ECS404U: Computer Systems & Networks

Week 09: CPU Design and Performance Techniques (pipelining and its hazards)

Prof Edmund Robinson, Dr Arman Khouzani

November 16, 2020

EECS, QMUL

Agenda/Objectives

- Designing a 5-stage CPU
- Techniques to improve performance
  - Pipe-lining
    - Pipeline hazards
  - Super-scaling
- Practice MIPS Assembly

Implementing an ISA

Recall: one of the components of the CPU is a clock pulse (generated by an oscillator). That is what regulates the rhythm of operations of the CPU. Informally speaking, each unit of operation of CPU takes place during each clock cycle (the amount of time between the two rising edges of a pulse sequence).

- A new notion is a **CPU cycle**: the amount of time that the CPU takes to carry out an instruction.

Note that a CPU cycle can be comprised of many clock cycles.

Implementing an ISA

There are at least three paradigms about the hardware implementation of an ISA:

- ▷ **Single-Cycle:** each instruction takes the same amount of time: one CPU cycle (so the length of each CPU cycle is the same).
    - easiest to implement (finite state machine)
    - each CPU cycle should be long enough for the slowest/most complex instruction (to ensure that the results are correct), so the speed of operation is determined by the slowest instruction;
- ▷ How can we improve efficiency (speed)?

Implementing an ISA (Designing a CPU)

- **Multi-Cycle:** allowing CPU cycles with different durations, shorter CPU cycles for faster/simpler instructions, while longer CPU cycles for slower/more complicated ones.
  - Improved efficiency, but more complicated hardware, and some parts of the hardware may still stay idle.
- **Pipelined** having a fixed number of stages, but allowing overlap between different stages of consecutive instructions.

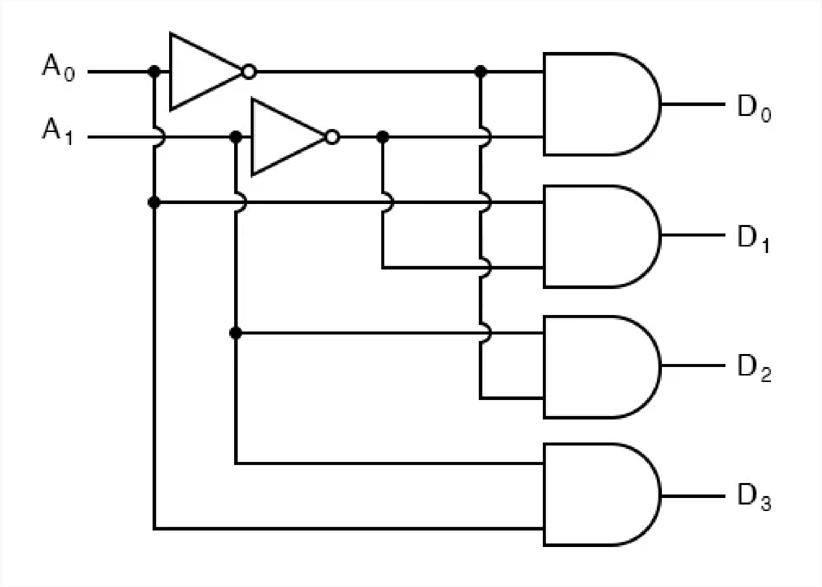
Implementing an ISA (Designing a CPU)

- **Multi-Cycle:** allowing CPU cycles with different durations, shorter CPU cycles for faster/simpler instructions, while longer CPU cycles for slower/more complicated ones.
  - Improved efficiency, but more complicated hardware, and some parts of the hardware may still stay idle.
- **Pipelined** having a fixed number of stages, but allowing overlap between different stages of consecutive instructions.

