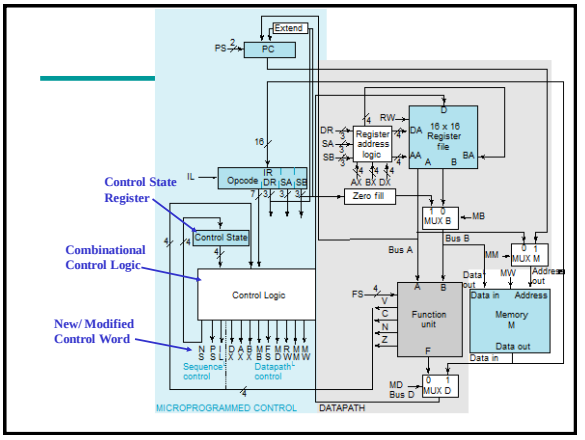
ECE 3401 Lecture 23

Pipeline Design



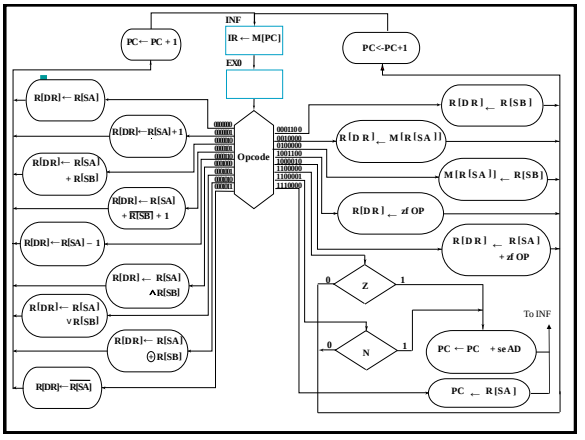
ISA: Instruction Specifications (for reference)

Instruction Specifications for the SimpleComputer - Part 1

Instruction	Opcode	Mnemonic	Format	Description	Status Bits
Move A	0000000	MOVA	RD,RA	R[DR] ← R[SA]	N, Z
Increment	0000001	INC	RD,RA	R[DR] ← R[SA] + 1	N, Z
Add	0000010	ADD	RD,RA,RB	R[DR] ← R[SA] + R[SB]	N, Z
Subtract	0000101	SUB	RD,RA,RB	R[DR] ← R[SA] − R[SB]	N, Z
Decrement	0000110	DEC	RD,RA	R[DR] ← R[SA] − 1	N, Z
AND	0001000	AND	RD,RA,RB	R[DR] ← R[SA] ∧ R[SB]	N, Z
OR	0001001	OR	RD,RA,RB	R[DR] ← R[SA] ∨ R[SB]	N, Z
Exclusive OR	0001010	XOR	RD,RA,RB	R[DR] ← R[SA] ⊕ R[SB]	N, Z
NOT	0001011	NOT	RD,RA	R[DR] ← R[SA]	N, Z

Instruction Specifications for the SimpleComputer - Part 2

Instruction	Opcode	Mnemonic	Format	Description	Status Bits
Move B	0001100	MOVB	RD,RB	R[DR] ← R[SB]	
Shift Right	0001101	SHR	RD,RB	R[DR] ← sr R[SB]	
Shift Left	0001110	SHL	RD,RB	R[DR] ← sl R[SB]	
Load Immediate	1001100	LDI	RD, OP	R[DR] ← zf OP	
Add Immediate	1000010	ADI	RD,RA,OP	R[DR] ← R[SA] + zf OP	
Load	0010000	LD	RD,RA	R[DR] ← M[SA]	
Store	0100000	ST	RA,RB	M[SA] ← R[SB]	
Branch on Zero	1100000	BRZ	RA,AD	if (R[SA] = 0) PC ← PC + se A D	
Branch on Negative	1100001	BRN	RA,AD	if (R[SA] < 0) PC ← PC + se A D	
Jump	1110000	JMP	RA	PC ← R[SA]	



