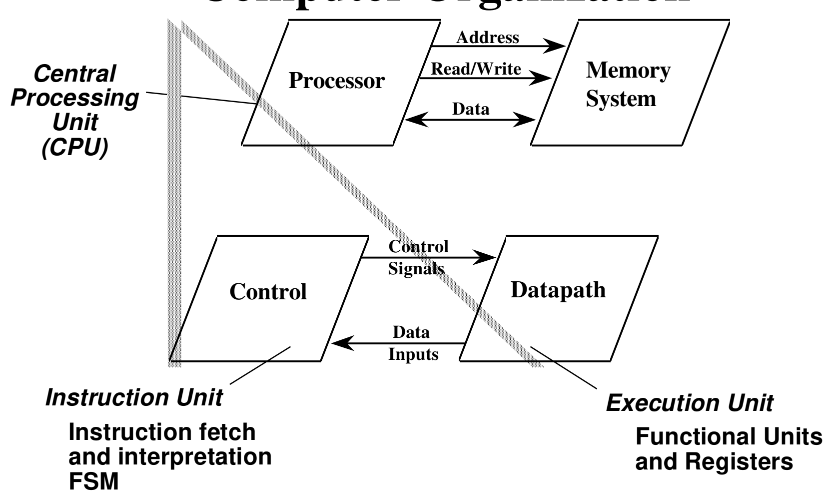
# Outline

- ❍ Last time:
  - ➜ Examples for Review
- ❍ This lecture:
  - ➜ Introduction to Computer Organization
  - ➜ Control
  - ➜ Datapath
  - ➜ I/O Interface
  - ➜ Bussing Strategies
  - ➜ Deriving the State Diagram & Datapath



# Computer Organization



# Computer Organization

*Example of Instruction Sequencing*

**Instruction: Add Rx to Ry and place result in Rz**

**Step 1:** ***Fetch the Add instruction from Memory to Instruction Reg***
**Step 2:** ***Decode Instruction***
**Instruction in IR is an ADD**
**Source operands are Rx, Ry**
**Destination operand is Rz**
**Step 3:** ***Execute Instruction***
**Move Rx, Ry to ALU**
**Set up ALU to perform ADD function**
**ADD Rx to Ry**
**Move ALU result to Rz**



# Structure of a Computer

*Instruction Types*

- **Data Manipulation**
  **Add, Subtract, etc.**
- **Data Staging**
  **Load/Store data to/from memory**
  **Register-to-register move**
- **Control**
  **Conditional/unconditional branches**
  **Subroutine call and return**

# Control

**Elements of the Control Unit (aka Instruction Unit):**

**Standard FSM things:**
- **State Register**
- **Next State Logic**
- **Output Logic (datapath control signaling)**

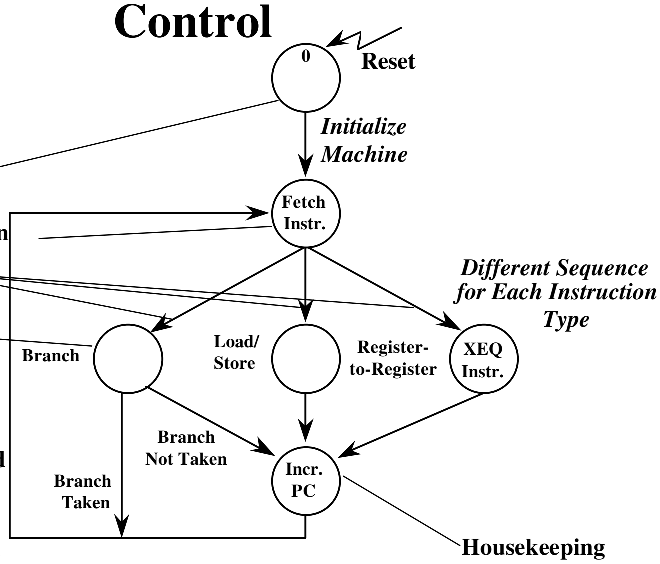
**Plus Additional "Control" Registers:**
- **Instruction Register (IR)**
- **Program Counter (PC)**



