

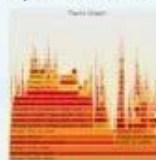
# Compiler-supported ILP

## STOP DOING Optimization

- Code was never meant to be optimized
- YEARS OF OPTIMIZING yet NO REAL-WORLD USE FOUND for BETTER PERFORMANCE
- Wanted to get better performance anyways? We had a tool for that, it was called "Upgrading hardware"
- "Yes please give me a low memory footprint. Please give me 5% CPU utilization" - Statements dreamed up by the utterly Deranged

Look at what Low-level programmers have been demanding your Respect for all this time, with all the RAM & CPU cores we built for them

(This is REAL optimizations, done by REAL programeers):



?????



???????

```
x2 = number * 0.5F;  
y = number;  
i = * ( long * ) &y;  
i = 0x5f3759df - ( i >> 1 );  
y = * ( float * ) &i;  
y = y * ( threehalfs - ( x2 * y * y ) );
```

????????????????

"I spent the entire week reducing the system latency by 2ms"

**They have played us for absolute fools**

# Macro-op fusion

source

Intel® Core™ Microarchitecture – Front End

Intel® Software College

## Instruction Decode / Macro-Fusion Presented

Read five Instructions from Instruction Queue

Send fusable pair to single decoder

Single uop represents two instructions

Example

```
for (unsigned int i=0; i<100000; i++)  
{  
    ...  
}
```

**Instruction Queue**

- add ecx, 1
- mov [mem1], ecx
- mov edx, [mem1]
- cmp eax, [mem2]
- jae label

**Cycle 1**

- add ecx, 1
- mov [mem1], ecx
- mov edx, [mem1]
- cmpj aje eax, [mem2], label

dec0  
dec1  
dec2  
dec3

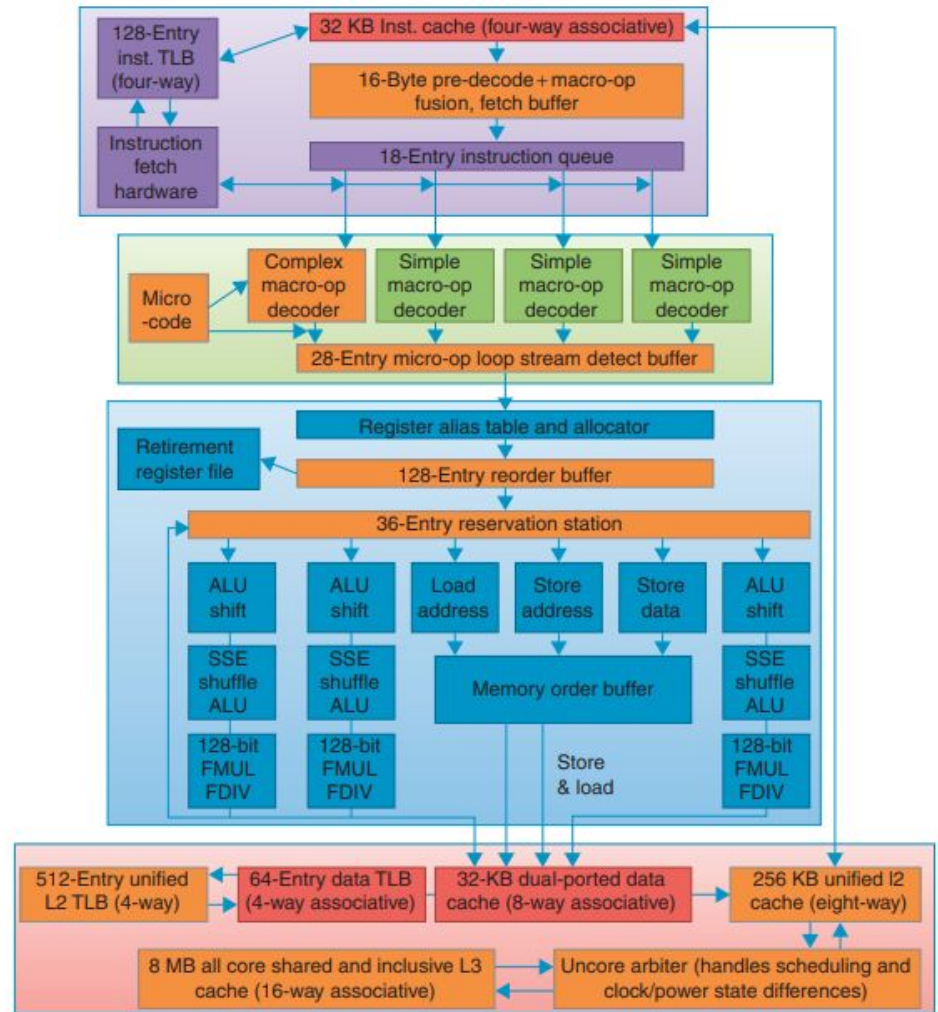
Intel® Processor Micro-architecture – Core®

