Computer Architecture Toolkit

Article 3: Estimating Performance

Pete Wilson

Version 1.01 • Mar 17, 2021

1. Performance

The simpleADL toolkit allows us to define an architecture (within annoying limitations), and tow write and execute programs for those architectures.

We can compare these architectures. We could compare their code footprints - how big an equivalent program is for each architecture. We could compare the number of instructions executed by each architecture for equivalent programs. We could compare the number of memory accesses made by each architecture.

Other things being equal, we might fall in love with the architecture with the smallest code footprint (memory costs money), and minimal instructions executed (fewer instructions executed surely translates to quicker execution time) and the least memory accesses (memory access consume energy). But this would be a bad decision, since what we want from an architecture is good implementations. And these have costs not exposed by the simple measures discussed here. For example, maybe we get minimal executed instruction counts by having rather complicated instructions which might, in a real implementation, require a lot more silicon, or make effective pipelining impossible, or...

So we'd like to measure implementations of our architectures. Now, in the absence of PowerPoint™-to-silicon converter technology, we will be limited in how confident we can be in estimating silicon cost, and in estimating how long certain operations (like multiply and divide) might take.

But we can get useful information by using knowledge accumulated from years of industrial practice. We know, for example, that we can make a 64-bit RISC processor[1] with 5-7 pipeline stages in 28nm CMOS and have the thing run at about a gigahertz (or more) and be somewhere between 0.05 and 0.1 sq mm (without memory).

In the same technology, we know we can get arrays of SRAM of a density that we can fit 200-400KB of SRAM into 1 sq mm (for L2 cache performance arrays)[2]. This is a shocking number - memory is very expensive in area compared to a processor - assuming the 400KB/sq mm density, we can get more than 20 64 bit cores in the same area as a megabyte of memory.

If we build processors 'similar to' RISC V - that is, RISC processors, single issue, in-order execution using a pipelined implementation - then we can reckon that they'll all be more or less the same size, and that any differences will be swamped by the need for lots of memory.

So we can compare implementations without doing the real silicon work, to some degree of accuracy (but we must be very careful if our comparisons lead us to want to proclaim our new Architecture X as head and shoulder better than anything else - if that's what the results show, then a very careful examination of the assumptions must be made).

Even with these caveats, we'd like to compare performance differences while varying some parameters of the design. For example, what happens if load instructions have a latency of 2 or 3 clocks instead of 1? What's the effect of *this* branch predictor vs *that* one? What happens if we have an 8-way set-associative cache rather than a direct-mapped one? What happens if we add an L2?

The right way of doing this is to write the Verilog, confirm it's correct, synthesise, and run on a clock-accurate gate level simulator. But we don't have those. So what *useful approximations* can we make use of?

Before addressing that, let's look at how processors are implemented.

2. Processor implementation

When processors were horribly simple, because transistors (or, more accurately, valves[3]) were expensive so you couldn't use very many, processor designs tended to be ad-hoc implementations. There was much less structure than we'd expect today, and generally the resulting computer executed one instruction completely - reading its operands, executing the operation, writing any results back - and then moved on to the next one.

Back then, computer architecture was, for better or for worse, not a settled field of achievement in which we *knew* what worked well and what worked less well and what was a really bad idea. So many architectures were created and fielded. Making these things out of custom-to-the model hardware designs was expensive and complicated and prone to errors.

The invention of *horizontal microcode* changed this. This introduced hierarchy into the computer implementation. While the programmers saw the advertised instruction set architecture, the hardware was rather different. At the lowest level, a very simple machine executed *microcode*. A microcode instruction was of fixed length - often a hundred bits or more - and had multiple fields which controlled a number of hardware black boxes which operated

1. The RISC-V Rocket implementation.

2. E6500: FREESCALE'S LOW-POWER, HIGH-PERFORMANCE MULTITHREADED EMBEDDED PROCESSOR,

David Burgess, Edmund Gieske, James Holt,, Thomas Hoy Gary Whisenhunt, IEEE Micro September/October 2012

3. "Tubes", to those of an USAian bent in parallel, and performed extremely low-level functions. The operations might be 'read register 7 onto internal bus B', or 'capture the values from internal buses A and C and add the results together, leaving it in register 17'.

This way, the chunks of hardware could be designed (mostly) once and re-used; and you could build a faster or slower machine by building a wider or a narrower microcode engine. And you could re-use the base hardware (given enough microcode memory) to implement many different architectures through simply re-programming the beast. And you could get higher performance than the prior approach (at least, under certain circumstances) because there was a great deal of controllable concurrency available. So in one microcode instruction, you could (in principle) working on various portions of the execution of several different instructions in any given microinstruction. Meanwhile, the design of the machine was constrained so that each of the parallel operations took just one clock (you could - obviously - introduce a 'not done yet' black box that would cause the micro engine to not proceed to the next instruction until all the components of this microinstruction were completed, but it was simpler and cheaper to stick to one operation per clock).

