flip-flops.
4.Signals to control common bus selection bits: S2, S1, and S0.
5.Signals to control the AC adder and logic circuit.
The specifications for the various control signals can be obtained directly from the list of register transfer statements in Table 5-6 of the textbook (page 159).
CONTROL UNIT
X = FSM Inputs = = {Di, Other inputs}
Combinational Circuits
Control Logic Gates
that implement:
δ(X,T) next state function
λ(X,T) output function
λ Output
T = present state
T+ = δ(X,T) Next state
State Register
1 HOT ENCODING
3-to-8 Instruction Decoder
4-to-16 State Decoder
4-bit Sequence Counter (SC)
Other inputs
AR
PC
IR
DR
ALU
AC
INPR
TR
OUTR
DATA PATH
BUS (MUX)
Sel
84

Control Unit Design

Inputs: X = {D_i, Other inputs}

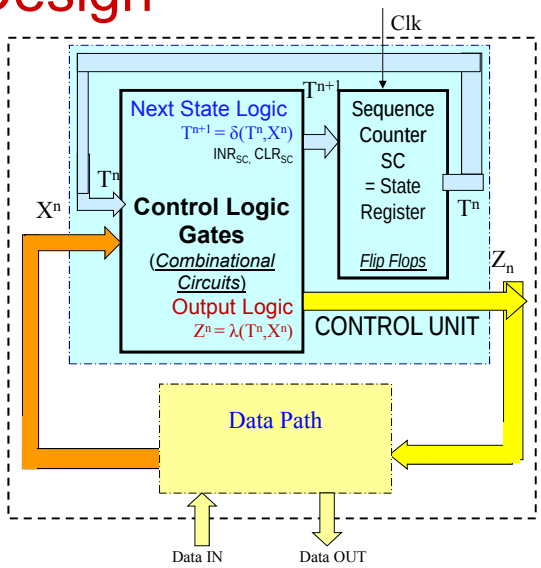
IR => D_i Instruction Decoder - AND, ADD, LDA, STA, BUN, BSA, ISZ, {RRI, IOI}

Outputs: Z = {LD_{AR}, INR_{AR}, CLR_{AR}, LD_{PC}, INR_{PC}, CLR_{PC}, Write, Read, LD_{IR}, LD_{DR}, INR_{DR}, CLR_{DR}, LD_{TR}, INR_{TR}, CLR_{TR}, LD_{AC}, INR_{AC}, …}

States : {T_0, T_1, T_2, T_3, … ,T_{15}}

$T^{n+1} = \delta(X^n, T^n)$ next state function

$Z^n = \lambda(X^n, T^n)$ output function



RTL ASM#2

Control Signals ASM#4

ASM#3

			$S_2S_1S_0$	
Fetch	$R'T_0$:	$AR \leftarrow PC$	010	LD_AR
	$R'T_1$:	$IR \leftarrow M[AR], PC \leftarrow PC + 1$	111	Read, LD_IR, INC_PC
Decode	$R'T_2$:	$D_0, \ldots, D_7 \leftarrow$ Decode IR(12-14), $AR \leftarrow IR(0\text{-}11), I \leftarrow IR(15)$	101	LD_AR, LD_I & IR(15)
Indirect	D'_7IT_3:	$AR \leftarrow M[AR]$	111	Read, LD_AR
Interrupt:				
$T_0T_1T_2IEN(FGI + FGO)$:		$R \leftarrow 1$		S_R,
	RT_0:	$AR \leftarrow 0, TR \leftarrow PC$	010	CL_AR, LD_TR,
	RT_1:	$M[AR] \leftarrow TR, PC \leftarrow 0$	110	Write, CL_PC
	RT_2:	$PC \leftarrow PC+1, IEN \leftarrow 0, R \leftarrow 0, SC \leftarrow 0$		INC_PC, R_{SC}, R_R, R_{IEN},
		Memory-reference:		
AND	D_0T_4:	$DR \leftarrow M[AR]$		
	D_0T_5:	$AC \leftarrow AC \wedge DR, SC \leftarrow 0$		
ADD	D_1T_4:	$DR \leftarrow M[AR]$		
	D_1T_5:	$AC \leftarrow AC + DR, E \leftarrow C_{out}, SC \leftarrow 0$		
LDA	D_2T_4:	$DR \leftarrow M[AR]$		
	D_2T_5:	$AC \leftarrow DR, SC \leftarrow 0$		
STA.	D_3T_4:	$M[AR] \leftarrow AC, SC \leftarrow 0$		
BUN	D_4T_4:	$PC \leftarrow AR, SC \leftarrow 0$		
BSA	D_5T_4:	$M[AR] \leftarrow PC, AR \leftarrow AR + 1$		
	D_5T_5:	$PC \leftarrow AR, SC \leftarrow 0$		
ISZ	D_6T_4:	$DR \leftarrow M[AR]$		
	D_6T_5:	$DR \leftarrow DR + 1$		
	D_6T_6:	$M[AR] \leftarrow DR$		
	D_6T_6DR:	if (DR = 0) then (PC <- PC + 1), SC<-0		