26

Copyright © 2006, Intel Corporation. All rights reserved.  
Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States or other countries. \*Other brands and names are the property of their respective owners.

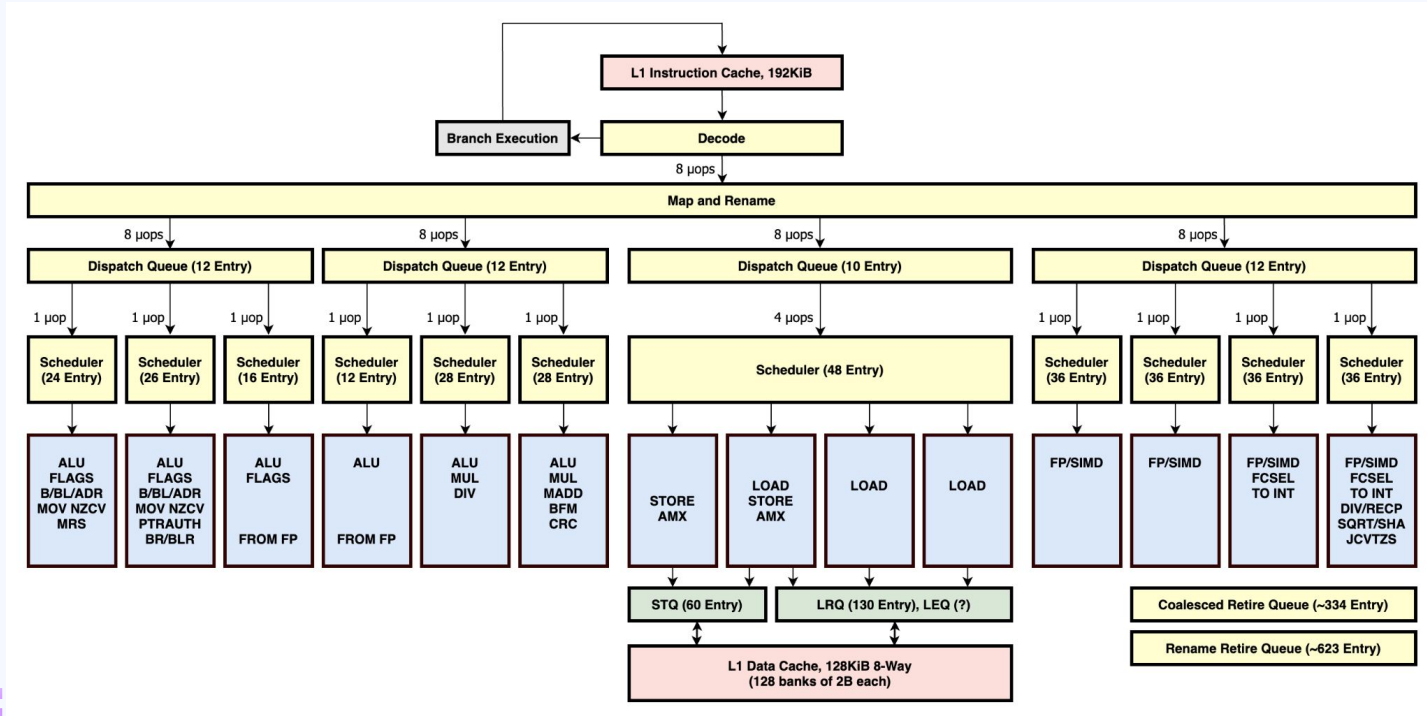
# Intel Core i7 (H&P fig. 3.41)

Pre-decode??  
Complex macro-op  
decoder??  
Loop stream detect??



