Lecture 10: Processing Instructions

William Gropp

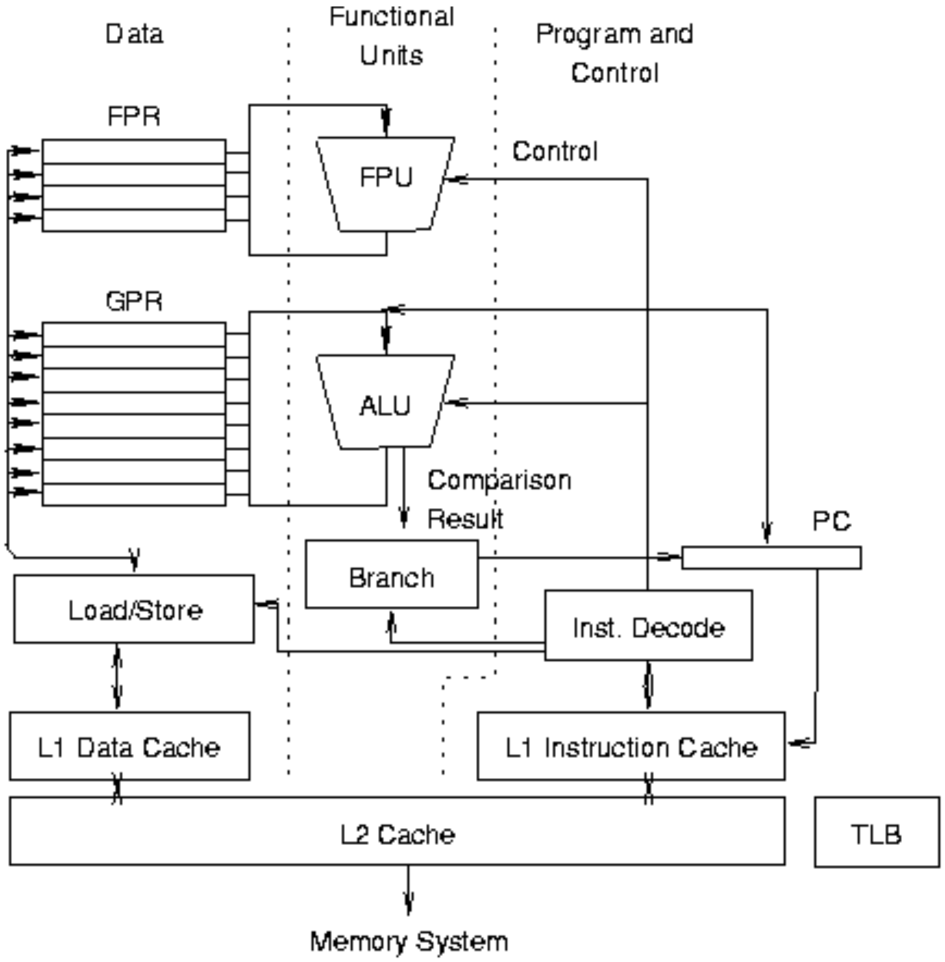
www.cs.illinois.edu/~wgropp

More on the CPU

- There are many details that we’ve ignored
  - Can more than one operation take place at a time?
  - Does each assignment require a store into memory?
  - What about the other operations (loop counts and tests, array indexing, etc.)?
- Before answering these, lets revisit the CPU

Basic Processor Architecture

- A representative processor architecture
- Key points:
    - Multilayered memory system
    - Multiple functional units permit concurrent actions (e.g., loads and floating point operations)
    - Some operations (e.g., address translation) have hardware assist (TLB) but may fall back on software



More Details

- Can more than one operation take place at a time?
  - Yes, if they involve different functional units
  - Here, operations are both arithmetic and memory load or store
  - Or if there are multiple units of the same type, as long as enough units are available
- Note: Quickest way to add to peak floating point performance is to add floating point units

