

Advanced processor designs

- We've only scratched the surface of CPU design.
- Today we'll briefly introduce some of the big ideas and big words behind modern processors by looking at two example CPUs.
 - The [Motorola PowerPC](#), used in Apple computers and many embedded systems, is a good example of state-of-the-art RISC technologies.
 - The [Intel Itanium](#) is a more radical design intended for the higher-end systems market.



General ways to make CPUs faster

- You can improve the chip manufacturing technology.
 - Smaller CPUs can run at a higher clock rates, since electrons have to travel a shorter distance. Newer chips use a “0.13 μm process,” and this will soon shrink down to 0.09 μm .
 - Using different materials, such as copper instead of aluminum, can improve conductivity and also make things faster.
- You can also improve your CPU design, like we’ve been doing in CS232.
 - One of the most important ideas is **parallel computation**—executing several parts of a program at the same time.
 - Being able to execute more instructions at a time results in a higher instruction throughput, and a faster execution time.

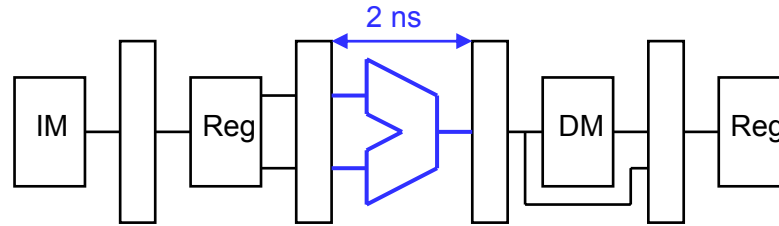
Pipelining is parallelism

- We've already seen parallelism in detail! A **pipelined processor** executes many instructions at the same time.
- All modern processors use pipelining, because most programs will execute faster on a pipelined CPU without any programmer intervention.
- Today we'll discuss some more advanced techniques to help you get the most out of your pipeline.

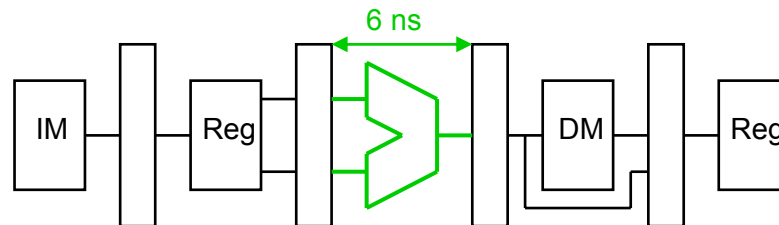


Motivation for some advanced ideas

- Our pipelined datapath only supports integer operations, and we assumed the ALU had about a 2ns delay.



- How can we add floating-point operations, which are roughly three times slower than integer operations, to the processor?



- The longer floating-point delay would affect the cycle time adversely. In this example, the EX stage would need 6 ns, and the clock rate would be limited to about 166MHz.

Deeper pipelines

- We could pipeline the floating-point unit too! We might split an addition into 3 stages for equalizing exponents, adding mantissas, and rounding.



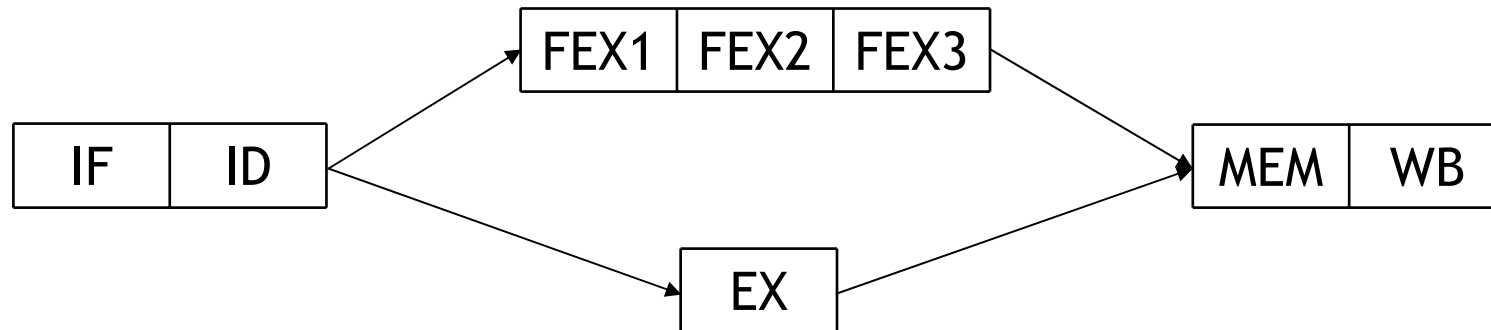
- A floating-point operation would still take 6ns total, but the stage length and cycle time can be reduced, and we can also overlap the execution of several floating-point instructions.