State Table for 2-Cycle Instructions

State	Inputs: Opcode	Inputs: VCNZ	Next state	Outputs: IL	PS	DX	AX	BX	MB	FS	MD	RW	MM	MW	Comments
INF	XXXXXXX	XXXX	EX0	1	00	XXXX	XXXX	XXXX	X	XXXX	X	0	1	0	IR ← M[PC]
EX0	0000000	XXXX	INF	0	01	0XXX	0XXX	XXXX	X	0000	0	1	X	0	MOVA R[DR] ← R[SA]*
EX0	0000001	XXXX	INF	0	01	0XXX	0XXX	XXXX	X	0001	0	1	X	0	INC R[DR] ← R[SA] + 1*
EX0	0000010	XXXX	INF	0	01	0XXX	0XXX	0XXX	0	0010	0	1	X	0	ADD R[DR] ← R[SA] + R[SB]*
EX0	0000101	XXXX	INF	0	01	0XXX	0XXX	0XXX	0	0101	0	1	X	0	SUB R[DR] ← R[SA] + R[SB] + 1*
EX0	0000110	XXXX	INF	0	01	0XXX	0XXX	XXXX	X	0110	0	1	X	0	DEC R[DR] ← R[SA] + (- 1)*
EX0	0001000	XXXX	INF	0	01	0XXX	0XXX	XXXX	0	1000	0	1	X	0	AND R[DR] ← R[SA] ∧ R[SB]*
EX0	0001001	XXXX	INF	0	01	0XXX	0XXX	0XXX	0	1001	0	1	X	0	OR R[DR] ← R[SA] ∨ R[SB]*
EX0	0001010	XXXX	INF	0	01	0XXX	0XXX	0XXX	0	1010	0	1	X	0	XOR R[DR] ← R[SA] ⊖ R[SB]*
EX0	0001011	XXXX	INF	0	01	0XXX	0XXX	XXXX	X	1011	0	1	X	0	NOT R[DR] ← R[SA] *
EX0	0001100	XXXX	INF	0	01	0XXX	0XXX	XXXX	0	1100	0	1	X	0	MOVB R[DR] ← R[SB]*
EX0	0010000	XXXX	INF	0	01	0XXX	0XXX	XXXX	X	XXXX	1	1	0	0	LD R[DR] ← M[R[SA]]*
EX0	0100000	XXXX	INF	0	01	XXXX	0XXX	0XXX	0	XXXX	X	0	0	1	ST M[R[SA]] ← R[SB]*
EX0	1001100	XXXX	INF	0	01	0XXX	0XXX	XXXX	1	1100	0	1	0	0	LDI R[DR] ← zf OP*
EX0	1000010	XXXX	INF	0	01	0XXX	0XXX	XXXX	1	0010	0	1	0	0	ADI R[DR] ← R[SA] + zf OP*
EX0	1100000	XXX1	INF	0	10	XXXX	0XXX	XXXX	X	0000	X	0	0	0	BRZ PC ← PC + se AD
EX0	1100000	XXX0	INF	0	01	XXXX	0XXX	XXXX	X	0000	X	0	0	0	BRZ PC ← PC + 1
EX0	1100001	XX1X	INF	0	10	XXXX	0XXX	XXXX	X	0000	X	0	0	0	BRN PC ← PC + se AD
EX0	1100001	XX0X	INF	0	01	XXXX	0XXX	XXXX	X	0000	X	0	0	0	BRN PC ← PC + 1
EX0	1110000	XXXX	INF	0	11	XXXX	0XXX	XXXX	X	0000	X	0	0	0	JMP PC ← R[SA]