# Control

**Control State Diagram**

- **Reset**
- **Fetch Instruction**
- **Decode**
- **Execute**



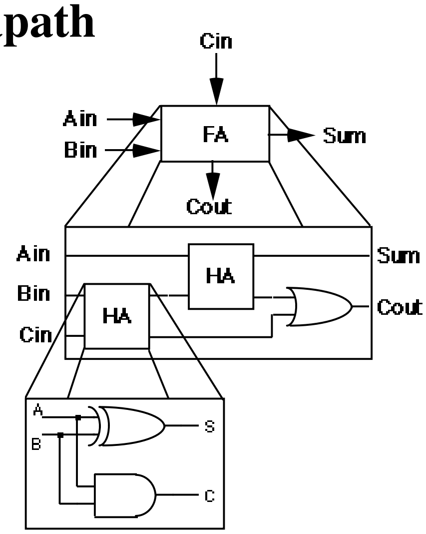

**Instructions partitioned into three classes:**
- **Branch**
- **Load/Store**
- **Register-to-Register**

# Datapath

Arithmetic Circuits
constructed in
hierarchical and
iterative fashion

Each bit in datapath
is functionally identical

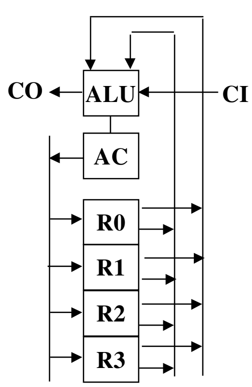
4-bit
8-bit
16-bit
32-bit
Datapaths




**Hierarchical Construction of Full Adder**

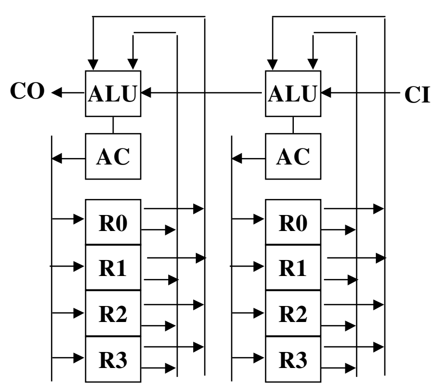


# *Datapath*




**1 bit wide**

***Bit Slice Concept***

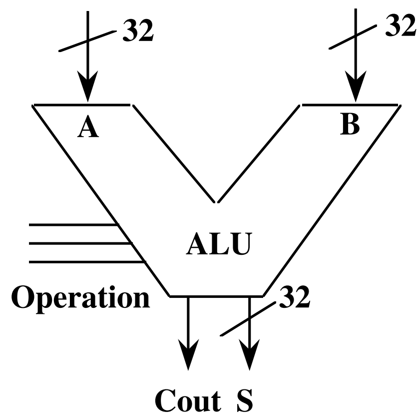



**2 bits wide**

**iterate to build n-bit wide datapaths**

# Datapath

ALU Block Diagram



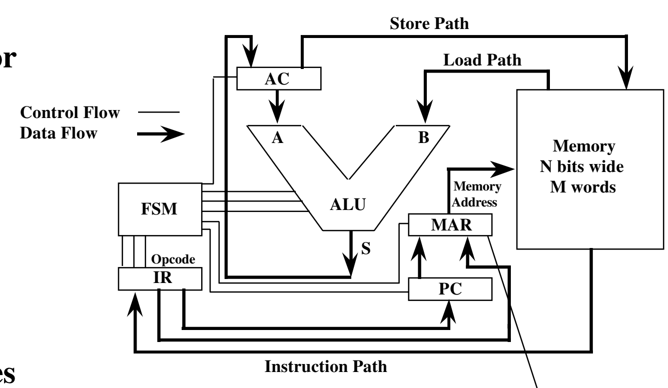



# Block Diagram/Register Transfer View

**Single Accumulator Machine**

**AC := AC <op> Mem**

**"single address instructions"**
**AC implicit operand**




**Arrowed Lines represent dataflows**

**others are control flows**

***Memory Address Register***
**Hold address during memory accesses**

## Block Diagram/Register Transfer View

**Placement of Data and Instructions in Memory:**

- **Data and instructions mixed in memory:**
  **Princeton Architecture**
- **Data and instructions in separate memory:**
  **Harvard Architecture**