- The homeworks already mentioned how the Pentium 4 uses a 20-stage pipeline, breaking an instruction execution into 20 steps—that's one of the reasons the P4 has such high clock rates.

Superscalar architectures

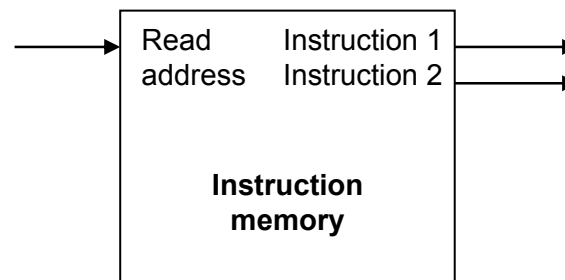
- What if we include both an integer ALU *and* a floating-point unit in the datapath?



- This is almost like having two pipelines!
- A **superscalar processor** can start, or dispatch, more than one instruction on every clock cycle.

Instruction memory contention

- One difficulty with superscalar designs are **structural hazards** that occur when many instructions need to use the same functional units.
- For example, to execute two instructions in one cycle, the memory must be able to fetch *two* instructions at once.

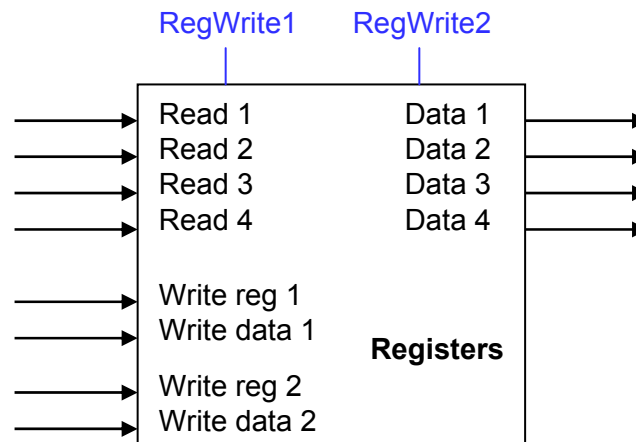


Register file contention

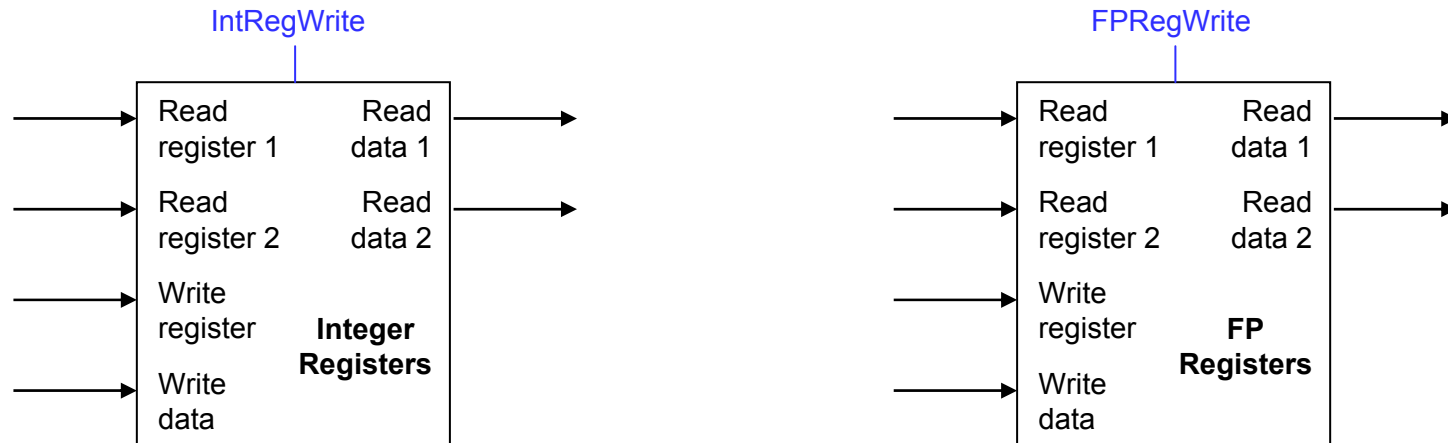
- Executing one integer and one floating-point instruction together would also require reading up to *four* register source operands, and writing up to *two* register destinations.

```
add    $s0, $s1, $s2
fadd   $t0, $t1, $t2
```

- We could add more read and write ports to the register file, as we did for one of the homework questions.



