

Chapter 7

Processing Unit

- Processing Unit
 - ◆ Datapath
 - ◆ Internal Bus Architecture
 - ◆ Internal Processing
 - Hard-wired
 - Microinstruction method (briefly)
- Next Lecture
 - ◆ Pipelining



Fundamental Concepts

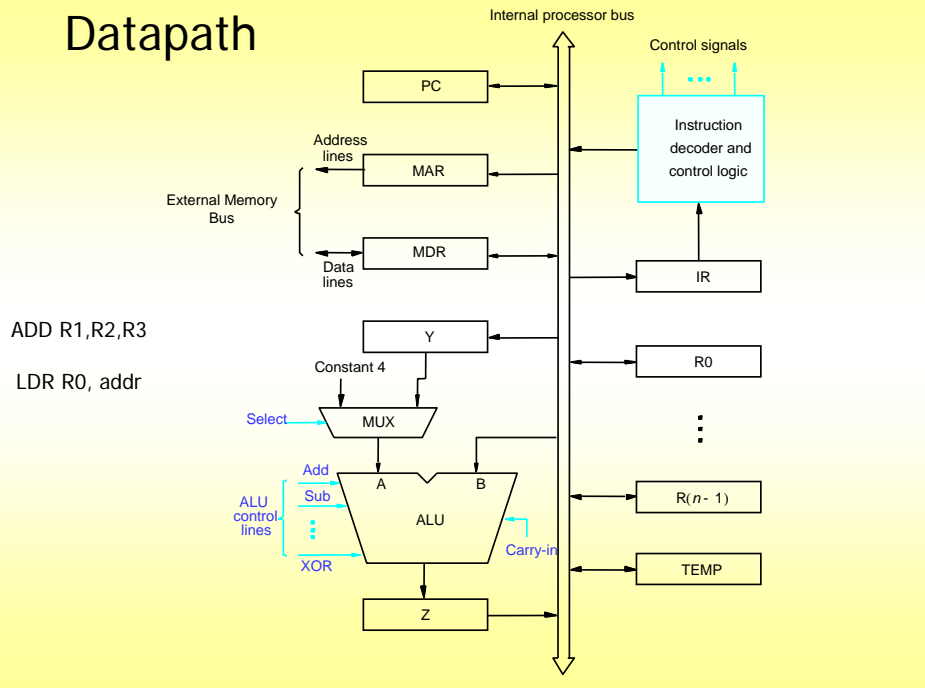
- For simplicity, assume that each instruction occupies one memory word

Instruction execution stages

- Fetch stage
 - ◆ Fetch the contents of the memory location pointed to by PC and load it into IR : $[IR] \leftarrow [[PC]]$
 - ◆ Increment the contents of PC : $[PC] \leftarrow [PC] + 4$
- Execution stage
 - ◆ Carry out the instruction fetched
 - ◆ Accessing ALU, register, memory, bus (using internal and external resources)



Datapath



Datapath

- ALU and all registers are on a single common bus
- The common bus is internal to the CPU (do not be confused with external buses connecting CPU to memory and I/O devices)
- The external memory bus connects to the CPU via MDR and MAR
- The number and function of registers R0 through R(n-1) varies from one CPU to another
- Registers can either be general purpose or special purpose
- Register Y, Z and TEMP are transparent to the program, they are used only by the CPU for temporary storage
- **Datapath:** ALU, registers, and the interconnecting bus
- Assume all the registers have a clock input



Processing

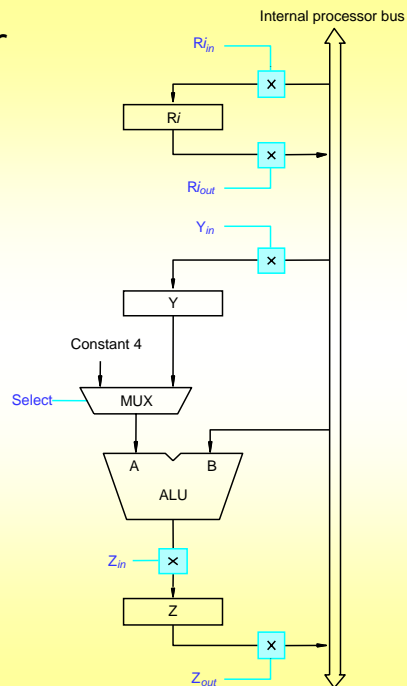
- Most of the operations needed to execute an instruction can be carried out by performing one or more of the following functions
 - ◆ Fetch the contents of a given memory location and load them into a CPU register (e.g., LD R0, addr)
 - ◆ Store a word of data from a CPU register into a given location in memory (e.g., STO R0, addr)
 - ◆ Transfer a word of data from one CPU register to another or to the ALU (e.g., MOV R2,R3 or ADD R1,#1)
 - ◆ Perform an arithmetic or logic operation and store the result in a CPU register (e.g., ADD R1,R2,R3)

5



Register Transfer

- Registers need input and output gating
- Ri_{in} control signal for input of Ri : when $Ri_{in}=1$, data available on the common bus is loaded in Ri
- Ri_{out} control signal for output of Ri when $Ri_{out}=1$, the content of Ri are placed on the bus
- Example: transfer the content of R1 to R4
 - ◆ Enable output of R1 :
 - ◆ $R1_{out}=1$
 - ◆ Enable input of R4:
 - ◆ $R4_{in}=1$



Arithmetic & Logic Operation

- ALU is a combinational circuit that has no internal storage
- To add two numbers, the two operands have to be available to the ALU simultaneously
- Register Y holds one of the two numbers
- The other number is gated onto the bus
- The result is stored temporarily in Z

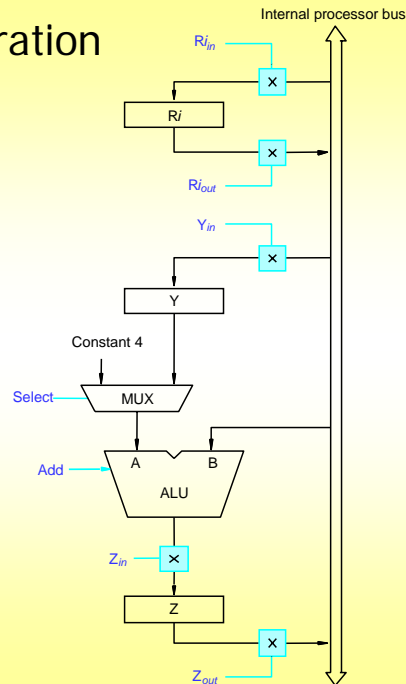
Example : ADD R1, R2, R3 ($R3=R1+R2$)