ASM#5

$LD\_AR = R'T_0 + R'T_2 + D'_7IT_3$; $CLR\_AR = RT_0$; $INR\_AR = D_5T_4$

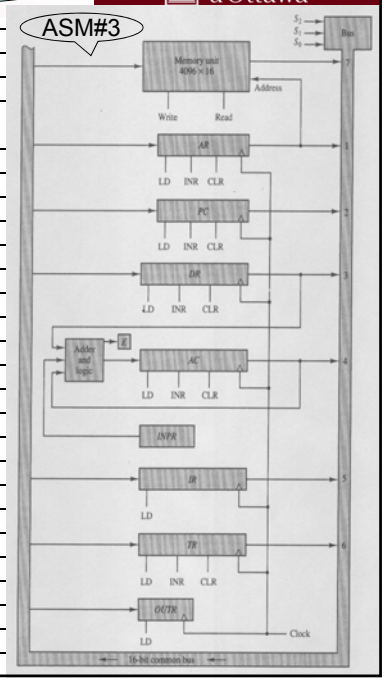
CEG 2136 Computer Architecture

uOttawa

Control Signals ASM#4

RTL ASM#2

ASM#3



			$S_2S_1S_0$	LD_{AR}	LD_{IR}	INC_{PC}	LD_{TR}	CLR_{PC}	LD_{DR}
Fetch	RT_0:	$AR \leftarrow PC$							
	RT_1:	$IR \leftarrow M[AR], PC \leftarrow PC + 1$							
Decode	RT_2:	$D_0, \ldots, D_7 \leftarrow$ Decode $IR(12\text{-}14)$,							
		$AR \leftarrow IR(0\text{-}11), I \leftarrow IR(15)$							
Indirect	D'_7IT_3:	$AR \leftarrow M[AR]$							
Interrupt:									
$T_0T_1T_2IEN(FGI + FGO)$:		$R \leftarrow 1$							
	RT_0:	$AR \leftarrow 0, TR \leftarrow PC$							
	RT_1:	$M[AR] \leftarrow TR, PC \leftarrow 0$							
	RT_2:	$PC \leftarrow PC+1, IEN \leftarrow 0, R \leftarrow 0, SC \leftarrow 0$							
		Memory-reference:							
AND	D_0T_4:	$DR \leftarrow M[AR]$							
	D_0T_5:	$AC \leftarrow AC \wedge DR, SC \leftarrow 0$							
ADD	D_1T_4:	$DR \leftarrow M[AR]$							
	D_1T_5:	$AC \leftarrow AC + DR, E \leftarrow C_{out}, SC \leftarrow 0$							
LDA	D_2T_4:	$DR \leftarrow M[AR]$							
	D_2T_5:	$AC \leftarrow DR, SC \leftarrow 0$							
STA.	D_3T_4:	$M[AR] \leftarrow AC, SC \leftarrow 0$							
BUN	D_4T_4:	$PC \leftarrow AR, SC \leftarrow 0$							
BSA	D_5T_4:	$M[AR] \leftarrow PC, AR \leftarrow AR + 1$							
	D_5T_5:	$PC \leftarrow AR, SC \leftarrow 0$							
ISZ	D_6T_4:	$DR \leftarrow M[AR]$							
	D_6T_5:	$DR \leftarrow DR + 1$							
	D_6T_6:	$M[AR] \leftarrow DR,$							
	D_6T_6DR':	if $(DR = 0)$ then $(PC \leftarrow PC + 1), SC \leftarrow 0$							