* For this state and input combinations, PC ← PC+1 also occurs

Control Unit

```
-- ------------------------------
-- controller                   --
-- ------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;
entity controller is
    Port ( clk : in std_logic;
           opcode : in std_logic_vector(6 downto 0);
           nzset : in std_logic;
           carry : in std_logic;
           neg : in std_logic;
           zero : in std_logic;
           overflw : in std_logic;
           IL : out std_logic;
           PS : out std_logic_vector(1 downto 0);
           DX : out std_logic_vector(3 downto 0);
           AX : out std_logic_vector(3 downto 0);
           BX : out std_logic_vector(3 downto 0);
           FS : out std_logic_vector(3 downto 0);
           MB : out std_logic;
           MD : out std_logic;
           RW : out std_logic;
           MM : out std_logic;
           MW: out std_logic);
end controller;

architecture Behavioral of controller is
    type state_type is (RES, INF, EX0);
    attribute enum_encoding : string;
    attribute enum_encoding of state_type : type is "0001 0010 0100";
    signal cur_state, next_state : state_type;
begin
        state_register: process(clk, reset)
        begin
                if(reset='1') then
                        cur_state<=RES;
                elsif (clk'event and clk='1') then
                        cur_state<=next_state;
                end if;
        end process;

out_func: process (cur_state, opcode, carry, zero, neg, overflw )
begin
    (IL, PS, MB, MD, RW, MM, MW) <= std_logic_vector'("00000000");
    (DX, AX, BX)<=std_logic_vector'(X"000");
    FS<="0000";
        case cur_state is
                when RES =>
                        next_state <= INF;
                when INF =>
                        next_state<=EX0;
                        MM<= '1';
                        IL <= '1';
                when EX0 =>
                        next_state<= INF;
                        case opcode is
                                when "0000000" =>
                                                PS <= "01";
                                                RW <= '1';
                                when "0000001" =>
                                                PS <= "01";
                                                RW <= '1';
                                                FS <= "0001";
                                                .....
                                when "1100000" =>
                                                if (zero = '1') then
                                                        PS <= "10";
                                                else
                                                        PS <= "01";
                                                end if;
                                                .....
                                when others=>
                                                report "Unrecognizable state"
                                                severity error;
                        end case;
        end case;
end process;

end Behavioral;
```

Outline for Pipelined Design

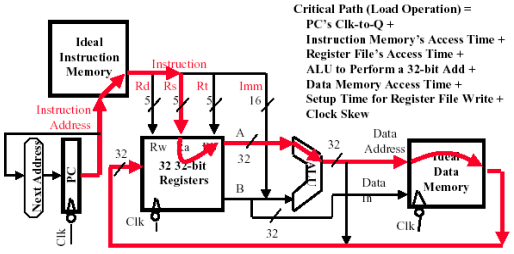
- Pipelined Design
  - Basic 5-stage pipe
    - Speedup of pipelined vs non-pipelined implementations
  - Pipeline hazards
    - Structural, data, control
- Parallel digital systems

7

Abstract View of Critical Path

- Register file and ideal memory:
  - The CLK input is a factor ONLY during write operation
  - During read operation, behave as combinational logic:
    - Address valid => Output valid after "access time."

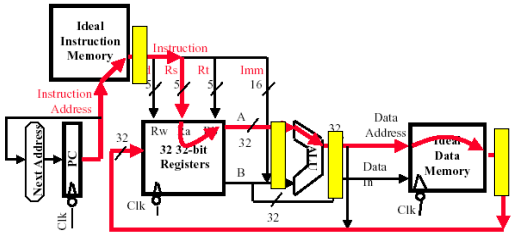
Critical Path (Load Operation) =
PC's Clk-to-Q +
Instruction Memory's Access Time +
Register File's Access Time +
ALU to Perform a 32-bit Add +
Data Memory Access Time +
Setup Time for Register File Write +
Clock Skew



8

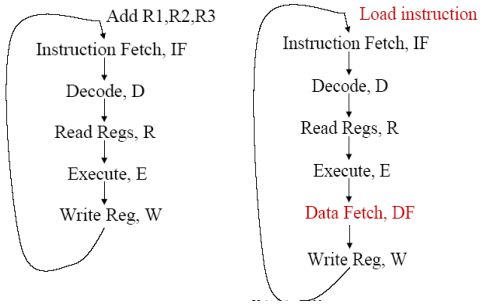
Pipelined critical path

- Critical path is longest path between stage registers



9

Steps in Instruction Processing



10

Un-pipelined (Non-overlapped) Implementation

- Consider loads with DF stage

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
I1	IF	DR	E	DF	W					
I2						IF	DR	E	DF	W
I3										
...										

