A Simple Implementation

EE380, Spring 2018

Hank Dietz

http://aggregate.org/hankd/

UNIVERSITY OF KENTUCKY

Where Is This Stuff?

- Not in the text per se...
- Primary reference is:

`http://aggregate.org/EE380/refss.html`

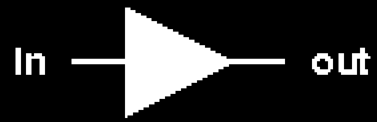
- Textbook appendix B has EE280 review stuff...

A Dumb Implementation

- A design like I learned as an undergrad...
  - Can be built with a pile of TTL parts
  - Can execute MIPS instructions
  - Slow; many clock cycles per instruction
- The key parts:
  - Memory
  - Processor
  - I/O – which we'll ignore for now...

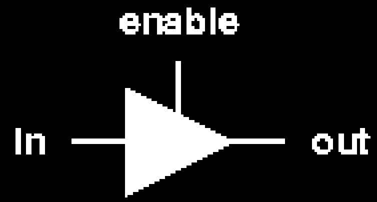
Our Favorite Gates

- In EE280, you never used one of these:



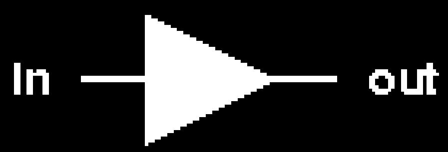
but they help keep signals digital...

- In EE380, we use lots of these:

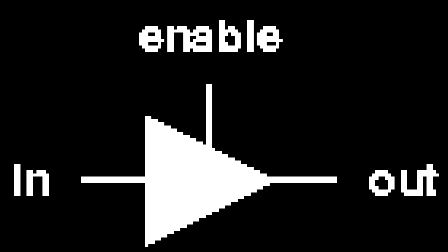


to make bus and mux structures...

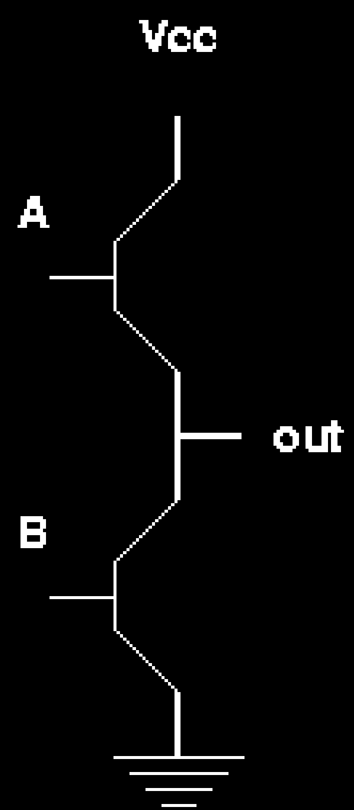
Tri-State (& Open Collector)