Step 1, $R1_{out}=1$ and $Y_{in}=1$

Step 2, $R2_{out}=1$, $Add=1$, $Z_{in} = 1$

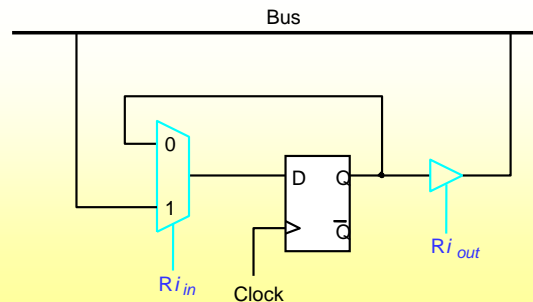
Step 3, $Z_{out} = 1$, $R3_{in}$: content of Z are transferred to R3

- Step3 cannot be done concurrently with step2, because only one register can be connected to the bus at any given time



Register Gating and Timing of Data Transfers

- Each bit of a register consists of a flip-flop (FF)
- While the $Ri_{in}=1$, the state of each FF changes to its corresponding data on the bus
- At a clock edge while $Ri_{in}=1$, the data stored in the FF immediately before the transition is locked until $Ri_{in}=1$ again
- The output of the register is capable of being disconnected from the bus, placing a 0 or placing a 1 on the bus: tri-state



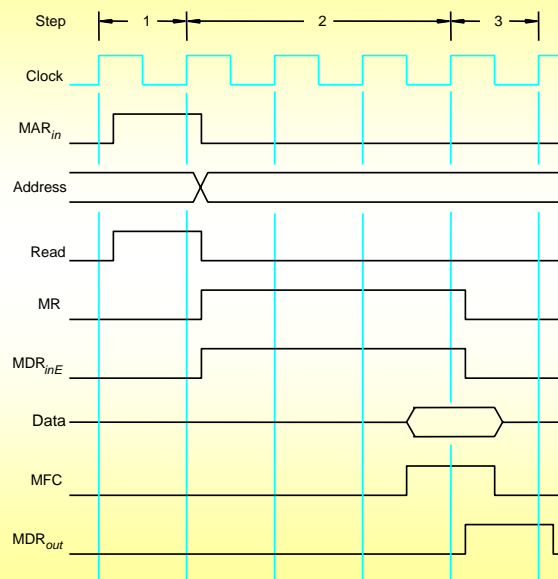
Fetch Operation

- CPU has to specify the address of the memory location and request a read operation
 1. Send an address ($MAR \leftarrow [R1]$) to memory
 - ◆ CPU transfers the address of the required word into MAR
 2. Start a Read operation
 - ◆ CPU uses the control lines of the memory bus to indicate a Read operation is needed
 3. Wait for MFC (memory function complete) response
 - ◆ CPU waits until it receives an answer from memory informing that the Read has been completed.
 - ◆ When MFC is set to 1, it indicates that the specified location has been read and the content is available on the data lines of the memory bus
 - ◆ The duration of this step depends on the speed of memory
 - ◆ Overall execution time of an instruction can be decreased by useful work, example: incrementing the PC
 4. $R2 \leftarrow [MDR]$
 - ◆ The information on the memory bus is loaded into MDR
 - ◆ The contents of the MDR are moved into a destination register

9



Read Timing



Synchronous Asynchronous Transfer

- **Asynchronous transfer**
 - ◆ One device initiates the transfer and waits until the other device responds
 - ◆ Enables transfer of data between two independent devices that have different speeds of operation
- **Synchronous transfer**
 - ◆ One of the control lines of the bus carries pulses from a clock running continuously at a fixed frequency
 - ◆ These pulses provide common timing signals to the CPU and main memory
 - ◆ Simpler implementation
 - ◆ Cannot accommodate devices of widely varying speed, except by reducing the speed of all devices to that of the slowest one
- **Mixed**

11



Store Operation

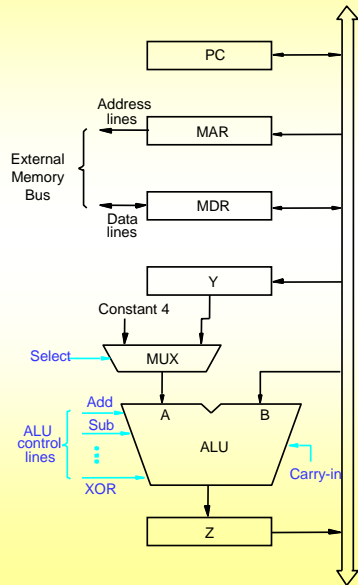
- STORE R2, [R1]

Step 1, MAR \leftarrow [R1]
Step 2, MDR \leftarrow [R2], Write
Step 3, Wait for MFC
- Steps 1 and 2 can be carried out simultaneously if the architecture allows it
- This is not possible with a single CPU bus
- Step 3 may be overlapped with other operations, provided that there is no conflict

12



Execution of a Complete Instruction Example Add (R3),R1

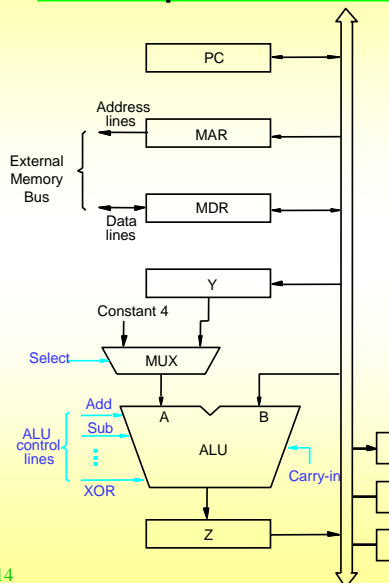


1. Instruction Fetch
2. Fetch operand(s)
3. Perform the addition
4. Store results into R1

13



Execution of a Complete Instruction Example Add (R3),R1



Step Action

- | Step | Action |
|------|--|
| 1 | PC_{out} , MAR_{in} , Read, Select4, Add, Z_{in} |
| 2 | Z_{out} , PC_{in} , WMFC |
| 3 | MDR_{out} , IR_{in} |
| 4 | $R3_{out}$, MAR_{in} , Read |
| 5 | $R1_{out}$, Y_{in} , WMFC |
| 6 | MDR_{out} , SelectY, Add, Z_{in} |
| 7 | Z_{out} , $R1_{in}$, End |

