Microarchitecture optimisation: Pipelining and caching COMSM1302 Overview of Computer Architecture

John Lapinskas, University of Bristol

Pipelining by analogy

Suppose you've let things pile up and you have many batches of laundry.

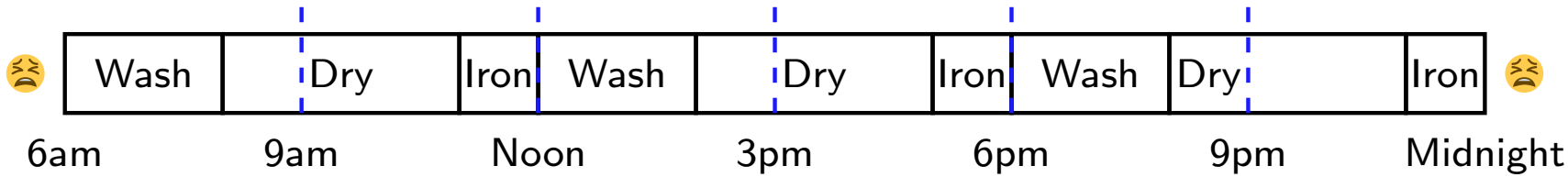
For one load, the washing machine takes 2 hours, the tumble dryer takes 3 hours, and ironing and folding takes 1 hour. How do you do it?

Pipelining by analogy

Suppose you've let things pile up and you have many batches of laundry.

For one load, the washing machine takes 2 hours, the tumble dryer takes 3 hours, and ironing and folding takes 1 hour. How do you do it?

The bad approach:

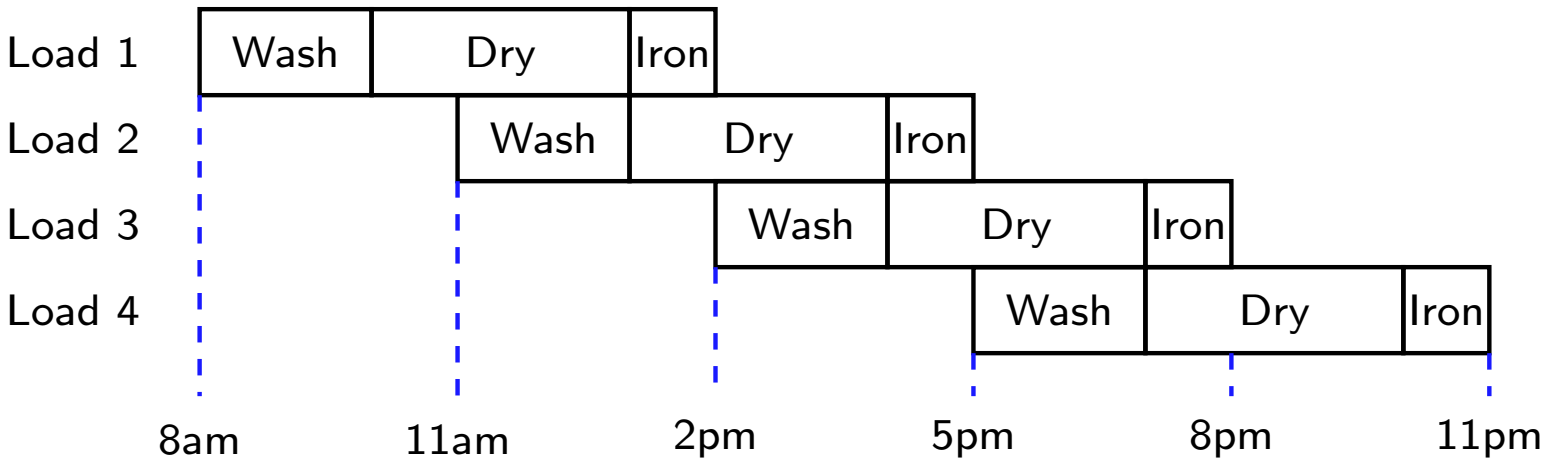


Pipelining by analogy

Suppose you've let things pile up and you have many batches of laundry.