Driver



Tri–State Driver

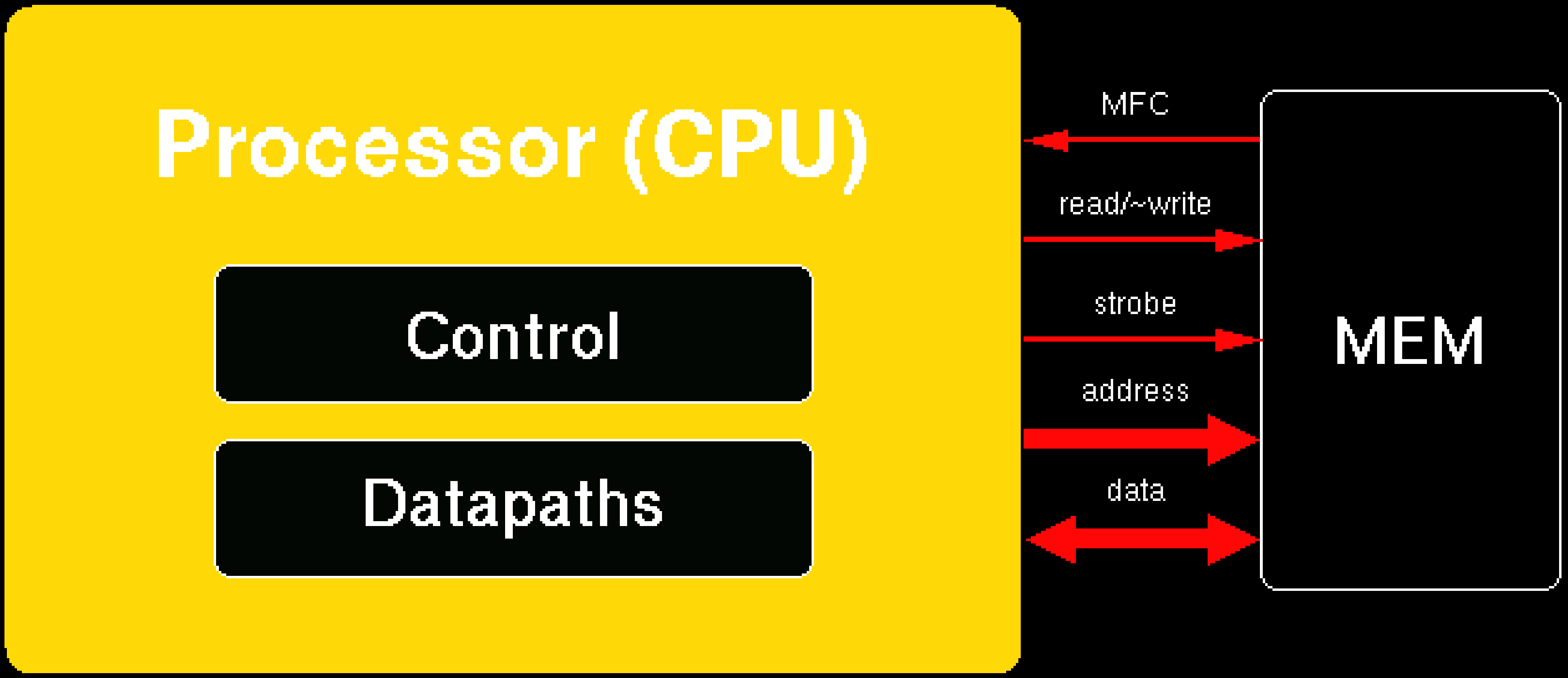


In	enable	A	B	out
X	0	*off*	*off*	*Z*
0	1	*off*	*on*	0
1	1	*on*	*off*	1
		on	*on*	*short!*