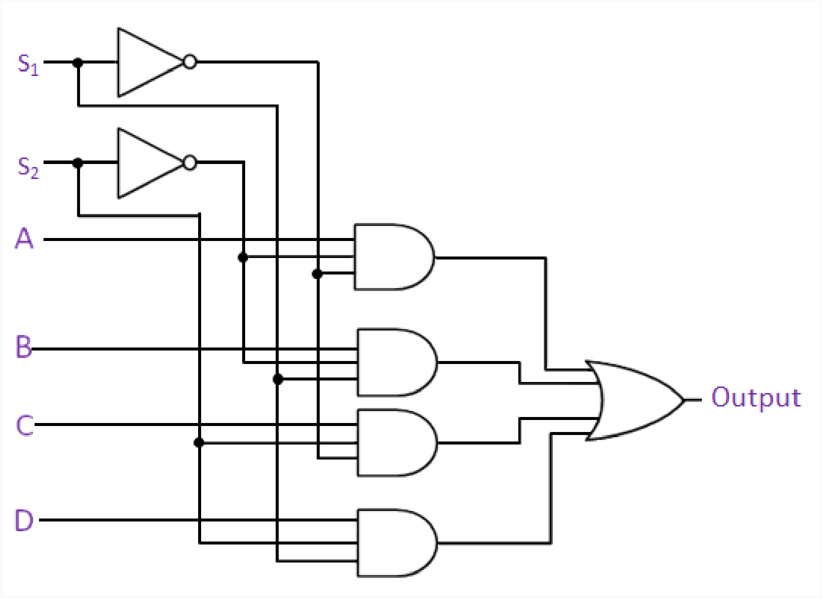
Implementing an ISA (Designing a CPU)

- **Multi-Cycle:** allowing CPU cycles with different durations, shorter CPU cycles for faster/simpler instructions, while longer CPU cycles for slower/more complicated ones.
  - Improved efficiency, but more complicated hardware, and some parts of the hardware may still stay idle.
- **Pipelined** having a fixed number of stages, but allowing overlap between different stages of consecutive instructions.

A few more circuits: Decoder



A few more circuits: Multiplexer (data selector)



Implementing an ISA (Designing a CPU)

The classic 5-stage implementation of a RISC architecture:

- ▷ **`IF`**: Instruction fetch
- ▷ **`ID`**: Instruction decode
- ▷ **`EX`**: Execute
- ▷ **`MEM`**: Memory Access
- ▷ **`WB`**: Write Back

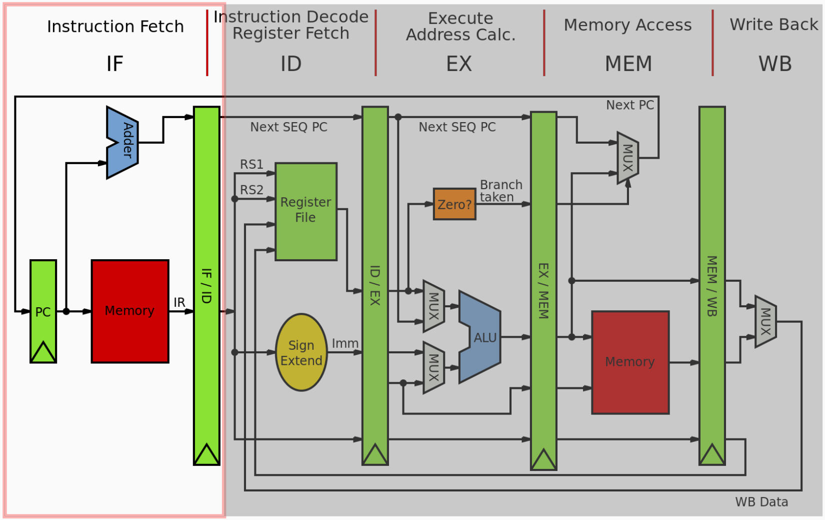
[the detail of the stages is not part of assessment!]

Designing a CPU: Instruction Fetch

We will start building a CPU by gathering the components we need. First, we need a hardware that does **Instruction Fetch (IF)**

- ▶ Fetch the next instruction from the main memory whose location (address) is in `PC`.
  - ▷ Increment `PC` by 4 (is there a problem here?)
  - ▷ Produce for the next stage: the instruction (32 bits) to be executed.