For one load, the washing machine takes 2 hours, the tumble dryer takes 3 hours, and ironing and folding takes 1 hour. How do you do it?

The good approach:

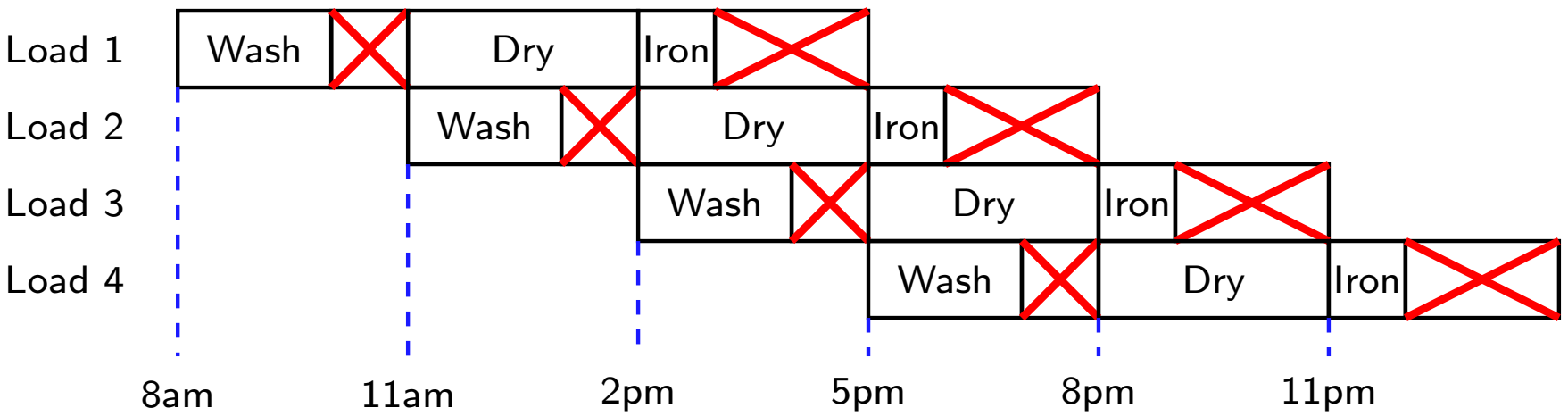


Pipelining by analogy

Suppose you've let things pile up and you have many batches of laundry.

For one load, the washing machine takes 2 hours, the tumble dryer takes 3 hours, and ironing and folding takes 1 hour. How do you do it?

Or, minimising the number of times we switch tasks (a.k.a. clock cycles):