**Princeton architecture simpler to implement**

**Harvard architecture has certain performance advantages:**
**overlap instruction fetch with operand fetch**

**We assume the more common Princeton architecture throughout**



## Block Diagram/Register Transfer View

**Trace an instruction: AC := AC + Mem<address>**

1. **Instruction Fetch:**

   **Move PC to MAR**
   **Initiate a memory read sequence**
   **Move data from memory to IR**

2. **Instruction Decode:**

   **Op code bits of IR are input to control FSM**
   **Rest of IR bits encode the operand address**

# Block Diagram/Register Transfer View

**Trace an instruction: AC := AC + Mem<address>**

**3. Operand Fetch:**
**Move operand address from IR to MAR**
**Initiate a memory read sequence**

**4. Instruction Execute:**
**Data available on load path**
**Move data to ALU input**
**Configure ALU to perform ADD operation**
**Move S result to AC**

**5. Housekeeping:**
**Increment PC to point at next instruction**



# Block Diagram/Register Transfer View

**Control: Transfer data from one register to another**
**Assert appropriate control signals**

***Register transfer notation***

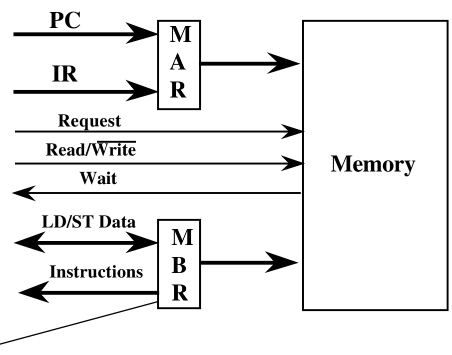
***Register to Register moves***

| | | |
|---|---|---|
| **Ifetch:** | **PC $\rightarrow$ MAR;** | **-- move PC to MAR** |
| | **Memory Read;** | **-- assert Memory READ signal** |
| | **Memory $\rightarrow$ IR;** | **-- load IR from Memory** |
| **Instruction Decode:** | **IF IR<op code> = ADD_FROM_MEMORY** | |
| | **THEN** | |
| **Instruction Execution:** | **IR<addr> $\rightarrow$ MAR;** | **-- move operand addr to MAR** |
| | **Memory Read;** | **-- assert Memory READ signal** |
| | **Memory $\rightarrow$ ALU B;** | **-- gate Memory to ALU B** |
| | **AC $\rightarrow$ ALU A;** | **-- gate AC to ALU A** |
| ***Assert Control*** | **ALU ADD;** | **-- instruct ALU to perform ADD** |
| ***Signal*** | **ALU S $\rightarrow$ AC;** | **-- gate ALU result to AC** |
| | **PC+1;** | **-- increment PC** |

# Memory Interface

**More Realistic Block Diagram:**

**Issue memory request**
**Is it a read or a write?**
**Memory asks CPU to wait**




**Decouple memory system from internal processor operation**

**Memory Buffer Register**

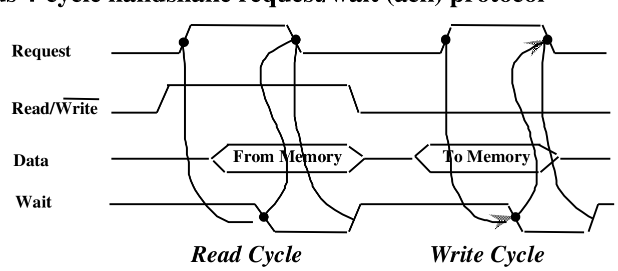


# Memory Interface

**No common clock between CPU and memory**

**Follow asynchronous 4-cycle handshake request/wait ($\overline{\text{ack}}$) protocol**

1. **Request Asserted**
2. **Wait Unasserted**
3. **Request Unasserted**
4. **Wait Asserted**

