Lecture 9: The EE183 Processor

David Black-Schaffer
davidbbs@stanford.edu
EE183 Spring 2003

Overview

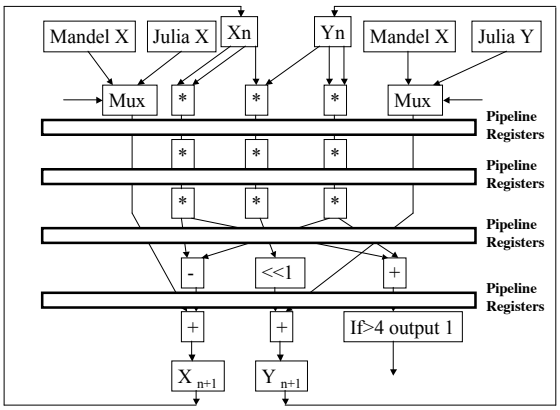
- Pipelining
  - Getting everything right is a very complicated control problem
  - Regularize the data path so we can use it more generically
  - Encode the control information in the data
- Hazards
  - Watch out for Data and Control Hazards
  - Use Forwarding and NOPs in lab 3

Public Service Announcement

- Xilinx Programmable World
  - Tuesday, May 6th
  - http://www.xilinx.com/events/pw2003/index.htm - free!
- Guest Lectures
  - Wednesday, May 7th
    Gary Spivey on ASIC & FPGA Design for Speed

Logistics

- Lab 2 due Friday by 5pm
- Any questions on Lab 2?



Pipeline Performance Analysis

- With the bad data path (3, 3 stage multipliers and 2 stages after that; **multiple** pixels at a time)

Clk	M1a	M2a	M3a	M1b	M2b	M3b	M1c	M2c	M3c	Add1	Add2
1	●	●	●								
2	▲	▲	▲	●	●	●					
3	●	●	●	▲	▲	▲	●	●	●		
4	❖	❖	❖	●	●	●	▲	▲	▲	●	
5	●	●	●	❖	❖	❖	●	●	●	▲	●

- We approach 100% utilization if there are no stalls or dependencies and we can keep getting new data

Key points on Pipelining

- Increased utilization of functional units ***only if you can keep the pipeline full***
- Keeping the pipeline full requires more complicated control logic
  - Data hazards
  - Control hazards

Two problems

- Control logic too complicated to keep the pipeline full
- Pipeline specific to one particular problem
- What to do?
  - Encode control information with the data
  - Use a generic pipeline

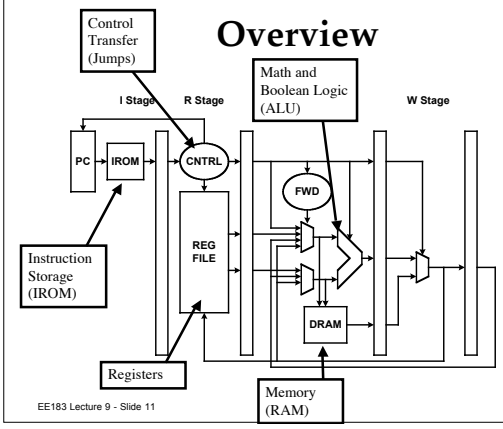
Motivation for Lab 3

- Take advantage of the performance increase from a pipelined architecture without limiting ourselves to a particular calculation.
- We want a programmable processor
  - Using a generic pipeline (so we can calculate anything)
  - Encoding the control with the data (to make the control logic simpler so we can keep the pipeline full)
- It's cool: you will have built a RISC processor

What do we want?

- Obvious stuff:
  - Instruction storage
  - Math
  - Load/Store from/to memory
  - Control Transfer (jump if…)
- Less obvious
  - Registers (so we don't have to wait for the RAM)
  - Boolean Algebra
  - External inputs/outputs
  - A compiler
  - Anything else?

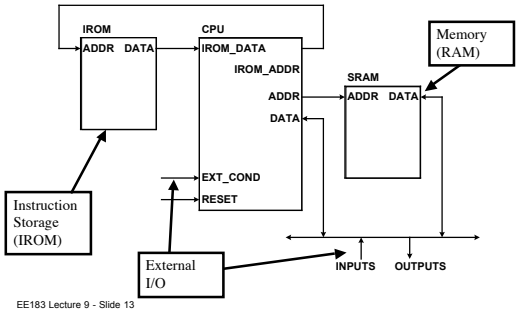
Overview



Processor Specs

- 12-bit RISC Microcontroller
  - What would having only 8 bits mean for the memory architecture?
- 8 General Purpose Registers
- 43 Instructions
- 3 operand instructions
- 4 stage pipeline
- Register indirect addressing mode
  - What does this mean?

