Processor Assignment

COMS30046 - Advanced Computer Architecture

Jacob Daniel Halsey

The Instruction Set

- I have chosen to simulate (a subset) of the ARM Cortex-M0 Thumb instruction set.
- I have implemented 41 of the 56 supported instructions.
- Each instruction takes (roughly) the same number of cycles as it would on the Cortex-M0
- In addition I have also repurposed the SVC (supervisor call) instruction as a means of communicating between the running program and the simulator:
  - SVC #1 to exit (Exit code in R0)
  - SVC #2 to print (Address in R0, length in R1)

The Programs - Toolchain

- The simulator can support any simple C program, by using:

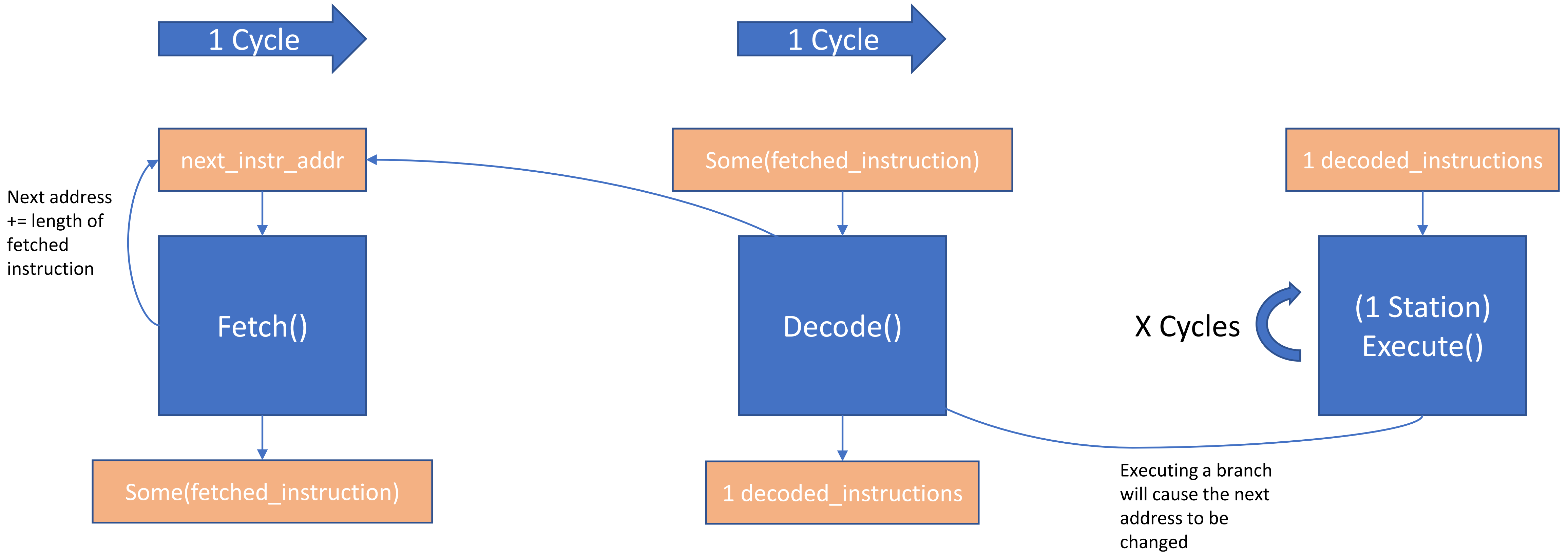
```
arm-none-eabi-gcc -mthumb -mcpu=cortex-m0 -ffreestanding sim.c example.c –o example.elf
```

- By default GCC will attempt to link with the Newlib implementation of the C standard library.
- `sim.c` contains implementations of a few system calls to enable newlib to function in this environment – including `_exit()`, `_write()`, and `_sbrk()` using a small statically allocated heap.
- Newlib also initialises the stack pointer to the address of the `_stack` symbol, defined in the default linker script as `0x80000`.

The Programs – Binaries & Memory

- `memory.rs` simulates a 32-bit address space, made up of “pages” mapped to particular address ranges within that space.
- I am using the rust-elf library to parse the ELF binary files.
- At launch each of the `PT_LOAD` program headers in the ELF has its data loaded as a “page” into the virtual memory, along with its specified write flag. These will correspond to:
  - The text and data segments (code & constants) as read only
  - The static variables as read/write
- An additional “page” is created for use as a stack (starting from the `0x80000` address to match Newlib).
- I am using the capstone-rs library to decode the ARM instructions.