CEG 2136 Computer Archit

uOttawa

Control Signals ASM#4

RTL ASM#2

			To	From $S_2S_1S_0$			LD_{AR}	INR_{AR}	CLR_{AR}	LD_{PC}	INR_{PC}	CLR_{PC}	Write	Read	LD_{IR}	LD_{DR}	INR_{DR}	CLR_{DR}	LD_{TR}	INR_{TR}	CLR_{TR}	LD_{AC}	INR_{AC}	CLR_{AC}
Fetch	RT_0:	$AR \leftarrow PC$	LD_{AR}	010			1																	
	RT_1:	$IR \leftarrow M[AR], PC \leftarrow PC + 1$	LD_{IR}	111	Read	INR_{PC}					1				1	1								
Decode	RT_2:	$D_0, \ldots, D_7 \leftarrow$ Decode $IR(12\text{-}14)$,																						
		$AR \leftarrow IR(0\text{-}11), I \leftarrow IR(15)$	LD_{AR}	101	LD_I		1																	
Indirect	D'_7IT_3	$AR \leftarrow M[AR]$	LD_{AR}	111	Read		1							1										
Interrupt	$T_0T_1T_2$	$IEN(FGI + FGO)$: $R \leftarrow 1$	S_R																					
	RT_0:	$AR \leftarrow 0, TR \leftarrow PC$	LD_{TR}	010	CLR_{AR}				1															
	RT_1:	$M[AR] \leftarrow TR, PC \leftarrow 0$	Write	110	CLR_{PC}							1												
	RT_2:	$PC \leftarrow PC+1, IEN \leftarrow 0, R \leftarrow 0, SC \leftarrow 0$	INR_{PC}	R_{IEN}	R_R	CLR_{SC}					1													
		Memory-reference:																						
AND	D_0T_4:	$DR \leftarrow M[AR]$	LD_{DR}	111	Read									1		1								
	D_0T_5:	$AC \leftarrow AC \wedge DR, SC \leftarrow 0$	LD_{AC}	AND	CLR_{SC}																	1		
ADD	D_1T_4:	$DR \leftarrow M[AR]$	LD_{DR}	111	Read									1		1								
	D_1T_5:	$AC \leftarrow AC + DR, E \leftarrow C_{out}, SC \leftarrow 0$	LD_{AC}	ADD	CLR_{SC}																	1		
LDA	D_2T_4:	$DR \leftarrow M[AR]$	LD_{DR}	111	Read									1		1								
	D_2T_5:	$AC \leftarrow DR, SC \leftarrow 0$	LD_{AC}	DR	CLR_{SC}																	1		
STA.	D_3T_4:	$M[AR] \leftarrow AC, SC \leftarrow 0$	Write	100	CLR_{SC}																			
BUN	D_4T_4:	$PC \leftarrow AR, SC \leftarrow 0$	LD_{PC}	001	CLR_{SC}					1				1										
BSA	D_5T_4:	$M[AR] \leftarrow PC, AR \leftarrow AR + 1$	Write	010	INR_{AR}			1																
	D_5T_5:	$PC \leftarrow AR, SC \leftarrow 0$	LD_{PC}	001	CLR_{SC}					1				1										
ISZ	D_6T_4:	$DR \leftarrow M[AR]$	LD_{DR}	111	Read									1		1								
	D_6T_5:	$DR \leftarrow DR + 1$	INR_{DR}														1							
	D_6T_6:	$M[AR] \leftarrow DR,$	Write	011																				
	D_6T_6DR'	if $(DR = 0)$ then $(PC \leftarrow PC + 1), SC \leftarrow 0$	INR_{PC}		CLR_{SC}						1													

AR Control Logic

RTL ASM#2

- Scan Table 5-6 & find all the RTL statements in which AR is modified.

- $R'T_0$: $AR \leftarrow PC$
- $R'T_2$: $AR \leftarrow IR(0\text{-}11)$
- D'_7IT_3: $AR \leftarrow M[AR]$
- RT_0 : $AR \leftarrow 0$
- D_5T_4 : $AR \leftarrow AR + 1$

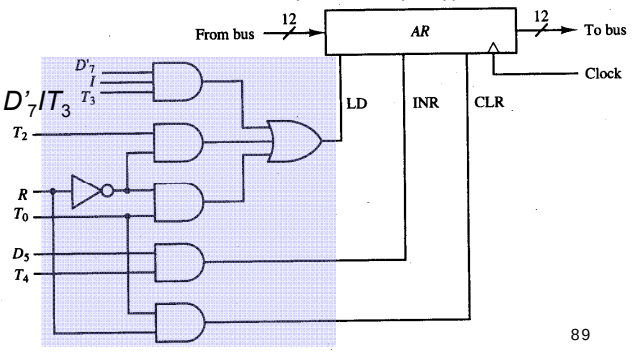
- The first three statements are performed by enabling the LD bit of AR (i.e., LD(AR)).
- The 4-th is done by activating the CLR
- The 5-fth statement is performed by enabling the INR bit of AR (i.e., INR(AR)).