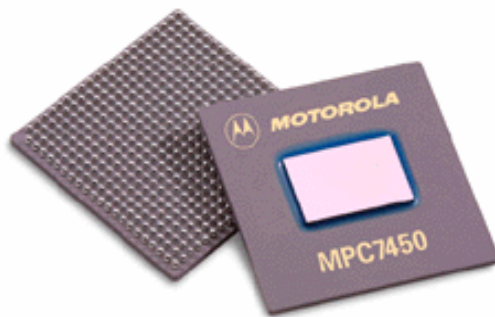
Separate register files



- A common alternative is to provide *two* register files, one for the integer unit and one for the floating-point unit.
 - Most programs do not mix integer and FP operands.
 - Double and extended precision IEEE numbers are 64 or 80-bits long, so they wouldn't fit in normal 32-bit registers anyway.
 - Explicit “casting” instructions can be used to move data between the register files when necessary.
- We saw this in the MIPS floating-point architecture too.

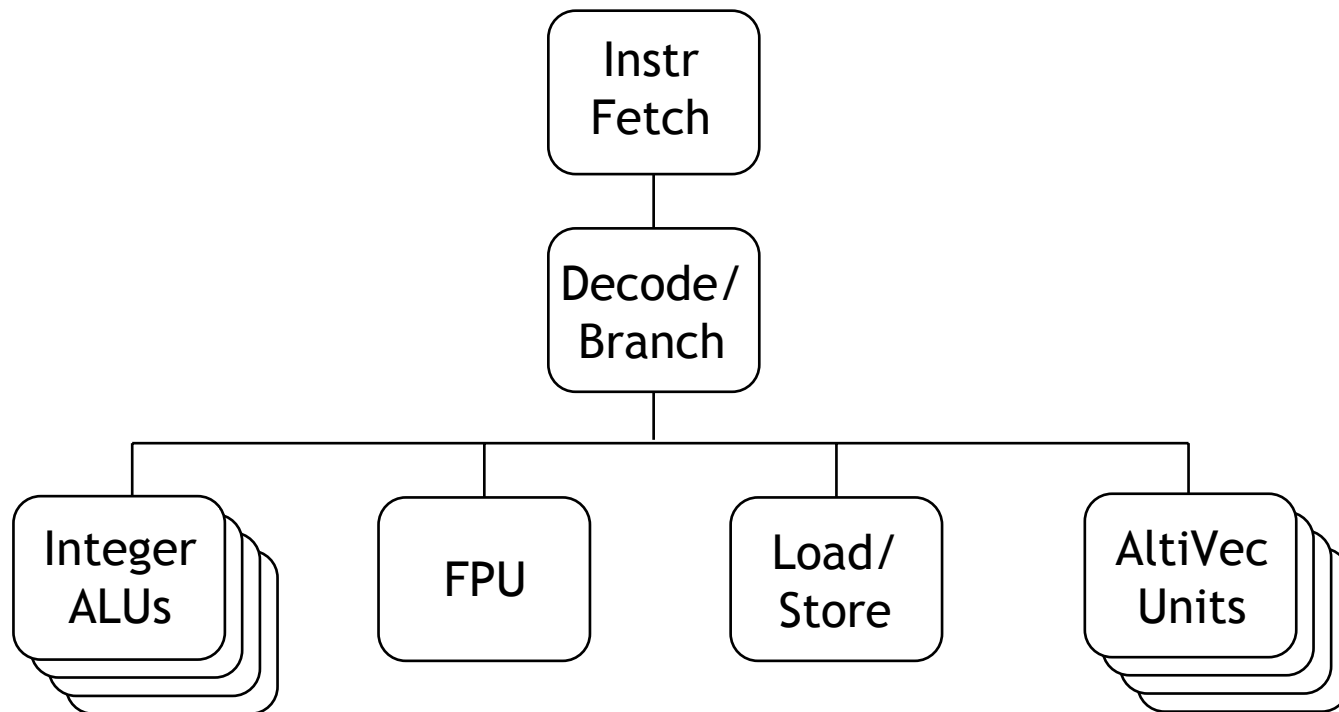
PowerPC introduction

- The PowerPC project was started in 1991 by Apple, IBM and Motorola.
 - The basis was IBM's existing POWER architecture, a RISC-based design used in older IBM RS/6000 workstations.
 - The first actual PowerPC processor was introduced in 1993.
 - The **G4** is the most current version, which runs at up to 1.42 GHz in Apple's Power Macintosh G4 computers.
- PowerPCs are noted for very low power usage: 20-30W for a 1 GHz G4, compared to 60W or more for newer Pentiums and Athlons.
 - Low power is especially good for laptops, which run on batteries.
 - It also helps keep desktop machines cool without fans.