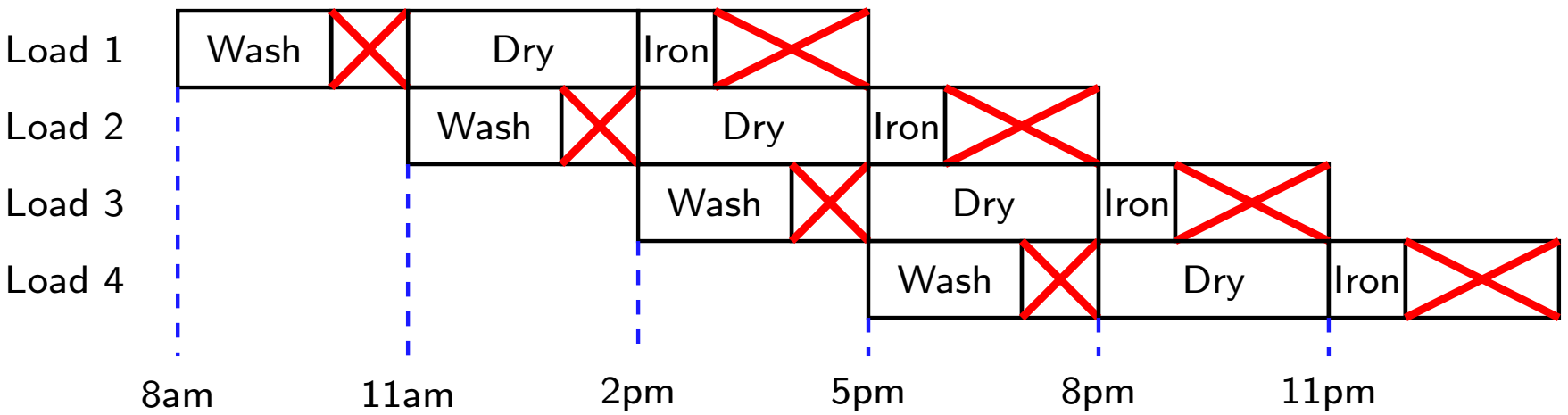


Pipelining by analogy

Suppose you've let things pile up and you have many batches of laundry.

For one load, the washing machine takes 2 hours, the tumble dryer takes 3 hours, and ironing and folding takes 1 hour. How do you do it?

Or, minimising the number of times we switch tasks (a.k.a. clock cycles):



The moral: By parallelising where we can to avoid letting components sit idle, we can massively increase throughput! This is called **pipelining**.

The ideal of pipelining

The main barrier to increasing a CPU's clock speed is **propagation delay**: The time taken for signals to pass through gates and assume stable values.

The ideal of pipelining

The main barrier to increasing a CPU's clock speed is **propagation delay**: The time taken for signals to pass through gates and assume stable values.

For C-instructions in Hack, we could divide the fetch-execute cycle into four stages (some of which would be skipped for A-instructions):

- **Fetch**: Fetch the next instruction from ROM[PC].
- **Decode**: Set the inputs for the ALU to the appropriate values.
- **Execute**: Set the outputs of the ALU to the appropriate values.
- **Writeback**: Write values to RAM and registers, update PC.

Each stage uses different hardware, so we can set the clock speed to the propagation delay of the slowest stage, rather than of the entire fetch-execute cycle!

The ideal of pipelining

The main barrier to increasing a CPU's clock speed is **propagation delay**: The time taken for signals to pass through gates and assume stable values.

For C-instructions in Hack, we could divide the fetch-execute cycle into four stages (some of which would be skipped for A-instructions):

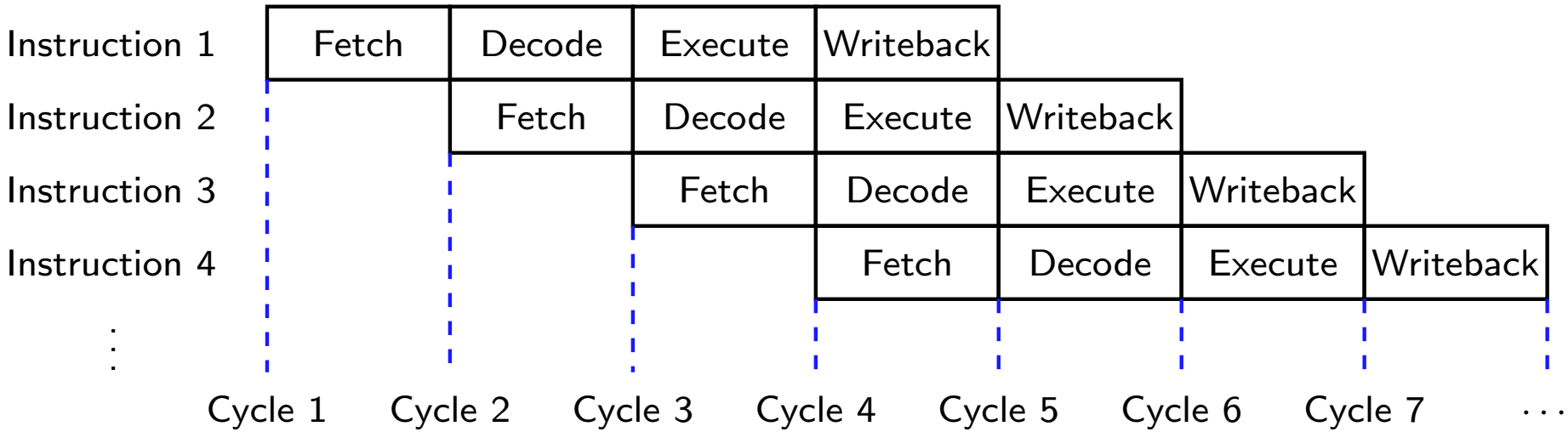
- **Fetch**: Fetch the next instruction from ROM[PC].
- **Decode**: Set the inputs for the ALU to the appropriate values.
- **Execute**: Set the outputs of the ALU to the appropriate values.
- **Writeback**: Write values to RAM and registers, update PC.

