Comparch II - Part 2 Notes

Ethan Graham

20th of June 2023

Pipelining

Basic Idea

Idea: Partition the circuit into stages using registers such that the functionality isn't affected, and so we can increase performance using intermediate results.

Latency λ is increased, but so is throughput ϕ.

$\lambda_{pipe} = N \cdot max_{i=0,...N-1}(T_i + T_{FF}) = N \cdot T_{CLK,pipe} = \frac{N}{f_{pipe}} > \lambda_{orig}$

$\phi_{pipe} = \frac{1}{max_{i=0,...,N-1}(T_i, T_{FF})} = f_{pipe}$

Pipelinling for Processors

Pipelining is only useful if we split logically. For a processor it makes sense to split, for instance, into stages **fetch, decode, execute, etc...**

This introduces **instruction level parallelism** to the processor.

All pipeline registers contain `nop` value initially.

Data Hazards

- **RAW:** Read after write
- **WAR:** Write after read
- **WAW:** Write after write

In this implementation of pipelining, we stall the pipeline to avoid hazards. For example

```
addi   $r0, $r0, 1
sub   $r2, $r0, $r1
```

Contains a dependency. In this case, we stall the program. This roughly translates to introducing a `nop` for as many cycles as necessary.

We can do better by introducing **forwarding paths** between different stages of the pipelined processor.

Structural Hazards

This happens when different instructions compete for the same resource. This cannot happen in the simple pipeline introduced in the course.

Control Hazards

This is when we try to fetch an instruction before knowing which one.

Branch Prediction

Try and guess the outcome of the branch and fetch the corresponding instruction. If the guess is correct, no cycles are lost and if it is incorrect it is *squashed* This has made architectures with delay slots (`nop`) extremely rare.

to summarize the solutions for this first section:

- **Data Hazard:** forwarding paths whenever possible, otherwise stall
- **Control Hazard:** Delay slots if architecture allows it, branch prediction, stalls if not.
- **Structural Hazards:** Rigid pipelines which cannot have them by construction, stalls otherwise.

refs: COD 5th edition, sections 4.5, 4.6, 4.7, 4.8, 4.13

Dynamic Scheduling

The idea

Lets say that we have a program with dependencies in some areas. For example the following:

```
divd $f0, $f2, $f4
addd $f10, $f0, $f8
subd $f12, $f8, $f14
```

We can see that there is a depdency on `line 2` for register `$f0`, so we are unable to execute it. **idea:** Why not execute `line 3` while we wait for `line 1` to finish and save some time?

In doing this, we need to relax a fundamental rule. ***instructions can be executed out of program order***.

Breaking the Rigidity of Basic Pipelining

The solution that we reach is to split the different tasks into independent units/pipelines. - Fetch and Decode - Execute - Writeback (write)

We can also make different *lanes* for different instructions (for example ALU instructions floating point instructions). We put `RS`(reservation station) between

before execution stages. We put `ROB` (re-order buffer) before the writeback stage so that everything is written back in correct program order.

We need to solve the following problems with a dynamically scheduled processor: - structural hazards (previously handled by rigid pipelining) - RAW data hazards (are operands ready to start execution?) - WAW and WAR Data Hazards (did the new data overwrite something that was needed?)

We note that WAW was impossible before. WAR often cannot occur.

Reservation Stations

This unit checks that operands are available (RAW) and that the **Execution Unit** is free (structural hazard), then starts execution.

They hold **instructions and their related operands** enabling them to be executed as soon as they become available.

- Operations that are fetched and decoded are sent to reservation station for corresponding unit (e.g. ALU, Floating Point)
- RS checks if operands are available. If they aren't, we wait
- Once all operands are available, instruction is issues to functional unit
- Instruction is executed using instructions fetched from reservation station
- If instructions require operands that are being computed by other instructions, forwarding mechanisms are in place to make this mechanism more efficient
- Once instruction has been executed and the result has been written back, we retire the instruction from the reservation station (frees it to leave more space for more)

We split a reservation station into lines. Lines are composed as follows: `OP tag1 tag2 arg1 arg2`

If one of the two args isn't computed yet, we wait. Tags say which unit the required missing operands are coming from (tag 1 and tag 2 respectively)

Example: We have the following line in an ALU RS `subd ALU3 - ??? 0xFFFFFEE1` This tells us that we want to compute a `subd`. `op1` is unavailable, and we know that it will come from `ALU3`. We provide no `tag2` because `op2` is already available at address `0xFFFFFEE1`.

The reservation station will loop around and when it sees that `ALU3`has finished and produced `op1`, we will send the full instruction to the execution unit.

`ALU3` corresponds to the third instruction in the reservation station for this ALU!!! `MUL3`corresponds to the third instruction for the multiplier reservation station (not this RS)

Dynamic Pipelines may create WAW hazards

Register Renaming

Eliminates false dependencies between registers. For example, if I do two operations using regiser `$r0` when they are totally independent. We could simply rename the register in this case and execute them with instruction-level parallelism to speed things up.

This is done implicitly by the processor. The programmer needn't have any knowledge of this.

We structure our processor now with a commit unit after the execution units. All results must pass here before getting written back to register file of forwarded back to reservation stations.

Reordering Instructions at Writeback

Reordering instructions becomes a problem for exceptions. When an exception is thrown, the program should know exactly which operation caused the exception. This becomes tricky if the instructions are all reordered.