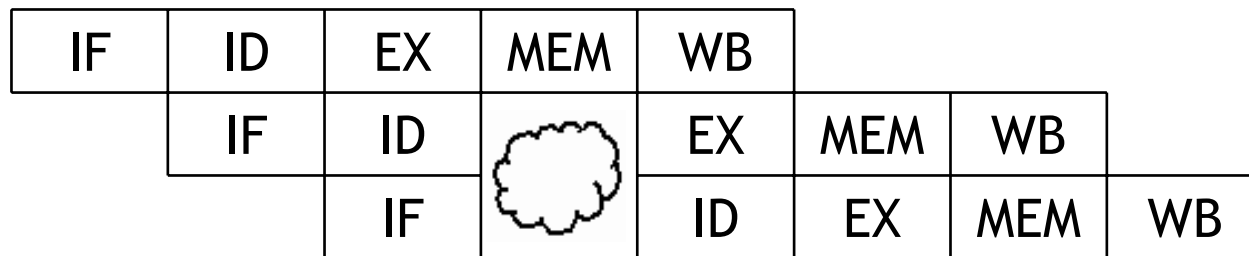
G4 superscalar architecture

- The PowerPC G4 has a total of eleven pipelined execution units.
 - There are four ALUs and one FPU for basic arithmetic operations.
 - A separate load/store unit manages memory accesses.
 - The AltiVec units support multimedia instructions (like MMX/SSE).
- The G4 is a **four-way superscalar** processor—the decoder can dispatch up to three instructions per cycle, as well as handle a branch instruction.

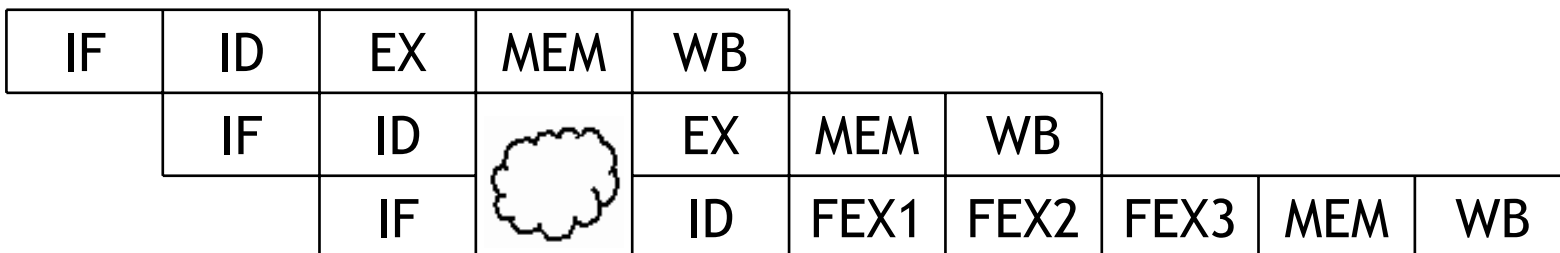


Stalls are even more dangerous here

- Remember that stalls delay all subsequent instructions in the pipeline.