Each stage uses different hardware, so we can set the clock speed to the propagation delay of the slowest stage, rather than of the entire fetch-execute cycle!

Modern CPUs have more complex pipelines than this, but the same principles hold and these four stages are still a good basic framework.

The ideal of pipelining

So our fetch-execute cycle would become:



What could go wrong?

The reality of pipelining: Stalls and hazards

- **Data hazards** arise when an instruction needs access to data modified by an earlier instruction that hasn't finished evaluating.
  - E.g. `D=D+1` followed by `A=A+D`. The second instruction needs the new value of D from the first instruction.

The reality of pipelining: Stalls and hazards

- **Data hazards** arise when an instruction needs access to data modified by an earlier instruction that hasn't finished evaluating.
  - E.g. `D=D+1` followed by `A=A+D`. The second instruction needs the new value of D from the first instruction.
- **Conditional hazards** arise at a conditional branch. The next instruction to be loaded into the pipeline will depend on whether or not the branch occurs.
  - E.g. after `M;JEQ`, the next instruction might be at either ROM$[PC+1]$ or ROM$[A]$.

The reality of pipelining: Stalls and hazards

- **Data hazards** arise when an instruction needs access to data modified by an earlier instruction that hasn't finished evaluating.
  - E.g. `D=D+1` followed by `A=A+D`. The second instruction needs the new value of D from the first instruction.
- **Conditional hazards** arise at a conditional branch. The next instruction to be loaded into the pipeline will depend on whether or not the branch occurs.
  - E.g. after `M;JEQ`, the next instruction might be at either ROM[$PC + 1$] or ROM[A].
- **Structural hazards** arise when two instructions need access to the same physical resource, e.g. both using the same parts of the ALU. (These would not be an issue in creating a pipeline for Hack.)

The reality of pipelining: Stalls and hazards

- **Data hazards** arise when an instruction needs access to data modified by an earlier instruction that hasn't finished evaluating.
  - E.g. `D=D+1` followed by `A=A+D`. The second instruction needs the new value of D from the first instruction.
- **Conditional hazards** arise at a conditional branch. The next instruction to be loaded into the pipeline will depend on whether or not the branch occurs.
  - E.g. after `M;JEQ`, the next instruction might be at either ROM[$PC + 1$] or ROM[A].
- **Structural hazards** arise when two instructions need access to the same physical resource, e.g. both using the same parts of the ALU. (These would not be an issue in creating a pipeline for Hack.)

Any of these hazards can lead to a **stall** (a.k.a. **bubble**), where the rest of the pipeline stops moving until the offending instruction completes.

Avoiding and mitigating stalls is a complex part of modern CPU design.

What about multiple cores?

A CPU **core** is a self-contained circuit capable of executing machine code instructions. The Hack CPU consists entirely of one core. Modern desktop CPUs often have 4–16 cores. Modern GPUs have thousands.