14



Steps 1, 2 and 3. Fetch & Increase PC

- $PC_{out}, MAR_{in}, Read, Select\ 4, Add, Z_{in}$
 - ◆ Load the content of the PC into MAR, and send a read request
 - ◆ **$PC_{out}, MAR_{in}, Read$**
 - ◆ While waiting for a response, increment PC
 - ◆ Select constant 4 in MUX
 - ◆ ALU input B is receiving the current value in PC,
 - ◆ Specify Add operation
 - ◆ In **step 2**, move updated value back into PC and wait MFC (**$Z_{out}, PC_{in}, WMFC$**)
 - ◆ In **step 3**, the word fetched from memory is loaded into IR
 - ◆ **MDR_{out}, IR_{in}**

15



Steps 4, 5, 6 and 7

Step 4 and 5:

- ◆ Fetch the first operand: the content of the memory location pointed to by R3
 - ◆ **$R3_{out}, MAR_{in}, Read$**
 - ◆ **$R1_{out}, Y_{in}, WMFC$**

Step 6:

- ◆ Perform the addition
 - ◆ **$MDR_{out}, Select\ Y, Add, Z_{in}$**

Step 7:

- ◆ Load results into R1
 - ◆ **$Z_{out}, R1_{in}, End$**

16



Branch Instructions

Step Action

- | | |
|---|--|
| 1 | PC_{out} , MAR_{in} , Read, Select4, Add, Z_{in} |
| 2 | Z_{out} , PC_{in} , Y_{in} , WMFC |
| 3 | MDR_{out} , IR_{in} |
| 4 | Offset-field-of- IR_{out} , Add, Z_{in} |
| 5 | Z_{out} , PC_{in} , End |
-

Control sequence for an unconditional branch instruction

17



Steps of Unconditional Branching

- Branching: branch address is obtained by adding an offset X (given in the branch instruction) to the current value of the PC
1. Fetch instruction
 - ◆ PC_{out} , MAR_{in} , Read, Select 4, Add, Z_{in}
 - ◆ Z_{out} , PC_{in} , Y_{in} , WMFC
 - ◆ MDR_{out} , IR_{in}
 2. Execute
 - ◆ Offset-field-of- IR_{out} , Add, Z_{in}
 - ◆ Z_{out} , PC_{in} , End
- PC is incremented during the fetch phase before knowing the type of instruction being executed
 - When the offset is added to the contents of the PC, the PC has already been updated to the instruction following the branch
 - The offset is the difference between the branch address and the address immediately following the branch

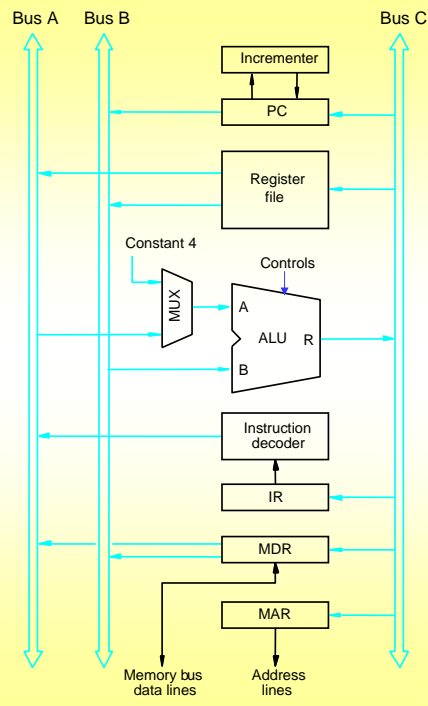
18



Steps of Conditional Branching

- Check the status of the condition codes before loading the new value into the PC
 - ◆ Offset-field-of- IR_{out} , Add, Z_{in}
 - ◆ If conditions do not match, then End

19



Multiple-Bus

- All general purpose registers are combined into a register file
- Register file can be implemented in VLSI using an array of memory cells similar to the one used in RAM chips
- The register file has two outputs, allowing the contents of two registers to be placed on buses A and B simultaneously
- Compared to the single bus organization, this organization requires fewer control steps (i.e., faster)

Multiple Bus Operation Example

Add R4,R5,R6

Step Action

1	PC_{out} , $R=B$, MAR_{in} , Read, IncPC
2	WMFC
3	MDR_{outB} , $R=B$, IR_{in}
4	$R4_{outA}$, $R5_{outB}$, Select BusA, Add, $R6_{in}$, End

Control sequence for the instruction

- Steps 1...3: Instruction fetch
- Step 4: Addition

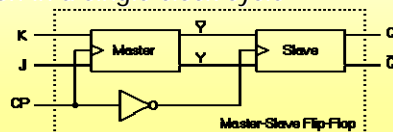
21



Multiple Bus Operation Example

Add R4,R5,R6

- Buses A and B are used to transfer the source operands
- Bus C is used to transfer the destination
- The path from the source to the destination goes through the ALU (where the operation is performed)
- Copies of one register to another also go through the ALU
- Temporary storage registers (Y, Z) are not needed
- Ensuring that a register can serve as both a source and a destination
 - ◆ not possible if registers are simple latches
 - ◆ the register file must be implemented using edge triggered master-slave flip-flops
- The three-bus architecture allows execution of register to register operation in a single clock cycle



22



Enhancements

- Overlap fetch and execute phases
 - ◆ Instruction unit: fetch instructions and place them into a queue ready for execution
 - ◆ It generates memory addresses based on the address of the last instruction fetched
 - ◆ Attempts to **prefetch** the correct instruction on branches based on a history of previous branches
 - ◆ **Prefetching with branch prediction**
- Including a fast **cache** on the same chip as the CPU
 - ◆ Hides the memory response time
 - ◆ If the desired data is found in the cache: cache hit; otherwise a cache miss
 - ◆ If a cache miss occurs, it is necessary to go to the main memory

23



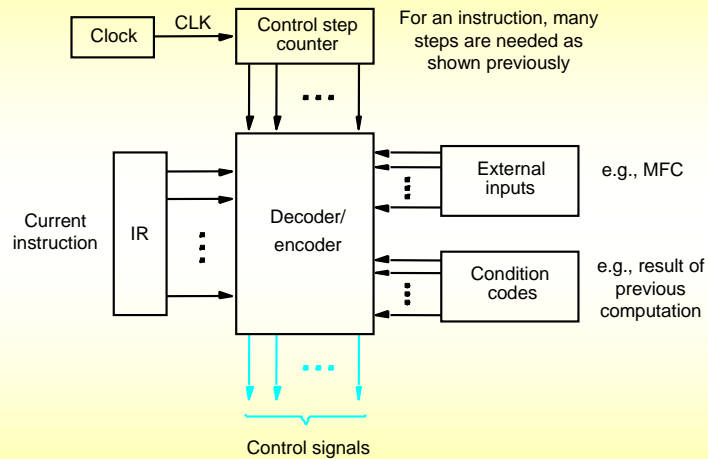
Generating Control Signals

- To execute an instruction, the CPU must generate control signals corresponding to the current instruction
- Two types of approaches
 - ◆ Hard-wired
 - ◆ Microprogrammed