These machines (now that LSI was available) were constructed from many separate semiconductor chips, so that a simple processor - sans memory and I/O - could well occupy a complete circuit board (and complicated machines were multi-board things).

One well-documented example of this approach is the Stanford Emmy. This research machine was intended to be used to experiment with ISA's, and it's well-described in Technical Report 118[4] by Flynn, Hoevel and Neuhauser. This design was later commercialised by Palyn Laboratories, and was licensed to Britain's International Computers Ltd as a basis for a mid-range computer.

Over time, folk started wondering about the usage of instructions. It turns out that to write large programs, you really need a good high level language, which means that you need a good compiler. At the time, computer capacity and performance limited just how clever even the best compilers could be, and when folk looked at what really got executed they discovered that the compilers rarely were able to make use of the cunning and exotic instructions engineered into the extant architectures, instead concentrating on the simpler instructions (like register-register operations, and loading and storing values between memory and the registers).

Well, that sounds like a useful simplification opportunity. The story is told in Sequin and Patterson's DESIGN AND IMPLEMENTATION OF RISC I[5]. In short, they designed a machine which would fit on a single chip and outperform other designs of the day because it had instructions which were simple to decode, could be pipelined, and could be clocked at a relatively high frequency (even when implemented by students).

The pipelining was the way they got higher performance. Pipelining breaks down the execution of an instruction into separate steps - fetch it from memory, decode it (oh, it's an add) so you know what it needs, fetch its operands (r1 and r5), do the operation requested (add), write the result into the specified register. It arranges these operations in a sort of bucket brigade - each operation reads values from its prior stage at the beginning of a clock, performs its task, and then writes its result just before the next clock occurs.

If you ignore the highly-structured inter-stage data passing, we could build an almost-equivalent machine which did these stages sequentially (in the classic model of fetching and executing instructions), and then went on to the next instruction. Such a machine would take about 5 clocks to execute an add (because that's how long it takes to do the five operations in the pipeline). We might be able to get it down to 3 or 4 clocks in such a machine - because some operations are naturally simpler and thus take less time than others - , but at a given clock rate, the classic machine will need multiple clocks to execute an instruction, and won't start the next until the current one is finished.

The pipelined machine, by comparison, can start a new instruction every clock. Effectively, it's 3-5 times as fast, but hardly any bigger. A measure of performance here is "instructions per clock", or ipc. The classic machine achieves around 0.3 - 0.25 ipc (ignoring the effects of the memory system) while our RISC implementation manages 1.0 ipc (again, ignoring the memory system).

The standard texts have a lot more information on the desirability and the problems with these sorts of pipelines, so we won't go any deeper here.

2.1. Approximating a RISC machine's performance

4. Digital System Laboratory: Stanford Electronics Laboratories, Technical Report No. 118, June 1976, THE STANFORD EMULATION LABORATORY, by Michael J. Flynn, Lee W. Hoevel and Charles J. Neuhauser. Available at time of writing from http://i.stanford.edu/pub/cstr/reports/csl/tr/76/118/CSL-TR-76-118.pdf. Related documents exist, including https://www.computer.org/csdl/proceedings/afips/1975/5083/00/50830085.pdf

5. DESIGN AND IMPLEMENTATION OF RISC I, Proc. Advanced Course an VLSI Architecture

University of Bristol, England, July 19-30, 1982. Available at the time of writing at http://digitalassets.lib.berkeley.edu/techreports/ucb/text/CSD-82-106.pdf

To create a timing accurate model of a RISC processor is simple in principle. Create a simulation environment in which you can describe a stage, with its input latches, and its action, and it's writing of the the results to the input latches of the next stage, write code for each stage, hook 'em up appropriately, and let 'er rip.

Unfortunately, these pipelines have feedback paths (think of the instruction *bne r1, r2, destinationoffset*: This compares the values of r1 and r2, and if they're not equal, branches somewhere else *destinationoffset* words away. By the time this instruction is executed, the pipeline will already have fetched and decoded and read the operands for more instructions. How does it know what instructions to fetch? Well, it can't *know*, because the comparison hasn't happened yet.

One thing you can do is to simply stop fetching instructions the moment that you see a conditional branch instruction. Then the fetch stage waits until it gets one of two signals from the execute stage - one signal says 'continue', and the other says 'ok, start fetching form this address'. It will need to wait a few clocks to receive this signal - two or three, depending on the aggressiveness of the implementation.

This works, and is simple. But on average there's a conditional branch roughly every 7 instructions in ordinary code. If every branch makes the machine wait three clocks, we're going to need 10 clocks to execute 7 instructions - down from 1.0 ipc to 0.7ipc.