What about multiple cores?

A CPU **core** is a self-contained circuit capable of executing machine code instructions. The Hack CPU consists entirely of one core. Modern desktop CPUs often have 4–16 cores. Modern GPUs have thousands.

With multiple cores, all these problems get much worse!

Source: Imgur (here)

Most benefits come from either the operating system dividing cores between applications, or from the developer writing explicitly parallel code.

Memory caching

AMD's Zen 4 microarchitecture (used in Ryzen 7000 series CPUs) has maximum clock speed 5.7GHz, i.e. 5.7×10^9 CPU cycles per second.

That's roughly 0.175ns per cycle.

How long does it take to retrieve a value from memory (the **latency**)?

Memory caching

AMD's Zen 4 microarchitecture (used in Ryzen 7000 series CPUs) has maximum clock speed 5.7GHz, i.e. 5.7×10^9 CPU cycles per second.

That's roughly 0.175ns per cycle.

How long does it take to retrieve a value from memory (the **latency**)?

Roughly 73.3ns, or 418 cycles. 😱😱😱

We don't want to do this (at least) once per fetch-execute cycle!
Why is it so slow?

Memory caching

AMD's Zen 4 microarchitecture (used in Ryzen 7000 series CPUs) has maximum clock speed 5.7GHz, i.e. 5.7×10^9 CPU cycles per second.

That's roughly 0.175ns per cycle.

How long does it take to retrieve a value from memory (the **latency**)?

Roughly 73.3ns, or 418 cycles. 😱😱😱

We don't want to do this (at least) once per fetch-execute cycle!
Why is it so slow?

- A 16+GB address space means more complex electronics and therefore more propagation delay.
- 16+GB of SRAM would be prohibitively expensive, so we use DRAM.
- Physical distance from the chip! (Electricity moves at light speed, which is roughly 3×10^8m/s, so 3.33ns to move one metre...)

Modern CPUs address this with smaller, faster, closer **memory caches**.

How caching works

Modern CPUs use three caches: L1, L2, and L3. For Zen 4 (roughly):

Data location	Capacity	Cycles to access	Time to access
Registers	1.8KB/core[1]	1	0.175ns
L1 cache	64KB/core	4	0.7ns
L2 cache	1MB/core	14	2.45ns
L3 cache	4MB/core	50	8.75ns
DDR5 RAM	16–64GB	418	73ns
Fast SSD	0.5–4TB	140,000	25μs
Fast HDD	0.5–4TB	28,500,000	5ms

Instructions are loaded into the L1 cache in advance. Where possible, at conditional branches, we "look ahead" and load both possible options.

Otherwise, the most frequently used data (e.g. memory used for common variables) goes to the L1 cache, then L2, then L3, then main memory.

[1] This is for the 224 integer registers — there are a few hundred others as well.

How caching works

Modern CPUs use three caches: L1, L2, and L3. For Zen 4 (roughly):

Data location	Capacity	Cycles to access	Time to access
Registers	1.8KB/core[1]	1	0.175ns
L1 cache	64KB/core	4	0.7ns
L2 cache	1MB/core	14	2.45ns
L3 cache	4MB/core	50	8.75ns
DDR5 RAM	16–64GB	418	73ns
Fast SSD	0.5–4TB	140,000	25μs
Fast HDD	0.5–4TB	28,500,000	5ms

Instructions are loaded into the L1 cache in advance. Where possible, at conditional branches, we "look ahead" and load both possible options.

Otherwise, the most frequently used data (e.g. memory used for common variables) goes to the L1 cache, then L2, then L3, then main memory.

Minimising the number of accesses to main memory ("**cache misses**") is a very important part of both CPU design and code optimisation.

[1] This is for the 224 integer registers — there are a few hundred others as well.

Applications of architecture: Predication

Understanding pipelining tells us ifs and loops require branches, which create control hazards and are disproportionately slow.