Open Collector replaces A with a resistor

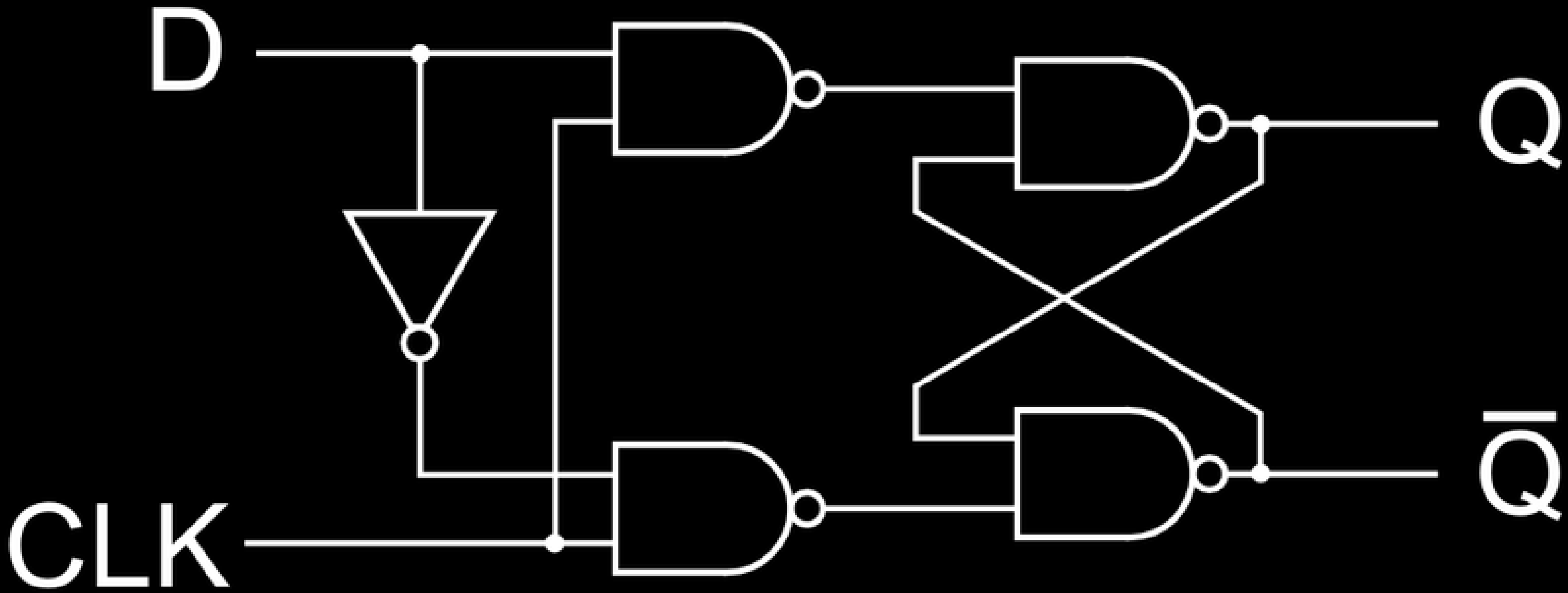
TTL input floats high; CMOS doesn't

