COS375
Fall 2023
github/jovyntls github/JothamWong

Instruction Set Architecture

- Architecture: Abstraction layer provided to SW to interface with HW.
  - Examples: ARM, x86, MIPS
  - Designed for stability and to hide implementation details.
  - Includes programmer visible state/operations/instructions/data types/sizes/execution semantics.
- Microarchitecture: Physical instance of contract. Implementation details.
  - Includes additional state not visible to programmer such as cache size.
- Design goals
  1. Portability
  2. Programmability
  3. Efficiency (power/energy/speed/code size/cost)

MIPS

- MIPS: Microprocessor without Interlocked Pipelined Stages
- word size: 32 bits or 4 bytes (1 byte = 8 bits)
- Branching in MIPS has delay slots: the next instruction after a branch command happens before the PC is updated to the branch

Data/Control Flow Graph

- Basic block: Single entry/single exit. You cannot execute instructions in the middle without executing from the start.
- Liveness: A register is live at a point in code if it holds a value that may be needed later.
  - All reads expose a live range for that register upward.
  - All writes prevent a live range for that register from continuing upward.
- Live out registers: registers who values are used afterwards. Cannot be overridden.
- Data dependency: Three forms of dependencies
  - Flow/RAW (True dependency):
  - Output/WAW (False dependency):
  - Anti/WAR (False dependency):
- False dependency == can be removed with register renaming
- Dominance: A node (basic block or instruction) M dominates a node N if every path in the CFG from entry to N passes through node M.
- All nodes dominate themselves, so the entry node dominates all nodes, including itself.
- Optimization tricks
  - Register renaming
  - Constant propagation: The constant of a move immediate operation replaces later uses of the destination register.
    - Safe to perform when 1) move-immediate dominates the use, 2) no intervening writes of the destination register on any path between the move-immediate to the use.
  - Constant Folding: Constants within an ALU instruction can be replaced with the computed value. Always safe to perform.
  - E.g: r1 = 3 + 5 becomes r1 = 8
  - Dead Code Elimination: Instructions that only write to dead registers (dead == value not used) can be eliminated. If an instruction writes to a dead register but has side effects (throws exceptions, modifies memory), it is not dead code.
- Sign Extension: Copy the most significant bit until 32 (or whatever) bits. Returns a signed integer.
- Zero Extension: Pad up with 0s until 32 (or whatever) bits. Returns an unsigned integer.

Integer representation

Bit Pattern	Unsigned	Sign Magnitude	One's Complement	Two's Complement
000	0	+0	+0	0
001	1	+1	+1	+1
010	2	+2	+2	+2
011	3	+3	+3	+3
100	4	-0	-3	-4
101	5	-1	-2	-3
110	6	-2	-1	-2
111	7	-3	-0	-1

- Sign magnitude and one's complement: Have duplicate zeros
- Two's complement: most significant bit is sign bit, 1 if negative 0 if positive
- The sign bit is multiplied by -1 and the rest are positive to get value
- Negating a two's complement: invert all bits then add 1 to result
- Sign extension works because 2 complement's really have infinite 0/1s on the left if positive/negative
- Overflow: With w bits, overflow is just (u + v) mod 2^w
  1. Hardware does not detect overflow (MIPS: addu, addiu, subu). Onus is on software to check.
  2. Hardware maintains a flag to detect overflow but does nothing.
  3. Hardware raises exception, PC jumps to predefined addr for exception and prior addr is saved for possible recovery. (MIPS: add, addi, sub)

Pseudo Instructions

Name	Example	Equivalent MISP Instructions
Load address	`la $t0, label`	`lui $at, hi-16-bits` `ori $t0, $at, lower-16-bits`
Load immediate	`li $t0, 0xdeadbeef`	`lui $at, 0xdead` `ori $t0, $at, 0xbeef`
Branch if less or equal	`ble $s0, $s1, label`	`slt $at, $s1, $s0` `beq $at, $zero, label`
Move	`move $s0, $s1`	`add $s0, $s1, $zero`
No operation	`nop`	`sll $zero, $zero, 0`

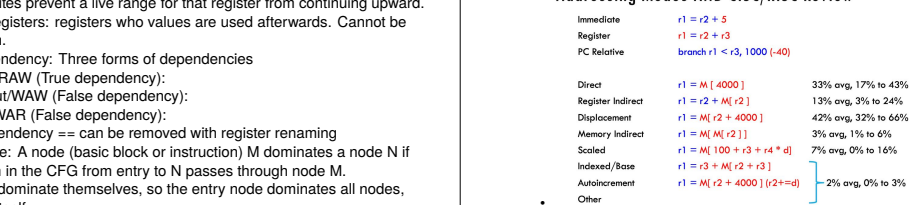
at (register no 1) is the assembler temporary and is reserved by the assembler.