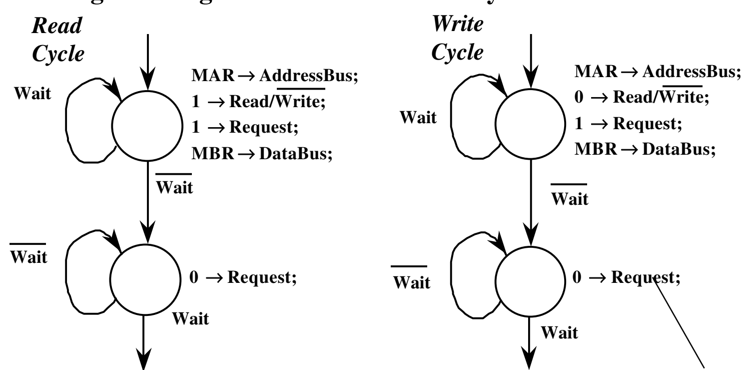



**Memory cannot make request unless Wait signal is asserted**

**Hi-to-Lo transition on Wait implies that data is ready (read) or data has been latched by memory (write)**

# Memory Interface

**State Diagram Fragments for Read/Write Cycles**




**State 1: drive address bus**
**assert read request**
**catch data into MBR**
**State 2: unassert request**
**hold in state until Wait reasserted**

***Normal Convention:***
**If register transfer op NOT asserted, it need not be mentioned in state diagram**



# I/O Interface

**Memory-Mapped I/O**
- **I/O devices share the memory address space**
- **Control registers manipulated just like memory word**
- **Read/write register to initiate I/O operation**

**Polling**
- **Programs periodically checks whether I/O has completed**

**Interrupts**
- **Device signals CPU when operation is complete**
- **Software must take over to handle the data transfers from the device**
- **Check for interrupt pending before fetching next instruction**
- **Save PC & vector to special memory location for next instruction**
- **Instruction set includes a "return from interrupt" instruction**

# Register-to-Register Communications

- Point-to-point
- Single shared bus
- Multiple special purpose busses

**Tradeoffs between datapath/control complexity and amount of parallelism supported by the hardware**

**Case study:**

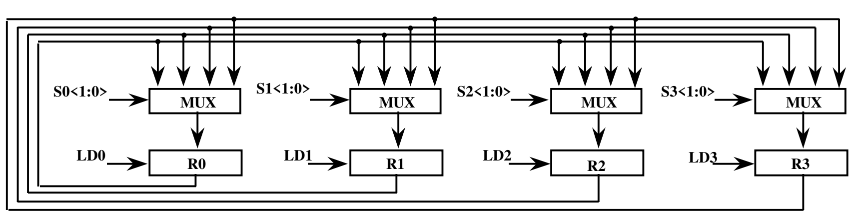
**Four general purpose registers that must be able to exchange their contents**

**Swap instruction must be supported:**

**SWAP(Ri, Rj)**
**Ri → Rj;**
**Rj → Ri;**



# Point-to-Point Connection Scheme




**Four registers interconnected via 4:1 Mux's and point-to-point connections**

- **Edge-triggered N bit registers controlled by LDi signals**
- **N x 4:1 MUXes per register, controlled by Si<1:0> signals**

# Point-to-Point Connections

**Example:**
**Register transfers R1 $\rightarrow$ R0 and R2 $\rightarrow$ R3**

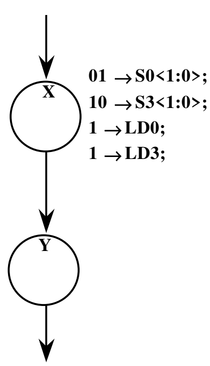
**Register transfer operations:**

| | |
|---|---|
| **01 $\rightarrow$ S0<1:0>;** | **Enable path from R1 to R0** |
| **10 $\rightarrow$ S3<1:0>;** | **Enable path from R2 to R3** |
| **1 $\rightarrow$ LD0;** | **Assert load for R0** |
| **1 $\rightarrow$ LD3;** | **Assert load for R3** |