Designing a CPU: Instruction Fetch



Designing a CPU: Instruction Decode

- **Instruction Decode (ID):** parse the instruction (gather information from the instructions fields) create appropriate control signals based on which operation to do given the opcode, determine which registers to read from (and read them), determine which registers to write to, what immediate value, how to extend it (and extend it), etc.);
  - Input from the previous stage (IF): The instruction to be executed;
  - Output to the next stage (EX): control signals for choosing the operation, and the operands;

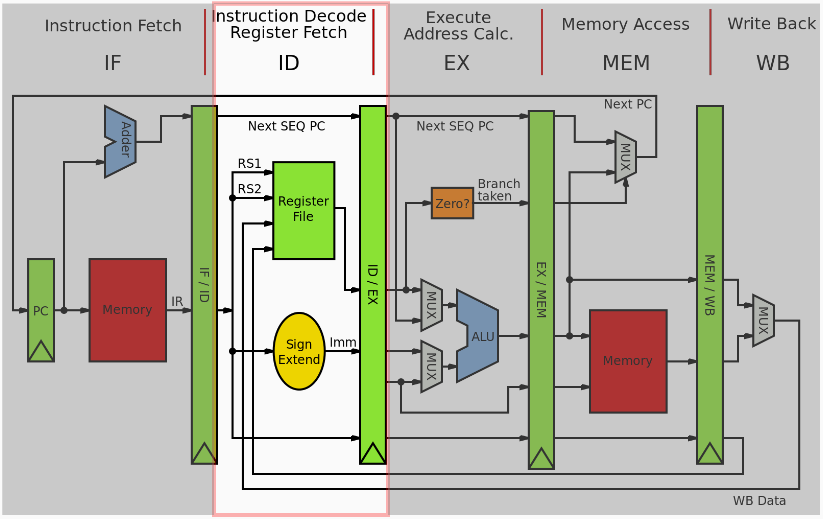
Designing a CPU: Instruction Decode

- **Instruction Decode (ID):** parse the instruction (gather information from the instructions fields) create appropriate control signals based on which operation to do given the opcode, determine which registers to read from (and read them), determine which registers to write to, what immediate value, how to extend it (and extend it), etc.);
  - Input from the previous stage (IF): The instruction to be executed;
  - Output to the next stage (EX): control signals for choosing the operation, and the operands;

Designing a CPU: Instruction Decode

- **Instruction Decode (ID):** parse the instruction (gather information from the instructions fields) create appropriate control signals based on which operation to do given the opcode, determine which registers to read from (and read them), determine which registers to write to, what immediate value, how to extend it (and extend it), etc.);
  - Input from the previous stage (IF): The instruction to be executed;
  - Output to the next stage (EX): control signals for choosing the operation, and the operands;

Designing a CPU: Instruction Decode



Designing a CPU: Execute (the ALU)

- ▶ **Execute (EX)**: Calculating the arithmetic/logic operation (mostly just combinatorial logic circuits)
  - arithmetic, shifting, logical operations directly from the instruction, **`add, sub, and, or, xor, sll`**, etc.
  - arithmetic operations required indirectly by the instruction, e.g. calculating the address for memory access operations (**`lw, sw`**), or comparing registers (and calculating memory address) for branching instructions (**`beq, bne`**)
- ▷ Input from the previous stage (ID): The instruction to be executed;
- ▷ Output to the next stage (MEM): control signals for choosing the operation, and the operands;

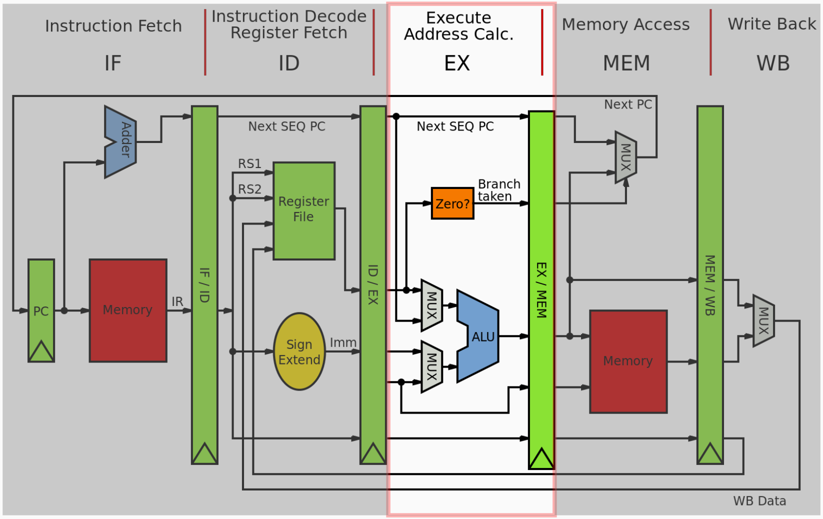
Designing a CPU: Execute (the ALU)

