NTNU – Trondheim
Norwegian University of
Science and Technology

Instruction Level Parallelism

Jan.Christian.Meyer@ntnu.no

Today's topic

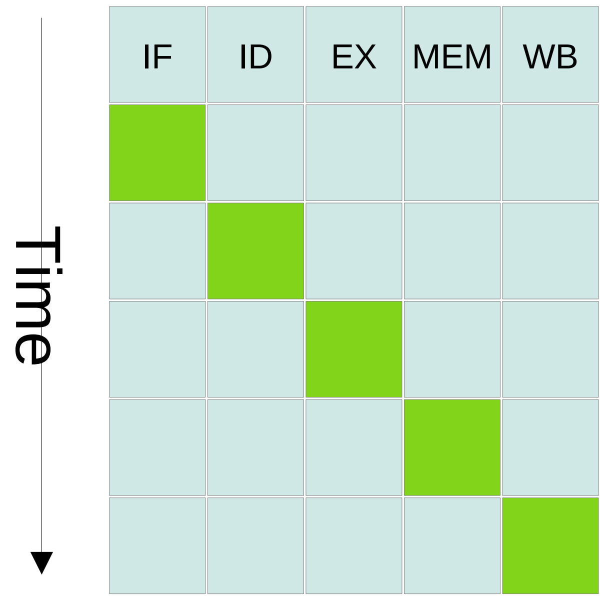
- A thorough treatment of all the ways to exploit parallelism at the instruction level requires a long chapter in a very thick book
- We don't need to design processors, so a simplified mental picture of what's going on will suffice
- I have chosen to highlight
  - Pipelining
  - Out-of-order execution
  - Prefetching & branch prediction
  - Vectorization

Pipelining

- During the clock race, improved switching frequencies made it tricky to complete one complex instruction each clock cycle
- Early RISC architectures broke each operation into 5 stages, to keep up:
  - IF (Instruction Fetch)
  - ID (Instruction Decode)
  - EX (Execute)
  - MEM (Memory)
  - WB (Write Back)

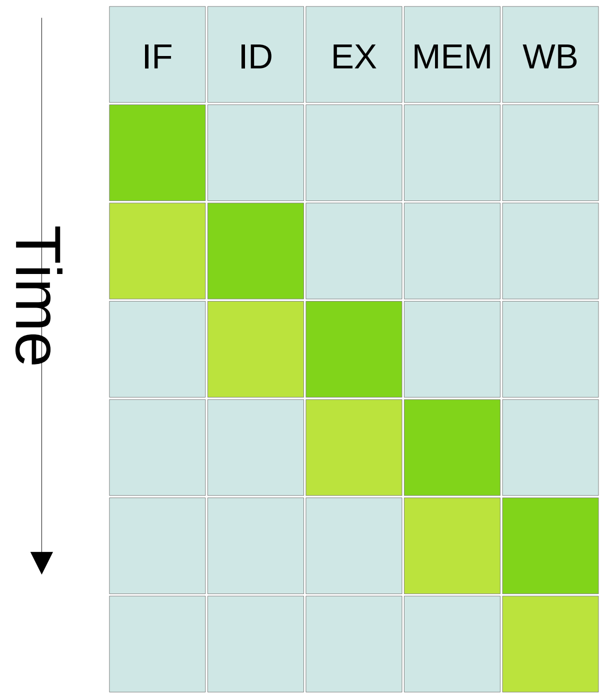
One instruction takes 5 steps

- Alone, an instruction is no faster than it would be with a 5x slower clock:



