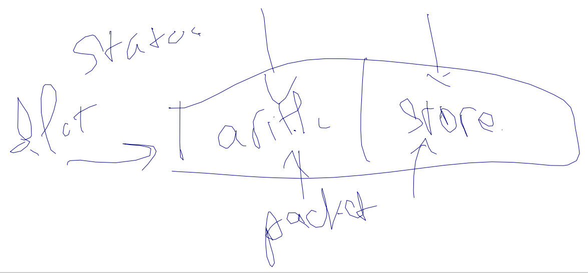
Chapter 4c - Instruction-Level Parallelism (ILP)

1. ILP

ILP is used to achieve higher throughput and lower overall latency by executing multiple instructions in parallel.

There are 2 ways to implement parallelism in MIPS.

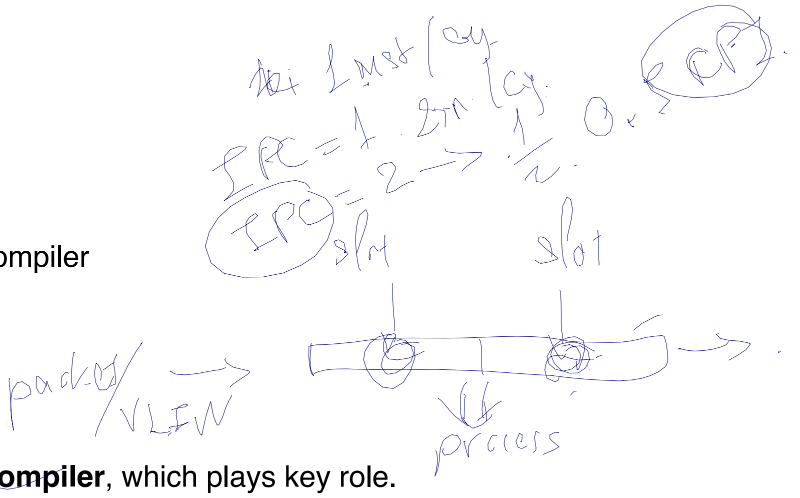
1. Pipeline
2. **Multiple Issue**



2. Multiple Issue

> **Definition of multiple issue**: start multiple instructions in multiple pipelines per clock cycle.

- **Issue Width**: the number of issue slots per instruction, for example 1 slot/instruction.
- **Pros**: Better peformance
  - increase instruction throughput
  - decrease CPI (to below 1) since IPC increases. If a process can run 4 instructions per cycle (IPC), that means it needs only ¼ cycle for 1 instruction, i.e. CPI = ¼

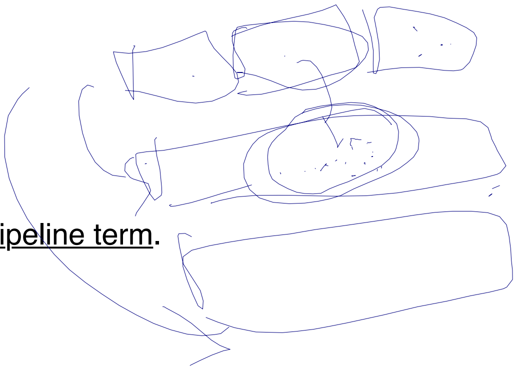


- **Cons**:
  - Greater hardware complexity
  - harder code scheduling job for the compiler
  - Some dependencies still limit ILP.

There are 2 types of Multiple Issue:

1. **Static Multiple Issue**.
   - Static = **Software**, represented by **Compiler**, which plays key role.
   - Compiler groups instruction into "issue packets" to be issued together in 1 slot, i.e. 1 single cycle, and detect & avoid hazards.
   - Issue packet = very long instruction (combined of multiple smaller instructions) = **Very Long Instruction Word (VLIW)**
2. **Dynamic Multiple Issue**
   - Dynamic = **Hardware**, represented by **CPU** aka "superscalar" processor, which plays key role.
   - CPU examines instruction stream to choose which instructions for issuing each cycle. CPU also resolves hazards, by using advanced techniques at runtime.

2.1. Mechanisms



2.1.1. Speculation

- Speculation in Multiple Issue is similar to "Guess/prediction" in pipeline term.
- Apply to both static and dynamic multiple issue
- For **handling branch hazard**.