We add a reorder buffer in the commit unit

This allows us to write results back to memory in the order in which they arrived. It functions as a FIFO. We check to see if we know the value at the head; if we do, return it. If we don't, do nothing and wait.

This commit unit is the middleman between the execution units and the register file. At any point in time, we choose to either commit the **oldest** instruction in the commit unit, or not. This ensured program order. Check class slides for detailed illustration.

What about exceptions?

When a synchronous exception occurs, we do not do anything immediately. **We mark the corresponding entry in the ROB**. When we are ready to commit the instruction (head of FIFO), we raise the exception. When this happens, we also **trash the contents of the ROB and all RSs**. We call this *squashing*

We mark the ROB with a bit that says whether or not the operation raised an exception.

Decoding and Dependencies

The reorder buffer knows of all instructions not yet committed and their destination registers Possible situations for the decoding: - No dependencies: read the **value** from the RF - Dependency from an ongoing instruction:

a. if the value is computed, get **value** from ROB b. if the value isn't yet computed, get the **tag** from ROB

We check for depencies in the ROB.

Example 1: I need `$f1`. I look in ROB for it. If it isn't there, then I know that it's safe to read it from the RF.

Example 2: I need `$f2`. I look in ROB and it is present - an ongoing instruction has produced it's value of `0xA732 FF64`, so I read this value from ROB.

Example 3: I need `$f3`. I look in RO and it is present - an ongoing instruction is still computing it in `MUL2`. Hence we send tag `MUL2` back.

If we look for a register value that will be produced by multiple ongoing instructions, we choose the most recent one to return.

Dependencies Through Memory

The way we detect and resolve dependencies through memory (*a store and subsequent load from the same address*) is handled the same was as with the ROB.

- if there is no store to the same address in the ROB, get the value from memory.
- if there is a store to the same address in the ROB, either get the value (if ready) or the tag
- If there is a store to an unknown address in the ROB or if the address of the load is unknown, wait! (this is a new situation that wasn't present for the register version)

In practice, the memory part of the ROB is implemented in the load-store queue.

Result of Dynamic Scheduling:

We have now significantly spedup our processor and implemented a tangible amount of ***instruction level parallelism***

Superscalar Processors

The idea is that we want to exploit dynamic scheduling even further. Why not more than one instruction-beginning execution issues per cycle?

Fetching more instructions in a cycle: not that hard (provided that the instruction cache has the bandwidth necessary)

Design Elements

referring to slide 6 We take the design previously discussed for dynamically scheduled processors and make some changes.

Before Execution - Widen up the bandwidth of the fetch and decode unit to allow for multiple instructions per cycle - "Park" more instructions in the reservation stations - Add more execution units *if the program has loads of integer operations, why not add more ALUs?* (these ALUs will share the same reservation station!)

After Execution - Commit unit must be able to commit multiple instruction results per cycle - at the same rate as the fetch and decode unit.

Difficulty in the reorder buffer: Not only do I have to see what's in the order buffer, but I have to do it while I haven't finished the cycle (result hasn't been written yet). I have to solve dependencies on an unfinished cycle. This can be easy-ish in a language like `MIPS` because every instruction is a word. However other instruction sets don't necessary have a fixed instruction length.

Hyperthreading: Multiple programs running on the same CPU. On non-hyperthreaded CPUs, we send interrupts at intervals so that another program can take over.

Cache Requirements

Given the following code:

```
lw $t2, 0($t0) ; mem[t0]
lw $t3, 0($t1) ; mem[t1]
addi $t3, $t3, 123
andi $t3, $t3, 0xff
```

A cache miss for `mem[t0]` will require a slow main memory access. With a superscalar processor we want to be able to continue execution as far as dependencies allow it.

Non-blocking Cache A second load cannot be served by main memory since there is only one port. The cache could serve another load while waiting for main memory (provided the data is in cache)

We do this every cycle:

- Cache hits are served immediately
- If there's a miss: *"I'll send a request to main memory. In the meantime, ask me for something else. If I have it, I'll answer immediately"*
- If I get a second request that misses, then I will have two pending requests. In this case do the following: search through pending misses - is this a new miss or part of a previous one?

Dynamic Branch Prediction

Statically predicting branches is rarely accurate enough. To remedy this, we do it dynamically for example by learning from history.

Branch History table Take branch addresses, put them in a table, and annotate them with a bit that says whether they were taken or not. This allows us to keep track of frequently read instructions.

Control Speculation

The next refinedment after branch prediction is control speculation. Assume that the prediction is correct and execute - was it right? Great. Was it wrong? Squash it.

refs: COD 5th edition sections 4.8 and 4.10

VLIW - Very Long Instruction Word

Dynamic scheduling is complicated (in terms of hardware). It is also super difficult to test. *can we do something simpler*?

Static Scheduling Do all the scheduling work turing compile-time instead of during runtime. Instead of feeding instructions, we structure the instructions **very long words that compose of sub-instructions destined for each execution unit**

The problem with it is that the code tends to be exceptionally bloated (too many `nop`s), we need advanced compiler technology, and we get binary compatibility issues.

Overall, VLIW architectures are rarely ever used for general purpose processors. However they are much more common in signal processing (wifi chips, bluetooth chips, DSP, etc. . .)