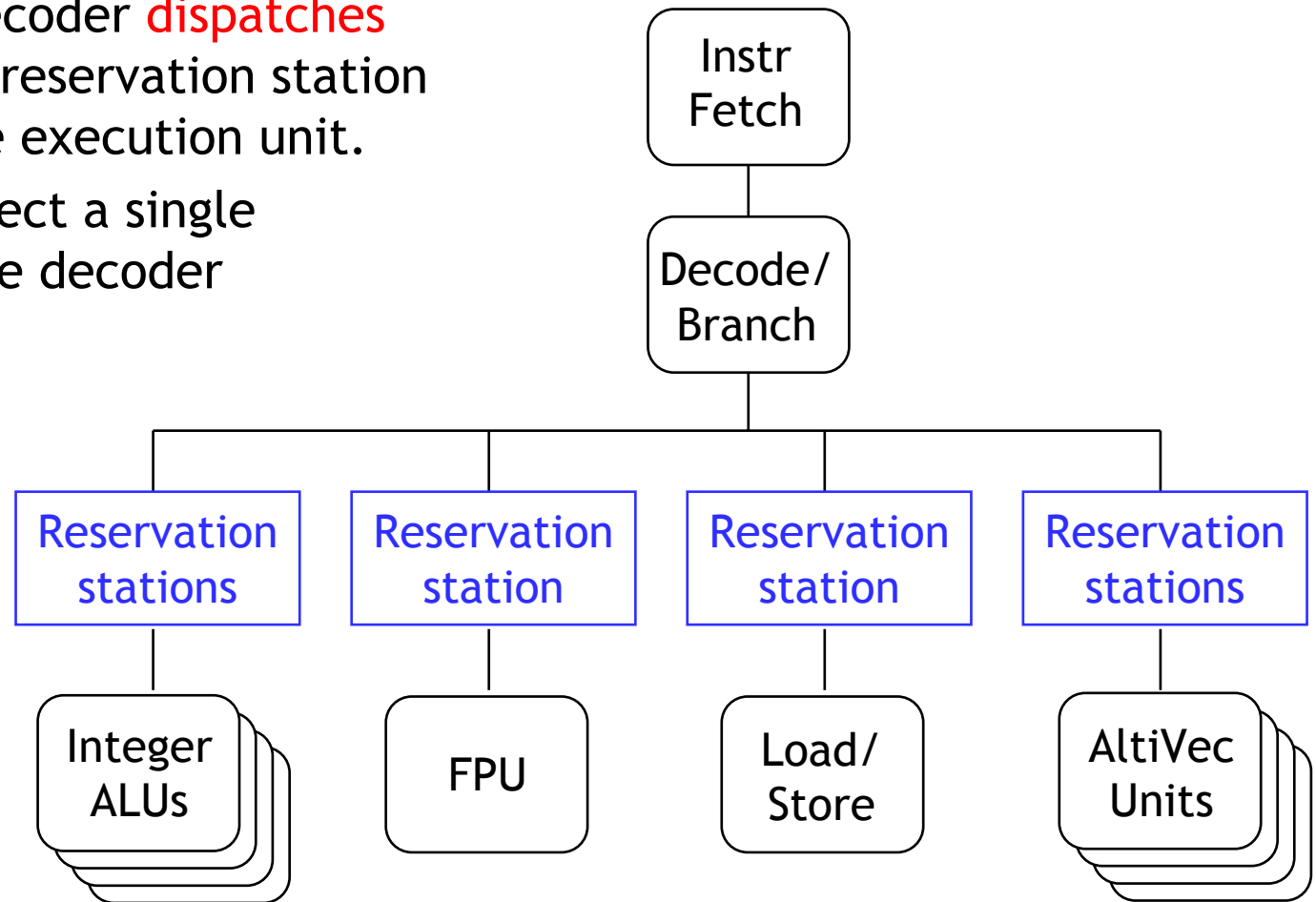


- Stalls are especially undesirable in a superscalar CPU because there could be *several* functional units sitting around doing nothing.
 - Below, when the integer ALU stalls the FPU must also remain idle.
 - The more functional units we have, the worse the problem.



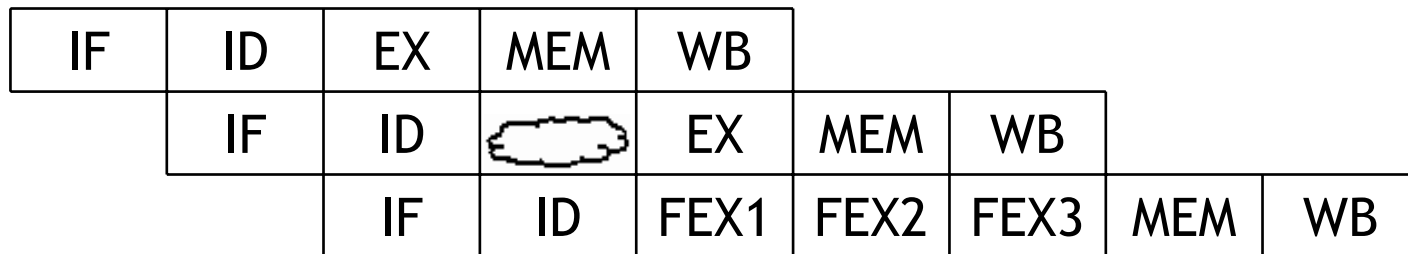
Dynamic pipelining

- One solution is to give each execution unit a **reservation station**, which holds an instruction until all of its operands become available, either from the register file or forwarding units.
- The instruction decoder **dispatches** instruction to the reservation station of the appropriate execution unit.
- Stalls will only affect a single execution unit; the decoder can continue dispatching *subsequent* instructions to other execution units.



Out-of-order execution

- If stalls only affect a single hardware element, the decoder can continue dispatching subsequent instructions to other execution units.



- But then a later instruction could finish executing before an earlier one!