- **Execute (EX)**: Calculating the arithmetic/logic operation (mostly just combinatorial logic circuits)
  - arithmetic, shifting, logical operations directly from the instruction, **`add, sub, and, or, xor, sll`**, etc.
  - arithmetic operations required indirectly by the instruction, e.g. calculating the address for memory access operations (**`lw, sw`**), or comparing registers (and calculating memory address) for branching instructions (**`beq, bne`**)
- ▷ Input from the previous stage (ID): The instruction to be executed;
- ▷ Output to the next stage (MEM): control signals for choosing the operation, and the operands;

Designing a CPU: Execute (the ALU)

- ► **Execute (EX)**: Calculating the arithmetic/logic operation (mostly just combinatorial logic circuits)
  - arithmetic, shifting, logical operations directly from the instruction, **`add, sub, and, or, xor, sll`**, etc.
  - arithmetic operations required indirectly by the instruction, e.g. calculating the address for memory access operations (**`lw, sw`**), or comparing registers (and calculating memory address) for branching instructions (**`beq, bne`**)
- ▷ Input from the previous stage (ID): The instruction to be executed;
- ▷ Output to the next stage (MEM): control signals for choosing the operation, and the operands;

Designing a CPU: Execute (the ALU)



Designing a CPU: Memory Access

- **Memory Access (MEM)** only for load and store instructions (e.g. **`lw`**, **`sw`**)
  - uses the memory address calculated for the load or store instructions
  - to read from or write some data to the main memory
  - for all other instructions, nothing happens in this stage (idle, just passing previous data)
- Input from the previous stage (EX): calculated memory address (if instruction has been **`lw`** or **`sw`**) and the data to be stored in memory (for **`sw`**)
- Output to the next stage (WB): the loaded data from memory to be written to the registers (for **`lw`**).

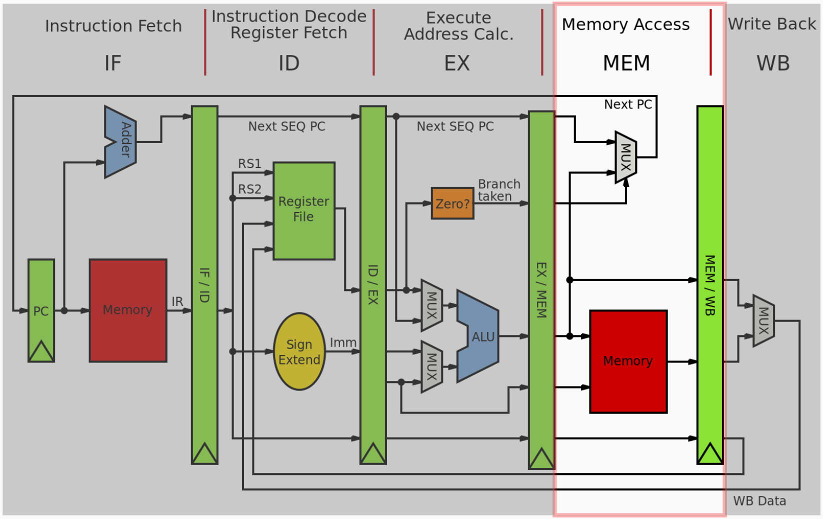
Designing a CPU: Memory Access

- **Memory Access (MEM)** only for load and store instructions (e.g. `lw`, `sw`)
  - uses the memory address calculated for the load or store instructions
  - to read from or write some data to the main memory
  - for all other instructions, nothing happens in this stage (idle, just passing previous data)
- ▷ Input from the previous stage (EX): calculated memory address (if instruction has been `lw` or `sw`) and the data to be stored in memory (for `sw`)
- ▷ Output to the next stage (WB): the loaded data from memory to be written to the registers (for `lw`).