Processor/Memory Interface

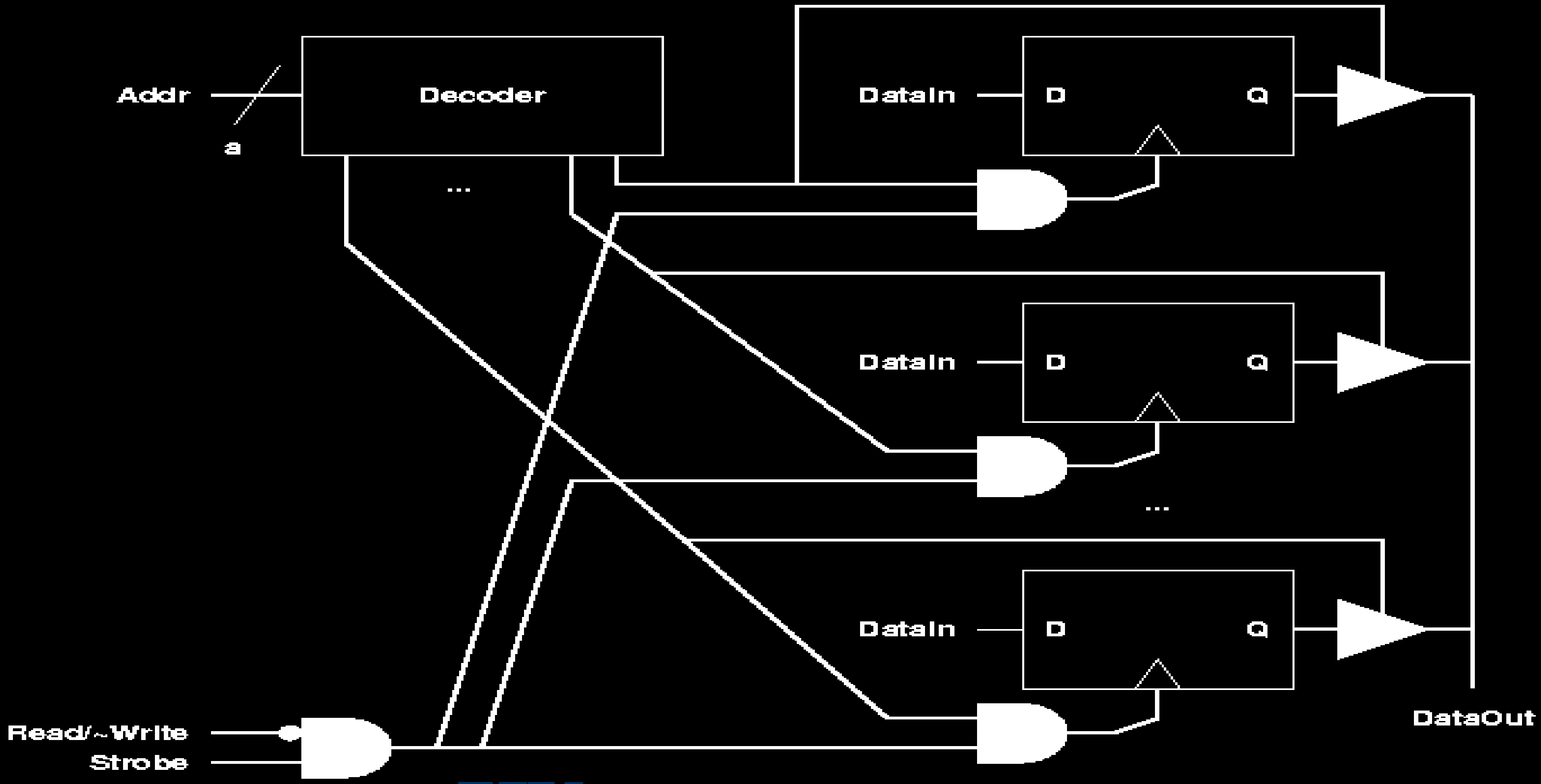


UK UNIVERSITY OF KENTUCKY

A bit Of SRAM (D Flip Flop)