24



Hard-wired Control



25



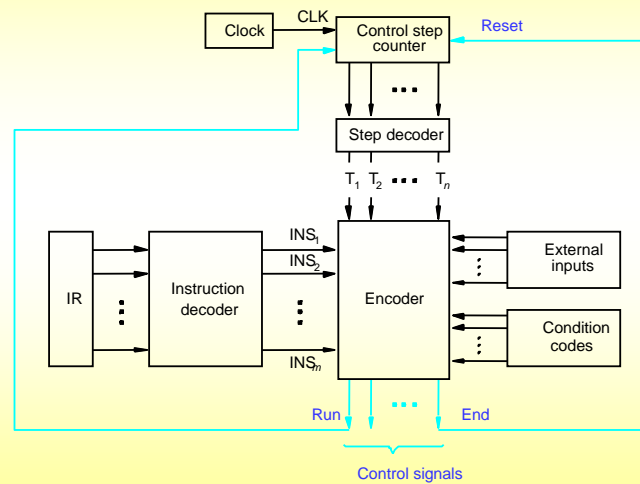
Hard-wired Control

- Several non overlapping time slots (i.e., steps) are required for executing an instruction
- Each time slot must be long enough for the functions specified in the step to be completed
- Assume all time slots are equal
- The control unit may be based on the use of a counter driven by CLK
- The required control signals are uniquely determined by
 - ◆ contents of the control step counter
 - ◆ contents of the instruction register fetched
 - ◆ contents of the condition code and other status flags (e.g. MFC status signal)
- The decoder/encoder is a combinational circuit that generates the required control outputs depending on the state of all its inputs

26



Separation of Decoding and Encoding Functions



27



Separation of Decoding and Encoding Functions

- Diagram with decoding and encoding function separated
- The step decoder provides a separate signal line for each step in the control sequence
- The output of the instruction decoder consists of a separate line for each machine instruction
- All input signals to the encoder block should be combined to generate individual control signals (e.g. Yin, PCout, Add, End)
- Examples

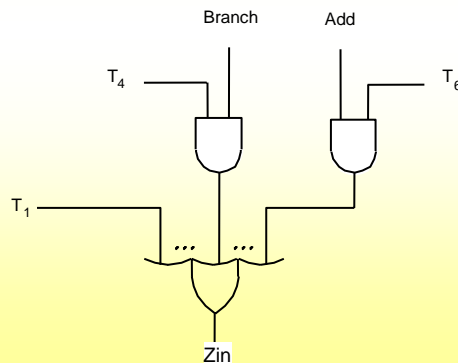
28



Generation of the Zin Control Signal

Example encoder structure, $Z_{in} = T_1 + T_6 \cdot ADD + T_4 \cdot BR + \dots$

- Zin is turned on during
 - ◆ slot T1 for all instructions
 - ◆ slot T6 for an ADD instruction (e.g., Add (R3),R1)
 - ◆ slot T4 for an unconditional branch



29



Example: Add (R3),R1

Step	Action
1	PC_{out} , MAR_{in} , Read, Select4, Add, Z_{in}
2	Z_{out} , PC_{in} , Y_{in} , WMFC
3	MDR_{out} , IR_{in}
4	$R3_{out}$, MAR_{in} , Read
5	$R1_{out}$, Y_{in} , WMFC
6	MDR_{out} , SelectY, Add, Z_{in}
7	Z_{out} , $R1_{in}$, End

Control sequence for instruction Add (R3),R1
 (Y_{in} at step 2 is there b/c steps 1~3 are common for all instructions)

30



Unconditional Branch

Step Action

- | | |
|---|---|
| 1 | PC _{out} , MAR _{in} , Read, Select4, Add, Z _{in} |
| 2 | Z _{out} , PC _{in} , Y _{in} , WMFC |
| 3 | MDR _{out} , IR _{in} |
| 4 | Offset-field-of-IR _{out} , Add, Z _{in} |
| 5 | Z _{out} , PC _{in} , End |

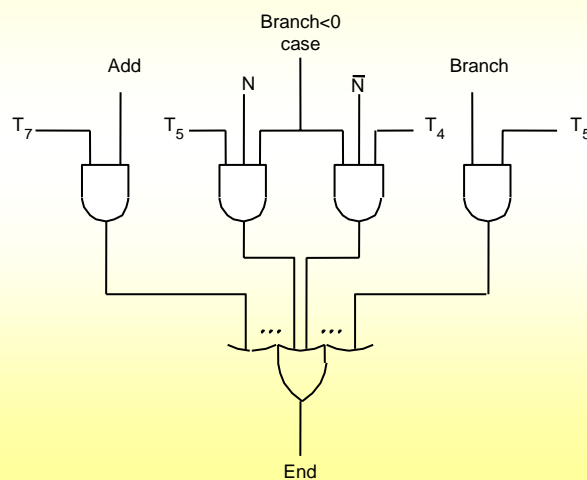
Control sequence for an unconditional branch instruction

31



Generation of the End Control Signal

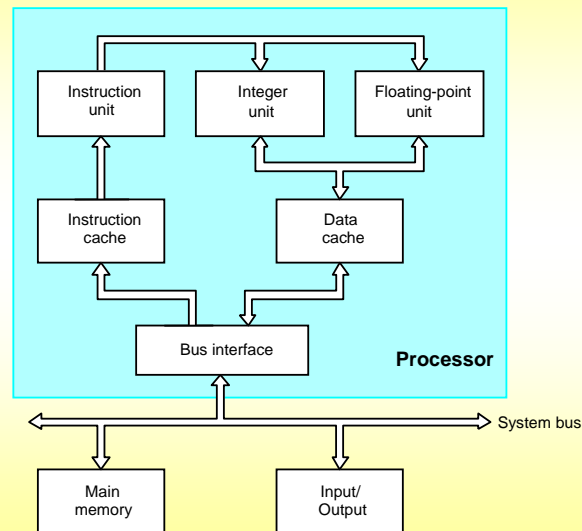
- Example encoder structure, $End = T_7 \cdot ADD + T_6 \cdot BR + (T_6 \cdot N + T_4 \cdot N') \cdot BRN + \dots$