11

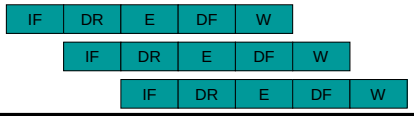
Pipelined Implementation

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
I1	IF	DR	E	DF	W					
I2		IF	DR	E	DF	W				
I3			IF	DR	E	DF	W			
I4				IF	DR	E	DF	W		
I5					IF	DR	E	DF	W	
I6						IF	DR	E	DF	W

12

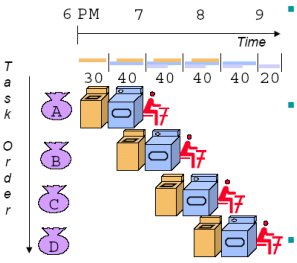
5-stage Pipeline

- CPU stages
  - IF: Instruction fetch
  - DR: Instruction decode & Register read
  - E: Execute
  - DF: Data fetch(Memory load/store)
  - W: Write Back Regs
- Another set of mnemonic names
  - IF, ID, E, MEM, WB



13

Pipelining Lessons



- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stages
  - Potential speedup = number of pipe stages
  - Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup

14

Computer Pipelines

- Execute billions of instruction, so throughput is what matters
- Throughput versus latency
  - + Throughput increases
  - - :Latency for a single instruction increases
    - May have to wait longer for single instruction to complete
- Allows much faster clock cycle
- RISC pipeline architecture features:
  - All instructions same length
  - Registers located in same place in instruction format
  - Memory operands only in loads and stores

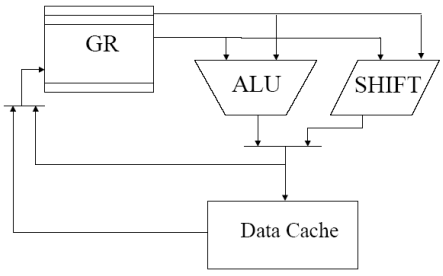
15

Pipelining

- Every clock cycle requires
  - New instruction fetch
  - New ALU operation
  - New data word to/from memory
- Memory Requirements
  - Faster memory (5x faster)
  - Separate instruction and data paths
  - Better caches

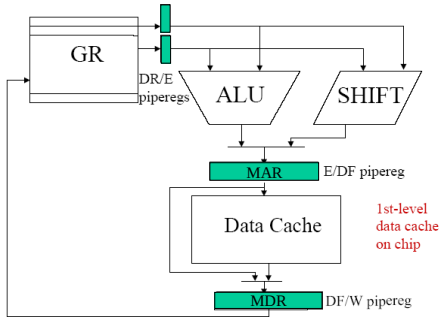
16

Unpipelined Datapath



17

Pipelined Datapath



18

Outline

- Pipelined Design
  - Basic 5-stage pipe
    - Speedup of pipelined vs. non-pipelined implementations
  - Pipeline hazards
    - Structural, data, control
- Parallel digital systems

19

Pipelining Hazards

- Hazards cause the pipe to stall because of some conflict in the pipe (prevents the next instruction in pipe from executing in its turn)
- Types of hazards
  - Structural: contention for same hardware resource
  - Data: dependency on earlier instruction for the correct sequencing of register reads and writes
  - Control: branch/jump instructions stall the pipe until get correct target address into PC