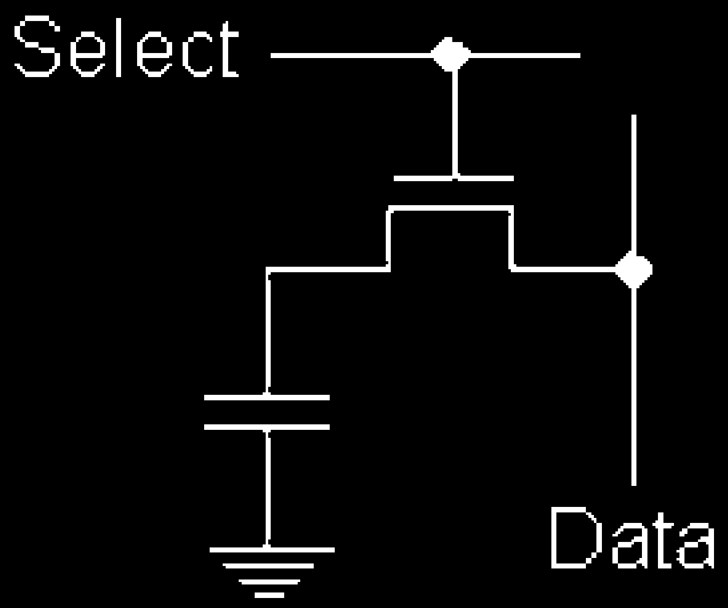


A Simple Memory



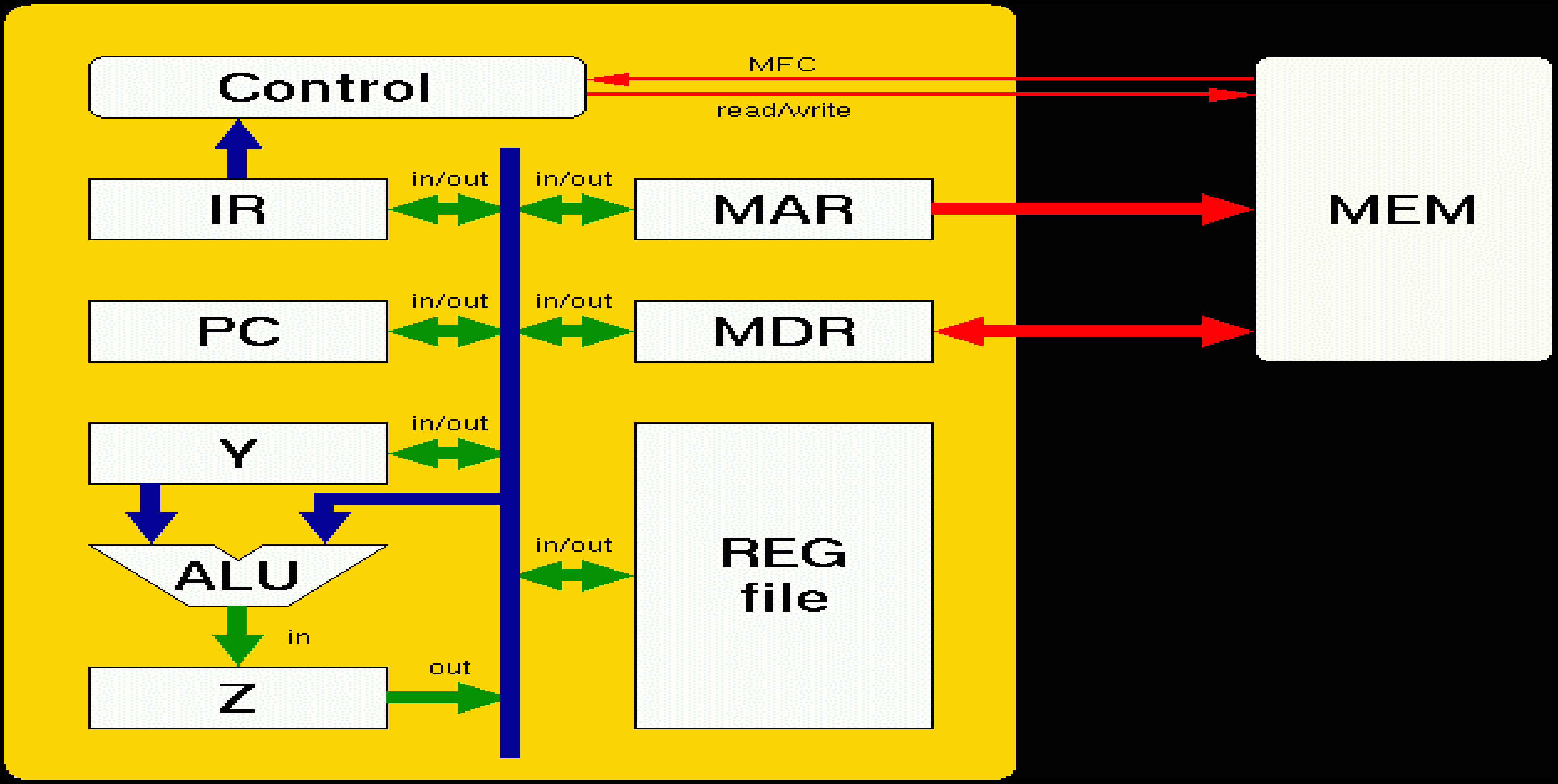
UK UNIVERSITY OF KENTUCKY

A bit Of DRAM



- Data to Vcc to store 1
- Data to Gnd to store 0
- Dump charge, amplify, & threshold to read...
  - Analog – slow & noise sensitive
  - Destructive (need to refresh value)
- Charge slowly leaks (need to refresh value)