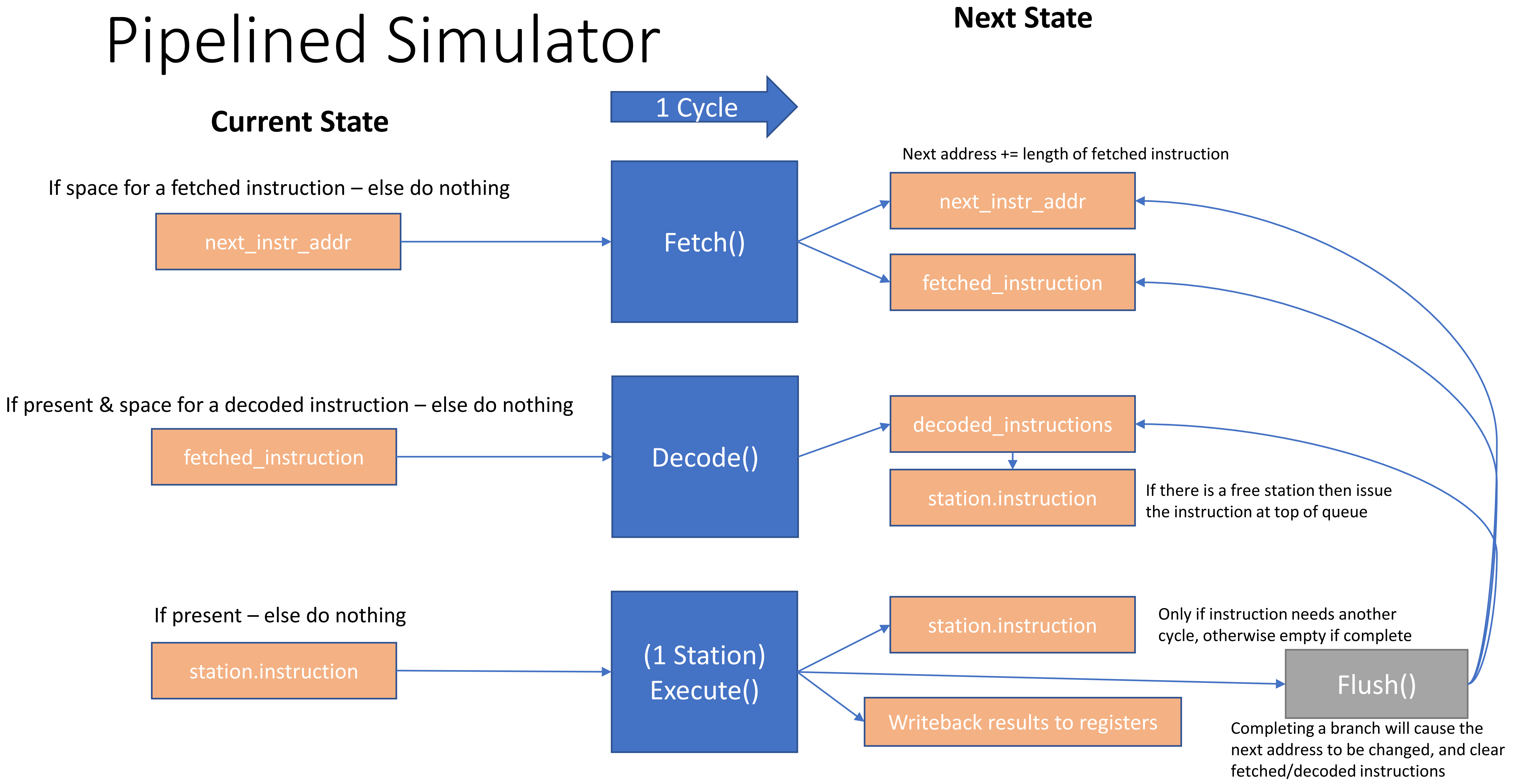
Scalar Simulator


1 Cycle
1 Cycle
next_instr_addr
Next address += length of fetched instruction
Fetch()
Some(fetched_instruction)
Some(fetched_instruction)
Decode()
1 decoded_instructions
1 decoded_instructions
X Cycles
(1 Station) Execute()
Executing a branch will cause the next address to be changed

Pipelined Simulator

- The pipelined simulator runs the fetch, decode, and execute steps in parallel. The code actually runs them concurrently on multiple threads.
- This is not actually faster in practice (overhead exceeds performance benefits!) but it does effectively demonstrate that the stages can indeed be executed in parallel.
- The pipeline means that instructions can be speculatively fetched and decoded – I am using a static predictor of **not taken** for any branching instruction.
- A not taken branch therefore requires only 1 cycle, and a taken requires 3 cycles due to flushing the pipeline (much like the actual Cortex-M0).