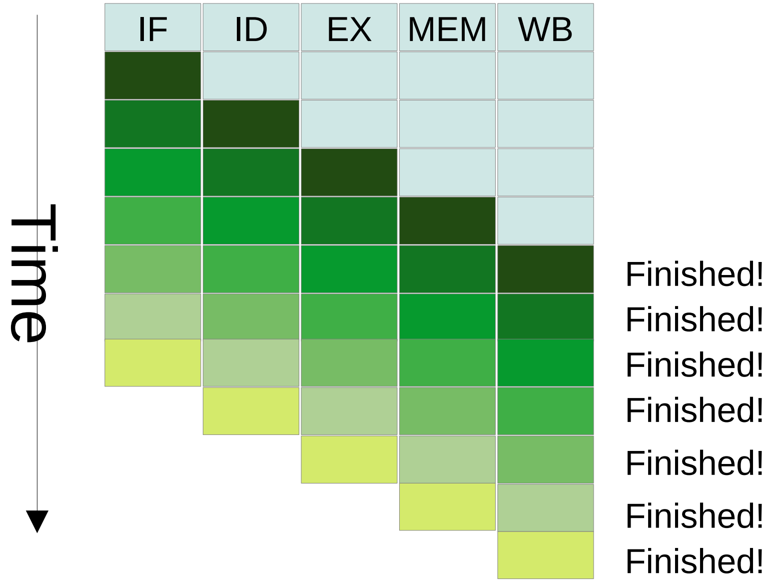
Two instructions take 6 steps

- With the operations broken into stages, we can start a new instruction when its predecessor goes to stage 2:



Full capacity

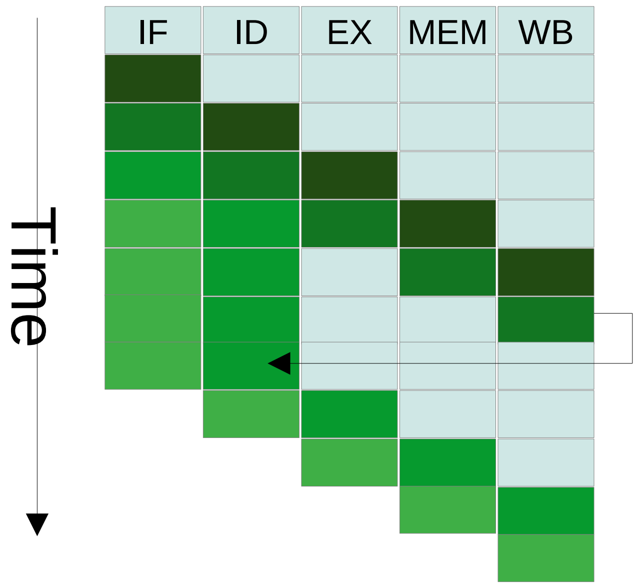
- When the pipeline is full (after a 5 step warmup), it produces a finished result for every step:



NTNU – Trondheim
Norwegian University of Science and Technology

Stall & flush

- If instruction 3 needs the result from instruction 2, the whole thing has to wait:



Out-of-order execution

- *Bernstein's conditions* define that statements S_1, S_2 will produce the same result when run in any order if
  - $R(S_1) \cap W(S_2) = \emptyset$ (S1 doesn't read what S2 writes)
  - $W(S_1) \cap R(S_2) = \emptyset$ (S1 doesn't write what S2 reads)
  - $W(S_1) \cap W(S_2) = \emptyset$ (S1 and S2 don't write in the same place)

  (this makes them *independent* statements)

- It's arguably just common sense, but looks better when dressed up as sets and operators
- When we have a window of instructions and their operands, their independence can be checked automatically

Dependences

- When instructions *aren't* independent, there is some kind of… dependence.
- We have three types:
  - Data dependence
  - Name dependence
  - Control dependence

Data dependence

- This is when the result of one operation is an input to a following operation:

```
// These two expressions are independent
t1 = a*c;
t2 = b*d;
// This one depends on both
x = t1 - t2;
```

Name dependence

- When a name is re-used for a different purpose, its first use must be finished before the second can begin:

```
norm = x[0]*x[0] + y[0]*y[0];
sum += norm;
norm = x[1]*x[1] + y[1]*y[1];
sum += norm;
…
```

- Programmers don't often re-use their names explicitly

  (and advanced compilers invisibly rename the variables if they do)

  but loop iterations can create this effect

- At the instruction level, the CPU has a limited number of registers to use, they have to be recycled every so often

Control dependence

- Branches in the program make it impossible to start operations simultaneously:

```
result = 0.0;
rad = x*x + y*y;
if ( rad < 1.0 )
   result = sqrt(rad);
```