Control instructions

- In assembly, the branch is typically short distance, range is lower than jump's range.
- The reason for padding right with 2 zeros is to make the address divisible by 4. The immediate specifies the i-th multiple of 4, not the i-th address.
- Jump addr can be anywhere.

Addressing Modes

- Memory can be viewed as a single 1-dimensional array of words (32 bits/4 bytes).
- MIPS words are aligned: this means we need to times 4 when indexing (see later).



- Indexed/base, scaled, and memory indirect not supported in MIPS

Endianness

How are bytes of a word laid out?

➢ **Little Endian:** Least-significant byte is stored in the location with the lowest address (little end first)

Address	0000	0001	0002	0003
Byte #	0	1	2	3

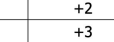
➢ **Big Endian:** Most-significant byte is stored in the lowest address (big end first)

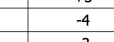
Address	0000	0001	0002	0003
Byte #	3	2	1	0

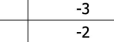
Registers vs Memory

- Registers are faster to access than memory as operating on memory requires load and store ops but have comparatively little space.
- Space available
  - Registers: 32 32 bits registers = 32 * 4 bytes = 2^7 bytes
  - Memory: 2^{32} bytes (2^{25} times more space)
- Compilers use registers as much as possible, spill to memory when required (known as register pressure).

Boolean Algebra

- NOT 

- AND 

- OR 

- Boolean Algebra Laws
  1. Identity : A.1 = A, A + 0 = A
  2. Zero and one: A.0 = 0, A + 1 = 1
  3. Inversion : $A.\overline{A} = 0$, $A + \overline{A} = 1$
  4. Idempotence : A.A = A, A + A = A (Useful for generating extra terms)
  5. Commutativity: A.B = B.A, A + B = B + A
  6. Associativity: A.(B.C) = (A.B).C, A + (B + C) = (A + B) + C
  7. Distribution: A + (B.C) = A.B + A.C, A + (B.C) = (A + B).(A + C)
  8. DeMorgan's : $\overline{A.B} = \overline{A} + \overline{B}$, $\overline{A + B} = \overline{A} \,.\, \overline{B}$

Performance

- Million Instructions Per second (MIPS) = $\frac{\text{Instruction count}}{\text{Execution Time} \times 10^6}$
- The Iron Law of Computer Performance:
  $\text{Time} = \text{Instructions} \times \frac{\text{Cycles}}{\text{Instructions}} \times \frac{\text{Time}}{\text{Cycles}}$
- $\frac{\text{Cycles}}{\text{Instructions}}$ = CPI (1 / IPC)
- $\frac{\text{Time}}{\text{Cycles}}$ = 1/(Clock Frequency)

Category	Example	Instruction Count	CPI	Cycle Time (1/frequency)
Algorithm	Sort	✓	✓	✗
Programming Language	C, Python	✓	✓	✗
Compiler	gcc -> LLVM	✓	✓	✗
ISA	MIPS->x86	✓	✓	✓
µArch		✗	✓	✓
Technology		✗	✗*	✓

- * If memory takes the same amount of time, loads might have to wait more cycles – CPI would change in that case

Index Card Quiz

$\text{Time} = \text{Instructions} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$

You are designing a new CPU and a new ISA. The base design has 32 general-purpose registers and operates at a clock rate of 1GHz. A member of your CPU's compiler team has suggested that if you can architect in 64 registers instead of 32 (i.e., change the instruction encodings, etc.), they will be able to reduce instruction counts for a key target program from 10M to 9M. But you suspect that the larger register file will be slower to access, and so the cycle time will get worse. What is the "break-even" cycle time and clock rate for this?

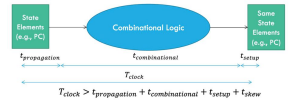
NOTE: 1GHz (1 billion cycles per second) = 1/1ns (each cycle is 1 nanosecond or 1 billionth of a second)
Also Associate:
MHz and microsecond
KHz and millisecond
Hz and second

9M · cpi. 1 = 10M · cpi. 1
1/1.111 ns/cycle ≈ 0.9GHz or 900 MHz

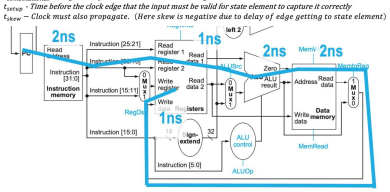
- A cycle is a step. An instruction may take any number of steps. Frequency is the number of cycles per second.
- Single cycle forces all instructions to take 1 step hence CPI = 1, at the expense of long cycle time.