2.1.1.1. Speculation in Compiler (software) / CPU (hardware)

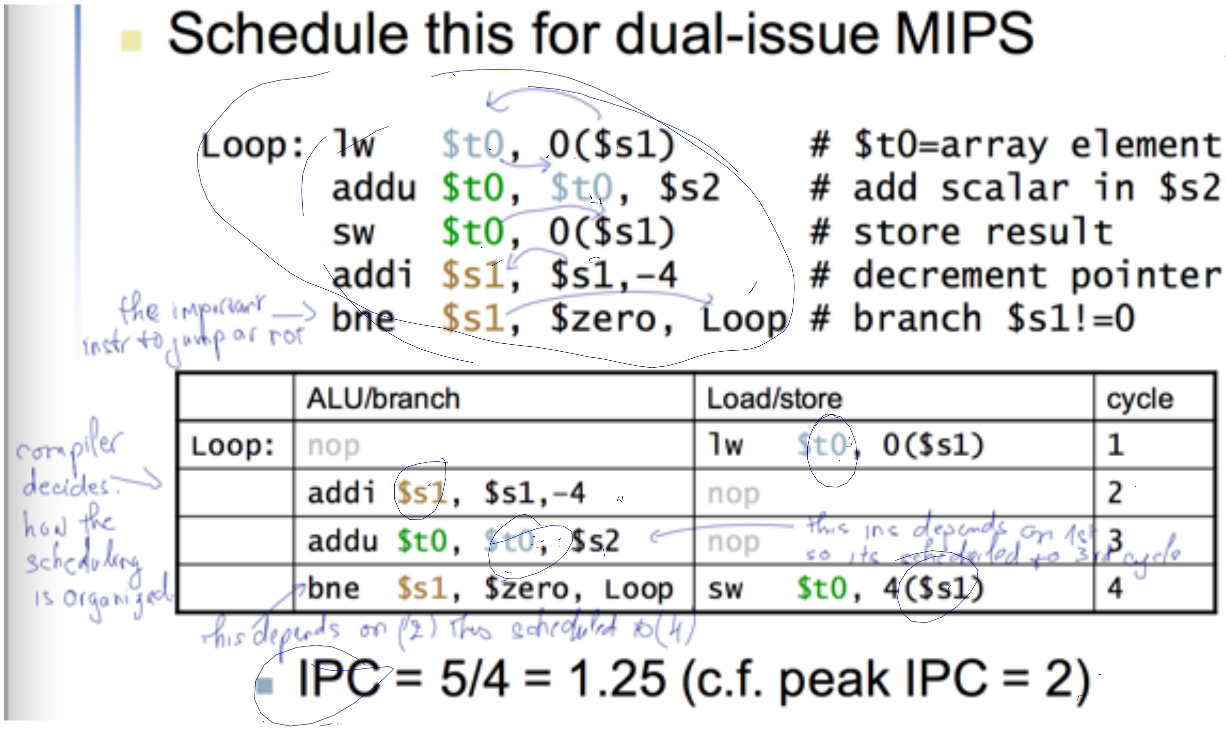
Compiler (software)	CPU (hardware)
Reorder instructions	Look ahead for instructions to execute
Include "fix-up" instructions (nop/bubble) to recover from incorrect guess	Buffer results until it determine when they are needed, flush buffer on incorrect speculation.
Exception handling: use ISA support for deferring exceptions.	Exception handling: Buffer exceptions until instruction completes (may not occur).

2.1.2. Scheduling

2.1.2.1. Scheduling Static Multiple Issue

- Compiler analyszes and reorders instructions into issue packets. Issues in the same packet has no dependency on each other. Possible dependencies across packets, but compiler knows, try to detect and remove hazards.
- Pad with nop if necessary.
- In looping situation, compiler uses a technique called **loop unrolling**, it checks how many loop runs (n), expand & run the unimportant instructions n times (using different registers - i.e. register renaming) and run the important instruction (the loop condition check) 1 time at last. In this case the loop requires less instruction count (minus the unecessary checking of loop condition in each loop), thus **results in better CPI**.

  Example: **Scheduling with Static Dual Issue**



Schedule this for dual-issue MIPS

```
Loop: lw   $t0, 0($s1)       # $t0=array element
      addu $t0, $t0, $s2     # add scalar in $s2
      sw   $t0, 0($s1)       # store result
      addi $s1, $s1,-4       # decrement pointer
      bne  $s1, $zero, Loop  # branch $s1!=0
```

the important instr to jump or not

compiler decides how the scheduling is organized

	ALU/branch	Load/store	cycle
Loop:	nop	lw $t0, 0($s1)	1
	addi $s1, $s1,-4	nop	2
	addu $t0, $t0, $s2	nop	3
	bne $s1, $zero, Loop	sw $t0, 4($s1)	4

this ins depends on 1st so its scheduled to 3rd cycle

this depends on (2) thus scheduled to (4)

IPC = 5/4 = 1.25 (c.f. peak IPC = 2)