Another thing you can do is to architect branches in such a manner that the machine *always* executes the instruction following the branch. Then the fetch unit can always fetch the next instruction after the branch. In the architecture we were just discussing, this would make the machine run at 0.8 ipc - nearly 15% more performance for the cost on one more register and a few gates. But this only works if you can nearly always find an instruction that does useful work to slip in behind the branch (a no-op works fine architecturally, but doesn't get much useful work done.

You can try to fix this with cunning. For example, you can have a magic bit which is set by *compare* instructions. With a bit of luck, you can do the comparison a few instructions before you need to do the conditional branch. If you arrange for the bit to be held in the fetch stage, then by the time you get to the conditional branch, you can do the change of flow *in the fetch stage*, with no need to wait for the comparison to be done. If you can *hoist* the compare operations far enough, you can make the branch penalty (of several clocks) go away most of the time.

But can't the fetcher get confused? How does it know whether the magic bit has actually been set?

To answer that, consider a multiply instruction. Multipliers are bigger and slower than adders. If you make your RISC run every stage at one operation per clock, your clock rate will be markedly slowed. To avoid this, you pipleline the multiplier. It accepts a new pair of operands every clock, but takes (say) four clocks to produce a result.

Thus, if you don't do anything about it, an instruction sequence like

```
r1 = r3 * r4;
r2 = r1 + r6;
```

Will put the wrong value into r2, because the result from the multiply will take several clocks to get written into r1.

We can fix this problem quite simply. Add a valid bit to every register. When reading operands for any instruction, *stall* the pipe stage until all the operands are valid, and set the destination operand to invalid.

Stalling a stage means that it repeatedly tries the operation without producing any new results and without reading anything from its inputs. It's a busy-wait in hardware.

When the machine writes the result of the multiply into the register, it also sets the valid bit for that register. With this scheme, our code works correctly.

So to make our branchery work, we just add a valid bit to the magic bit. The fetch stage sees that its's just got a compare instruction and so it makes the magic bit invalid. Later in the pipeline the compare happens and the value is sent back to the magic bit, which also has its valid bit set. When the fetch stage sees a conditional branch, it waits (if necessary) until the magic bit is valid, and then does the right thing.

Now reading and writing the register file *should* take less time than doing an add. So we can fine-tune our pipe by being slightly cunning:

- Fetch takes a clock
- Decode happens in half a clock
- Register read happens in half a clock
- Operate takes a clock
- Register write takes half a clock

If we can live with this (it has implications for clock rate), we only need to mess with valid bits on the register file when we have multi-cycle instructions, because we read the register file in the last half of a clock and write it in the first half. So a sequence like:

r1 = r3 + r4;

r2 = r1 + r6;

Has written the result to r1 in the half clock while the second instruction is being decoded, and before the register is read.

It also means that branches have an inherent penalty of one clock if nothing is done, so we don't need to hoist our compare instruction too far.

But to return to the simulation. If we build an 'exact model' of the pipeline, we need to get every handshake between the stages exactly correct, or the simulation won't operate correctly. When you're faffing with architecture and microarchitecture, you need useful results quickly and with a lot less labour than essentially implementing the whole processor in C.

So is there a useful approximation available?

It looks like there is. We can be inspired by the valid bits on the registers.

Let's take our interpreter for the r32 architecture. It has code looking like this, after the instruction has been fetched:

```
switch (fmt) {
    default:
        stop = unimplementedInstruc(iptr, instruction);
    break;

    case 2:      {
            switch (op12) {
            default:
                stop = unimplementedInstruc(iptr, instruction);
                break;

                case 0:      {        // halt
                    if (traceIt) printf("\n\thalt();");
                    stop = 1;
                    }
                    break;

                case 1:      {        // add
                    if (traceIt) printf("\n\tadd();");
                    // assembler instruction definition
                    r[rd] = r[rs1] + r[rs2];      // action when executed
                    }
                    break;

                case 2:      {        // sub
                    if (traceIt) printf("\n\tsub();");
                    r[rd] = r[rs1] - r[rs2];
                    }
                    break;

                case 3:      {        // mul
                    if (traceIt) printf("\n\tmul();");
                    r[rd] = r[rs1] * r[rs2];
                    }
                    break;
```

Well, for the add and sub instructions, we know that if the current time is *now*, then the result of the operation will be written at *now + 1*. And we know that each instruction takes one clock to kick off, so we can start playing with the next instruction at *now + 1*.

With multiply, though, we can start the next instruction at *now + 1*, but the destination register won't get written until *now + 4* (we have a 4-cycle pipelined multiply).

It looks as though we can get *exactly* the same timing information if we 'tag' every resource with the time at which it will be valid, and then for each instruction compute now to be the maximum of the computed now and the validity times for all the operands. And on every instruction, update the time tag for any resource that's written to to be now plus latency of the operation.

This is remarkably easy to do.