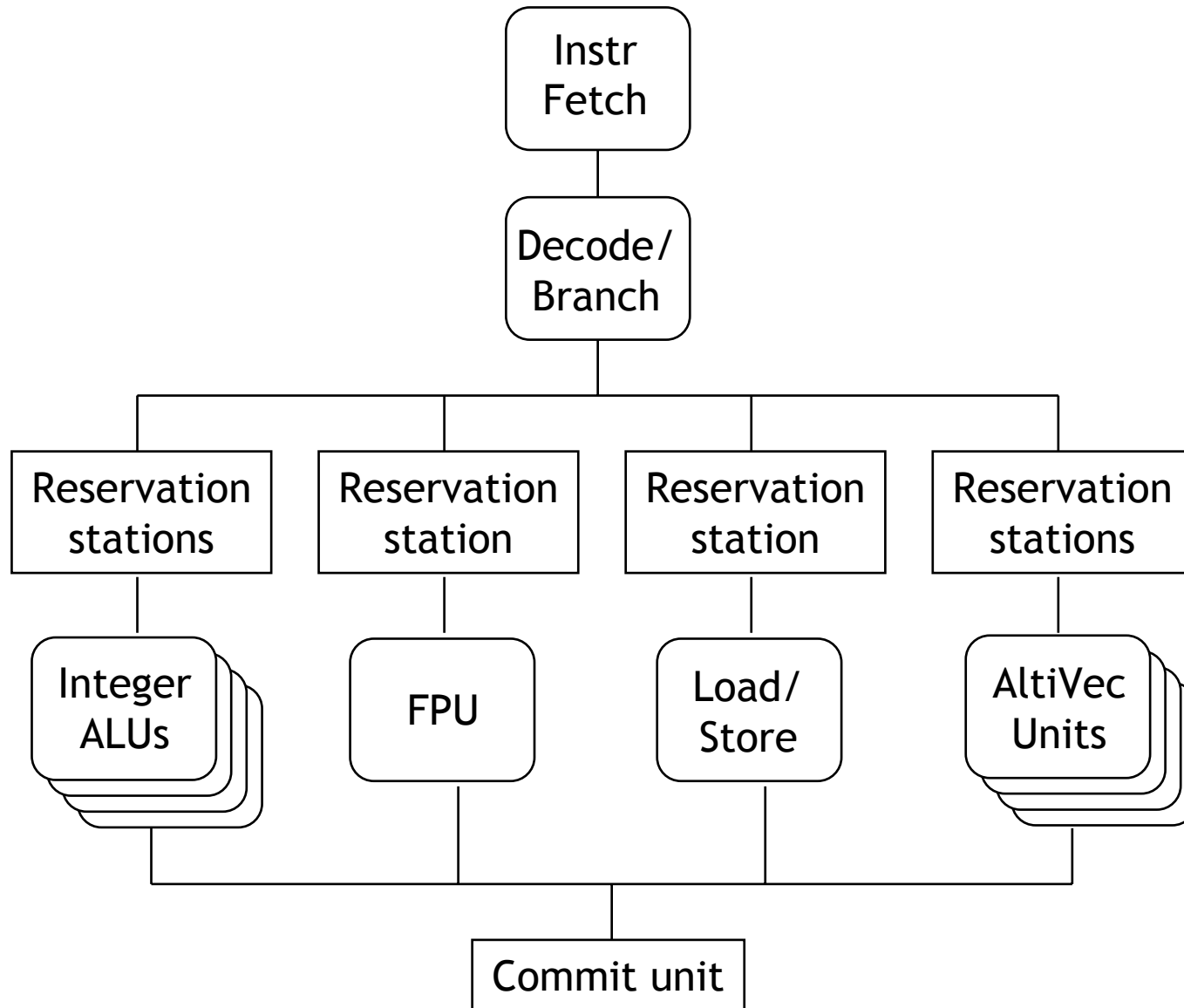
```
lw    $t0, 4($sp)
mul   $t0, $t0, $t0
add   $t0, $s2, $s3
```

- This can be troublesome for writing the correct destination registers as here, and for handling precise interrupts.

Reordering

- To prevent problems from out-of-order execution, instructions should not save their results to registers or memory immediately.
- Instead, instructions are sent to a **commit unit** after they finish their EX stage, along with a flag indicating whether or not an exception occurred.
- The commit unit ensures that instruction results are stored in the correct order, so later instructions will be held until earlier instructions finish.
 - Register writes, like on the last page, are handled correctly.
 - The flag helps implement precise exceptions. All instructions before the erroneous one will be committed, while any instructions after it will be flushed.
 - The commit unit also flushes instructions from mispredicted branches.
- The G4 can commit up to six instructions per cycle.

G4 block diagram



G4 summary

- The G4 achieves parallelism via both pipelining and a superscalar design.
 - There are eleven functional units, and up to four instructions can be dispatched per cycle.
 - A stall will only affect one of the functional units.
 - A commit buffer is needed to support out-of-order execution.
- The amount of work that can be parallelized, and the performance gains, will depend on the exact instruction sequence.
 - For example, the G4 has only one load/store unit, so it can't dispatch two load instructions in the same cycle.
 - As usual, stalls can occur due to data or control hazards.
- Problem 3 from today's homework demonstrates how rewriting code can improve performance.

Itanium introduction



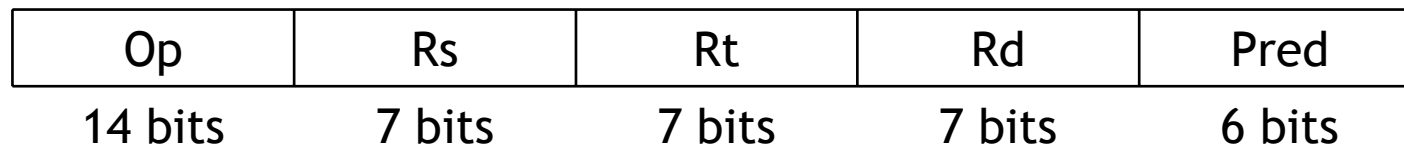
- Itanium began in 1993 as a joint project between Intel and HP.
 - It was meant as a study of very different ISA and CPU designs.
 - The first Itaniums, at 800MHz, appeared about two years ago.
 - Current performance is rumored to be poor, but it uses many new, aggressive techniques that may improve with better compilers and hardware designs.

