Multi-Core Computer Architecture: Storage and Interconnects Week 1

Aronya Baksy

August 2021

1 Introduction

1.1 Instruction Execution Cycle

- Every executable program is represented as a sequence of instructions that are executed on the CPU. The execution cycle of one instruction is as follows:
  1. **Instruction Fetch**: Obtain the instruction from the program storage
  2. **Instruction Decode**: Determine the required action to be performed and the instruction size
  3. **Operand Fetch**: Locate the operands (may be in registers or memory)
  4. **Execute**: Compute the result or modify the status (as per the action determined in step 2)
  5. **Result Store (memory)**eposit results into appropriate storage for later use
  6. **Next instruction**: Determine the next instruction.
     - If not branch, then next instruction is in sequence
     - If unconditional branch, jump to the appropriate label
     - If conditional branch, check the status flags and appropriately jump to the next instruction
- Registers in a generic CPU:
  - **General Purpose Registers**: used to store operands and results
  - **Program Counter** (PC): Holds the memory address of **next instruction** to be fetched from memory
  - **Memory Address Register** (MAR): Holds the address of the memory location that is to be accessed (for a load or a store)
  - **Memory Data Register** (MDR): Holds the data read from or stored to memory (who's location is inside the MAR)
  - **Instruction Register** (IR): Holds the currently executing instruction
- Memory organization can be either big endian or little endian
  - **Big Endian**: Most significant byte (leftmost byte) is in the lowest memory address
  - **Little Endian**: Most significant byte (leftmost byte) is in the highest memory address

1.2 Processor-Memory Interaction

1.2.1 Instruction Execution

- PC value is supplied to MAR
- MAR enables those specific address lines, and the data at the location is read from memory and stored in the MDR
- From the MDR, the control logic stores that value into the IR
- The decode logic gets the appropriate operands, loads them into the registers, and then triggers the ALU with the correct signals (as per the opcode)
- The output of the ALU is stored in a temporary register, and from there transferred to the output register

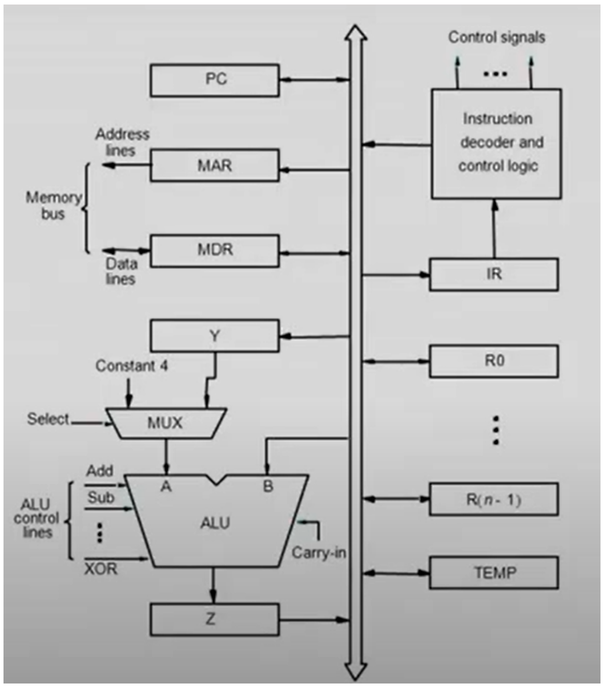


Figure 1: Generic Layout of CPU

1.2.2 Role of MAR and MDR

- The MAR is connected to a $n : 2^n$ decoder (where n is the size of MAR in bits) called the **address decoder**.
- The address decoder output selects one particular data line through which data will flow.
- Data flows from MDR to memory in case of a store operation, or from memory to MDR in case of a load operation.

1.2.3 Using multiple memory chips

- Let there be 8KB (2^{13} bytes, hence addresses are 13 bits long) of memory to be addressed by the CPU. Let this 8KB be organized into 8 modules each of 1 KB.
- In this case, the highest 3 bits will index which chip (out of 8) will be used. Hence the highest 3 bits are connected to a 3:8 decoder that selects the chip.
- The remaining 10 bits of the address are connected to all the 8 modules. The location selected is indexed by a combination of the decoder output and the 10 lower order bits of address.