Designing a CPU: Memory Access

- **Memory Access (MEM)** only for load and store instructions (e.g. `lw`, `sw`)
  - uses the memory address calculated for the load or store instructions
  - to read from or write some data to the main memory
  - for all other instructions, nothing happens in this stage (idle, just passing previous data)
- ▷ Input from the previous stage (EX): calculated memory address (if instruction has been `lw` or `sw`) and the data to be stored in memory (for `sw`)
- ▷ Output to the next stage (WB): the loaded data from memory to be written to the registers (for `lw`).

Designing a CPU: Memory Access



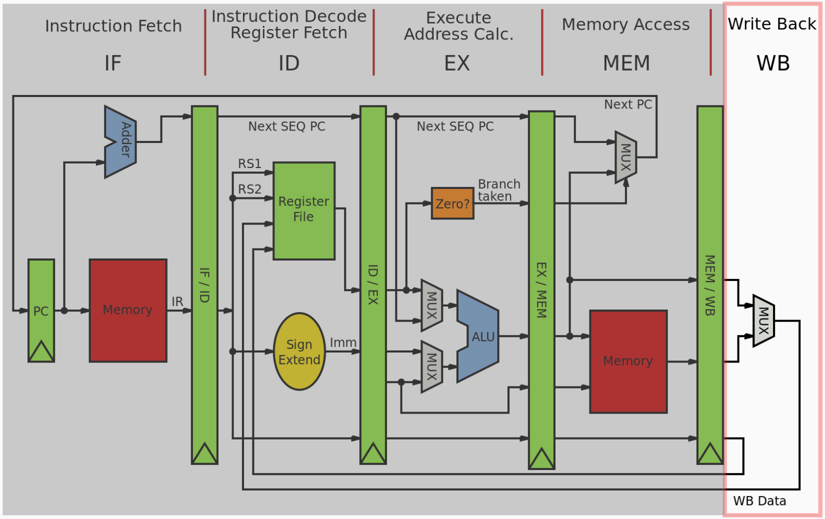
Designing a CPU: Register Write Back

- **Register Write Back (WB)** only for load and store instructions (e.g. `lw`, `sw`)
  - writes the result of the arithmetic/logic or memory read (`lw`) back into the destination register (so need destination register number and computed result)
  - for branching instructions, jumps or memory store instructions nothing happens in this stage (idle)
- Input from the previous stage (MEM): calculated result from ALU or loaded data from memory.

Designing a CPU: Register Write Back

- **Register Write Back (WB)** only for load and store instructions (e.g. `lw`, `sw`)
  - writes the result of the arithmetic/logic or memory read (`lw`) back into the destination register (so need destination register number and computed result)
  - for branching instructions, jumps or memory store instructions nothing happens in this stage (idle)
- Input from the previous stage (MEM): calculated result from ALU or loaded data from memory.

Designing a CPU: Write Back



Speedup Techniques: Pipelining & Parallelism

- ▷ Do we always have 5-stage implementation?
- ▷ But why 5 stages, when some (actually most) of the instructions don't use all?
- ▷ How can we do better?
- ► **Pipelining**, **Super scaling**
    - and other more advanced techniques like out-of-order execution, multi-core and multi-threading: super-threading, hyper-threading, etc.

Speedup Techniques: Pipelining & Parallelism

- ▷ Do we always have 5-stage implementation?
- ▷ But why 5 stages, when some (actually most) of the instructions don't use all?
- ▷ How can we do better?
- ► **Pipelining**, **Super scaling**
    - and other more advanced techniques like out-of-order execution, multi-core and multi-threading: super-threading, hyper-threading, etc.

Speedup Techniques: Pipelining & Parallelism

- ▷ Do we always have 5-stage implementation?
- ▷ But why 5 stages, when some (actually most) of the instructions don't use all?
- ▷ How can we do better?
- ► **Pipelining**, **Super scaling**
  - and other more advanced techniques like out-of-order execution, multi-core and multi-threading: super-threading, hyper-threading, etc.