Overview 2



Instruction Set Architecture (ISA)

- 8 General Purpose Registers
- ALU Instructions
  - 28 Instructions
  - 3 operands
- Control Transfer Instructions
  - 12 Instructions
  - Conditional/Unconditional branches
- Memory Instructions
  - 2 instructions
  - Load/Store

Instruction Format

- Different for different types of instructions
- ALU instructions (similar to others):

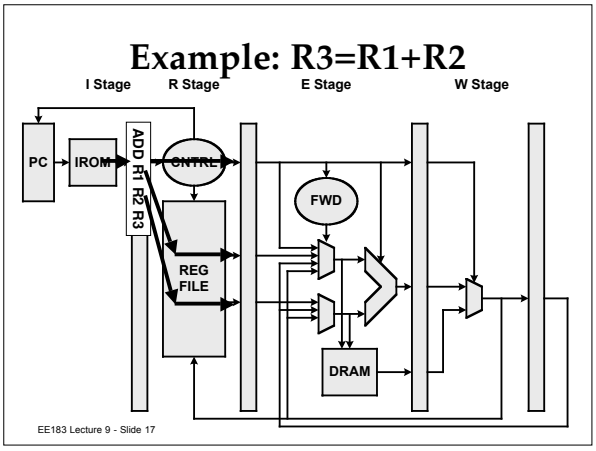
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1														
0	1	WC			OP					RA			RB		

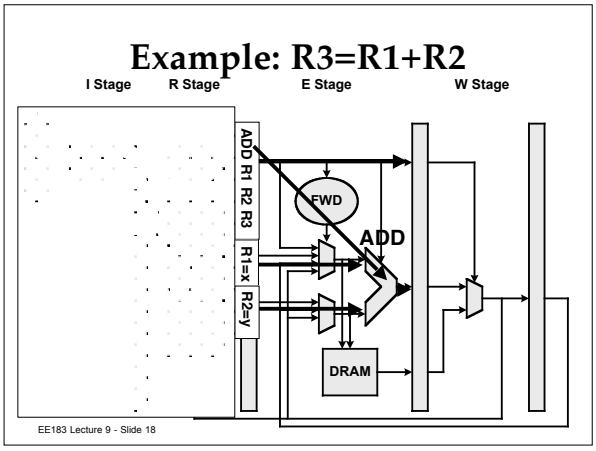
Writeback Register | Operation Code | Source Register A | Source Register B

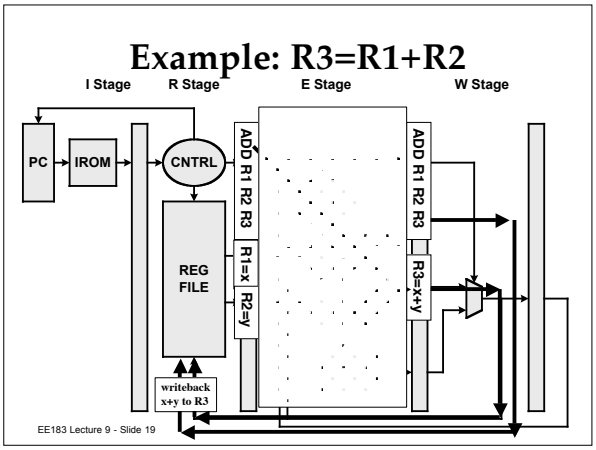
- See Lab 3 handout

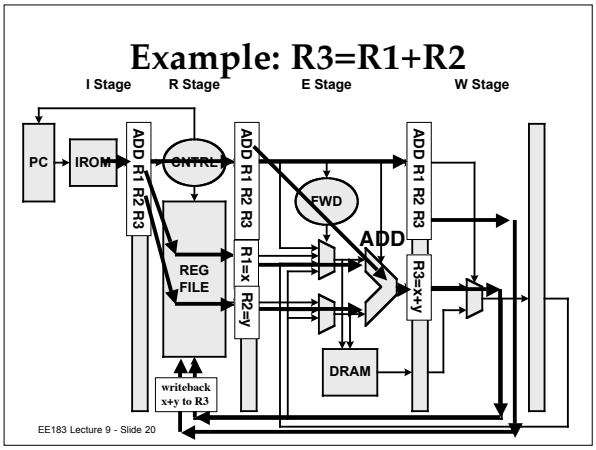
Instruction Execution Steps

- 4 Step Sequence
  - Step **I** Fetch instruction from Instruction Memory
  - Step **R** Read operands from registers (A, B)
  - Step **E** Execute instruction, set condition codes
  - Step **W** Write results to register C
- So how does this simplify the control logic?