# Firestorm (Apple M1)

Source (NOTE: reverse-engineered: might not be fully accurate)

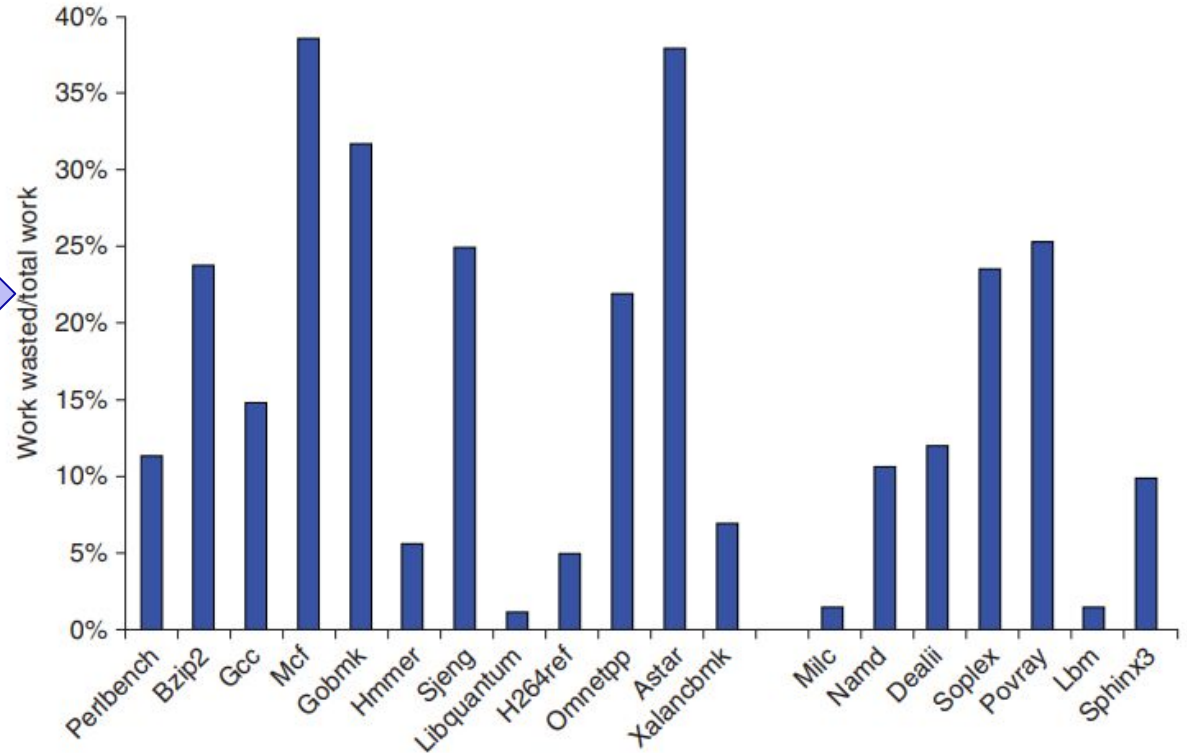




What performance metrics (beyond CPI)  
might become important in a speculative  
CPU?

# H&P fig. 3.42

% of executed  
uops that were  
not committed





What effects would speculative execution have  
on the memory system?  
(Hint: think protected access and/or caches)

# Remember this question?

?

?

?

What else could we parallelize here?

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



# A simpler example

```
addi t0, x0, 0    // t0/i = 0
addi t1, x0, 100  // t1 = 100
loop: bge t0, t1, end
slli t2, t0, 2    // t2 = t0/i * 4
add t2, a0, t2    // t2 = A + t2
lw t3, 0(t2)      // t3 = A[i]
add t3, t3, a1    // t3 = A[i] + c
sw t3, 0(t2)      // A[i] = A[i] + c
addi t0, t0, 1    // t0/i++
j loop
end: nop
```

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

# Reduce # of computations in loop

```
addi t0, x0, 0    // t0/i = 0
addi t1, x0, 100  // t1 = 100
loop: bge t0, t1, end
slli t2, t0, 2    // t2 = t0/i * 4
add t2, a0, t2    // t2 = A + t2
lw t3, 0(t2)      // t3 = A[i]
add t3, t3, a1     // t3 = A[i] + c
sw t3, 0(t2)      // A[i] = A[i] + c
addi t0, t0, 1    // t0/i++
j loop
end: nop
```

```
addi t0, x0, 0    // t0/i = 0
addi t1, x0, 100  // t1 = 100
addi t2, a0, 0    // t2 = A
loop: bge t0, t1, end
lw t3, 0(t2)      // t3 = A[i]
add t3, t3, a1     // t3 = A[i] + c
sw t3, 0(t2)      // A[i] = A[i] + c
addi t2, t2, 4    // advance pointer
addi t0, t0, 1    // t0/i++
j loop
end: nop
```