32



A Complete CPU



33



A Complete CPU

- An instruction unit that fetches instructions from an instruction cache, or from main memory on a cache miss
- Separate processing units to deal with integer and floating point
- Data cache is between the processing units and main memory
- Separate caches for instruction and data (split cache)
- Other processors may have one cache for both data and instructions (unified cache)
- The CPU is connected to the system bus (rest of the computer) through a bus interface

Alternatives

- More than two processing units: several units of the same type to increase parallelism
- Processors that execute instructions at a rate faster than one instruction per cycle are called : *superscalar*

34



Microprogrammed Control Approach

Add (R3), R1

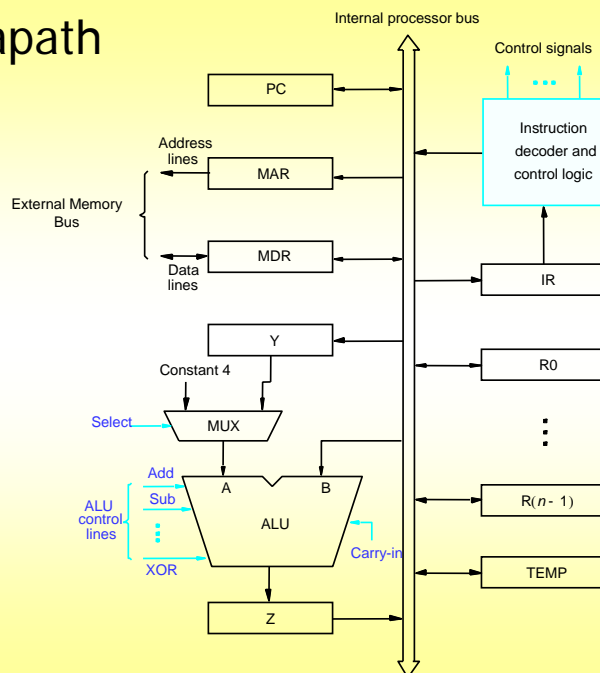
Micro - instruction	PC _{in}	PC _{out}	MAR _{in}	Read	MDR _{out}	IR _{in}	Y _{in}	Select	Add	Z _{in}	Z _{out}	R1 _{out}	R1 _{in}	R3 _{out}	VMFC	End	?
1	0	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0	
2	1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	
3	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	
4	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	
5	0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	
6	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	
7	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	

Figure 7.15 An example of microinstructions for Figure 7.6.

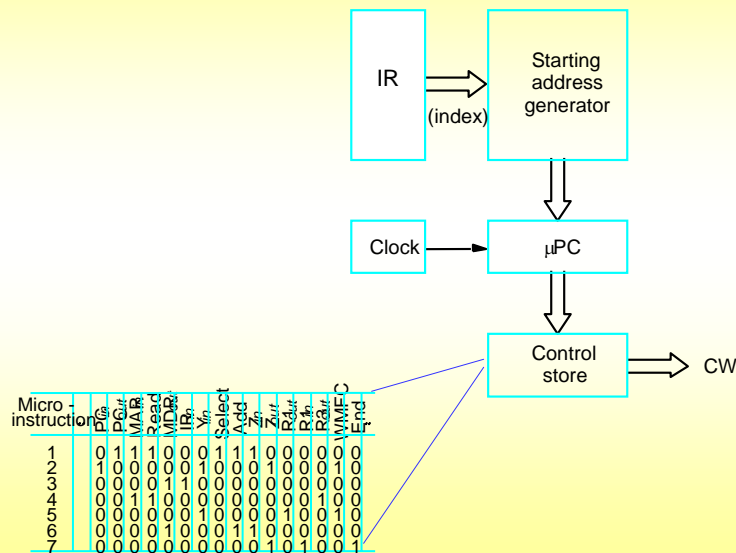
35



Datapath



Basic Organization of a Microprogrammed Control Unit



37



Microprogrammed Control

- Control signals are generated by a program similar to machine language programs
- Individual bits of a control word (CW) correspond to control signals
- Each of the control steps defines a unique combination of 1s and 0s in the CW
- Microroutine**: a sequence of CWs corresponding to the control sequence of a machine instruction
- Individual control words are called **microinstructions**

microroutine \approx subroutine
 microinstruction \approx instruction
 microprogram counter \approx program counter

38



Basic Organization of a Microprogrammed Control Unit

- Assume that the microroutines for all instructions are stored in special memory called a **control store**
- The control unit can generate the control signals for any instruction by sequentially reading the CWs in the corresponding microroutine
- A **microprogram counter (μ PC)** is used to point to the next microinstruction
- When a new instruction is fetched into IR, the starting address generator loads the starting address of the corresponding microroutine into the μ PC
- The μ PC is incremented to access successive microinstructions

39



Branch Instructions

- How does the control unit check the status of the condition flags or status flags on conditional branches
- The microinstruction set needs to be expanded to include conditional branch microinstructions
- In addition to the branch address, these microinstructions specify the flag or bit that should be checked as a condition

Example: Microroutine for the instruction Branch on negative

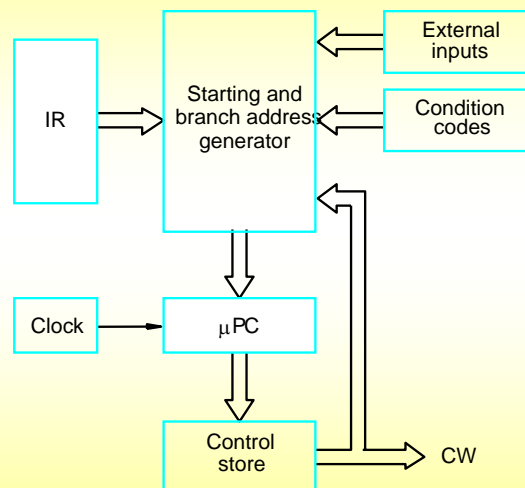
Address	microinstruction
0	PC_{out} , MAR_{in} , Read, Clear Y, Set carry-in to ALU, Add, Z_{in}
1	Z_{out} , PC_{in} , WMFC
2	MDR_{out} , IR_{in}
3	Branch to starting address of an appropriate microroutine
..
25	PC_{out} , Y_{in} if $N=0$ then branch to microinstruction 0
26	Offset field of IR_{out} , Add, Z_{in}
27	Z_{out} , PC_{in} , End

After loading the instruction into IR, a branch microinstruction transfers control to the microroutine starting at location 25