Itanium basic architecture

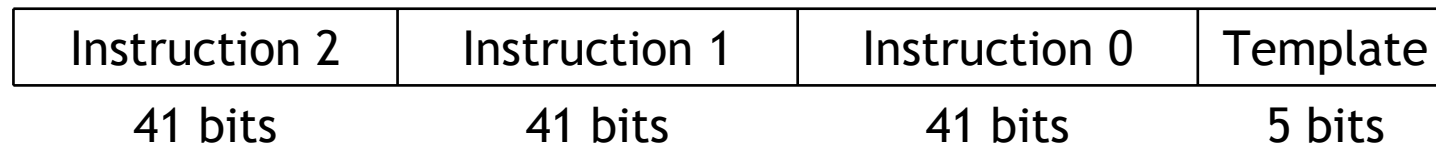
- Six instructions can be fetched and dispatched per cycle.
- To reduce the chance of structural hazards and hardware contention, the Itanium has a whopping seventeen execution units.
 - Four ALUs and four multimedia units, like the G4.
 - Two single-precision and two extended-precision floating-point units.
 - Three branch units handle multi-way branches (like “switch” in C++).
 - Two memory units.
- There are hundreds of 64-bit registers to support all of this hardware.
 - 128 general purpose registers.
 - 128 floating-point registers.
 - 128 registers for implementing a stack, instead of using slower RAM.
 - 8 branch registers for function calls.
 - 1 predicate register for conditional tests for branches.

VLIW instruction set architecture

- The Itanium uses a **VLIW** (Very Long Instruction Word) architecture.
- A single Itanium instruction is 41 bits long. Notice that with 128 registers, the register specifiers have to be 7 bits long.



- A 128-bit **bundle** contains up to three instructions and a “template” field.



- Each bundle contains only instructions that can be executed in parallel.

Explicit and implicit parallelism

- There are different kinds of parallelism!
 - The Itanium features **explicit parallelism**, since programs must contain information about what instructions can run in parallel (in the form of bundles).
 - Pipelining is sometimes referred to as **implicit parallelism** because programs can be pipelined without any special programmer actions (but compilers can help out a great deal).
- The Itanium still supports regular pipelining, but it's up to the compiler to generate bundles that take full advantage of the Itanium's features.
 - Making the compiler do more work is a common trend, from the first RISC processors to these VLIW architectures.
 - The Itanium is much more dependent upon compilers for generating good code than a processor like the G4.

Tradeoffs

- Having a VLIW compiler find parallelism has some advantages.
 - The compiler can take its time to analyze a program, whereas the CPU has just a few nanoseconds to execute each instruction.
 - A compiler can also examine and optimize an entire program, but a processor can only work on a few instructions at a time.
- Some things can still be done better by the CPU, since compilers can't account for dynamic events.
 - Compilers can't predict branch patterns accurately.
 - They can't adjust for delays due to cache misses or page faults.
- These issues illustrate a general tradeoff between compilation speed and execution speed.

Other VLIW architectures

- The [Transmeta Crusoe](#) processors emulate Intel 8086 code with low power consumption.



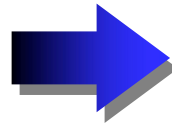
- The [Sony Playstation 2](#) graphics processor also has a VLIW architecture.



Branch predication

- Having a lot of hardware lets you do some interesting things.
- The Itanium can do **branch predication**: instead of guessing whether or not branches are taken, it just executes *both* branches simultaneously!

```
if (r1 == r2)
    r9 = r10 - r11;
else
    r5 = r6 + r7;
```

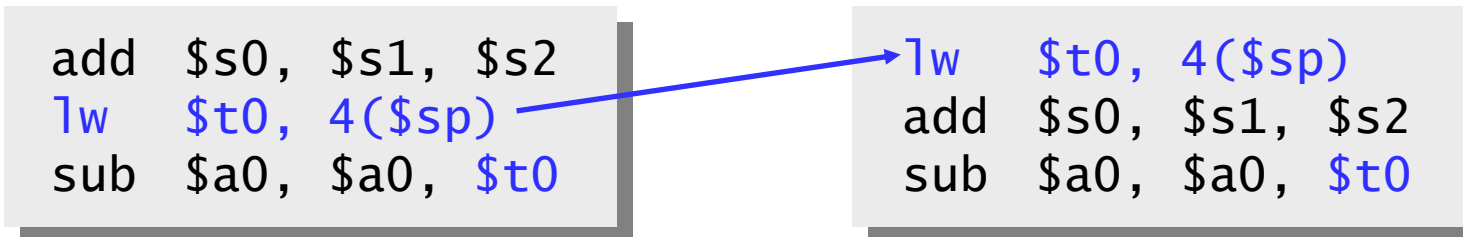


```
cmp.eq p1, p2 = r1, r2 ;;
(p1) sub r9 = r10, r11
(p2) add r5 = r6, r7
```