Pipelined Simulator


Current State
Next State
1 Cycle
If space for a fetched instruction – else do nothing
next_instr_addr
Fetch()
Next address += length of fetched instruction
next_instr_addr
fetched_instruction
If present & space for a decoded instruction – else do nothing
fetched_instruction
Decode()
decoded_instructions
station.instruction
If there is a free station then issue the instruction at top of queue
If present – else do nothing
station.instruction
(1 Station) Execute()
station.instruction
Only if instruction needs another cycle, otherwise empty if complete
Flush()
Writeback results to registers
Completing a branch will cause the next address to be changed, and clear fetched/decoded instructions

Out of Order Simulator

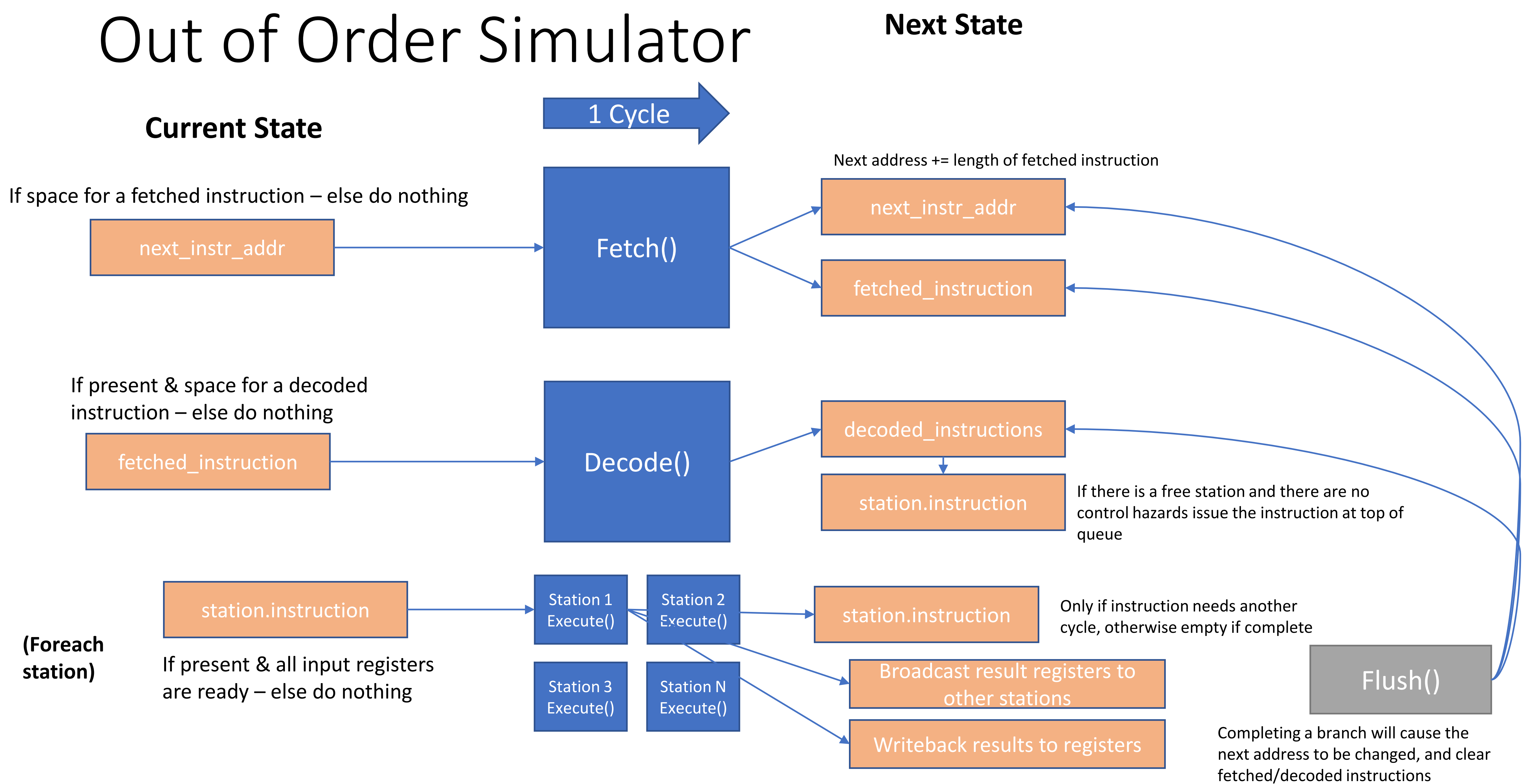
- The out of order simulator uses Tomasulo’s algorithm to deal with hazards and implement register renaming.
- Each input register is filled in as ready if it is already available, or pending if it is dependent on another station currently executing.

```
pub source_registers: HashMap<RegId, Register>,

pub enum Register {
    Ready(u32),
    Pending(StationId, RegId),
}
```