ASM#5

$LD(AR) = R'T_0 \vee R'T_2 \vee D'_7IT_3$

$CLR(AR) = RT_0$

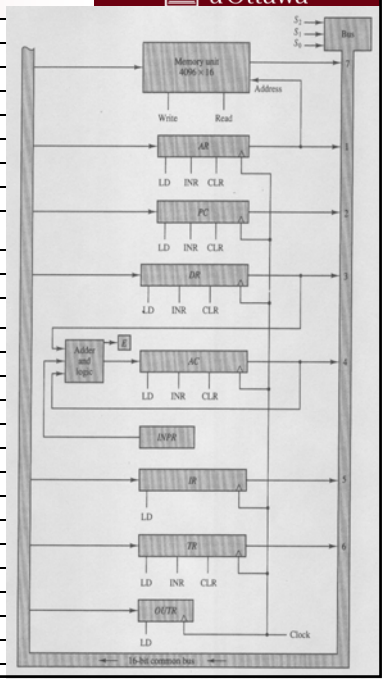
$INR(AR) = D_5T_4$



Control Signals ASM#4

		Register-reference:	R_{IEN}	S_{IEN}				
	$D_7I'T_3 = r$	(common to all RRI)						
	$IR(i) = B_i$	$(i = 0, 1, 2, \ldots, 11)$						
	r:	$SC \leftarrow 0$						
CLA	rB_{11}:	$AC \leftarrow 0$						
CLE	rB_{10}:	$E \leftarrow 0$						
CMA	rB_9:	$AC \leftarrow AC'$						
CME	rB_8:	$E \leftarrow E'$						
CIR	rB_7:	$AC \leftarrow shr\ AC, AC(15) \leftarrow E, E \leftarrow AC(0)$						
CIL	rB_6:	$AC \leftarrow shl AC, AC(0) \leftarrow E, E \leftarrow AC(15)$						
INC	rB_5:	$AC \leftarrow AC + 1$						
SPA	$rB_4AC'(15)$	$If\ (AC(15) = 0)\ then\ (PC \leftarrow PC + 1)$						
SNA	$rB_3AC(15)$:	$If\ (AC(15) = 1)\ then\ (PC \leftarrow PC + 1)$						
SZA	rB_2AC':	$If\ (AC = 0)\ then\ (PC \leftarrow PC + 1)$						
SZE	rB_1E':	$If\ (E = 0)\ then\ (PC \leftarrow PC + 1)$						
HLT	rB_0:	$S \leftarrow 0$						
		Input-output:						
	$D_7IT_3 = p$	(common to all IO instructions)						
	$IR(i) = B_i$	$(i = 6, 7, 8, 9, 10, 11)$						
	p:	$SC \leftarrow 0$						
INP	pB_{11}:	$AC(0\text{-}7) \leftarrow INPR, FGI \leftarrow 0$						
OUT	pB_{10}:	$OUTR \leftarrow AC(0\text{-}7), FGO \leftarrow 0$						
SKI	pB_9FGI:	$If\ (FGI = 1)\ then\ (PC \leftarrow PC + 1)$						
SKO	pB_8FGO:	$If\ (FGO = 1)\ then\ (PC \leftarrow PC + 1)$						
ION	pB_7:	$IEN \leftarrow 1$		1				
IOF	pB_6:	$IEN \leftarrow 0$	1					

RTL ASM#2



IEN FF Control Logic

- The control logic circuit for the individual flip-flops of the control unit can be determined in a similar manner.
- For example, Table 5-6 (of the textbook) shows that flip-flop IEN is loaded in the following statements:

RTL ASM#2

$pB_7: \ IEN \leftarrow 1 \quad S_{IEN} = 1$

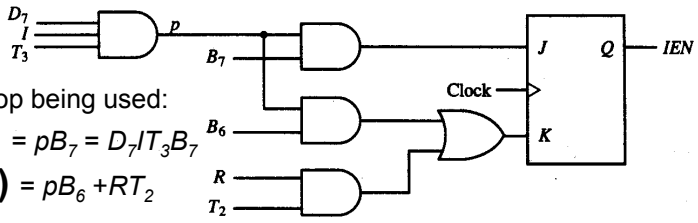
$pB_6: \ IEN \leftarrow 0 \quad R_{IEN1} = 1$