Single Cycle Processor

- Critical path: The clock cycle is determined by the operation that requires the longest time.
- Clocking Methodology: Way to prevent unreliable results in computer hardware due to reading/writing at same time.
- Clock-Edge: Only read/write on a rising/falling clock edge. Point is it's standardized.



$t_{propagation}$ - Time for signal to propagate through state element
$t_{combinational}$ - Longest delay time for the combinational logic
t_{setup} - Time before the clock edge that the input must be valid for state element to capture it correctly
t_{skew} – Clock must also propagate. (Here skew is negative due to delay of edge getting to state element)

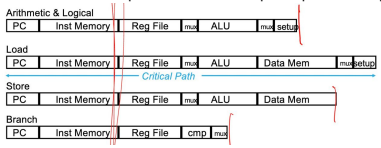


Load Word is longest running instruction
(prove to yourself by computing time for sw, br, R-type)

memory (2ns), ALU/adders (2ns), register file access (1ns)

2ns + 1ns + 2ns + 2ns + 1ns = **8ns**

- Combinational circuits: Output solely depends on input. Implements a boolean function.
- State elements: Sequential circuits. Output depends on input and state.



1. Long cycle time, wasted inefficiency.
2. Real memory takes a variable amount of time to read and write.
3. Depending on the ratio of loads/stores/adds/branches, the ideal average time could be much lower.
4. Haven't even accounted for floating point operations yet which are much slower.
5. Some units hold value long after job is complete, ALU could be reused, underutilized resources.

Control

Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

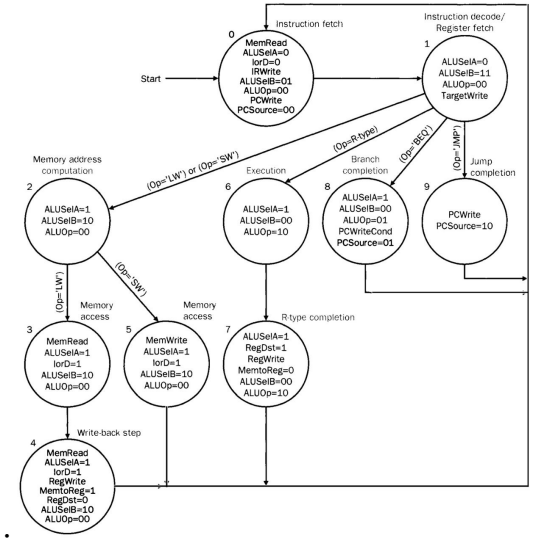
Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

Multicycle Processor

- From "The best way to design a machine": Break instructions into steps, one per cycle, such that
  - Amount of work in each step is balanced
  - Restrict each cycle to only one major functional unit
  - Shorter cycle time
- At the end of a cycle, store values for use in later cycles.
- Introduce "internal" registers for values between cycles.
- Each instruction broken into multiple steps, each step defined as one clock cycle.

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR = Memory[PC] PC = PC + 4			
Instruction decode/register fetch	A = Reg [IR[25-21]] B = Reg [IR[20-16]] ALUOut = PC + (sign-extend (IR[15-0]) << 2)			
Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + sign-extend (IR[15-0])	if (A ==B) then PC = ALUOut	PC = PC [31-28] II (IR[25-0]<<2)
Memory access or R-type completion	Reg [IR[15-11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
Memory read completion		Load: Reg[IR[20-16]] = MDR		

- The control now needs to be specified as a FSM instead of a combinational circuit since there is some state to it.



Pipelining

- Why can't single cycle (from lecture) do pipeline?
- Why can't multi cycle (from lecture) do pipeline?
- Pipelining increases throughput but does not decrease latency (the time taken to complete one instruction)
- We use the same 5 stages as multicycle: IF, ID, EX, MEM, WB.

Pipeline Hazards

- **Structural Hazard**: Hardware cannot support the combination of instructions that we want to execute in the same clock cycle.
- Example: Only a single memory instead of two. Cannot have an instruction access data from memory and another fetch instruction from memory.
- **Data Hazard**: Occurs when the pipeline must be stalled because one step must wait for another to complete. Arises due to RAW dependencies.
- Example: add $s0, $t0, $t1; sub $t2, $s0, $t3.
- This can be resolved via **forwarding/bypassing**. Store the computed value in a buffer and pass to the ALU.
- Forwarding is not enough to resolve load data dependency (known as **load-use data hazard**) since the mem is written one stage after the input. For example, R-type after a lw.
- The concept of **pipeline stall/bubble** where we either detect and reorder or stall.
- **Control Hazard**: The flow of instruction addresses is now what pipeline expects.
- Possible approaches include stalling, branch prediction and simply delayed decision (delay slots in MIPS)