Parts of an ISA: Class of ISA; Memory Addressing; Addressing Modes; Types and sizes of Operands; Operations; Control flow instructions; Encoding

\*Die Cost Formulas; although useful—are fairly straightforward

MTTF – mean time to failure; MTTR – mean time to repair

Availability = MTTF / (MTTF + MTTR)

Amdahl’s Law

Speedup = ExecTime Old / ExecTime New

Speedup\_overall = 1 / ( 1 – Frac\_enh) + Frac\_enh / Speedup\_enh )

CPU time = CPU cycles for program \* cycle time

CPU time = CPU cycles for program / clock rate

Instruction set design, Compilers and ISA (App B)

Four Instruction Set Architecture Classes: Stack; Accumulator; Register-Memory; Register-register/load-store

Register-Memory allows lots of freedom; since you do not necessarily need to load into a register before using a value.

|  |  |  |  |
| --- | --- | --- | --- |
| **Mode** | **Example** | **Meaning** | **When used** |
| Register | ADD R4, R3 | *Regs[R4] = Regs[R4] +*  *Regs[R3]* | When a value is in a register |
| \*Immediate | ADD R4, #3 | *Regs[R4] = Regs[R4] + 3* | For constants |
| Register indirect | ADD R4, (R1) | *Regs[R4] = Regs[R4] +*  *Mem[Regs[R1] ]* | Accessing with pointer or computed address |
| Direct or  absolute | ADD R4, (1001) | *Regs[R4] = Regs[R4] +*  *Mem[ 1001 ]* | Accessing static data; address constant |
| \*Displace  ment | ADD R4, 100 (R1) | *Regs[R4] = Regs[R4] +*  *Mem[ 100 + Regs[R1] ]* | Accessing local variables |
| Indexed | ADD R4, (R1 + R2) | *Regs[R4] = Regs[R4] +*  *Mem[Regs[R1] +Regs[R2]]* | Array addressing: R1 = base:R2 = index amount |
| Auto  increment | ADD R4, (R2) + | *Regs[R4] = Regs[R4] +*  *Mem[Regs[R2] ]*  *Regs[R2] = Regs[R2] + d* | arrays within a loop. R2 =start; R2+ d. |
| Auto decrement | ADD R4, -(R2) | *Regs[R2] = Regs[R2] – d*  *Regs[R4] = Regs[R4] +*  *Mem[Regs[R2] ]* | Same as above  also act as push/pop to implement a stack |
| Scaled | ADD R4, 100 (R2) [R3] | *Regs[R4] = Regs[R4] +*  *Mem[100 + Regs[R2] +*  *Regs[R3] \* d]* | Used to index arrays. |

Control Flow Instructions: conditional branches, jumps, calls, returns.

Basic 5 stage pipeline stages: Instruction Fetch; Instruction Decode; Execute; Memory Access; Write-back.

Register accesses can often run on first and second part of cycle.

Pipeline Hazards:

Structural Hazards – hardware cannot support instructions simultaneously in overlapped execution.

Data Hazards – instruction depends on result of previous instruction that cannot be overlapped.

Control Hazards – from the pipelining of branches and other instructions that change the Program Counter (PC).

Speedup = CPI unplined \* Clock cycle unplined / CPI plined \* clock cycle plined

CPI pipelined = 1 + Pipeline stall clock cycles per instruction

Speedup = Pipeline depth / (1 + Pipeline stall cycles per instruction)

Forwarding can be used to alleviate some data hazards.

Branch delay slot; also useful.

\*Review scheduling branch delay slot, pg A-24.

Pipeline speedup = Pipeline depth / (1 +Pipeline stall cycles from branches)

Pline stall cycles = branch frequency \* branch penalty

3 checks that must be done before issuing an instruction if the pipeline does all hazard detection in the ID stage.

Check for structural hazards

Check for a RAW data hazard

Check for a WAW data hazard

Sometimes register renaming can make loop unrolling work better.

Basically just unroll the loop if you know you’ll go through it at least so many times.

Scoreboard; Goal: Allow out of order execution and out of order completion.

Runs in basically four pieces: Issue; Read Operands; Execution Complete; Write Result.

Keeps track of instruction states, register states, and functional unit states.

**\*Also; stalls to avoid certain hazards-which hazards (unroll-scoreboard.pdf).**

Tomasulo:

Uses register renaming to avoid RAW and WAW.

Reservation stations have pending operations.

Results broadcast over common data bus.

Reorder buffer (ROB) provides additional registers. It holds the result of an instruction between the time it completes and commits. It has four fields: the instruction type (is it a branch, store, register op); the destination field; the value field. The ready field indicates that the value is ready.

Branch-Target Buffers: Predicts the next instruction address and will send it out before decoding the instruction. 3 fields: Look up; predicted PC; branch taken or untaken.

Branch Prediction

CPI = (1-branch%) \* non-branch CPI + branch% \* branch CPI

CPI = (1-branch%) \* 1 + branch% \* (1 + penalty)

CPI = 1 + (branch% \* penalty)

Penalty = not taken% \* not taken cost + taken% \* taken cost

2-bit predictor must be incorrect twice to change state.