$RT_2: \ IEN \leftarrow 0 \quad R_{IEN2} = 1$

Control Signals ASM#4

where $p = D_7 IT_3$, $B_7 = IR(7)$, and $B_6 = IR(6)$

ASM#5



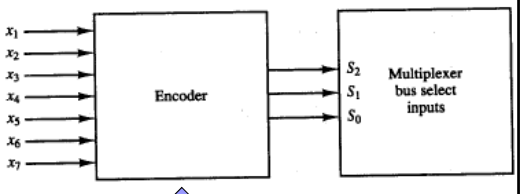
Should a JK flip-flop being used:

$J_{IEN} = \Sigma(S_{IEN}) = pB_7 = D_7 IT_3 B_7$

$K_{IEN} = \Sigma(R_{IEN}) = pB_6 + RT_2$

Bus Control

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	S_2	S_1	S_0
AR	1	0	0	0	0	0	0	0	0	1
PC	0	1	0	0	0	0	0	0	1	0
DR	0	0	1	0	0	0	0	0	1	1
AC	0	0	0	1	0	0	0	1	0	0
IR	0	0	0	0	1	0	0	1	0	1
TR	0	0	0	0	0	1	0	1	1	0
M	0	0	0	0	0	0	1	1	1	1



Source Register (from Table 5-6):

AR: $x_{AR} = x_1 = D_4T_4 \vee D_5T_5$

PC: $x_{PC} = x_2 =$

DR: $x_{DR} = x_3 =$

AC: $x_{AC} = x_4 =$

IR : $x_{IR} = x_5 =$

TR: $x_{TR} = x_6 =$

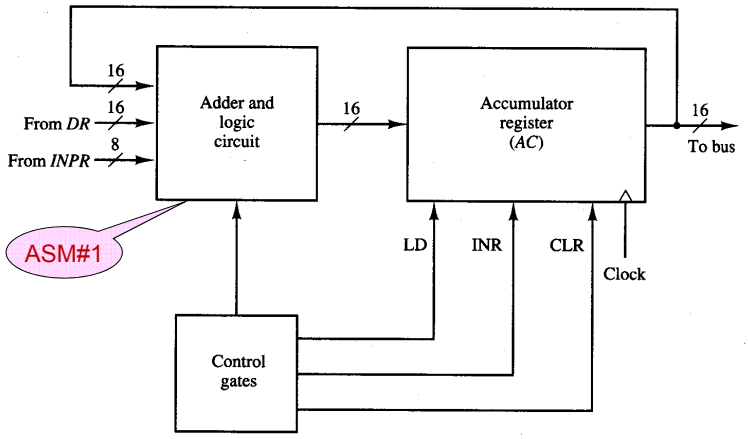
M : $x_M = x_7 = R'T_1 \vee D'_7IT_3 \vee (D_0 \vee D_1 \vee D_2 \vee D_6)T_4$