Example: R3=R1+R2
I Stage
R Stage
E Stage
W Stage
PC
IROM
ADD R1 R2 R3
CNTRL
REG FILE
FWD
DRAM
EE183 Lecture 9 - Slide 17


Example: R3=R1+R2
I Stage
R Stage
E Stage
W Stage
ADD R1 R2 R3
R1=x
R2=y
FWD
ADD
DRAM
EE183 Lecture 9 - Slide 18

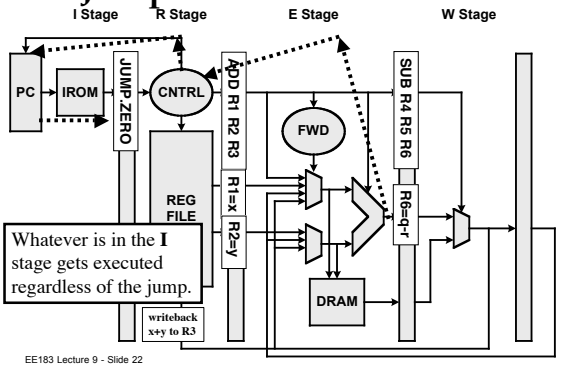

Example: R3=R1+R2
I Stage
R Stage
E Stage
W Stage
PC
IROM
CNTRL
ADD R1 R2 R3
REG FILE
R1=x
R2=y
R3=x+y
writeback x+y to R3
EE183 Lecture 9 - Slide 19


Example: R3=R1+R2
I Stage
R Stage
E Stage
W Stage
PC
IROM
CNTRL
ADD R1 R2 R3
REG FILE
FWD
ADD
R1=x
R2=y
R3=x+y
DRAM
writeback x+y to R3
EE183 Lecture 9 - Slide 20

Notes

- We do a little bit of the work in each stage and carry it along through the pipeline registers
  - Each stage will probably have more registers
- We want each stage to be as fast as possible and as independent as possible
- Can we do this?
- Not entirely…

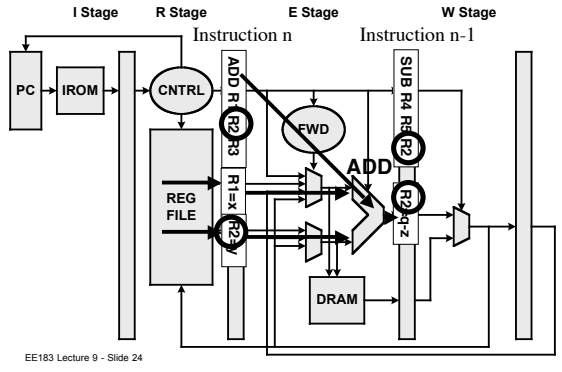
Jumps - Control Hazards



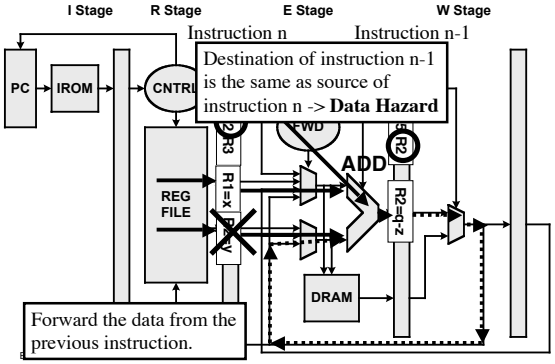
Control Hazards Solution

- Insert a NOP after each branch statement (JUMP)
- Now we don't care if the next instruction is executed because it never does anything

Data Hazards



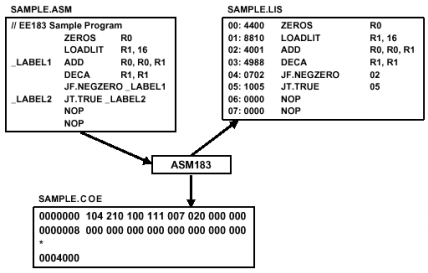
Data Hazards



Tricky Parts

- Prelab for Lab 3:
- Forwarding logic
  - Really only a few cases, but it takes a bit of thinking to work it out
- Branching (Jump) logic:
  - Not too complicated either, but if you get it wrong it will be hard to debug

ASM183.pl (Assembler)



What do you get?

- Template verilog file with some of the pipe registers
  - This also has the logic for memory-mapping the DIP switches, but there is a typo
- Register file
- Boolean logic for the ALU
- ASM183 Perl assembler

What do you have to do?

- Understand the pipeline
- Put together each stage
- Instantiate the RAM & IROM
- Put in the Forwarding

- Add a memory-mapped VGA display
- Add a free-running timer

Demo

- Run the three sample programs and display the output on the LEDs/VGA

- Write your own program which uses the free-running counter and the VGA

Lecture 6 Key Points

- Pipelining only gives you better performance if you can keep the pipeline full
- Regularize the data path and encode the control information with the data to make it easier
- Watch out for Hazards

- Logistics
  - **Lab 2 demo due Friday by 5pm**