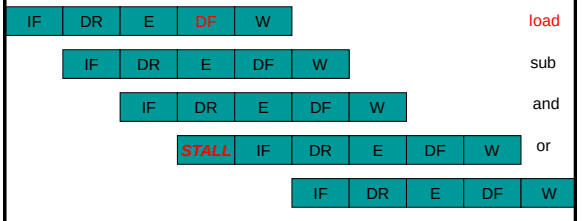
20

Structural Hazards

- Resource conflicts in the pipeline
- Examples
  - Single memory port shared for instruction and data access
  - Register file without a separate write port

21

Structural Hazards



22

Structural Hazards

- IF and DF compete for single memory port
- Ideal Machine
  - No stalls, 1 cycle per instruction
- Assume 30% of instructions access data
  - With structural hazard, 1.3 cycles per instruction
  - Performance has gone down by 30%
- Solutions:
  - Pipeline stall (insert bubble)
  - Have 2 memory ports for shared instruction-data cache-memory (expensive)
  - Have separate instruction cache-memory and data cache-memory

23

Three Generic Data Hazards (I) - RAW

- $Instr_1$ followed by $Instr_2$

  add r1, r3, r2
  add r4, r5, r1
- $Instr_2$ tries to read operand before $Instr_1$ writes it
  - Can be due to true “data dependency” (data must be produced before it can be consumed)
  - Or can be due to pipeline staging (data already produced, but not yet written to general register file

24

Data Hazards (II) - WAR

- $Instr_1$ followed by $Instr_2$

  ld r1, (r3)+
  add r3, r4, r1

- $Instr_2$ tries to write operand before $Instr_1$ reads it
  - $Instr_1$ gets wrong operand
  - Can't happen in the 5-stage RISC pipeline we just covered
    - All instruction take 5 stages
    - Reads are always in stage 2
    - Writes are always in stage 5

25

Data Hazards (III) - WAW

- $Instr_1$ followed by $Instr_2$

  mul r1, r0, r2
  add r1, r5, r6

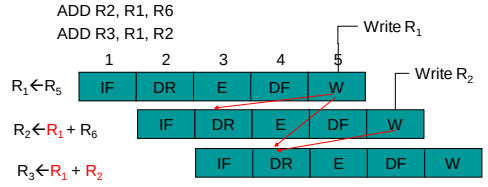
- $Instr_2$ tries to write operand before $Instr_1$ writes it
  - Leaves wrong result ($Instr_1$, not $Instr_2$)
  - Can't happen in our 5-stage pipeline because
    - All instructions take 5 stages
    - Writes are always in stage 5

26

Data Hazards

- Overlapping instructions cause dependencies on data (RAW)

e.g., MOVA R1, R5
ADD R2, R1, R6
ADD R3, R1, R2



27

Data Hazards Remedy - SW

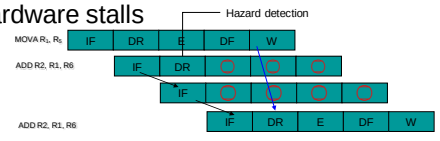
- Software delay (compiler or machine code programming to insert NOPs)

  MOVA R1, R5
  NOP
  NOP
  ADD R2, R1, R6
  NOP
  NOP
  ADD R3, R1, R2

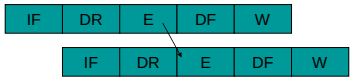
28

Data Hazards Remedy - HW

- Hardware stalls

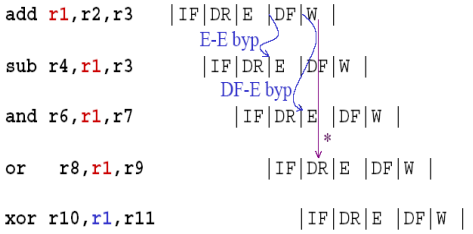


- Hardware Data Forwarding
  - Add an extra path connecting ALU outputs to ALU inputs on the next clock



29

Data Forwarding (Reg. Bypassing)



* Write regs in 1st. half of cycle, Read regs in 2nd. half

30

Pipelined Datapath – with data forwarding