We extend the Context definition from

```
    typedef struct {
        uint32 r[32];
        uint32 iptr;
        uint8 stop;
        uint32 instruction;
    } Context;
```

To be

```
    typedef struct {
        uint32 r[32];
        uint64 tt_r[32];
        uint32 iptr;
        uint64 tt_iptr;
        uint8 stop;
        uint64 tt_stop;
        uint32 instruction;
        uint64 now;
    } Context;
```

That is, we have a variable or an array of variables whose name is tt_XXX, where XXX is the name of the register or register file it's related to. We keep validity times in the tt_XXX variables.

Now we change the interpreter code in the obvious manner:

```
                case 2:      {
                        switch (op12) {
                            default:
                                stop = unimplementedInstruc(iptr, instruction);
                                now++;
                                break;

                            case 0:      {        // halt
                                if (traceIt) printf("\n\thalt():\n\tnow = %llu", now);
                                stop = 1;
                                now++;
                                }
                                break;

                            case 1:      {        // add
                                if (traceIt) printf("\n\tadd():\n\tnow = %llu", now);
                                // assembler instruction definition
                                r[rd] = r[rs1] + r[rs2];      // action when executed
                                now = 1 + u64max4(now, tt_r[rd], tt_r[rs1], tt_r[rs2]);
                                tt_r[rd] = now + l_simpleOp;
                                }
                                break;

                            case 2:      {        // sub
                                if (traceIt) printf("\n\tsub():\n\tnow = %llu", now);
                                r[rd] = r[rs1] - r[rs2];
                                now = 1 + u64max4(now, tt_r[rd], tt_r[rs1], tt_r[rs2]);
                                tt_r[rd] = now + l_simpleOp;
                                }
                                break;

                            case 3:      {        // mul
                                if (traceIt) printf("\n\tmul():\n\tnow = %llu", now);
                                r[rd] = r[rs1] * r[rs2];
                                now = 1 + u64max4(now, tt_r[rd], tt_r[rs1], tt_r[rs2]);
                                tt_r[rd] = now + l_mulOp;
                                }
                                break;
```

Where the functions *u64maxN()* compute the maximum value of N passed arguments. We always update *now* to be the latest time of all resources (and the prior value of now), and add 1 so time increments with single-clock instructions, and then update the destination's timetag with *now + the latency of the operation*. Done this way, single-cycle operations have a latency ((*l_simpleOp*))of 0.

This approach is horribly simple, and horribly quick, having little effect on the performance of the interpreter - probably because on a modern x86, the time tagging can be done in parallel with the instruction computation and the various computed branches done in the interpreter. As described, it's not quite complete - multiply is pipelined, but divide isn't. So we really need a time tag for function units which are incompletely pipelined - this isn't hard to do. And it's simple to see how we might have simpleADL generate a timetagged interpreter automatically (probably generate just one interpreter, with a compile-time #define turning on or off the tagging).

Hand-editing the r32 interpreter to provide the tagging for every operation, and providing latencies as #defines:

```
    // operation latencies - clocks to add to the newly-computed now
    #define l_simpleOp        (0)
    #define l_memreadOp       (2)
    #define l_branchOp        (2)
    #define l_mulOp           (4)
    #define l_divOp           (30)
    #define l_ioOp            (0)
```

Results in the label.asm program running (same 'binary', of course):

```
..executed 3275 instructions in 289 microseconds = 11 MIPS.
..3275 instructions in 5339 clocks -> 0.613 ipc
Done.
```

Now, we don't do much memory accessing in this program; there's just a loop repeated 32 times which reads characters and spits them out at the end of the program:

```
        cpyc r5, @datalabel;  // point at the data with r5
        //cpyc r5, 0x7c;
        cpyc r1, 0;
        cpyc r2, 32;
        outcharc '\n';
        outhex r5;                // output data start address
        outcharc '\n';
    [counting]
        ld8 r3, r5, r1;           // read the byte in mem
        outhex r1;                // output the address
        outcharc '\t';            // space
        outhex r3;                // the value
        outcharc '\n';            // newline
        addc r1, r1, 1;           // next character
        sub r4, r2, r1;           // compare
        bne r4, counting;     // branch if not finished

        halt;

    dataseg
    [otherlabel]
    [datalabel]

        d8 1 2 3 4 5 6 7 8 9 10;
        d16 0x1111 0x222 0x3333 0x4444 0x5555 0x6666;
        d32 0x12345678 0x4567890 0x98765 0x77 0x1235;

    [lastlabel]
```

The loop is between *[counting]* and the *halt* instruction., and consists of 8 instructions. The first use of the register loaded from memory (that is, r3) comes 3 instructions later, so we should get essentially the same performance if we set *l_memreadOp* to 1 or 2, and see a slight slowdown when we set it to 3:

Set to 1:

```
..3275 instructions in 5339 clocks -> 0.613 ipc
```

Set to 2 - same ipc as with a shorter memory cycle, because there's time for the value to get to the register before we need to use it:

```
..3275 instructions in 5339 clocks -> 0.613 ipc
```

Set to 3 - a bit slower, because we need to wait a clock to use r3:

```
..3275 instructions in 5371 clocks -> 0.610 ipc
```

Let's calculate how much worse it should be if memory reads took 100 cycles. Then each iteration of the loop would take an extra 98 clocks, so the overall run time of the program would be 5339 + (98 * 32), which is 8475, for an ipc of 3275/8475 = 0.386 ipc

Let's check:

```
..3275 instructions in 8475 clocks -> 0.386 ipc
```

Yes; we got it right.