- The Itanium executes both the sub and add instructions immediately after the comparison. When the comparison result is known later, either sub or add will be flushed.
 - **p1** is a **predicate register** which will be true if $r1 = r2$, and **p2** is the complement of p1.
 - **(p1)** is a **guard**, ensuring that the sub is committed only if p1 is true.
- Predication won't always work (e.g., if there are structural hazards), so regular branch prediction must still be done.

Code motion

- One way to avoid memory-based stalls is to *move* the load instruction to a position before the data is needed.



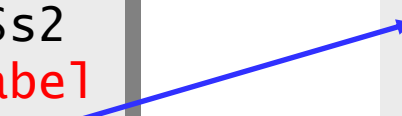
- These two code sequences are semantically equivalent.
- However, the sequence on the left would stall one cycle, whereas the one on the right performs an add in place of the stall.
- Good compilers can do this **code motion**.
 - Something like this also appears on the homework due today.
 - This is especially important in machines where loads might stall for more than one cycle.

Code motion across branches

- Itanium compilers can do code motion *across* branches and stores, while preserving precise exceptions.

```
add $s0, $s1, $s2
beq $v0, $0, Label1
lw $t0, 4($sp)
sub $a0, $a0, $t0
```

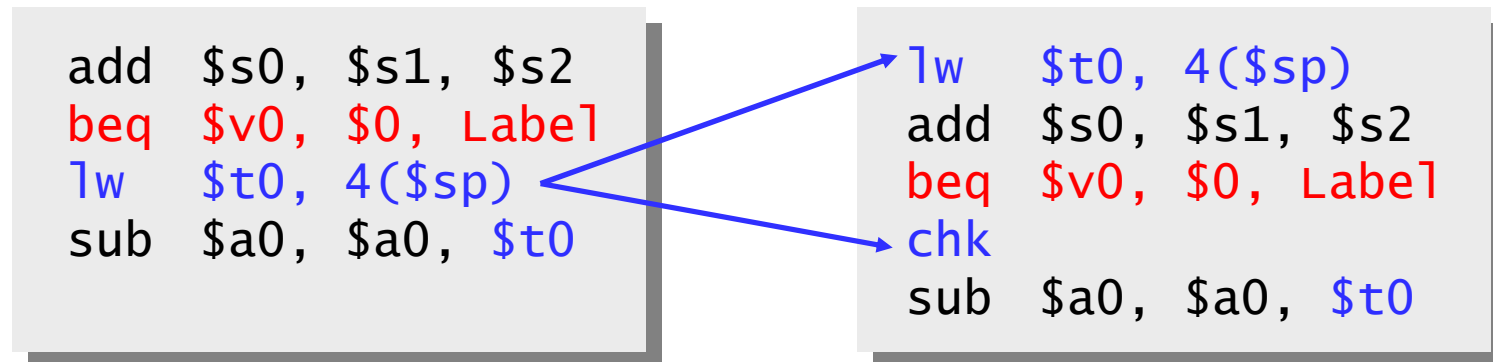
```
lw $t0, 4($sp)
add $s0, $s1, $s2
beq $v0, $0, Label1
sub $a0, $a0, $t0
```



- Ordinarily, this is problematic! For instance, assume that 4(\$sp) contains an invalid address.
 - If the branch is taken on the left, the load will not be executed and no exception will occur.
 - The code on the right tries to avoid a stall by executing the load first, but this would result in an exception!

Speculation

- In the Itanium, any potential exception from a load is *deferred*.
- The original load instruction is actually replaced by a deferred exception check instruction `chk`, as shown below.



- If the load on the right causes an exception, the destination register `$t0` will be flagged, but the exception is not raised until the `chk` instruction executes.
- This is most useful for architectures like the Itanium, where loads can stall for multiple cycles; the `chk` can be done in just one cycle instead.

Summary

- Modern processors try to do as much work in parallel as possible.
 - Superscalar processors can dispatch multiple instructions per cycle.
 - Lots of registers are needed to feed lots of functional units.
 - Memory bandwidths are large to support multiple data accesses.
- A lot of effort goes toward minimizing costly stalls due to dependencies.
 - Out-of-order execution reduces the impact of stalls, but at the cost of extra reservation stations and commit hardware.
 - Dedicated load/store units handle memory operations in parallel with other operations.
 - Control and data speculation help keep the Itanium pipeline full.
- Compilers are important for good performance. They can greatly help in maximizing CPU usage and minimizing the need for stalls.