After first incorrect prediction it becomes less certain, basically. The same goes for the opposite case.

(m,n) Correlating Predictors

Record m most recently executed branches as taken or not taken, and use that pattern to select the proper branch history table

(m,n) predictor means record last m branches to select between 2m history tables each with n-bit counters

This one does the pattern matching prediction stuff.

Tournament Predictors are just state transition tables where the state is actually which predictor to use. And when they’re wrong you move around as though it was one large predictor.

Cache:

Hit time\_L1 + Miss Rate\_L1 \* (Hit Time\_L2 + Miss Rate\_L2 \* Miss Penalty\_L1)

Direct-Mapped Cache – fairly straightforward…

Cache block address = (Block address) modulo (Number of cache blocks)

And each entry has: a Valid bit, a Tag, and the data.

If you have multiple words/block Block address = (byte address) / (bytes per block).

Average Access Time = Hit Time \* (1 - Miss Rate) + Miss Penalty \* Miss Rate

Set associative allows for blocks that would map to the same location to map to other locations, allowing for better cache utilization.

Set number = (Block number) modulo (Number of sets in the cache)

For a fully associative cache: Compare the Cache Tags of all cache entries in parallel.

Two write strategies: write-through and write-back. Write-back has to deal with blocks being dirty or clean, but write-through has to always send writes to memory.

Write-through can have nice write-buffer.

Cache Misses

Compulsory – because it has to be read in at some point.

Capacity – because we don’t have room

Conflict – too many blocks in set, etc.

Ways of improving cache:

Larger block size reduces miss rate

Bigger caches reduce miss rate

Higher associativity reduces reduces miss rate

Multi-level caches reduce miss penalty

Priority to reads over writes reduces penalty

Avoid translation during indexing of the cache to reduce hit time

Advanced ways:

Victim Cache: holds things just dropped out of cache, because they are often needed again quickly.

Critical Word First: request the missed word first; send to CPU; then get rest.

Early restart: normal order, but as soon as the block arrives, send it to CPU.

Compiler Optimization:

Code/Data Rearrangement—branch straightening

Loop Interchange – access in order stored in memory

Blocking – work on blocks of data, versus loading entire rows/column

Hardware Prefetching of Instructions and Data

Cache Coherency Models:

Snooping Solution (Snoopy Bus)

–Send all requests for data to all processors

–Processors snoop to see if they have a copy and respond accordingly

–Requires broadcast, since caching information is at processors

–Works well with bus (natural broadcast medium)

–Dominates for small scale machines (most of the market)

Write Invalidate Protocol:

–Write to shared data: an invalidate is sent to all caches which snoop and invalidate any copies

–Cache invalidation will force a cache miss when accessing the modified shared item

–For multiple writers only one will win the race ensuring serialization of the write operations

–Read Miss:

–Write-through: memory is always up-to-date

–Write-back: snoop in caches to find most recent copy

Write Broadcast (Update) Protocol (typically write through):

Directory-Based Schemes

-Three States: Shared, Uncached, Modified

–Keep track of what is being shared in one centralized place

–Distributed memory ⇒ distributed directory for scalability (avoids bottlenecks)

–Send point-to-point requests to processors via network

–Scales better than Snooping

–Actually existed before Snooping-based schemes

Terms: typically 3 processors involved

–Local node where a request originates

–Home node where the memory location of an address resides

–Remote node has a copy of a cache block, whether exclusive or shared

Virtual Memory

Advantages

–Allows efficient and safe data sharing of memory among multiple programs

–Moves programming burdens of a small, limited amount of main memory

–Simplifies program loading and avoid the need for contiguous memory block

–Allows programs to be loaded at any physical memory location

Uses an in memory page table

Translation Look-aside Buffer

A TLB has a fixed number of slots containing page table entries, which map virtual addresses onto physical addresses.

RAID 0: no redundancy

RAID 1: mirroring, two copies of everything

RAID 2: Memory-style ECC

RAID 3: Bit-interleaved parity

RAID 4: Block-interleaved parity

RAID 5: Block-interleaved distributed parity

RAID 6: Row-diagonal parity, EVEN-ODD

Disk Access Time = Seek time + Rotational Latency + Transfer time

+ Controller Time + Queuing Delay

HDL & Simulation, Parallel Systems (class PDFs)

Varying levels of abstraction for Simulation.

VHDL is for writing models of a system

Levels of Parallelism:

Bit-level parallelism

-ALU parallelism: 1-bit, 4-bits, 8-bit, …

-Instruction-level parallelism (ILP)

-Pipelining, Superscalar, VLIW, Out-of-Order execution

-Process/Thread-level parallelism

-Divide job into parallel tasks

-Job-level parallelism

-Independent jobs on one computer system

Uniform Memory Access; all processors have a uniform latency form memory.

Complex parallel systems

Hypercube structure; something about neighbors with bits that are similar