But of course all this is ignoring the elephant in the room. Instructions come from memory, too. So if memory has an effective access time of 100 clocks, we'll get an instruction only every 100 clocks (unless there's some cunning buffering, whereby you read 256 bits from memory each access into a buffer, and fetch the instructions from the buffer, giving you somewhat greater performance).

In short, to get performance even close to what the cycle time of the processor logic can support, we need to have instructions and data fetched - at least most of the time - from very fast memory, which means it's a small memory, and close to the processor.

3. Caches

So we can see that real processors have their performance dominated by their memory systems. It's easy to get processors to run at 1-2GHz. It's easy to write programs that need megabytes of memory. It's easy to supply that memory with cheap DRAM.

But the DRAM is much, much, slower than the processor.

Since load instructions are about 20% of the executed instructions in a RISC[6], a DRAM with an effective access time of 100 clocks really isn't going to do much for performance. In every 100 instructions, we'll have 80 instructions which don't read memory, and 20 that do. So those 100 instructions will take about 80 + (20 * 100) clocks, or 2080 clocks, for a staggering 0.05 ipc.

So two things are clear - we need to do something to vastly improve the access time of at least *most* memory reads; and suddenly approximate processor models don't seem too much of a concern.

To fix memory access time statistically, we take advantage of program locality - we can observe that in the real world, programs tend to re-use quite soon code and data they've just used. We can make use of this with a *cache*.

The first thing to know is that small chunks of SRAM (a few kilobytes) are faster than large chunks (a few megabytes) and the large chunks are much faster than the commercial DRAM that's used for the main memory of practically all computer systems today (cunning new technologies promise much improvements in the near future, but even then, it takes time and energy to get offchip to a few gigabytes of memory, so the problem while improving isn't likely to go away.)

So the obvious thing to do is to write your software so that the first thing it does is to copy the 'interesting functions' - the ones that get executed a lot - into some SRAM which is local to your processor - right next door, at way less than a picosecond of travel. Then you arrange to execute the code that's been copied into the SRAM rather than the (same) code that lives in the DRAM. In practice, there won't be enough SRAM, so rather than doing this once, you'll need to do it repeatedly.

In the good old days of early mainframes, this approach was known as *overlays*, and there were toolkits to try to help you manage it. (To be honest, the code existed on a drum or a disk, not in DRAM, and what you did was to arrange to copy stuff off the spinning storage into the computer's main memory; but the principle is the same).

But data can be even bigger than code, and so in general you have to be prepared to do the same thing with data. This is feasible but suffers the same problem as the code overlays - it's a lot more code to write; you generally have to write your code in a weird manner, it's more to get wrong; you'll make more errors.

And it's worse than that for data. Suppose, rather than some giant image, you have actual data structures - ones with pointers in. When you copy the data from DRAM into your SRAM, the pointers will still point into DRAM (because you've *copied* the data). Suddenly all your work is for naught.

So just as we *can* do multiplication with simple single cycle instructions and a loop, but we generally prefer to have a chunk of hardware to do the operation, perhaps we could have a chunk of hardware to manage the copying...

Indeed we can; that's what a *cache* is.

The basics are that a cache has two chunks of memory - a *data array* and a *tag array*.

The *data array* - holds the copied data, and the tag array holds *metadata* to allow us to manage moving data in and out of the data array. We'd like this management to be as simple and quick as possible.

6. Various sources will suggest 20-25% load instruction frequency, and 10-15% store instruction frequency.

When the cache is given a read request, it's obviously given the address of the data. Things will be simpler if we can organise things so that *if* there's a copy of that data in the cache, there's exactly one place that it could be. Then we can look to see if it's there. Looking to see if we have a copy must mean keeping track not only of the data, but the address of the data - we look in the obvious place and see if the address we have stored matches the address we're looking for.

First thing is that it's rather inefficient to keep a possibly quite long address for each byte of data. So we don't keep data as single bytes - we keep it in chunks. Typical size of a chunk is 64 bytes. Now we don't need the whole address - with 64 bytes per chunk, we can store a shrunken address, which is the address we were given shifted right 6 places (6 places because 64 is 2**6). Next question is how we organise the storing of these addresses.