- When an instruction completes executing, its outputs are written back to the registers, as well as broadcast so that any other stations with pending registers for this station may be resolved.

Out of Order Simulator


Current State
1 Cycle
Next State
If space for a fetched instruction – else do nothing
next_instr_addr
Fetch()
Next address += length of fetched instruction
next_instr_addr
fetched_instruction
If present & space for a decoded instruction – else do nothing
fetched_instruction
Decode()
decoded_instructions
station.instruction
If there is a free station and there are no control hazards issue the instruction at top of queue
(Foreach station)
station.instruction
If present & all input registers are ready – else do nothing
Station 1 Execute()
Station 2 Execute()
Station 3 Execute()
Station N Execute()
station.instruction
Only if instruction needs another cycle, otherwise empty if complete
Broadcast result registers to other stations
Writeback results to registers
Flush()
Completing a branch will cause the next address to be changed, and clear fetched/decoded instructions

Experiments

Hypothesis: Using a 3 stage pipeline will reduce the total cycles by up to a maximum factor of 3. The fewer taken branches the closer it will be to the maximum of 3 (because a taken branch flushes the pipeline).

Experiment: I wrote two versions of the same function, one with and one without an unrolled loop (because the unrolled version will have a lower number of branches taken). I tested both with and without the pipeline enabled.

Program	Instructions	Branches Taken	Non-Pipelined Cycles	Pipelined Cycles	Factor
Bitcount O3	630	62	1896	762	2.488
Bitcount Unrolled O3	433	16	1306	474	2.755

Result: Both programs did result in a reduction of cycles, each of less than 3x. The version with fewer taken branches had a noticeable improvement in the cycle reduction when enabling the pipelined mode.

Hypothesis: Adding an execution unit / reservation station will increase the instructions per cycle rate, each time by a decreasing amount, until reaching zero.

Experiment: I ran the test2.elf program (which contains a large variety of instructions and patterns) with different numbers of execution units:

Stations / Units	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Cycles	34294	30847	29924	29793	29780	29772	29772
Change		-3447	-923	-131	-13	-8	0
Instructions / Cycle	0.581	0.646	0.651	0.654	0.654	0.654	0.654

Result: As expected the first increase in units leads to a larger reduction in processor cycles, but this then diminishes until we reach a limit where we cannot run any more instructions in parallel.

Hypothesis: Using loop unrolling to increase instruction level parallelism will lead to a greater improvement in cycle rate when adding additional execution units.

Experiment: I compared the regular bitcount and unrolled version, when run in order (1 unit) and out of order (4 units). (I am using the unoptimized version because I don't want the compiler to make unexpected optimizations that may diminish the difference between the two versions).

Program	**Instructions**	**1 Unit**	**4 Units**	**Change**
Bitcount O0	1611	0.647	0.840	+29.8%
Bitcount Unrolled O0	1067	0.637	0.836	+31.3%

Result: Enabling out of order execution increases the performance for both, but with only a very small increase in change for the unrolled version. Perhaps this is because the unrolled instructions mainly consist of single cycle arithmetic operations, that when combined with single issue do not create many chances for overtaking / out of order execution.

(End)