# Get rid of i

```
addi t0, x0, 0    // t0/i = 0
addi t1, x0, 100  // t1 = 100
addi t2, a0, 0    // t2 = A
loop: bge t0, t1, end
lw t3, 0(t2)      // t3 = A[i]
add t3, t3, a1     // t3 = A[i] + c
sw t3, 0(t2)      // A[i] = A[i] + c
addi t2, t2, 4    // advance pointer
addi t0, t0, 1    // t0/i++
j loop
end: nop
```

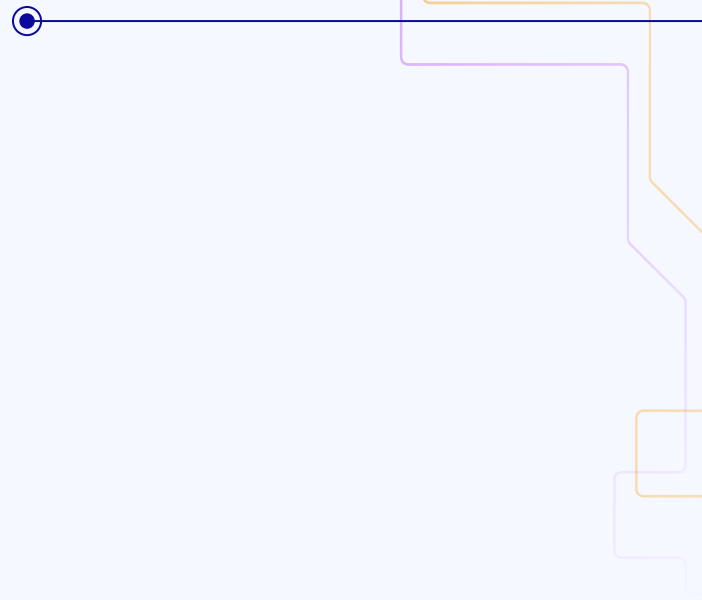
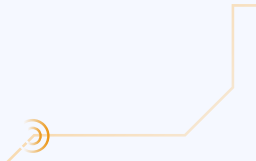
```
addi t2, a0, 0    // t2 = A
addi t1, a0, 400  // stop before A[100]
loop: bge t2, t1, end
lw t3, 0(t2)      // t3 = A[i]
add t3, t3, a1     // t3 = A[i] + c
sw t3, 0(t2)      // A[i] = A[i] + c
addi t2, t2, 4    // advance pointer
j loop
end: nop
```



What else could a compiler do?

```
addi t2, a0, 0 // t2 = A
addi t1, a0, 400 // stop before A[100]
loop: bge t2, t1, end
lw t3, 0(t2) // t3 = A[i]
add t3, t3, a1 // t3 = A[i] + c
sw t3, 0(t2) // A[i] = A[i] + c
addi t2, t2, 4 // advance pointer
j loop
end: nop
```

# Loop unrolling





What if we don't know the # of iterations  
through the loop at compile time?

```
int (i = 0; i < input_max; i++)
```

# Dynamic loop size

```
for (int i = 0; i < input_max % 4; i++) {
```

```
    // do loop body
```

```
}
```

```
for (int j = input_max % 4; j < input_max; j += 4) {
```

```
    // unrolled loop body (4x)
```

```
}
```



Besides compiler complexity, what is a  
downside to loop unrolling?  
(Could aggressive unrolling reduce  
performance?)



?

?

?

Could we unroll this loop?

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



Could we unroll this loop?