Speedup Techniques: Pipelining & Parallelism

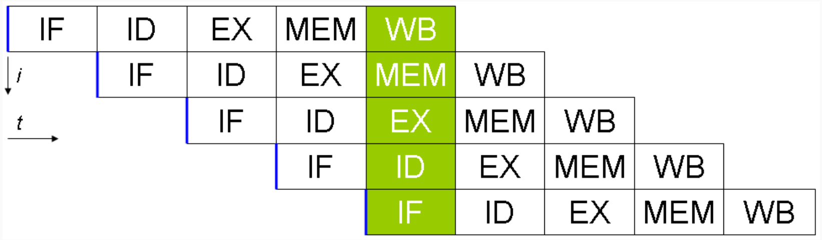
- ▷ Do we always have 5-stage implementation?
- ▷ But why 5 stages, when some (actually most) of the instructions don’t use all?
- ▷ How can we do better?
- ▶ **Pipelining**, **Super scaling**
    - and other more advanced techniques like out-of-order execution, multi-core and multi-threading: super-threading, hyper-threading, etc.

Speedup Techniques: Pipelining

- **pipelining**: trying to keep all stages busy, so instead of waiting for an instruction to finish all stages before processing the next instruction, feeding the next instruction in as soon as the previous one finishes a stage (like a moving assembly line in a car factory!)
  - So instructions are now only one stage apart, and their process is overlapping.
  - For example, in a 5-stage pipeline, at the same time, 5 different instructions can be executed. (But how?!)

Speedup Techniques: Pipelining

This is how:[1]



IF	ID	EX	MEM	WB				
i	IF	ID	EX	MEM	WB			
t		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB

- ▷ So after 5 clock cycles), all 5 stages of the CPU will be busy (no stage will be idle)

[1] Figure: https://en.wikipedia.org/wiki/Classic_RISC_pipeline

Speedup Techniques: Pipelining

- So what speed-up do we get using this 5-stage pipeline (ideally)?

Recall that each stage takes exactly one clock cycle. Now, suppose we have *N* instructions, then:

without pipelining (sequentially processing)	takes $5N$ clock cycles
with pipelining	takes $N + 4$ clock cycles

So the speed-up is $\frac{5N}{N+4}$. For large *N* this is almost 5-fold (so effectively, each instruction is taking only one clock cycle!). This is amazing, however, we have *hazards!*

Pipeline Hazards

- **Pipeline hazards** (term coined by Henessy-Patterson!) a.k.a. **pipeline stall**: situations which cause problem with the ideal (fully packed) pipeline, i.e., the fully packed pipeline would create wrong results.
   - so the pipeline has to be interrupted, e.g. by introducing *pipeline bubbles* by inserting **`nop`** (no operation) instructions, or re-flushing the pipeline, which reduces the efficiency of the pipeline

Pipeline Hazards

Types of pipeline hazards:

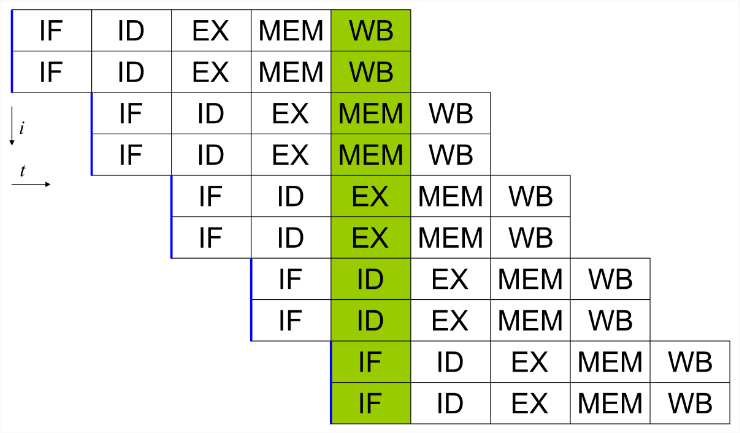
- *Structural hazards:* two instructions want to use the same hardware resource (e.g. ALU)
  - can be avoided by replicating the hardware
- *Data Hazards:* a subsequent instruction uses data that is not resolved yet by the previous instruction
  - e.g., consider the following consecutive instructions:
    ```
    add $t0, $t1, $t2
    sub $t4, $t3, $t0
    ```
  - can be avoided by *bypassing (operand **forwarding**)*
  - However, sometimes it is inevitable to introduce stalls which reduces efficiency, by forcing some stages to be idling until the data is settled. The circuit to detect such hazards and enforce stalls is pipeline “interlock”.