# Point-to-Point Connections

**When control signals are asserted and when they take place:**




**Moore Machine State Diagram**

**Enter state X:**
**Multiplexor control signals asserted**
**R1 outputs arrive at R0 inputs**
**R2 outputs arrive at R3 inputs**

**LD signals asserted**
**Do not take effect until next rising clock**

**On entering state Y:**
**LD signals are synchronous and take effect at the same time as the state transition!**

# Bussing Schemes

## Implementation of Register SWAP operation

SWAP(R1, R2):

01 → S2<1:0>;
10 → S1<1:0>;
1 → LD2;
1 → LD1;

Establish connection paths

Swap takes place at next state transition

**Point-to-Point Scheme Plusses and Minuses:**

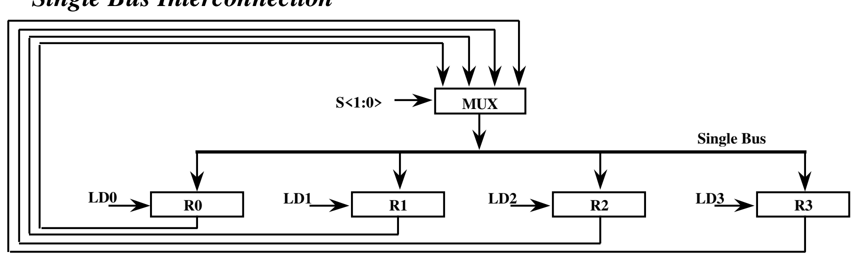
+ transfer a new value into each of the four registers at same time
+ register swap implemented in a single control state

- 5 gates to implement 4:1 MUX
  32 bit wide datapath implies 32 x 5 x 4 registers
  = 640 gates!
  very expensive implementation



# Bussing Strategies

*Single Bus Interconnection*




- per register MUX block replaced by single block
- 25% hardware cost of previous alternative
- shared set of pathways is called a BUS

Single bus becomes a critical resource --
used by only one transfer at a time

# Bussing Strategies

*Single Bus Interconnection*

Example: R1 $\rightarrow$ R0 and R2 $\rightarrow$ R3

State X: (R1 $\rightarrow$ R0)
01 $\rightarrow$ S<1:0>;
1 $\rightarrow$ LD0;

State Y: (R2 $\rightarrow$ R3)
10 $\rightarrow$ S<1:0>;
1 $\rightarrow$ LD3;

Datapath no longer supports two simultaneous transfers!
Thus two control states are required to perform the transfers



# Bussing Strategies

*Single Bus Interconnection*

SWAP Operation

A special TEMP register must be introduced ("Register 4")
MUX's become 5:1 rather than 4:1

State X: (R1 $\rightarrow$ R4)
001 $\rightarrow$ S<2:0>;
1 $\rightarrow$ LD4;

State Y: (R2 $\rightarrow$ R1)
010 $\rightarrow$ S<2:0>;
1 $\rightarrow$ LD1;

State Z: (R4 $\rightarrow$ R2)
100 $\rightarrow$ S<2:0>
1 $\rightarrow$ LD2;

Three states are required rather than one!

plus extra register and wider mux

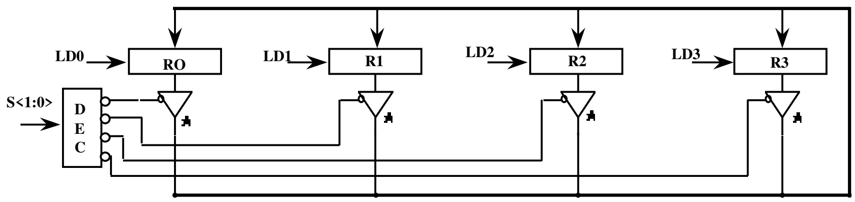
More control states because this datapath supports less parallel activity

Engineering choices made based on how frequently multiple transfers take place at the same time