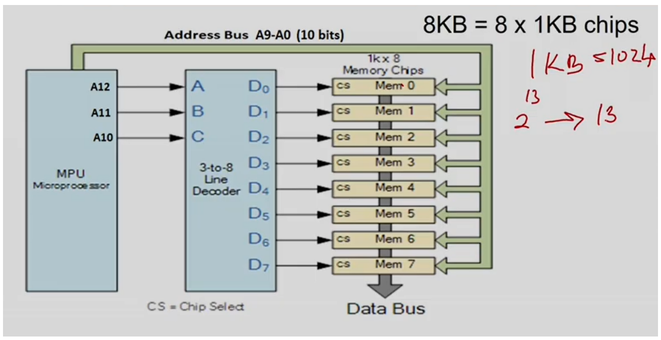


Figure 2: Memory Addressing

2 Performance Metrics

- Choice of performance metric depends on the application area (desktop - execution time, server - transactions per unit time aka throughput)
- Performance comparisons are typically made using ratios
- Some performance metrics:
  - Response time
  - Throughput (number of tasks per unit time)
  - Execution time
  - CPU time (ignoring I/O and other overheads from execution time)
- Standard benchmarks for performance measurement:
  - **Synthetic Benchmarks**: Do not represent any real-world applications, only simulate loads on CPU
  - **Benchmarking Suites**: SPEC2006, SPLASH etc.
- The SPECRatio is a metric defined as:

$$SPECRatio_A = \frac{\text{Execution time on reference machine}}{\text{Execution time on machine } A}$$

- Each benchmark suite has a pre-defined reference machine (For SPEC 2006, Sun Ultra Enterprise 2 workstation, 296 MHz UltraSparc II processor)
- 2 architectures A and B are compared using their SPECRatios as follows:

$$\frac{SPECRatio_A}{SPECRatio_B} = \frac{\frac{\text{Execution time on reference machine}}{\text{Execution time on machine } A}}{\frac{\text{Execution time on reference machine}}{\text{Execution time on machine } B}} = \frac{\text{Execution time on machine } B}{\text{Execution time on machine } A}$$

- One program yields one such ratio. To combine results from multiple benchmark programs, take the Geometric Mean of the SPECRatios.

2.1 Amdahl's Law

- "The speedup of a program using multiple processors in parallel is limited by the time needed for the sequential fraction of the program" - Gene Amdahl, 1967
- Mathematically,

$$S = \frac{1}{s + \frac{1-s}{P}} \tag{1}$$

  where S is the speedup obtained, s is the sequential (serial) fraction of the program (hence $1 - s$ is the parallelizable fraction) and P is the number of processing units available to work in parallel.

- As $P \to \infty$, we have $S \to \frac{1}{s}$ (the **maximum theoretical speedup**, which clearly depends only on the serial fraction s)

2.2 Average CPI Calculations

- Average CPI is calculated as

$$CPI = \frac{\sum_{i=1}^{n} IC_i \times CPI_i}{\text{Instr. Count}}$$

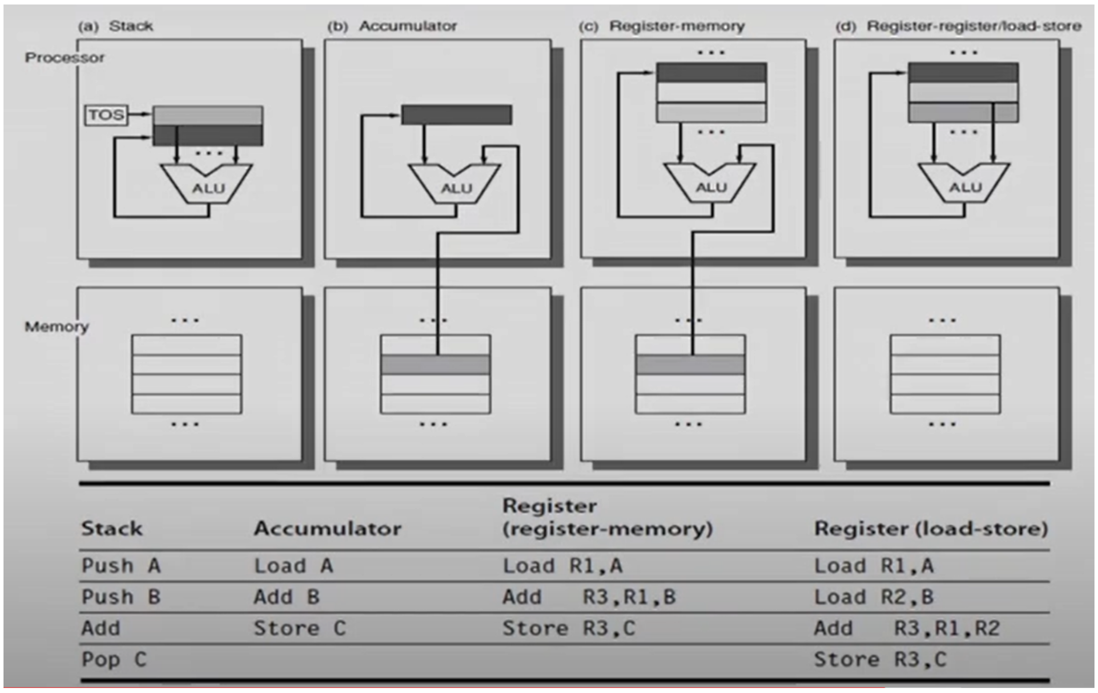
  where IC_i is the number of instructions of type i having CPI equal to CPI_i