- We can't overwrite `result` before the comparison between `rad` and `1.0` is complete

Superscalar processors

- With automatic (in)dependence detection in place, we can
  - replicate the ALU parts of the Neumann machine
  - make the control path dispatch several instructions at once
- This is fondly known as *multiple issue* in computer architecture

Prefetching & branch prediction

- Programs typically spend most of their time in long loops

  (at least when they spend most of their time using the CPU)

- Many loops create regular access patterns in memory:

```
double a[1000];
for ( int j=0; j<1000; j++ )
    a[j] = j*j;
```

  - This loop will ask for 1000 consecutive addresses that are all 8 bytes apart

Prefetching

- By default, that kind of loop will regularly create cache misses at the end of every cache line
- By (inexpensively) adding a small counter device that tracks how many times we've been fetching stride-8 addresses lately, upcoming misses can be avoided by starting the memory transfer early
- It'll often be correct, and when it isn't, the cache space will just be re-used anyway

Branch prediction

- When pre-loading instructions, `if()` statements, loop tails, *etc.* make it hard to decide which branch to pre-load
- A simple variant is just to always guess that branches will be taken

  ...or, for that matter, that they won't...
- Wrong guesses require the pipeline to be flushed, but without any kind of guess, we couldn't fill it in the first place

Slightly fancier branch prediction

- At a moderate increase in complexity,
  - A unit with 2 states can switch policies based on whether the last branch was actually taken or not
  - A unit with 4 states can switch policies based on the last two branches
  - Etc. etc.
- It is also possible to store branch statistics next to a table of the branch instructions' addresses in the code

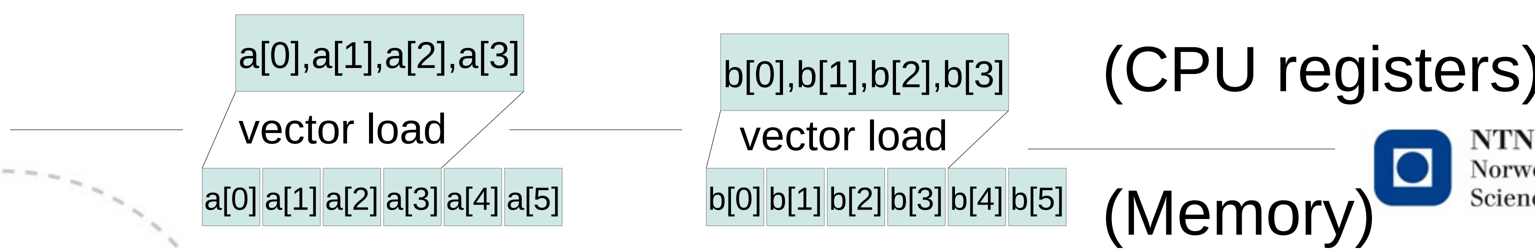
Vectorization

- Loops like this are not uncommon:

```
for ( int j=0; j<N; j++ )
    c[i] = a[i] + b[i];
```

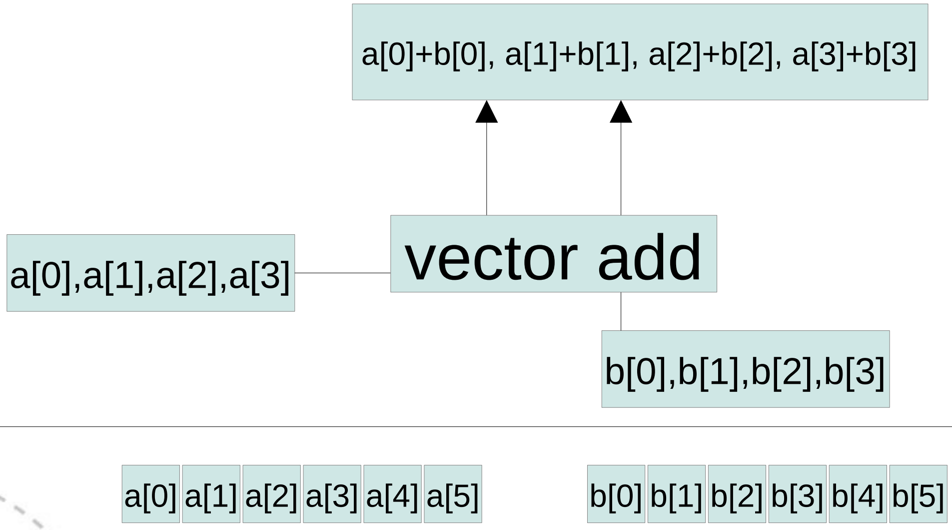
This can cause a register to contain a[0], a[1], a[2]… in sequence, another one to contain b[0], b[1], b[2], … and so on