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

# Detecting loop dependences

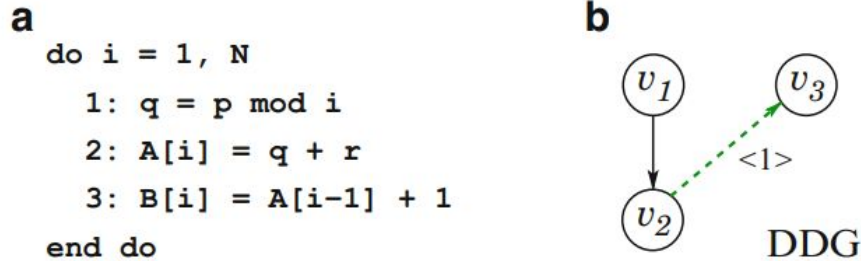


Figure 6.5 : Intra-iteration and loop carried dependences

From *Instruction Level Parallelism* by  
Aiken, Banerjee, Kejariwal, Nicolau

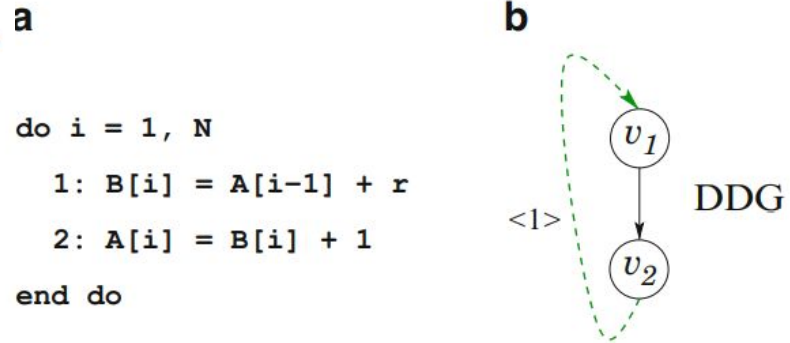


Figure 6.6 : Loop recurrences

# Other static ILP approaches

Software pipelining

Pipeline dependent instructions within a loop

Global scheduling

Move instructions across basic blocks

Trace scheduling

Find a trace (common path through program) of multiple basic blocks

Rearrange and parallelize instructions within trace

Need “compensation code” in case branching into/out of trace

# VLIW (Very Long Instruction Word)

Compiler packs instructions into one long instruction word

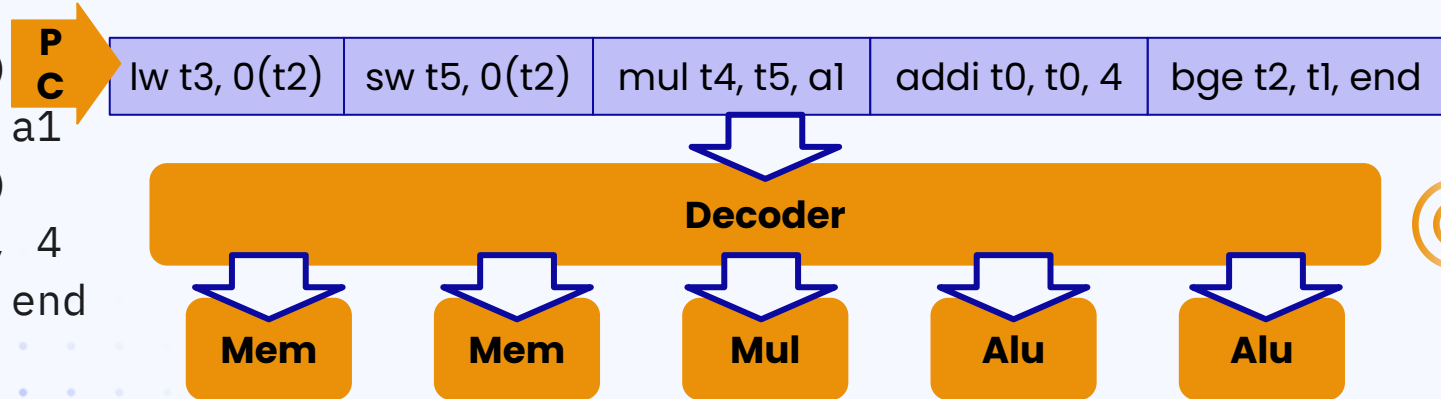
Early VLIW: no dependences between instructions, units operate in lockstep

Pairs with loop unrolling, trace scheduling

Pros:

Cons:

```
lw t3, 0(t2)
mul t4, t5, a1
sw t5, 0(t2)
addi t2, t2, 4
bge t6, t1, end
```





What tradeoffs do you see between dynamic and static ILP? Which do you like more? Can you imagine ways to combine them?