- Hence, total CPU time:

$$Time = \left(\sum_{i=1}^{n} IC_i \times CPI_i \right) \times \text{Clock Cycle Time}$$

3 Fundamentals of ISA

- Things to remember:
  - Instruction denotes a single machine task consisting of an opcode and operands
  - Multiple instructions make up a program. Multiple programs make up software
- Types of ISA: stack, accumulator, register-memory and register-register



Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R3,R1,B	Load R2,B
Add	Store C	Store R3,C	Add R3,R1,R2
Pop C			Store R3,C

Figure 3: Classifications of ISA

3.1 Stack Architecture

- Operands are taken only from a single stack.
- Every operation operates on the top 2 operands in the stack and stores the result back on the top of stack
- e.g.: in the figure, since A and B are on top of stack, the `Add` operation uses A and B to store the result in a new variable on top of stack called C

3.2 Accumulator Architecture

- One operand from the memory directly, the other one from a special register called the **accumulator**.
- Result of the operation is stored in the accumulator

3.3 Register-Memory Architecture

- One operands is taken from the register file, the other one from memory
- Result is stored in a register and can be written to memory from the register

3.4 Register-Register Architecture

- Both operands are taken from the registers
- Result is stored back in register
- This architecture is als called the **load-store architecture** as it involves loading operand values from memory to register, and then storing the output from register to memory.

4 RISC Instruction Pipeline

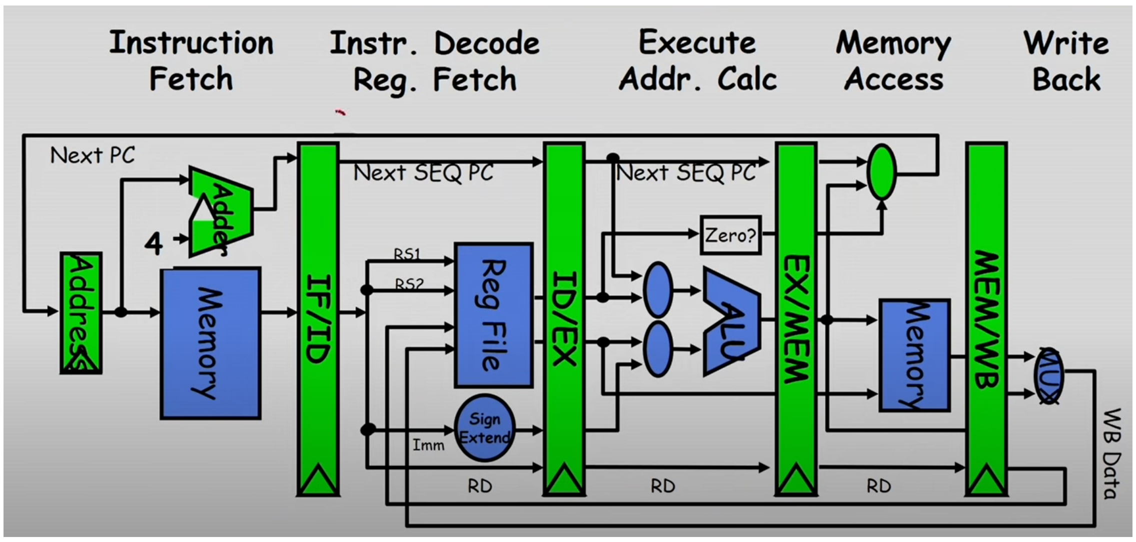


Figure 4: RISC Pipeline

- Characteristics:
  - Improves throughput of the entire workload, but doesn't affect latency of a single task
  - Limited by the slowest stage of the pipeline
  - Potential speedup by pipelining = number of stages
  - Unbalanced pipeline (where each stage takes different time) has less speedup (time to fill the pipeline and drain the pipeline reduces speedup)
- **Important**: For an ideal pipeline, CPI = 1 (one instruction should finish per cycle, this ignores the time needed to fill up the pipeline)

4.1 Issues with Pipelining

- The RISC pipeline is not balanced, some stages take more time than others. (since memory access takes most time, number of memory accesses must be minimized, the number of memory addressing modes must be minimized, and fast caches should be used)
- Some stages are not used in all instructions (e.g.: memory stage not used by ALU instructions). This is mitigated by reducing the complexity of the ISA and using uniform stage instructions
- Instructions are never independent of one another. Dependent instructions will cause hazards that lead to pipeline stalls. This is mitigated by reducing number of memory addressing modes (making dependency deteciton easy) and using register addressing modes.