40



Allowing Conditional Branch in Microprogram



41



Allowing Conditional Branch in Microprogram

Support for microprogram branching

- Starting and branch address generator
- The block loads a new μ PC when a microinstruction requires a branch
- Input to the block include: status flags, condition flags, IR
- The μ PC is incremented by one every time except in the following situations
 - ◆ When a new instruction is loaded into IR, μ PC is loaded with the starting address of the microroutine for that instruction
 - ◆ When a branch microinstruction is encountered and the branch condition is satisfied
 - ◆ When an End microinstruction is encountered: the μ PC is loaded with the first microinstruction (i.e., address 0) to fetch a new instruction to IR

42



Implementation of Microinstructions

1st alternative : Assign one bit position to each control signal

- - Resulting in long microinstructions
- Only few bits are set to 1 in any given microinstruction
- Example of the single bus organization
 - ◆ 4 general purpose registers
 - ◆ Some of the connections to the CPU are permanently enabled: the output of IR to the decoding circuit, the two inputs of the ALU
 - ◆ A total of 20 gating signals are needed
 - ◆ Additional signals include : Read, Write, Clear Y, Set Carry-in, WMFC and End
 - ◆ Signals to specify with ALU operation to perform: 16 operations
→ 16 bits
- **Total of 42 bits of control signals**

43



Microinstructions

2nd alternative: Encoded control signals

- Most signals are not needed simultaneously
- Many signals are mutually exclusive
- Only one function of the ALU is needed at a time
- Read and write signals to memory cannot be active at the same time
- The source for a data transfer must be unique: cannot gate the contents of two registers simultaneously on a single bus
- Signals can be grouped so that mutually exclusive signals are placed in the same group
- 4 bits are needed to represent the 16 functions of the ALU
- Register output control signals can be in a group consisting of PC_{out} , MDR_{out} , Z_{out} , $Address_{out}$, $R0_{out}$, $R1_{out}$, $R2_{out}$, $R3_{out}$ and $TEMP_{out}$: encoding with 4 bits
- **Control signals can be grouped and encoded to reduce the number of bits in microinstructions**

44



Field-encoded Microinstructions

Microinstruction

F1	F2	F3	F4	F5
F1 (4 bits)	F2 (3 bits)	F3 (3 bits)	F4 (4 bits)	F5 (2 bits)
0000: No transfer 0001: PC_{out} 0010: MDR_{out} 0011: Z_{out} 0100: $R0_{out}$ 0101: $R1_{out}$ 0110: $R2_{out}$ 0111: $R3_{out}$ 1010: $TEMP_{out}$ 1011: $Offset_{out}$	000: No transfer 001: PC_{in} 010: IR_{in} 011: Z_{in} 100: $R0_{in}$ 101: $R1_{in}$ 110: $R2_{in}$ 111: $R3_{in}$	000: No transfer 001: MAR_{in} 010: MDR_{in} 011: $TEMP_{in}$ 100: Y_{in}	0000: Add 0001: Sub : : : 1111: XOR 16 ALU functions	00: No action 01: Read 10: Write

Total 20 bits

F6	F7	F8	...
F6 (1 bit)	F7 (1 bit)	F8 (1 bit)	
0: SelectY 1: Select4	0: No action 1: WMFC	0: Continue 1: End	

45



Field-encoded Microinstructions

- Most fields must include one inactive code for the case where no action is required
- No active code is reserved in the ALU; thus the ALU is active at all times; the control on Z_{in} makes sure that the result of an operated is gated only when appropriate
- Grouping control signals requires more hardware to decode bit patterns
- The cost of the additional hardware is amortized by having the smaller control store

46



Microprogram Sequencing

- Each machine instruction is implemented by a microroutine
- A microroutine is entered by decoding an instruction into a starting address that is loaded into the μ PC
- Branching capabilities are introduced through branch microinstructions
- Having a separate microroutine for each machine instruction leads to a large control store
- There are several instructions and several addressing modes
- Organize the microprogram so that microroutines share as many common parts as possible
- Sharing common parts requires several branch microinstructions
- Longer time is needed to execute branch microinstructions

47

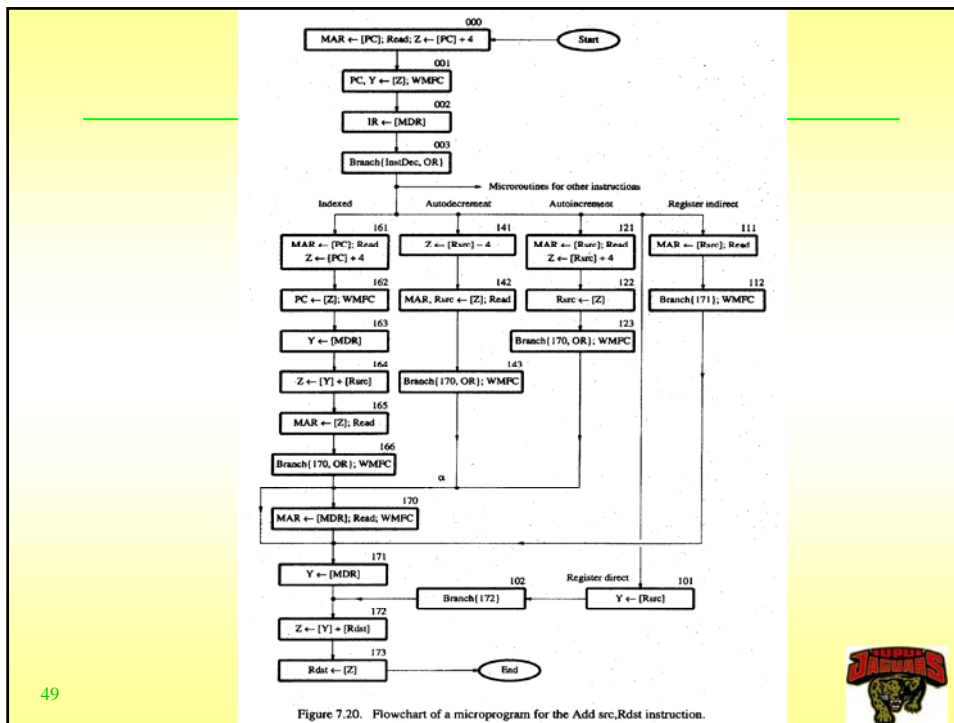


Example: ADD src, Rdst

- Assume that the source operand can be specified using: register, autoincrement, autodecrement, indirect and indirect forms of all of these modes
- A suitable microprogram will combine all the modes
- See next slide