If we have N data chunks, then we can quickly compute a unique place to look if we store the addresses as successive elements of an array of N elements. It's simplest if N is a power of two, because then we can find the element that must be holding our wanted address (if we have it) by simply masking off the bottom n bits of the shrunken address, where N = 2**n. This gives us a bit pattern which selects exactly one of our own elements. If we store the data chunks as a matching array, then we have a cache access algorithm which works like this when we have 2**c chunks of 2**l bytes each, with the addresses held in an array *addresses[]*

```
• Compute match_address = address >> l;
• Compute mask = (1 << c) - 1;
• Compute index = match_address & mask;
• Is addresses[index] == match_address?
    • If so, our data is in the data array, at a simple-to-calculate data offset:
        • Data_offset = address & ((1 << l) - 1);
    • If not, then the data isn't in the data array, and we must read main memory for the data and its
      surrounding bytes, update the data array and update the addresses[] array.
```

This brings up the need for a bit more information in the *addresses[]* array. The corresponding data chunk may never have had any data placed into it, so we need one extra bit to say whether the element holds a valid data address. It's pretty clear that over the execution of a program, we're very likely to have to re-use chunks and thus address[] elements. If we have changed the data in a chunk, by writing to it, we need to copy that data back to main memory when we re-use the chunk (but if we haven't written to it, there's no reason to write it back - we can just overwrite). To keep track of whether we've written to the data chunk, we need an extra bit which is initialised to zero and set to 1 every time we write to the data. So we have a little data structure, commonly called a *tag*, to hold the needed information, comprising

- The shifted address
- The validity bit v
- The written bit d

The written bit is generally called the *dirty* bit - if we write to the data, it's no longer a "clean" copy.

The data chunks are generally called *lines*. The number of bytes in a line is generally known as the *line length*.

This arrangement of a cache is called a *direct-mapped cache*. It has the advantage of simplicity, but the disadvantage that address *clashes* are possible. If we happen to be in a loop, reading from two arrays and writing to a third, we may be unlucky enough to find that given the size of our cache, the line in which we hold data from one array is by the inexorable logic of the lookup algorithm exactly the same as the line needed for one (or, worse, both) of the others. If we have 2**c lines in the cache, all we need is the bad luck to have corresponding array elements to be separated in memory by some multiple of 2**c - because then they'll all map to the same line. This results in extremely disappointing performance; the situation is known as *thrashing the cache*..

Other cache arrangements are known.

The *fully associative cache* has not only the tag described here, but an address comparator circuit for each tag. Then, given an address, the hardware can compare that address with all the tags in parallel. This means that a given address and its data can be stored anywhere in the cache, so that the problem of clashing addresses outline for the direct-mapped cache will not occur. But such a cache is expensive in hardware and in energy - all those comparators firing every clock costs energy -and in time - discovering that there was a hit and which line had it requires more logic, which takes time; plus the extra hardware makes the tag structure larger, and thus slower. Fully-associative caches are used in modern computers, but generally in rather small arrays, and generally for address translation rather than straightforward caches.

The *set-associative cache* is a compromise between the fully-associative and the direct-mapped cache. It's basically some number of direct-mapped caches operated in parallel - if we hook up *w* direct-mapped caches, we are said to have a *w-way set-associative cache*. Fairly clearly, such a cache can avoid the clashing problem of the direct-mapped cache, but we have now added one more piece of complexity - when we have to throw out data to make room for some new, which of the w ways do we choose? Various studies suggest that we're best off throwing away the data was last accessed the longest time ago - the *least recently used* replacement algorithm. But keeping an accurate timestamp in the tags uses bits and incurs a write cycle even when reading, so this isn't done. There are various pseudo-LRU algorithms available[7], but picking at random isn't too bad, and is very easy to do.

In a direct-mapped cache we have an option - since the tag array index is the same as the data array index, we can read out the data array in parallel with the tag array match operations - then, if we have a hit, the data is already read. This saves time, at the expense of one data line read that may not have been useful (if a miss)

In a set-associative cache, we can do the same thing (so in a 4-way set-associative cache, we do four lookups in four complete chunk reads in parallel), and then have to choose to read the value from the correct data chunk.

This approach provides minimal latency, but wastes power - in an n-way set associative cache, we waste n-1 data array accesses at best. In power-sensitive systems, therefore, some implementations choose to do the tag lookup first, then just one data array access(when there's a hit).

3.1. Adding caches to the performance estimator.

We obviously need to be able to include the effects of caches in our performance estimating model. One thing to note is that we don't have to model the whole cache, just the tag lookup. After determining whether we have a hit or a miss, we return a time-tagged value which we read directly from memory as usual. This reduces the performance impact usefully.