4.2 Pipeline Hazards

- Hazards prevent the next instruction in the instruction stream from being executing during its designated clock cycle.
- Hazards reduce the performance from the ideal speedup gained by pipelining.
- Types of hazards:
  - **Structural Hazards**: Attempting to use the same hardware to do 2 different things at the same time
  - **Data Hazards**: Instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline. (e.g.: WAW, WAR, RAW)
  - **Control Hazards**: Arise from pipelining of branches and other instructions that change the PC.

4.2.1 Structural Hazards

- Caused by access to hardware by multiple instructions at the same time
- e.g.: one instr attempts to write to memory while another attempts to fetch the instruction from memory, but memory has only one port for r/w.
- Solutions for structural hazards:
  1. Detect a structural hazard and ask one instruction to wait until the hardware is free
  2. Duplicate the required hardware so that it can be used parallely (e.g.: use separate instruction and data caches, also called I-cache and D-cache to store instructions and data, instead of storing both together in memory)

4.2.2 Data Hazards

Let instruction I and J be in sequence such that J comes after I

- **Read-after-Write Hazard**: In a RAW hazard, I writes to a location and J tries to read the value that I wrote. Because of pipeline organization, J may read a stale value of data. (also called **true dependency**)
- **Write-after-Read Hazard**: In a WAR hazard, instruciton J writes to a register before I can read it. In RISC 5 stage pipeline this doesn't occur because read happens in stage 2, write happens in stage 5. Also called **anti dependency**.
- **Write-after-Write Hazard**: I and J both write to same register, but this must happen in order or else data inconsistencies will occur. Also not seen in 5 stage pipeline
- WAR and WAW hazards occur in out-of-order processors.
- RAW hazards (but not all of them) are handled by **operand forwarding**.
- In case of a load followed by an ALU which uses the result of the load, there is no choice but to stall the pipeline (Forwarding will not work here)
- Software scheduling of instructions by the compiler also helps in resolving data hazards (instruction reordering)

4.2.3 Control Hazards

- In a conventional MIPS pipeline, the target address is known only after the 4th stage (memory stage).
- The optimization here is to not allow register comparision during branching, but only allow the branch to test value inside one register
- This allows branches to be resolved in the decode stage (2nd stage)
- e.g.: Instructions like `BEQ R1, R3, 690` will not be allowed as `R1` and `R3` have to be compared for equality. The branch instructions will be of the form `BEZ R1, 690` which branches to instruction at address 690 only if `R1 == 0`
- Handling control hazards:
  - Stall pipeline till branch is resolved
  - Use branch prediction technique
  - Delayed branching: fill up stalls with useful instructions

4.3 Deeper Pipelines

- Splitting the execution stage (3rd pipeline stage) into multiple functional units
- Each functional unit has its own latency, and it may be pipelined or not pipelined.
- Since functional units have varying latency, it can lead to *out-of-order completion* even though instructions were issued in order

4.3.1 Issues with high-latency deep pipelines

- Structural hazards caused by the fact that division unit is not pipelined (FP add and FP multiply are pipelined)
- Instructions have varying runtime, hence more register read/write per cycle (this can be resolved at the decode stage)
- WAW hazards can occur (due to out-of-order completion)
- Stalls for RAW hazards are longer (because of longer delays in execution)

4.4 Superscalar Pipeline

- A pipeline that completes more than one instruction in a single clock cycle
- **Static Scheduling**: reorder instructions in a manner that reduces number of stalls while maintaining the correctness of the program. This is done at compile time
- **Dynamic Scheduling**: rearrange the execution order of instructions in a way that maintains the data flow between instructions
- Dynamic scheduling separates decode stage into 2 parts:
  1. **Issue**: Decode the instruction and check for structural hazard
  2. **Read Operand**: Wait for absence of data hazard then read operand
- **Multithreading**: Allow instructions from multiple streams to be fetched, but only one stream is decoded in each decode slot
- **Hyperthreading**: Allow instructions from multiple streams to be fetched, and multiple streams to be decoded in each decode slot. Also called simultaneous multithreading