Pipeline Hazards

- ▷ *Control hazards*: Branching
    - Branching is problematic for pipelining, when a branching instruction is being processed, the *next* instruction to be fed into the pipeline is not going to be known until stage 3 of the pipeline.
    - So to know what instruction to feed to the pipeline, we either have to wait, or just *speculate* what is the likely next instruction and feed that in (e.g. assume **`PC+4`** is the address of the next instruction)
    - However, if our guess is wrong, then the wrongly fed instruction has to be flushed from the pipeline and the correct next instruction fed in. That means for some clock cycles some stages will be idling, which undermines the overall efficiency of the pipeline.

Speedup Techniques: Super scaling

Super scaling: simultaneously dispatching multiple pipelines:[2]



IF	ID	EX	MEM	WB				
IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB
				IF	ID	EX	MEM	WB

[2]Figure: https://en.wikipedia.org/wiki/Superscalar_processor

Practice Example for MIPS Assembly

Suppose we have four variables initially loaded into registers **`t0`**, **`t1`**, **`t2`**, **`t3`**. Let's implement a *while loop* in MIPS Assembly:

```
while (t0> t1) {
    t2 = t2 + t0;
    t0 = t0 - t3;
}
```

Let's do this through steps. First, the easy part: the inside part of the loop:

```
    add $t2, $t2, $t0    # t2 <- t2+t0
    sub $t0, $t0, $t3    # t0 <- t0-t3
```

Practice Example for MIPS Assembly

Let's turn it to a loop, and for now, without any conditions:

```
LOOP:
         # branching instruction to come here!
    add $t2, $t2, $t0    # t2 <- t2+t0
    sub $t0, $t0, $t3    # t0 <- t0-t3
    j LOOP
EXIT:    # exit, or the rest of the code
```

So what should we put as the branching condition to exit from the loop?

Practice Example for MIPS Assembly

The first (incorrect) attempt in the class was the following:

```
LOOP:
    beq $t0, $t1, EXIT  # goto exit if t0==t1
    add $t2, $t2, $t0   # t2 <- t2+t0
    sub $t0, $t0, $t3   # t0 <- t0-t3
    j LOOP
EXIT:   # exit, or the rest of the code
```

- although **`$t0`** keeps decrementing by **`$t3`**, it may never become equal to **`$t1`**: it may skip over it!
- even before entering the loop, we can have **`$t0<$t1`**, then the **`while`** loop should have not executed at all!

Note that though this is wrong, this will get $\sim 6/10$ mark!

Practice Example for MIPS Assembly

The second (still incorrect) attempt was the following:

```
LOOP:
    slt $t4, $t0, $t1        # t4 <- (t0<t1)
    bne $t4, $zero, EXIT     # if t4!=0 goto EXIT
    add $t2, $t2, $t0        # t2 <- t2+t0
    sub $t0, $t0, $t3        # t0 <- t0-t3
    j LOOP
EXIT:   # exit, or the rest of the code
```

- This implements **`while (t0>= t1)`** instead!
  (because even if **`$t0==$t1`**, the loop is executed)

Note that **`slt`** itself is not a branching instruction.

Again even though this is wrong, it'd get $\sim 8/10$ mark.

Practice Example for MIPS Assembly

And finally, a correct answer:

```
LOOP:
    slt $t4, $t1, $t0        # t4 <- (t1<t0)
    beq $t4, $zero, EXIT     # if t4==0 goto EXIT
    add $t2, $t2, $t0        # t2 <- t2+t0
    sub $t0, $t0, $t3        # t0 <- t0-t3
    j LOOP
EXIT:   # exit, or the rest of the code
```

Note that there may be more than one correct implementation, any correct answer will get 10/10 mark!