A model which uses caches would replace use of the *readMem()* and *writeMem()* functions by *readDcache()* and *writeDcache()* functions (plus *fetchIcache()* for instructions). These would make use of data structures mirroring the functionality of the hardware to perform the lookup operation, determine either that the access was a hit, and call *readMem()* to get a value from memory, returning it with its timetag showing a couple of clocks of latency; or that it was a miss, do the *readMem()* to get the value, and return that with a timetag representing main memory access time plus the time necessary to get through any system interconnect, and return the value with that timetag.

Since we're just accounting for time, not actually delaying operations, each chunk of data in the cache needs its own timetag - so that when we read a value, our *now* gets updated to the timetag of that cache chunk.

Interestingly, we don't need to actually model the cache's lines - avoiding copying the data from our simulated main memory into the cache saves execution time, and reading the data from main memory is at worst no slower than reading it from the simulated cache line. (Well, in principle, our cache model should offer better cache behaviour than simulated main memory accesses, but in practice we'll just cache the main memory lines…)

In principle, we should do this with main memory as well, but the cost of tagging every chunk of main memory is rather extreme. So we'll rely on the fact that every read from main memory results in the processor (one way or another) advancing its time into the future to cover the latency, and have just one timetag for the whole memory, set when a store arrives. We'll probably find that memory's timetag is always in the past.

We can write these cache functions fairly straightforwardly. Here's the declaration of a direct-mapped cache, and the read and write functions:

```
#define written_bit      (1)
#define valid_bit    (2)
// a tag element
typedef struct {
    uint64 ttag;
    uint32 match;
    uint32 bits;     // written_bit, valid_bit,...
} cacheTag;

// a cache
typedef struct {
    char * name;
    uint32 rhits;
    uint32 rmisses;
    uint32 whits;
    uint32 wmisses;
    uint32 reads;
    uint32 writes;
    uint32 rows;
    uint32 linelength;
    uint32 l2rows;
    uint32 l2linelength;
    uint32 l_cacheHit;
    uint32 l_cacheMiss;
```

7. Wikipedia has information: https://en.wikipedia.org/wiki/Cache_replacement_policies

```
    uint32 l_cacheWriteBack;
    cacheTag * tags;
} Cache;

// ----------------- newCache ----------------

Cache * newCache(char * name, uint32 l2rows, uint32 l2linelength, uint32 l_cacheHit, uint32 l_cacheMiss,
uint32 l_cacheWriteBack) {
    Cache * c = malloc(sizeof(Cache));
    if (c) {
        c -> name = strdup(name);
        c -> l2rows = l2rows;
        c -> l2linelength = l2linelength;
        c -> rows = 1 << l2rows;
        c -> linelength = 1 << l2linelength;
        c -> tags = malloc(c -> rows * sizeof(cacheTag));
        c -> l_cacheHit = l_cacheHit;
        c -> l_cacheMiss = l_cacheMiss;
        c -> l_cacheWriteBack = l_cacheWriteBack;
        c -> reads = 0;
        c -> writes = 0;
        c -> rhits = 0;
        c -> rmisses = 0;
        c -> whits = 0;
        c -> wmisses = 0;
        // initialise the tags
        for (int i = 0; i < c -> rows; i++) {
            c -> tags[i].ttag = 0;
            c -> tags[i].bits = 0;
            c -> tags[i].match = 0;
        }
        printf("\nCreated cache %s: %d rows %d bytes per line", c -> name, c -> rows, c -> linelength);
        }
    return c;
}

// ----------------- cacheRead ---------------

uint64 cacheRead(uint64 now, Cache * cache, uint32 address, Memory * mem, uint32 width, uint64 * result) {
    // we have 64 byte lines
    // the addresses are byte addresses
    // extract index and match
    uint32 indexmask = cache -> rows - 1;
    uint32 index = (address >> (cache -> l2linelength)) & indexmask;
    uint32 match = address >> (cache -> l2linelength);
    uint32 latency = 0;
    cacheTag * tag = &cache -> tags[index];
    int hit = 0;
    cache -> reads++;
    // look up the address
    if ((tag -> bits & valid_bit) && (tag -> match == match)) {
        // hit
        cache -> rhits++;
        latency = cache -> l_cacheHit;
        hit = 1;
        if (cacheTrace) printf("\n%s\tnow=%llu: index %d, addr 0x%x -> --hit--- [val=%d match = %d]",
                cache -> name, now, index, address, tag -> bits & valid_bit, tag -> match == match);
    }
    else {
        // miss
        cache -> rmisses++;
        if (cacheTrace) printf("\n%s\tnow=%llu: index %d, addr 0x%x -> **MISS** [val=%d match = %d]",
                cache -> name, now, index, address, tag -> bits & valid_bit, tag -> match == match);
        if (tag -> bits & written_bit) {
            // is the line dirty? If so, we need to write it back then read
```

```
                latency = cache -> l_cacheMiss + cache -> l_cacheWriteBack;
                memWriteBack(mem, cache -> linelength);
                hit = 1;
            }
            else {
                latency = cache -> l_cacheMiss;
            }
            memLineFill(mem, cache -> linelength);
            // we'll be filling in the tags, so make it valid and fill in the address
            tag -> bits = valid_bit;
            tag -> match = match;
            hit = 0;
        }
        // update the time
        uint64 t = u64max3(now, tag -> ttag, mem -> ttag);
        t += latency;
        tag -> ttag = t;
        *result = readMem(mem, width, address);
        return t;
    }
```

