

Summary for Lecture 1

- ❑ **Components of a computer**
- ❑ **Level of abstraction; instruction set architecture (ISA)**
- ❑ **Performance; throughput and response time**
- ❑ **CPU time (execution time): 1/performance**
- ❑ **Quantities**
 - **Total number of instruction per program**
 - **Total number of cycles per program**
 - **Cycle time; clock frequency**
 - **Cycle per instruction (CPI); Instruction per cycle (IPC)**
 - **Millions of instructions per cycle (MIPS)**
- ❑ **CPU time**

$$\text{CPU time} \equiv \frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}$$

$$\text{CPU Time (or, Execution Time)} = \frac{\# \text{ of instructions}}{\text{program}} \times \frac{\# \text{ of cycles}}{\text{instruction}} \times \frac{\# \text{ of seconds}}{\text{cycle}}$$

Summary for Lecture 2

$$\text{average CPI of a program} = \frac{\text{total \# of cycles}}{\text{total \# of instructions}}$$

$$\text{MIPS} = \frac{\text{total millions of instructions}}{\text{total execution time in seconds}}$$

- ❑ One program does not characterize a CPU; SPEC
- ❑ Amdahl's Law

$$T_{\text{after}} = \left(1 - f + \frac{f}{S}\right) T_{\text{before}}$$

$$\text{Speedup} = \frac{T_{\text{before}}}{T_{\text{after}}} = \frac{1}{1 - f + \frac{f}{S}}$$

- ❑ First Quiz: 4/15/04, in class

Summary for Lecture 3 (slide 32 to 40)

□ MIPS instruction set architecture (ISA)

- **Arithmetic instructions use only register values as operands**
 - Add `r_dest, r_src1, r_src2`
- **Memory instructions use register and mem. address as operands**
 - Lw `r_dest, offset(r_base)`
 - Sw `r_src, offset(r_base)`

□ Memory addressing

- **Byte address**
- **Word address: multiple of 4 (assuming 4 byte per word)**

Summary for Lecture 4 (slide 41 to 59)

□ MIPS instruction set architecture (ISA)

– Instructions that change the control flow:

- Beq $r_src1, r_src2, displacement$
- Bne $r_src1, r_src2, displacement$
 - Target address = $PC + 4 + displacement \times 4$
- Slt r_dest, r_src1, r_src2
- J $partial\ address$
 - Target address = $PC[31..26] || partial\ address$

– Instructions that handle small constants:

- Addi(slti, andi, ori) $r_dest, r_src1, immediate\ value$
- Lui $r_dest, immediate\ value$

□ Three instruction formats

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

Summary for Lecture 5 (slide 60 to 70)

❑ RISC and CISC

- Memory operand; instruction length

❑ PowerPC vs. 80x86

❑ Binary representations

- Sign-magnitude, one's complement, two's complement
- Represent-able value range using 2's complement given n bits: $-2^{(n-1)} - 2^{(n-1)} - 1$

❑ “negate” vs. “invert”

❑ Converting short binaries into long binaries

- Sign extension

Summary for Lecture 6 (slide 71 to 77)

❑ Binary addition and subtraction

- Subtraction is done through adding the negative subtrahend

❑ Detecting overflow of arithmetic operations.

- Occurs when the result is out of value range of an N bit binary ($-2^{(N-1)} - 2^{(N-1)-1}$)

❑ How to built logics for a function

1. Build truth table
2. Write boolean equations
3. Implement the equations using AND, OR, NOT, et.c gates

❑ How to build a 32-bit ALU

- We implement these operations: *andi*, *ori*, *add*, *sub*, *slt*, *test equal*
- Build 1-bit ALU for the operations first, and connect them into 32 bits
- Use a mux to select the desired results according to operations

Summary for Lecture 7 (slides 78 to 85)

❑ Use one-bit adder to build 32-bit adder

- $c_{out} = a b + a C_{in} + b C_{in}$
- $sum = a \text{ xor } b \text{ xor } C_{in}$

❑ Subtraction needs to negate subtrahend first

- Invert every bit, $C_0 = 1$

❑ SlT is implemented through subtraction

- The sign of the result determines the output
- Output 1 if the result is negative, 0 otherwise

❑ Equality is tested also through subtraction

- Output is 1 if the result is 0

❑ How to determine the control lines

- 000 = and
- 001 = or
- 010 = add
- 110 = subtract
- 111 = slt



Summary for Lecture 9 (C5 slides 1 to 14)