Inside The Processor



REGISTER control signal	Effect
ALUadd	Configures the ALU to add its inputs
ALUand	Configures the ALU to bitwise AND its inputs
ALUxor	Configures the ALU to bitwise eXclusive OR its inputs
ALUor	Configures the ALU to bitwise OR its inputs
ALUsll	Configures the ALU to shift left logical; the result is (bus << Y)
ALUslt	Configures the ALU to compare its inputs; the result is (Y < bus)
ALUsrl	Configures the ALU to shift right logical; the result is (bus >> Y)
ALUsub	Configures the ALU to subtract the bus input from Y
CONST(*value*)	Places the constant *value* onto the bus
HALT	Halt the machine (stop the simulator without error) at the end of the current state
IRaddrout	Tri-state enables the portion of the Instruction Register that contains the (26 bit, MIPS "J" format) address, along with the top 6 bits of the Program Counter, to be driven onto the bus
IRimmedout	Tri-state enables the portion of the Instruction Register that contains the (16 bit, MIPS "I" format) 2's complement immediate value to be sign-extended to 32 bits and driven onto the bus
IRin	Latches the bus data into the Instruction Register at the trailing edge of the clock cycle
IRoffsetout	Tri-state enables the Instruction Register's shifted and sign extended value from the offset field to be driven onto the bus (used for branches)
JUMP(*label*)	Microcode jump to *label*
JUMPonop	Microcode jump to label named like the opcode; e.g., if an "Addi" is in the IR, jumps to the microcode label Addi
MARin	Latches the bus data into the Memory Address Register at the trailing edge of the clock cycle
MARout	Tri-state enables the Memory Address Register's output to be driven onto the bus
MDRin	Latches the bus data into the Memory Data Register at the trailing edge of the clock cycle
MDRout	Tri-state enables the Memory Data Register's output to be driven onto the bus
MEMread	Initiate a memory read from the address in the MAR; here, you may assume that the memory will take 2 clock cycles to respond
MEMwrite	Initiate a memory write using the data in the MDR and the address in the MAR; in this simple design, you may assume that a memory write takes precisely 1 clock cycle
PCin	Latches the bus data into the Program Counter at the trailing edge of the clock cycle
PCinif0	Only if the value in Z is zero, latch the bus data into the Program Counter at the trailing edge of the clock cycle
PCout	Tri-state enables the Program Counter's output to be driven onto the bus
REGin	Latches the bus data into whichever register is selected by SELrs, SELrt, or SELrd; the value is latched at the trailing edge of the clock cycle
REGout	Tri-state enables the output of whichever register is selected by SELrs, SELrt, or SELrd; the selected value is driven onto the bus
SELrs	Selects the rs field of the IR to be used to control the register file's decoder
SELrt	Selects the rt field of the IR to be used to control the register file's decoder
SELrd	Selects the rd field of the IR to be used to control the register file's decoder
UNTILmfc	Repeat this state until the memory has issued a memory fetch complete signal, indicating that the fetched value will be valid to read from the MDR in the next clock cycle
Yin	Latches the bus data into the Y register at the trailing edge of the clock cycle; this register is needed because, with only one bus, one of the two operands for a binary operation (e.g., Add) must come from somewhere other than the bus
Yout	Tri-state enables the Y register's output to be driven onto the bus
Zin	The ALU is always producing a result, but we only make note of that result if we latch the ALU's output into the Z register at the trailing edge of the clock cycle
Zout	Tri-state enables the Z Register's output to be driven onto the bus

Control Logic

- A big state machine (spec. by names)
  - Begins by fetching instruction
  - Decoding instruction sends us to particular instruction's state sequence
  - Ends by going to fetch next instruction
- Instruction decode logic