- *Vector registers* are extra-wide registers that can store several consecutive values at once:



Combining multiple elements

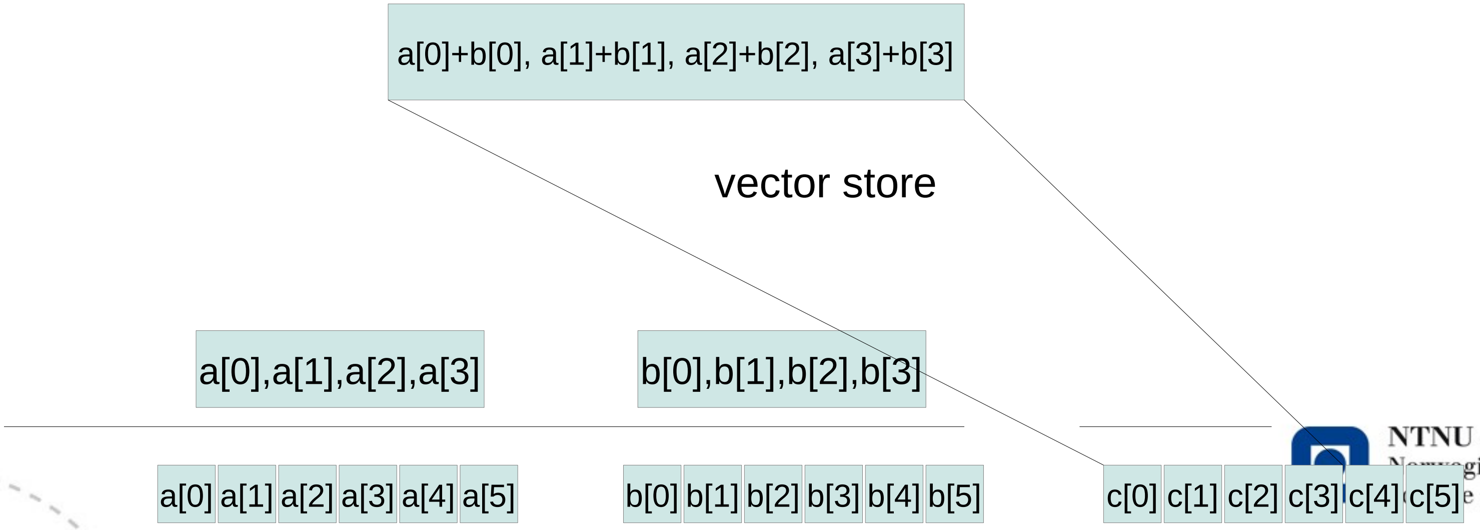
- With the vector registers loaded, there are instructions that do the same thing to all of their elements in parallel:



Combining multiple elements

- Packing data into wide registers like this, we can do 4 times the work in one of the *read-modify-write* cycles of Neumann

  (as long as the data are laid out consecutively in memory)



In practice

- These were all simple illustrations of principles
  - Real pipelines are often much deeper than 5 stages
  - Superscalar processors can “typically” issue 2-8 simultaneous operations
  - Highly sophisticated branch predictors and prefetchers have been invented, but most practical ones aren’t super complicated
  - Vectorization is ideally something for the compiler to divine from your source code, but that doesn’t always succeed
    - (We will look at how to do it by hand, one of these days)