48





49



Branch Addressing

Branch address modification using bit-ORing

- Branches are not always made to a single branch address
- A direct consequence of combining microroutines
- At the point α of the previous example, it is necessary to choose between the actions required by direct and indirect addressing modes
- Indirect mode: microinstruction at location 170 (fetch an operand from memory)
- Direct mode: microinstruction at location 171 (fetching an operand is bypassed)
- Efficient branching: Bit-ORing technique
 - ◆ having the preceding instruction specify 170
 - ◆ use an OR gate to change the least significant bit of 170 if direct addressing mode

Wide Branch Addressing

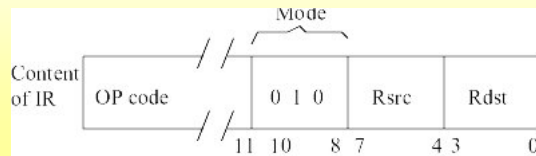
- Generating branch addresses means that the circuitry becomes more complex
 - ◆ E.g., the machine instruction fetch is completed, and an appropriate microroutine should be selected according to addressing modes
- A simple and inexpensive way of generating required branch addresses is using a PLA
- The OP code of a machine instruction is translated into a starting address

50



Detailed Example: ADD (Rsrc)+, Rdst

Address (octal)	Microinstructions
000	PC _{out} , MAR _{in} , Read, Clear Y, Set carry-in, Add, Z _{in}
001	Z _{out} , PC _{in} , WMFC
002	MDR _{out} , IR _{in}
003	μBranch μPC ← 101 (from instruction decoder); μPC _{5,4} ← [IR_ _{10,9}]; μPC ₃ ← [IR _{10}] · [IR_{9}] · [IR_{8}]}}}
121	Rsrc _{out} , MAR _{in} , Read, Clear Y, Set carry-in, Add, Z _{in}
122	Z _{out} , Rsrc _{in}
123	μBranch { μPC ← 170 ; μPC ₀ ← [IR _{8}] }, WMFC}
170	MDR _{out} , MAR _{in} , Read, WMFC
171	MDR _{out} , Y _{in}
172	Rdst _{out} , Add, Z _{in}
173	Z _{out} , Rdst _{in} , End



51



Detailed Example: ADD (Rsrc)+, Rdst

- 3 bit field used to specify the addressing mode for the source operand
- Bits 10 and 9 denote indexed (11), autodecrement (10), autoincrement (01), and register modes (00)
- Bit 8 is used to specify the indirect version of the addressing mode
- E.g., 010: direct version of the autoincrement
- Assume CPU has 16 registers that can be used for addressing purposes
- Bits 7 through 4 specify the source operand
- Bits 3 through 0 specify the destination operand

52



Detailed Example: ADD (Rsrc)+, Rdst

- Any of the 16 general purpose registers may be involved in determining the source and destination operands
- Microinstructions refer to control signals only as $Rsrc_{out}$, $Rsrc_{in}$, $Rdst_{out}$ and $Rdst_{in}$
- These signals are translated into a specific register by the decoding circuit connected to Rsrc and Rdst address fields of IR
- Requires a two level decoding
 - ◆ The microinstruction field must be decoded to determine that an Rsrc or Rdst is involved
 - ◆ The decoded output is used to gate the contents of the Rsrc or Rdst field in the IR into a second decoder which produces the gating signals for the actual registers R0 through R15

53



Detailed Example: ADD (Rsrc)+, Rdst Using Bit-Oring Scheme

- Consider Address 123:
 $123 \mu\text{Branch} \{ \mu\text{PC} \leftarrow 170 ; \mu\text{PC}_0 \leftarrow [IR_8]'\}$, WFMC
 unmodified version causes a branch to location 170
- When a direct addressing mode appears, the fetch is bypassed by ORing the inverse of the indirect bit in the src address (bit 8 of IR) with the 0 bit position of the μPC

$$003 \mu\text{Branch} \{ \mu\text{PC} \leftarrow 101 \text{ (from Instruction decoder)} ; \mu\text{PC}_{5,4} \leftarrow [IR_{10,9}] ;$$

$$\mu\text{PC}_3 \leftarrow [IR_{10}]' \cdot [IR_9]' \cdot [IR_8] \}$$
- The five branch addresses differ in the middle octal digit only
- The octal pattern 101 is obtained from the PLA
- The 3 bits to be ORed with the middle octal digit are supplied by the decoding circuitry connected to the src address mode field (bits 8, 9 and 10 of IR)
- Bits 4 and 5 of the μPC are set directly from bits 9 and 10 of IR
- These bits select the appropriate microinstruction for all src address except the register indirect mode
- Register indirect mode: set bit 3 of μPC to 1 using the AND of $[IR_{10}]'$, $[IR_9]'$ and $[IR_8]$

54



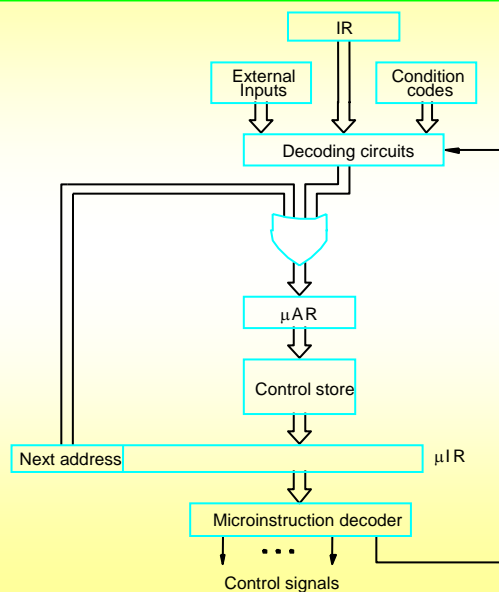
Microinstructions with Next-Address Field

- The previous microprogram requires several branch microinstructions
- This reduces the operating speed of the computer
- A powerful alternative is to include an address field as a part of every microinstruction to indicate the location of the next microinstruction
- Thus, every microinstruction becomes a branch
- Advantages: flexibility
- Disadvantages: expense of the additional bits for the address field
- Typical microprogram: 4k microinstructions with 50 to 80 bits per microinstruction → 12 bit address field is needed

55



Microinstruction Sequencing



56



Microinstruction Sequencing

- Advantage: separate branch microinstructions are virtually eliminated, makes this scheme very attractive
- The μ PC is replaced by a microinstruction address register (μ AR)
- The μ AR holds the address of the next microinstruction