But evaluating a logical expression to 0 or 1 doesn't create a control hazard. It might create a data hazard, but it's still faster overall. E.g.:

```
if (x <= 50) {
    y += 10;
} else {
    y = 3;
}
```

becomes

```
y = (y+10)*(x <= 50) + 3*(x > 50);
```

This technique is called **predication** or **branchless programming**.

Applications of architecture: Predication

Understanding pipelining tells us ifs and loops require branches, which create control hazards and are disproportionately slow.

But evaluating a logical expression to 0 or 1 doesn't create a control hazard. It might create a data hazard, but it's still faster overall. E.g.:

```
if (x <= 50) {
    y += 10;
} else {
    y = 3;
}
```

becomes

```
y = (y+10)*(x <= 50) + 3*(x > 50);
```

This technique is called **predication** or **branchless programming**.

As with inserting assembly fragments, it's *only* a good idea for code you already know is a performance bottleneck. Otherwise it's not worth the bugs, the time spent, or the loss of clarity and maintainability.

(Modern compilers are also smart, so you may do more harm than good!)

Applications of architecture: Loop unrolling

Loop unrolling is the practice of writing out several iterations of a simple loop by hand to minimise the number of branches. For example:

```c
// Copy the first [count] entries of [from] to [to].
int broke_array_copy(int from[], int to[], int count)
{
    for(int i=0; i<=count; i++)
        to[i] = from[i];
}
```

becomes

```c
// Copy the first [count] entries of [from] to [to].
int woke_array_copy(int from[], int to[], int count)
{
    while (count >= 8) {
        *to = *from;
        to += 1;
        from += 1;
        // Repeat these three lines 6 more times
        *to = *from;
        to += 1;
        from += 1;
        count -= 8;
    }
    while (count > 0) {
        *to = *from;
        to += 1;
        from += 1;
        count -= 1;
    }
}
```

Say $\texttt{count} = 500$. Then the left code will use $\texttt{count} + 1 = 501$ conditional branches. The right code will only use $(1 + \texttt{count}/8) + (1 + \texttt{count}\,\%8) = 68$. (We've also saved 500 addition operations by removing i.)

Applications of architecture: Loop unrolling

Loop unrolling is the practice of writing out several iterations of a simple loop by hand to minimise the number of branches. For a scary example:

```
// Copy the first [count] entries of [from] to [to].
int bespoke_array_copy(int from[], int to[], int count)
{
    register int n = (count + 7) / 8;
    switch (count % 8)
    {
        case 0:                 do {  *to++ = *from++;
        case 7:                       *to++ = *from++;
        case 6:                       *to++ = *from++;
        case 5:                       *to++ = *from++;
        case 4:                       *to++ = *from++;
        case 3:                       *to++ = *from++;
        case 2:                       *to++ = *from++;
        case 1:                       *to++ = *from++;
                                  } while (--n > 0);
    }
}
```

Applications of architecture: Loop unrolling

Loop unrolling is the practice of writing out several iterations of a simple loop by hand to minimise the number of branches. For a scary example:

```
// Copy the first [count] entries of [from] to [to].
int bespoke_array_copy(int from[], int to[], int count)
{
    register int n = (count + 7) / 8;
    switch (count % 8)
    {
        case 0:                do {  *to++ = *from++;
        case 7:                      *to++ = *from++;
        case 6:                      *to++ = *from++;
        case 5:                      *to++ = *from++;
        case 4:                      *to++ = *from++;
        case 3:                      *to++ = *from++;
        case 2:                      *to++ = *from++;
        case 1:                      *to++ = *from++;
                                  } while (--n > 0);
    }
}
```

The code above is (a version of) **Duff's device**. Its creator, Tom Duff, said:

"Many people (even [Brian Kernighan]?) have said that the worst feature of C is that switches don't break automatically before each case label. This code forms some sort of argument in that debate, but I'm not sure whether it's for or against."