□ State elements

– Unclocked

- S-R latch

– Clocked

- Latches: D-latch – positive level sensitive
- Flip-flop: D-flip-flop – negative edge sensitive

□ Register file

– Read port

– Write port

Summary for Lecture 10 (C5 slides 15 to 24)

- ❑ **Datapath for R-type, memory instructions, and branch instructions**
- ❑ **How are the control signals generated?**
 - **Two-level implementation**
 - **Main control signals**
 - Inputs: opcode field in the instruction;
 - Output: ALUOP + other control signals
 - **ALU control signals**
 - Inputs: ALUOP + funct field in the instruction
 - Outputs: three-bit control signals to the ALU
 - **In any case, build the truth table first, simplify it and then build the logic diagram**

Summary for Lecture 12 (C5 slides 25 to 46)

- ❑ **Single-cycle datapath: every instruction takes the same amount of time – longest instruction execution time**
 - Inefficient in performance
 - Waste of hardware resources – functional units need to be duplicated
- ❑ **Multi-cycle datapath: every instruction takes different number of cycles**
 - Splitting from single-cycle datapath needs to add internal registers and muxes
 - Thus, control signals need to be generated for some of those registers and all of the muxes
- ❑ **Five execution steps (cycles)**
 - fetch, decode/register_read, execution, memory access/R-type completion, write back
 - What are the control signals generated in each step?

Summary for Lecture 12 (continued)

- ❑ **Implementing the control logic for multi-cycle datapath**
 - Finite state machine
 - Each state represents the set of control signals for a certain instruction in a certain cycle
 - First two states, fetch (state 0) and decode/register_read (state 1) are common to all instructions
 - Transition from one state to the next state depends on the instruction type, or the opcode field in the instruction
- ❑ **PLA implementation, ROM implementation, and microprogramming implementation**

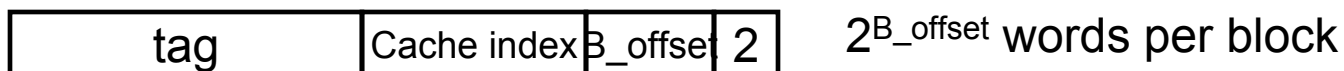
Summary for Chapter 7 (slides 1 to 15)

- ❑ **SRAM cells are fast, but big and more expensive; DRAM cells are slow, but small and cheap**
 - **Use a combination of them to build the memory hierarchy**
 - **Purpose: give CPU an illusion that the memory is big and fast!**
 - **The levels close to CPU are small and fast; levels further away from the CPU are bigger and slower**
- ❑ **Principle that makes memory hierarchy work – locality**
- ❑ **CPU always tries the 1st level (cache) first; 2nd level on 1st level misses, and so on. Meantime, it brings the contents found in lower levels to the higher levels in the hierarchy to better exploit locality.**

Summary for Chapter 7 (continued)

□ Direct mapped caches

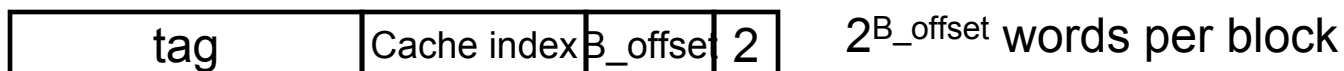
- Any memory block can be mapped into only one location in the cache
- How do find the location using simply the memory address?



- Read hit; read miss; write hit; write miss

□ Set-associative caches

- Any memory block can be mapped into multiple locations (a set) in the cache



Summary for Chapter 6

□ Pipeline instruction execution to increase its throughput

- Overlap their execution stages
- Five-stage execution – at most 5 instructions are in the pipeline
- Every stage a new instruction is fetched
- Add pipeline latches between stages; Latches store the context of the current instruction, including its control signals
- Latch content advances together with the instructions in the pipeline

□ Problems with pipelining

- Structural hazards
- Control hazards
- Data hazards

Summary for Chapter 6 (continued)

- ❑ **Resolve structural hazards by replicating functional units**
- ❑ **Resolve control hazards by scheduling branch delay slots**
 - **What kind of instructions can be put into delay slots?**
- ❑ **Resolve data hazards by data forwarding**
- ❑ **Add forwarding and hazard detection logic into the pipeline.**
When hazards can not be resolved, stall the pipeline
 - **Not enough functional units**
 - **Load results used by immediate following R-type instruction**
 - **No instruction in the delay slot**