High Performance Computing
Lecture 25

Matthew Jacob

Indian Institute of Science

Control Hazard Solutions

1. Static Branch Prediction
2. Delayed Branching

- Design hardware so that control transfer takes place after a few of the following instructions

```
BEQ   R1, R2, target
ADD   R3, R2, R3
```

- Delay slots: following instructions that are executed whether or not the branch is taken
- Stall cycles are avoided if the delay slots are filled with useful instructions

Delayed Branching: Filling Delay Slots

- Instructions that do not affect the branching condition can be put in the delay slot
  - by the compiler
- Where to get instructions to fill delay slots?
  - From the branch target address
    - only valuable when branch is taken
  - From the fall through (branch not taken path)
    - only valuable when branch is not taken
  - From before the branch
    - useful whether branch is taken or not

Delayed Branching…Compiler's Role

- When filled from branch target or fall-through, patch-up code may be needed

```
            BEQZ  R1,  target
                                        / Branch delay slot
fall through:
                . . .
 target:  ADDI   R7,  R7,  1
            LW  R8,  -8(R29)
                . . .
```

Delayed Branching…Compiler's Role

- When filled from branch target or fall-through, patch-up code may be needed

```
                BEQZ  R1,  target
                ADDI   R7,  R7,  1        / Branch delay slot
fall through:   SUBI   R7,  R7,  1
                  . . .

     target:  LW  R8,  -8(R29)
                  . . .
```

Delayed Branching…Compiler's Role

- When filled from branch target or fall-through, patch-up code may be needed
- It may still be beneficial, depending on branching frequency
- The more the number of delay slots, the harder it is to fill them usefully

If no instruction can be found…

- The compiler must insert an instruction that does nothing
  - other than occupying the delay slot, being fetched and decoded
  - Example: ADD R0, R0, R0
  - If an instruction that does nothing was included in the instruction set, it would be called a No-Operation instruction, or NOP for short
  - NOP might be included in the assembly language
  - It has practically the same effect as a STALL cycle

Pipeline and Programming

- Consider a simple pipeline with the following warnings in the ISA manual
  1. One load delay slot
  2. One branch delay slot
  3. 2 instructions after FP arithmetic operation can't use the value computed by that instruction
- We will think about a specific program, say vector addition

```
double A[1024], B[1024];
for (i=0; i<1024; i++) A[i] = A[i] + B[i];
```

Vector Addition Loop

```
                                    / R1: addr(A[0]), R2: addr(B[0])
Loop: FLOAD  F0,  0(R1)      / F0 = A[i]
      FLOAD  F2,  0(R2)      / F2 = B[i]
      FADD F4,  F0,  F2      / F4 = F0 + F2
      FSTORE  0(R1),  F4     / A[i] = F4
      ADDI  R1,  R1,  8      / R1 increment
      ADDI  R2,  R2,  8      / R2 increment
      BLE   R1,  R3,   Loop
                                    / R3: address(A[1023])
```

11 cycles per iteration

Vector Addition Loop

```
Loop: FLOAD  F0,  0(R1)
      FLOAD  F2,  0(R2)
      FADD F4,  F0,  F2
      FSTORE  0(R1),  F4
      ADDI  R1,  R1,  8
      ADDI  R2,  R2,  8
      BLE   R1,  R3,  Loop
```

11 cycles per iteration

```
Loop: FLOAD  F0,  0(R1)
      FLOAD  F2,  0(R2)
      ADDI  R1,  R1,  8
      FADD F4,  F0,  F2
      ADDI  R2,  R2,  8
      BLE   R1,  R3,  Loop
      FSTORE  -8(R1),  F4
```

7 cycles per iteration

An even faster loop? Loop Unrolling

- Idea: Each time through the loop, do the work of more than one iteration
  - More instructions to use in reordering
  - Less instructions executed for loop control
  - … but program increases in size

Loop Unrolling

```
Loop: FLOAD  F0,  0(R1)
      FLOAD  F2,  0(R2)
      FADD F4,  F0,  F2
      FSTORE  0(R1), F4
      ADDI  R1,  R1,  8
      ADDI  R2,  R2,  8
      BLE   R1,  R3, Loop
```

```
Loop: FLOAD  F0,  0(R1)
      FLOAD  F2,  0(R2)
      FADD F4,  F0,  F2
      FSTORE  0(R1), F4
      FLOAD  F0,  8(R1)
      FLOAD  F2,  8(R2)
      FADD F4,  F0,  F2
      FSTORE  8(R1),  F4
      ADDI  R1,  R1,  16
      ADDI  R2,  R2,  16
      BLE   R1,  R3,  Loop
```