$S_1 = x_1 + x_3 + x_5 + x_7$

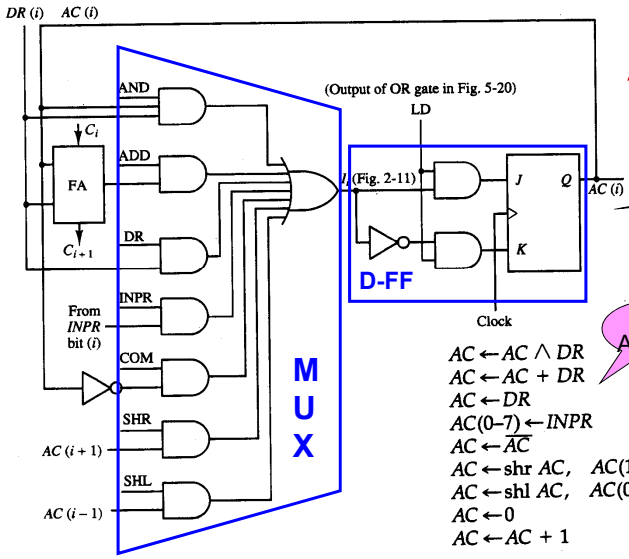
$S_1 = x_2 + x_3 + x_6 + x_7$

$S_2 = x_4 + x_5 + x_6 + x_7$

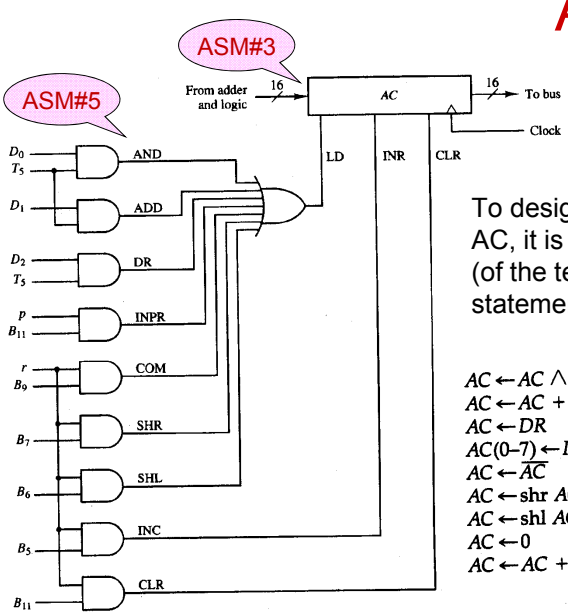
ALU Block Diagram



AC+ALU



AC Control Logic (ASM#5)



To design the logic associated with AC, it is necessary to scan Table 5-6 (of the textbook) and extract all the statements in which AC is loaded:

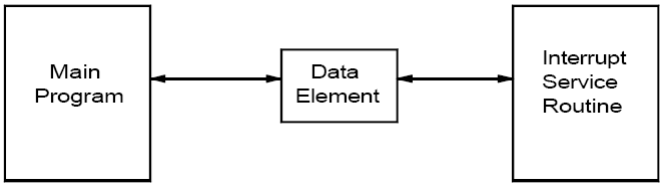
RTL ASM#2

$AC \leftarrow AC \wedge DR$	AND with DR
$AC \leftarrow AC + DR$	Add with DR
$AC \leftarrow DR$	Transfer from DR
$AC(0\text{–}7) \leftarrow INPR$	Transfer from $INPR$
$AC \leftarrow \overline{AC}$	Complement
$AC \leftarrow \text{shr } AC$, $AC(15) \leftarrow E$	Shift right
$AC \leftarrow \text{shl } AC$, $AC(0) \leftarrow E$	Shift left
$AC \leftarrow 0$	Clear
$AC \leftarrow AC + 1$	Increment

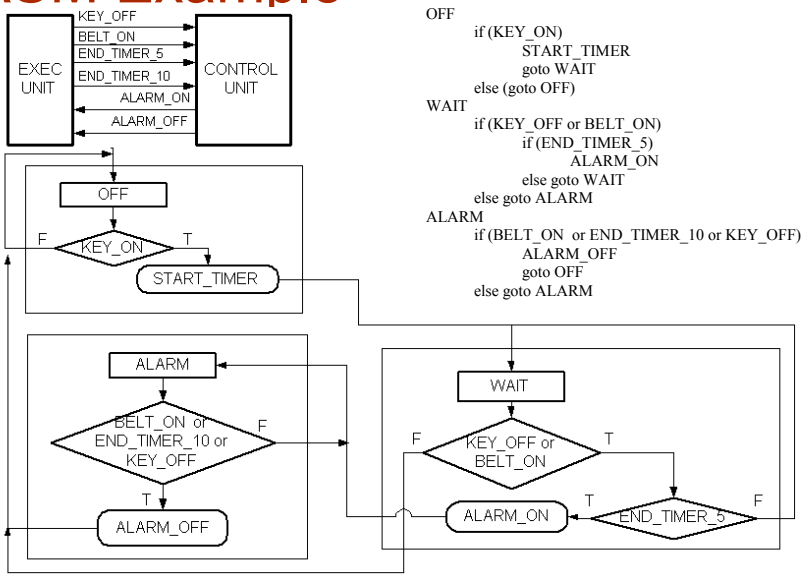
Data Exchange with ISRs

- Use global data
- May need to disable interrupts for critical regions of code using this data

Input-Output and Interrupt



ASM Example




```
OFF
        if (KEY_ON)
                START_TIMER
                goto WAIT
        else (goto OFF)
WAIT
        if (KEY_OFF or BELT_ON)
                if (END_TIMER_5)
                        ALARM_ON
                else goto WAIT
        else goto ALARM
ALARM
        if (BELT_ON  or END_TIMER_10 or KEY_OFF)
                ALARM_OFF
                goto OFF
        else goto ALARM
```

97