This code includes a variety of counters to provide insight into the cache behaviour when used in a simulation. For convenience of reporting, every cache is given a name, and contains counters for reads, read hits, read misses, writes, write hits and write misses. We construct a cache by calling *newCache()*, and passing it not the line length and number of lines, but *log2(linelength)* and *log2(number of lines)*; this way be can be sure to end up with power-of-two sizes so shifting and masking work. We keep the computed number of lines (as *rows*) and linelength in the structure. We also store the hit latency (in clocks), the miss latency and the latency needed to write back a dirty line (it would be better to keep just the overheads of these and rely on the latency of the main memory, but that's for later).

The tag structure is as described above, with the addition of the necessary timetag.

To allow us to also track main memory traffic, we make use of the *readMem()* and *writeMem()* functions to actually change main memory, but these functions don't update any main memory statistics. To do this, we use the functions *memLineFill()* and *memWriteBack()* - they simply update memory stats.

When we make the necessary changes in model() to use the cache models, we get output like the following when executing the (unchanged) labels.asm program:

```
architecture simulator 0.1v8 for 'r32_model 0.1v0'
default bin path '/Volumes/OxfordRoad/Users/pete/ArchProjDev/archModels/r32'
Going to load and execute file '/Volumes/OxfordRoad/Users/pete/ArchProjDev/archModels/r32/
programs/bin/label.ldr'
        loading software..

Created memory 'model memory' size 4096
        loaded; took 789 microsecs

Created cache icache: 16 rows 64 bytes per line
Created cache dcache: 8 rows 128 bytes per line
Run the software ..
0xa8
0x0     0x1
0x1     0x2
0x2     0x3
0x3     0x4
0x4     0x5
0x5     0x6
0x6     0x7
0x7     0x8
0x8     0x9
0x9     0xa
0xa     0x0
0xb     0x0
0xc     0x11
0xd     0x11
0xe     0x2
0xf     0x22
```

```
0x10   0x33
0x11   0x33
0x12   0x44
0x13   0x44
0x14   0x55
0x15   0x55
0x16   0x66
0x17   0x66
0x18   0x12
0x19   0x34
0x1a   0x56
0x1b   0x78
0x1c   0x4
0x1d   0x56
0x1e   0x78
0x1f   0x90

..executed 3275 instructions in 451 microseconds = 7 MIPS.
..3275 instructions in 5930 clocks -> 0.552 ipc

Cache 'icache':
       16 lines of 64 bytes
       reads  =   3275:   hitrate = 99.91%   hits =   3272  misses =      3
       writes =      0:   hitrate = 0.00%    hits =      0  misses =      0

Cache 'dcache':
       8 lines of 128 bytes
       reads  =     32:   hitrate = 96.88%   hits =     31  misses =      1
       writes =      0:   hitrate = 0.00%    hits =      0  misses =      0

Memory 'model memory':
       reads = 4:  bytes read =  320  avg read = 80  bandwidth = 0.05 bpc
       writes = 0: bytes written = 0  avg write = 0  bandwidth = 0.00 bpc
Done.
```

Cache sizes are set up in modelmain.c, in the invocations of newCache(). Different sizes do have an effect on performance:

```
..executed 3275 instructions in 414 microseconds = 7 MIPS.
..3275 instructions in 231643 clocks -> 0.014 ipc

Cache 'icache':
       2 lines of 4 bytes
       reads  =   3275:   hitrate = 30.50%   hits =    999  misses =   2276
       writes =      0:   hitrate = 0.00%    hits =      0  misses =      0

Cache 'dcache':
       2 lines of 4 bytes
       reads  =     32:   hitrate = 75.00%   hits =     24  misses =      8
       writes =      0:   hitrate = 0.00%    hits =      0  misses =      0

Memory 'model memory':
       reads = 2284:  bytes read =  9136 avg read = 4   bandwidth = 0.04 bpc
       writes = 0: bytes written = 0  avg write = 0  bandwidth = 0.00 bpc
```

Looking at the code in model.c it's pretty clear that the time-tagging would be pretty straightforward to generate from a tool like simpleADL. We'd want to have a .impl file which somehow referenced the appropriate .adl file, and we'd need to provide cache organisation and instruction latency information, and the tool would be able to create the time-tagged, cached, model. But we're not doing that in simpleADL - that'll come in a later version. Meanwhile, we've added the time-tagged model as *simpleTTModel* to the *archTools* directory.