  `when` *`mask match statename`*

  - Applied in state with **`JUMPONOP`**
  - if ((IR & *`mask`*) == *`match`*) goto *`statename`*;

Instruction Fetch Sequence

- Not dependent on instruction type – can't be
- Also does PC+=4

```
Start: PCout,MARin,MEMread,Yin
       CONST(4),ALUadd,Zin,UNTILmfc
       MDRout,Irin
       JUMPONOP,Zout,Pcin
       HALT /* illegal inst. */
```

MIPS Register Add

- `add` *$rd,$rs,$rt*
- Means *rd=rs+rt*

```
Add: SELrs,REGout,Yin
     SELrt,REGout,ALUadd,Zin
     Zout,SELrd,REGin,JUMP(Start)
```

MIPS Register And

- `and $rd,$rs,$rt`
- Means *rd*=*rs*&*rt*

```
And: SELrs,REGout,Yin
     SELrt,REGout,ALUand,Zin
     Zout,SELrd,REGin,JUMP(Start)
```

MIPS Load Word

- `lw $rt,immed($rs)`
- Means *rt*=mem[*immed*+*rs*]

```
Lw: SELrs,REGout,Yin
    IRIMMEDout,ALUadd,Zin
    Zout,MARin,MEMread
    UNTILmfc
    MDRout,SELrt,REGin,JUMP(Start)
```

MIPS Store Word

- `sw $rt,immed($rs)`
- Means mem[*immed+rs*]=*rt*
- Don't have to wait for write to complete

```
Sw: SELrt,REGout,MDRin
    SELrs,REGout,Yin
    IRIMMEDout,ALUadd,Zin
    Zout,MARin,MEMwrite,JUMP(Start)
```

Timing

- Clock period determined by slowest path in any state – try to minimize variation
- Number of clock cycles/instruction (CPI) is determined by counting
  - Not just count of states passed through
  - Time passed waiting counts (**`UNTILmfc`**)
- Clock period and CPI usually trade off; higher Hz often achieved by higher CPI

Clock Period

- Assume the critical state is:

```
SELrt,REGout,MDRin,ALUadd,Zin
```

- The paths are:

```
SELrt > REGout > MDRin
SELrt > REGout > ALUadd > Zin
```

Reducing Clock Period

- Clock speed can be increased by replacing:

```
SELrt,REGout,MDRin,ALUadd,Zin
```

- With:

```
SELrt,REGout,MDRin
MDRout,ALUadd,Zin
```

Counting CPI

- Instruction fetch time counts
- Time between `MEMread` and `UNTILmfc`

```
Lw: SELrs,REGout,Yin +1
    IRIMMEDout,ALUadd,Zin +1
    Zout,MARin,MEMread +1
    UNTILmfc +?
    MDRout,SELrt,REGin,JUMP(Start) +1
```

A Verilog Implementation

- Design for simulation, not rendering HW
- Key ideas:
  - `` `define `` control signals & constants
  - `module memory(…);`
    Models main memory
  - `module simple(halt, reset, clk);`
    Models the complete processor
  - `module bench;`
    Drives the simulation

Verilog Simulation

- `` `define `` control signals & constants
- **`module memory(…);`**
  Models main memory
- **`module simple(halt, reset, clk);`**
  Models the complete processor
- **`module bench;`**
  Drives the simulation

Verilog Simulation

- Could go very low level, e.g.:

```
module DFF(q, d, clk, reset);
input d, clk, reset;
output reg q;
always @(posedge reset or
         negedge clk)
  if (reset) q = 1'b0; else q = d;
Endmodule
```

Verilog Simulation

- Don't have to go low level:

  `http://aggregate.org/EE380/simplev.html`

- Don't have to feed it raw bits either;
  here's a (slightly mutant) MIPS assembler:

  `http://aggregate.org/EE380/simplev.html`