More Details (2)

- Does each assignment require a store into memory?
- Consider this code in C:

```
double sum = 0;
for (i=0;i <n; i++) {
    sum = sum + a[i];
}
```

- The value “sum” may be stored in register, requiring no load or store.
  - Making use of registers can be crucial in achieving high performance
  - Recall the CPU diagram: most operations take place between operands in register

More Details (3): Traps for the Unwary

- Consider these two codes in C:

```
Void sum(double *total, double *a, int n) {
  int I;
  for (I=0; I<n; I++) *total += a[I];
}
```

and

```
void sum2(double *total, double *a, int n) {
  double s = *total; int I;
  for (I=0; I<n; I++) s += a[I];
  *total = s;
}
```

- Do these codes compute the same result?

Perils of Aliasing

- They do not compute the same value!
- Consider this usage of the routines
  - Sum(&a[2], a, 3);
  - In the first case, the routine computes
    - A[2] + A[0] + a[1] + a[2] + a[0] + a[1]
    - Why?
  - In the second case, the routine computes
    - A[0] + a[1] + a[2]

Question for Review

- Stop here and show why
  Sum(&a[2], a, 3)
  computes
  - A[2] + A[0] + a[1] + a[2] + a[0] + a[1]
- Do this by writing out what happens at each iteration
- In Fortran, that would be
  call sum(a(3), a, 3)
  with a declared as double precision a(3)
  - Fortran experts will note that this is an *invalid* statement in Fortran

Aliasing of Data

- When two variables may describe overlapping memory regions, they are said to alias one another
  - Programming languages with pointers often permit aliasing (how can they prevent it)
  - The potential for aliasing can force the compiler to store a value (or in a different example, load it) even though the programmer does not intend to use aliased data

Impact on Compilers

- Most compilers do not generate code to check at runtime if aliasing is present – the decision is made at compile time, and if the compiler is not *certain* that aliasing is not present, it assumes that aliasing may be present

Helping the Compiler

- Additional information *may* help the C compiler:
  - const – data is "constant".
  - restrict – pointer is not aliased to any other pointer
- Nonstandard
  - pragma disjoint – specified pointers refer to separate (disjoint) memory areas
  - pragma ivdep – ignore "vector" dependencies
- Compiler command line options can sometimes be used (not recommended)

More Details (4)

- What about the other operations (loop counts and tests, array indexing, etc.)?
  - ♦ Operations on integers are relatively fast in modern CPUs
    - Exceptions include integer divide and modulus
  - ♦ Branches (conditional jumps to other parts of the code, such as at a loop test) are also relatively expensive
    - Many processors predict branches, and an incorrect prediction can be costly
  - ♦ However, most are still faster than an L2 cache miss

Can those Operations be Ignored in Performance Bounds?

- Not a priori – you should check
- To test whether they can be ignored, compute a *bound* on them:
  - Assume simple operations: add, integer multiply, compare, branch, etc.
  - Assume one cycle each
    - A very coarse assumption
  - Assume these can execute concurrently with loads, stores, and floating point operations
    - Remember the CPU diagram - each functional unit can run independently
  - In numerical calculations, it is *often* the case that the *sustained* load/store rate is the limiting step
    - But more computationally intensive calculations with complex control may be dominated by these "other" operations

Some Rules for Bounding Performance

- Most importantly remember: the goal is to create an effective (but possibly approximate) bound on performance - *not* an estimate!
  - Discussion Question: What's the difference?
- Count the number of operations in each *functional unit category:*
  - Loads/Stores
  - Floating Point (add, subtract, multiply - divides are a special subcase)
  - Other operations (integer arithmetic, branches, comparisons, etc.)
- For each of these, compute the time they will take
- The bound on the time is the **max** of these three
  - Note: not really a bound because weve ignored any dependencies between the different operations
  - You can refine each of these by including more detail
    - Refine load/store by considering cache