- A new control structure that supports next address field and bit-ORing
- The decoding circuit includes a PLA decoder that is used to generate the starting address of a given microinstruction on the basis of the OP code field in the IR

57



Microprogram Sequencing

Example : ADD (Rsrc)+, Rdst

- Rsrc and Rdst are used instead of referring to register R0 through R15 explicitly
- Actual control signals can be decoded using the data in the src and dst fields of IR

Microinstruction 003

- Bit-ORing is used to determine the next instruction based on the addressing mode of the source operand
- The addressing mode is indicated by bits 8,9 and 10 of IR
- Let OR_{mode} control whether or not this bit-ORing is used

Microinstructions 123, 143, and 166

- Bit-ORing is used to decide if indirect addressing of the source operand is used
- OR_{indsrc} signal is used for this purpose

58



Format for Microinstructions

Microinstruction

F0	F1	F2	F3
F0 (8 bits)	F1 (3 bits)	F2 (3 bits)	F3 (3 bits)
Address of next microinstruction	000: No transfer 001: PC_{out} 010: MDR_{out} 011: Z_{out} 100: $Rsrc_{out}$ 101: $Rdst_{out}$ 110: $TEMP_{out}$	000: No transfer 001: PC_{in} 010: IR_{in} 100: $Rsrc_{in}$ 101: $Rdst_{in}$	000: No transfer 001: MAR_{in} 010: MDR_{in} 011: $TEMP_{in}$ 100: Y_{in}

F4	F5	F6	F7	F8	F9	F10
F4 (4 bits)	F5 (2 bits)	F6 (1 bit)	F7 (1 bit)	F8 (1 bit)	F9 (1 bit)	F10 (1 bit)
0000: Add 0001: Sub ⋮ 1111: XOR	00: No action 01: Read 10: Write	0: SelectY 1: Select4	0: No action 1: WMFC	0: NextAdrs 1: InstDec	0: No action 1: OR_{mode}	0: No action 1: OR_{ndsrc}

59



Format for Microinstructions

- PLA
 - ◆ PLA is used initially to decode the instruction OP codes
 - ◆ One bit in the microinstruction is used to indicate when the output of the PLA is gated into the μ AR
- Address field
 - ◆ Each microinstruction contains an 8 bit address field that holds the address of the next microinstruction

60



Implementation of the Microroutine

Octal address	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
000	00000001	001	011	001	0000	01	1	0	0	0	0
001	00000010	011	001	100	0000	00	0	1	0	0	0
002	00000011	010	010	000	0000	00	0	0	0	0	0
003	00000000	000	000	000	0000	00	0	0	1	1	0
121	01010010	100	011	001	0000	01	1	0	0	0	0
122	01111000	011	100	000	0000	00	0	1	0	0	1
170	01111001	010	000	001	0000	01	0	1	0	0	0
171	01111010	010	000	100	0000	00	0	0	0	0	0
172	01111011	101	011	000	0000	00	0	0	0	0	0
173	00000000	011	101	000	0000	00	0	0	0	0	0

61



Implementation of the Microroutine

Microroutine for ADD (Rsrc)+, Rdst

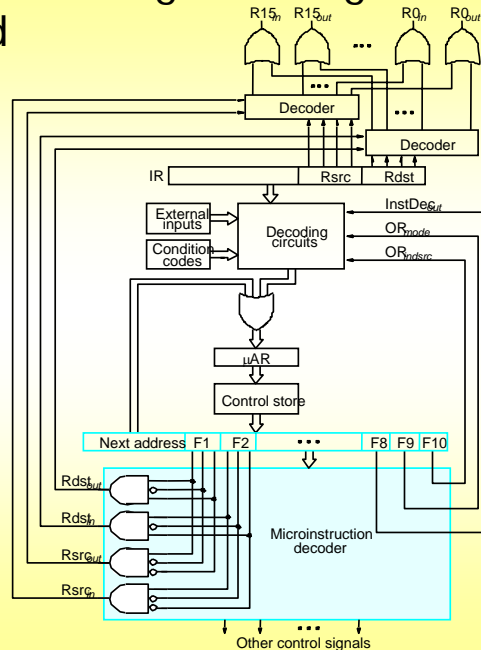
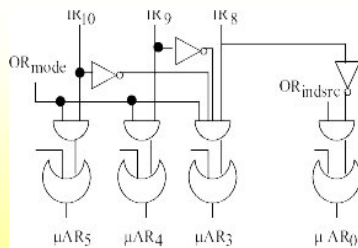
- Fewer microinstructions are needed because branch microinstructions are no longer required
- Locations 003 and 123 have been combined with the microinstructions immediately preceding them
- When microinstruction sequencing is controlled by a μ PC, the End signal is used to reset the μ PC to point to the starting address of the microroutine that fetches the next machine instruction
- With this scheme, the End signal is specified explicitly in the F0 field

62



Circuitry for the control signals using the next address field

- Details of bit-ORing circuitry for the control signals using the next address field



Prefetching Microinstructions

- Drawback of the microprogrammed control: slow operating speed
- Fetching microinstructions from the control store takes a long time
 - ◆ Fast control store
 - ◆ Long microinstructions
 - ◆ Prefetching
- Problems with prefetching
 - ◆ Next microinstruction may depend on the status flags and results of a current microinstruction
 - ◆ Prefetch a wrong microinstruction
 - ◆ Fetch must be repeated with a correct address
- Disadvantages are minor and prefetching is often used



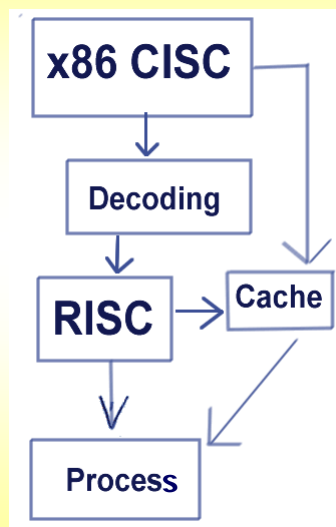
Emulation

- Microprogrammed control provides simple, flexible, and inexpensive way of executing machine instructions
- Allows diverse classes of instructions to be implemented
- It is possible to define additional machine instructions and implement them with microroutine
- We can add an instruction set of a different computer
- A given computer can emulate instructions of a different computer
- No software changes need to be made to legacy programs
- Emulation facilitates transition to a new computer system with minimal effort
- Example: Pentium 4 translates X86 CISC instructions into its RISC microinstructions inside

65



Pentium 4



66



Conclusion

- Speed: hardwired approach
- Flexibility: microprogrammed
- Most present day processors use hardwired

