# Compilers and HW

# VLIW (Very Long Instruction Word)

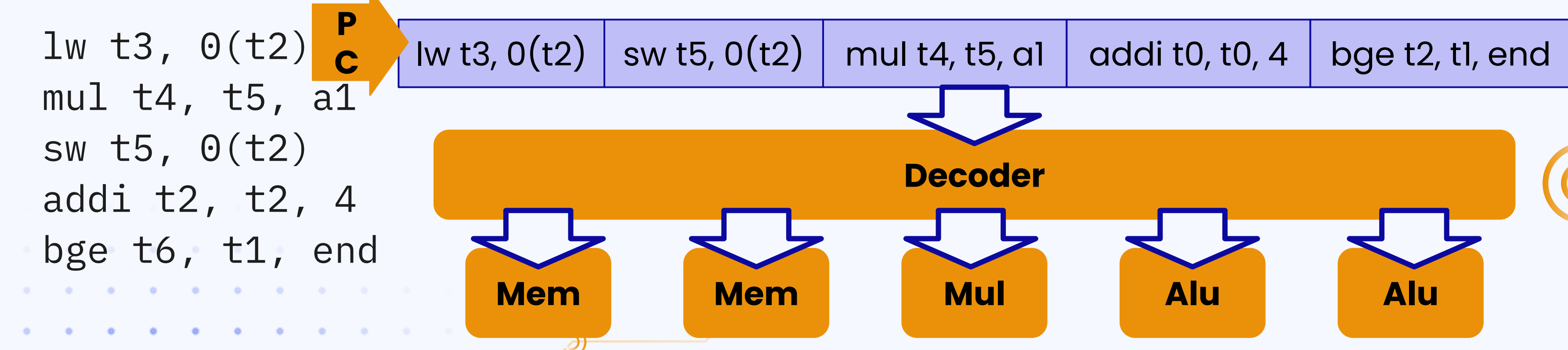
Compiler packs instructions into one long instruction word

Early VLIW: no dependences between instructions, units operate in lockstep

Pairs with loop unrolling, trace scheduling

Pros:

Cons:



# Follow up: loop dependence

```
for (int i = 0; i < 100; i++) {
    A[i + 1] = A[i] + C[i];
    B[i + 1] = B[i] + A[i + 1];
}
```

"loop carried dependence" cannot execute successive iterations in parallel

If we got rid of the loop carried dependence, we would need to make sure the two operations in the loop body are not reordered

# Follow up: loop dependence

Some seemingly dependent loops can be parallelized!

```
for (int i = 0; i < 100; i++) {
    A[i] = A[i] + B[i];
    B[i + 1] = C[i] + D[i]
}

A[0] = A[0] + B[0];
for (int i = 0; i < 99; i++) {
    B[i + 1] = C[i] + D[i];
    A[i + 1] = A[i + 1] + B[i + 1];
}
B[100] = C[99] + D[99]
```

# Follow up: software pipelining

```
lp: lw t2, 0(t1)
    addi t2, t2, 8
    sw t2, 0(t1)
    addi t1, t1, -4
    bne t1, t0, lp
```

```
lw t2, 0(t1)

addi t2, t2, 8
lw t3, -4(t1)

sw t2, 0(t1)
addi t3, t3, 8
lw t4, -8(t1)

addi -12(t1)
```

```
lp:
lw t2, 0(t1)
addi t4, t4, 8
sw t3, 8(t1)
addi t1, t1, -4
bne t1, t0, lp
```

```

addi t2, t2, 8
sw t3, 12(t1)

sw t2, t2, 4(t1)
```

What tradeoffs do you see between compiler scheduling and hardware (OOO/speculative) scheduling? Which do you like more? Do you think they can be combined?

# Why care about compilers?

HW techniques seem to rule the field: branch prediction, OOO, speculation…

BUT

- Understanding HW/SW tradeoffs and interactions is a useful exercise
- Some ISA features are designed to help compiler optimization
- Not every computer* has an Apple silicon or Intel chip

# Real-world Arduino compiler example (for Arm Cortex M0+)

```
while (i < N) {
   int f_next = f + f_prev;
   f_prev = f;
   f = f_next;
   i += 1;
```

```
cmp r4, r2
ble.n    2116 <setup+0x1a>
adds     r0, r1, r3
adds     r2, #1
movs     r1, r3
movs     r3, r0
b.n 2108 <setup+0xc>
```

1st Stage – Fetch and Pre-decode

2nd Stage – Decode and Simple Execute (DX)

Shift

ALU

Fetch and Pre-decode

Instruction Queue

Main decoder, data address generation

Read/write ports

# Conditional instructions

Some ISAs set condition codes as side effect of instruction

- overflow, zero, not zero, negative, etc

Often paired with branch instructions that don't have source registers

- ex: `cmp eax, 0; jne B` instead of `bne t1, x0`

Can sometimes be used in conjunction with non-branch instructions

- Conditional move (`CMOVcc`) in x86 will move value from memory or register to register based on condition (turn into a nop otherwise)

# Arm conditional execution

*source*

“Almost all ARM instructions can include an optional condition code. This is shown in syntax descriptions as {cond}. An instruction with a condition code is only executed if the condition code flags in the CPSR* meet the specified condition.”