# Bussing Strategies

*Alternatives to Multiplexors*

**Tri-state buffers as an interconnection scheme**




**Only one register's contents gated to shared bus at a time**



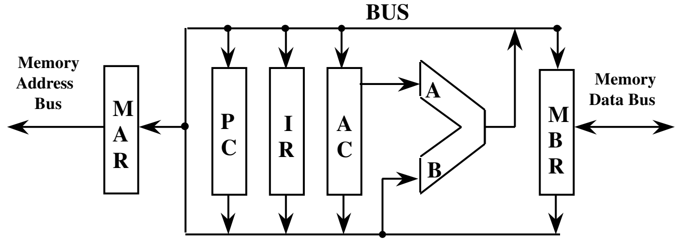
# Bussing Strategies

*Multiple Busses*

**Real datapaths are a compromise between the two extremes**

**Register Transfer Diagram**

**Single Bus Design**




**Register transfer operations:**

| | | |
|---|---|---|
| PC $\rightarrow$ BUS | BUS $\rightarrow$ PC | AC $\rightarrow$ ALU A |
| IR $\rightarrow$ BUS | BUS $\rightarrow$ IR | ("hardwired") |
| AC $\rightarrow$ BUS | BUS $\rightarrow$ AC | |
| MBR $\rightarrow$ BUS | BUS $\rightarrow$ MBR | |
| ALU Result $\rightarrow$ BUS | BUS $\rightarrow$ ALU B | |
| | BUS $\rightarrow$ MAR | |

# Bussing Strategies

*Multiple Busses*

**Example Register Transfer for Single Bus Design**

**Instruction Interpretation for "ADD Mem[X]"**

| | |
|---|---|
| ***Fetch Operand*** **Cycle 1:** | **IR<operand address> $\rightarrow$ BUS;**<br>**BUS $\rightarrow$ MAR;** |
| **Cycle 2:** | **Memory Read;**<br>**Databus $\rightarrow$ MBR;** |
| ***Perform ADD*** **Cycle 3:** | **MBR $\rightarrow$ BUS;**<br>**BUS $\rightarrow$ ALU B;**<br>**AC $\rightarrow$ ALU A;**<br>**ADD;** |
| ***Write Result*** **Cycle 4:** | **ALU Result $\rightarrow$ BUS;**<br>**BUS $\rightarrow$ AC;** |

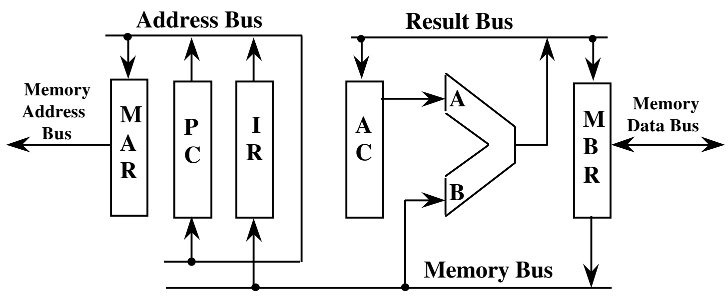
**Requires latch for ALU Result**



# Bussing Strategies

*Multiple Busses*

**Three Bus Design -- Supports more parallelism**




**Single bus replaced by three busses:**

- **Memory Bus (MBUS)**
- **Result Bus (RBUS)**
- **Address Bus (ABUS)**

# Bussing Strategies

***Multiple Busses***

**Instruction Interpretation for "ADD Mem[X]"**

| | | |
|---|---|---|
| ***Fetch Operand*** | | |
| **Cycle 1:** | **IR<operand address> → ABUS;**<br>**ABUS → MAR;** | |
| **Cycle 2:** | **Memory Read;**<br>**Databus → MBR;** | |
| ***Perform ADD*** | | |
| **Cycle 3:** | **MBR → MBUS;**<br>**MBUS → ALU B;**<br>**AC → ALU A;**<br>**ADD;** | **Implemented in three cycles rather than four** |
| ***Write Result*** | | |
| | **ALU Result → RBUS;**<br>**RBUS → AC;** | |

**Advantage of separate ABUS:**
**overlap PC → MAR with instruction execution**