18 cycles for 2 iterations (9 cycles/iteration)

Reorder to reduce to 5.5 cycles per iteration

Agenda

1. Program execution: Compilation, Object files, Function call and return, Address space, Data & its representation (4)
2. Computer organization: Memory, Registers, Instruction set architecture, Instruction processing (6)
3. Virtual memory: Address translation, Paging (4)
4. Operating system: Processes, System calls, Process management (6)
5. Pipelined processors: Structural, data and control hazards, impact on programming (4)
6. Cache memory: Organization, impact on programming (5)
7. Program profiling (2)
8. File systems: Disk management, Name management, Protection (4)
9. Parallel programming: Inter-process communication, Synchronization, Mutual exclusion, Parallel architecture, Programming with message passing using MPI (5)

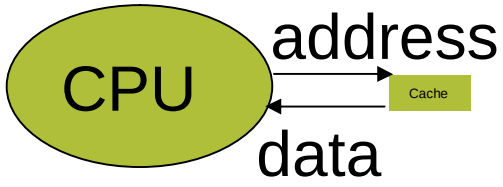
Cache Memory; Memory Hierarchy

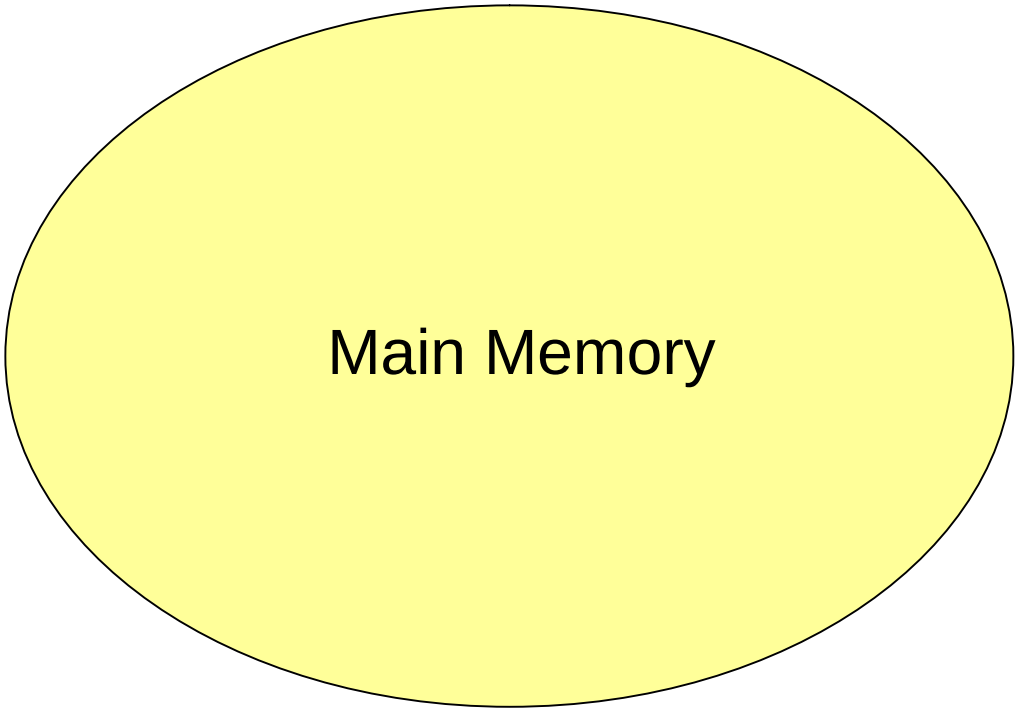
- Recall: In discussing pipeline, we assumed that memory latency will be hidden so that it appears to operate at processor speed
- Cache Memory: HW that makes this happen
  - Design principle: Locality of Reference
  - Temporal locality: least recently used objects are least likely to be referenced in the near future
  - Spatial locality: neighbours of recently referenced locations are likely to be referenced in the near future

Cache Memory Exploits This

Cache: Hardware structure that provides memory objects that the processor references

- directly (most of the time)
- fast





Cache Design