“Almost all ARM data processing instructions can optionally update the condition code flags according to the result. To make an instruction update the flags, include the S suffix as shown in the syntax description for the instruction.”

*Current Program Status Register (holds condition flags)

# Arm conditional GCD example

*Source*

? ? ?

Do conditional instructions introduce hazards in a traditional pipelined processor?

# What about RISC-V?

(from the spec)

*The conditional branches were designed to include arithmetic comparison operations between two registers (as also done in PA-RISC and Xtensa ISA), rather than use condition codes (x86, ARM, SPARC, PowerPC), or to only compare one register against zero (Alpha, MIPS), or two registers only for equality (MIPS). This design was motivated by the observation that a*

*We considered but did not include conditional moves or predicated instructions, which c effectively replace unpredictable short forward branches. Conditional moves are the simpler the two, but are difficult to use with conditional code that might cause exceptions (memo accesses and floating-point operations). Predication adds additional flag state to a system, ad tional instructions to set and clear flags, and additional encoding overhead on every instructio Both conditional move and predicated instructions add complexity to out-of-order microarchit tures, adding an implicit third source operand due to the need to copy the original value of t destination architectural register into the renamed destination physical register if the predic is false. Also, static compile-time decisions to use predication instead of branches can res in lower performance on inputs not included in the compiler training set, especially given th unpredictable branches are rare, and becoming rarer as branch prediction techniques improve*

# ISA vs uArch

*We note that various microarchitectural techniques exist to dynamically convert unpredictable short forward branches into internally predicated code to avoid the cost of flushing pipelines on a branch mispredict [13, 17, 16] and have been implemented in commercial processors [27].*

*IBM Power-7 paper*

If the effect of a conditional branch is only to conditionally skip over a subsequent FX or LS instruction and the branch is highly unpredictable, POWER7 can often detect such a branch, remove it from the instruction pipeline, and conditionally execute the FX or LS instruction. The conditional branch is converted to an internal "resolve" operation, and the subsequent FX or LS instruction is made dependent on the resolve operation. When the condition is resolved, depending on the taken or not-taken determination of the condition, the FX or LS instruction is either executed or ignored. This may cause a delayed issue of the FX or LS instruction, but it prevents a potential pipeline flush due to a mispredicted branch.

# What if the compiler could speculate?

Would want to move speculated instrs before condition evaluation

Why? Might help VLIW scheduling or reducing pipeline hazards

Compiler needs to be able to find such instrs and move them without affecting correctness

We also need to:

Ignore exceptions in speculated execution

Be able to exchange stores and loads/stores

Cannot do this with *just* a compiler; need HW support!

# Example of compiler speculation

```
if X == 0; X = Y; else X += 4;
```

```
lw t1, 0(t0)
bne t1, x0, B1
lw t1, 0(s0)
j B2
B1: addi t1, t1, 4
B2: sw t1, 0(t0)
```

Assume branch is *almost never* taken (X=Y much more likely than X+=4)

What could go wrong if x != 0?
- Unnecessary page fault
- Y's address could be invalid (memory protection exception)

```
lw t1, 0(t0)
lw t3, 0(s0) # speculative load
beq t1, x0, B3
addi t3, t1, 4
B3: sw r3, 0(t0)
```

# Four approaches to exceptions:

1) OS returns undefined value instead of ending execution

   Works fine for correct programs, yields incorrect results for programs that will have real exceptions

2) ISA has speculative instructions (do not result in exceptions) + exception check instructions used after speculation is resolved
3) Track exceptions using “poison bits” on registers that only activate exception when value is used

   Need to mark which instructions are speculative

4) Hardware buffers speculative instructions (like a ROB-lite)

# What about reordering loads/stores?

```
lw t3, 0(t2)
...
sw t3, 0(t2)
lw t4, 4(t2)
...
sw t4, 4(t2)
```

```
lw t3, 0(t2)
lw t4, 4(t2)
...
...
sw t3, 0(t2)
sw t4, 4(t2)
```

Compiler can probably compute that this is fine to do

But what about this:

```
sw t3, 8(t2)
lw t4, 4(t1)
```

```
lw t4, 4(t1) # speculative
sw t3, 8(t2) # compare addr
guardian instr # fix if needed
```

# Takeaways

Compilers should at least be hardware-aware to make optimizations

Advanced compiler optimizations require hardware and/or OS support

Tradeoff between statically and dynamically scheduled instructions