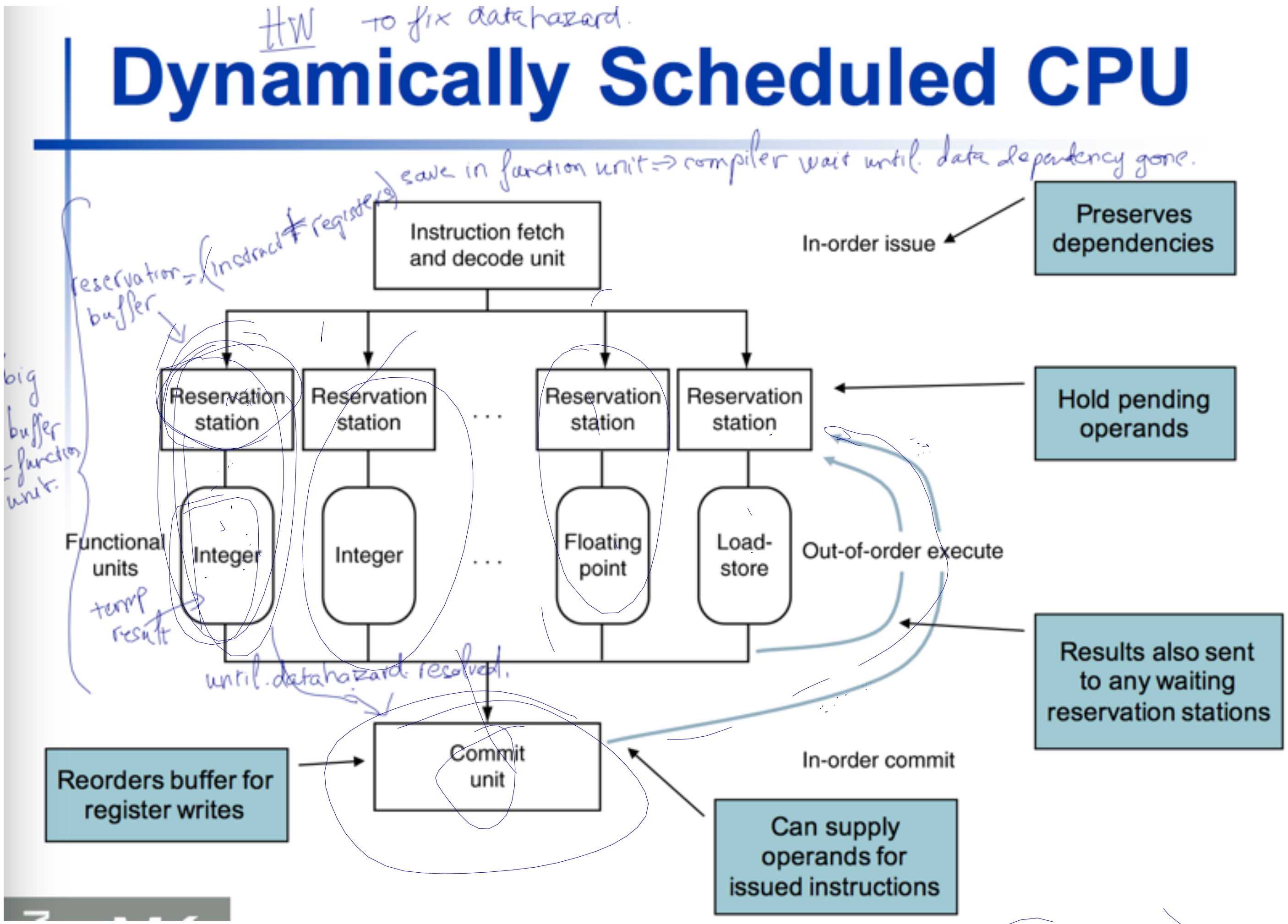
- **Static Dual Issue**: one ALU/branch instruction and one load/store instruction, 64-bit size, pad unused instruction with nop.

2.1.2.2. Scheduling Dynamic Multiple Issue

(aka Dynamic Pipeline Scheduling)

- CPU can execute instructions out of order to avoid stalls.
- But **commit result to registers in order**.
- **Why use dynamic scheduling?** Not all stalls are predicable, can't always schedule around branches, different implementations of ISA have different latencies and hazards.

ordered


HW to fix data hazard.
Dynamically Scheduled CPU
save in function unit ⇒ compiler wait until data dependency gone.
reservation = (instruct # registers) buffer
Instruction fetch and decode unit
In-order issue
Preserves dependencies
big buffer = function unit.
Reservation station
Reservation station
Reservation station
Reservation station
Hold pending operands
Functional units
Integer
Integer
Floating point
Load-store
Out-of-order execute
temp result
until data hazard resolved.
Results also sent to any waiting reservation stations
Reorders buffer for register writes
Commit unit
In-order commit
Can supply operands for issued instructions

- Overall system is called **big buffer function unit**, which consists of multiple **functional units**, and a **commit unit (aka reorder buffer)**.
  - Instructions are loaded from memory for processing.
  - If operand is available in register file or reorder buffer, its value is copied to reservation and no longer required in the register for recycle (used by others).
  - If operand is not yet available, it will wait for the function unit or reorder buffer to provision.

3. Questions

3.1. Is pipeline easy?

No, idea is easy, but implementation (details) is hard.

3.2. Is pipeline independent of technology?

No, more transistors (HW technology) makes more advanced piepline techniques feasible. Thus ISA design (for pipeline) needs to be aware of technology trends.

3.3. Is poor ISA design making pipelining harder?

Yes, complex instruction sets introduce overhead, complex addressing causes memory indirection, long delayed branches.

3.4. Relationship of ISA and datapath and control?

ISA influences design of data path and control, and vice versa.

3.5. What limits the parallelism?

Instruction dependencies, instruction complexity.