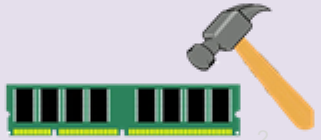
CSE 141: Introduction to Computer Architecture

Advanced Pipelines

Overview & Goals for the rest of the quarter

- **Parts I–V** went in-depth on core concepts
- **Parts VI and VII** are more about breadth
  - Expose you to a variety of concepts that you can go deep on
    - With more architecture classes! :D
- **Part VI (weeks 8-9): How to go even faster?**
  - *Guess (a lot) more* about what is going to happen
  - Find *more* opportunities for parallelism
- **Part VII (week 10): Oops… maybe we went too fast…**
  - Modern ISAs are *leaky abstractions*, they <u>only</u> specify abstract behavior
    - And people who understand the specifics can exploit that — what do we do??



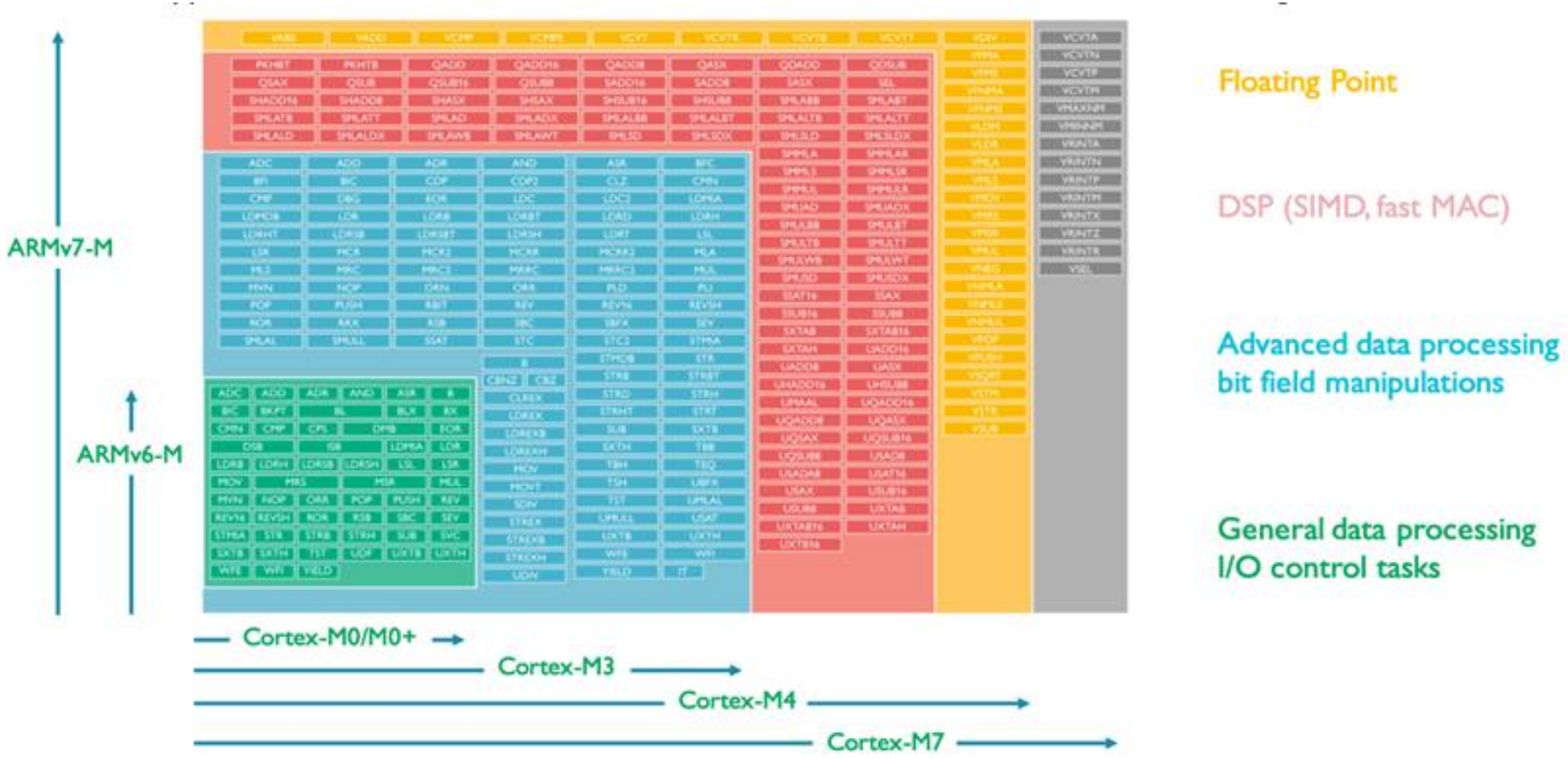
How much of a "real processor" have we implemented in 141 so far?

BEFORE WE DIVE INTO THE FANCY STUFF… DO WE NEED IT?

Acorn/Advanced RISC Machine (ARM) has three processor families

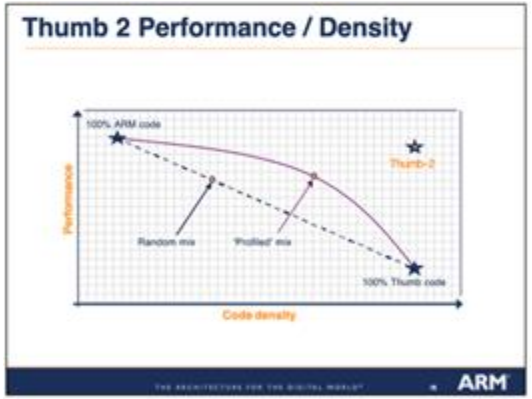
- Cortex A – "Application" processors
- Cortex R – "Real-Time" processors
- Cortex M – "Microcontroller" processors
  - (get it?)

The Cortex-M family exposes a wide tradeoff of capability and cost – measured mostly in $$, Joules, and die area


Floating Point
DSP (SIMD, fast MAC)
Advanced data processing bit field manipulations
General data processing I/O control tasks
ARMv7-M
ARMv6-M
Cortex-M0/M0+
Cortex-M3
Cortex-M4
Cortex-M7

Let’s look at the ARM Cortex-M3

- ISA: “Thumb2”, specifically ARMv7-M
  - Mixed 16/32-bit instructions [“hybrid length” instructions]
  - Compromise: many instructions can be compact, why waste bits? Still simple (just two cases)
- 3 stage, in-order, single issue pipeline
  - With single-cycle hardware multiply!
- It has a branch predictor…
  - It predicts Not Taken!
  - 2 cycle mis-predict penalty
- It has a 3-word prefetcher
  - Prefetchers help make unified memory designs fast
  - Q: How many instructions can prefetcher hold?



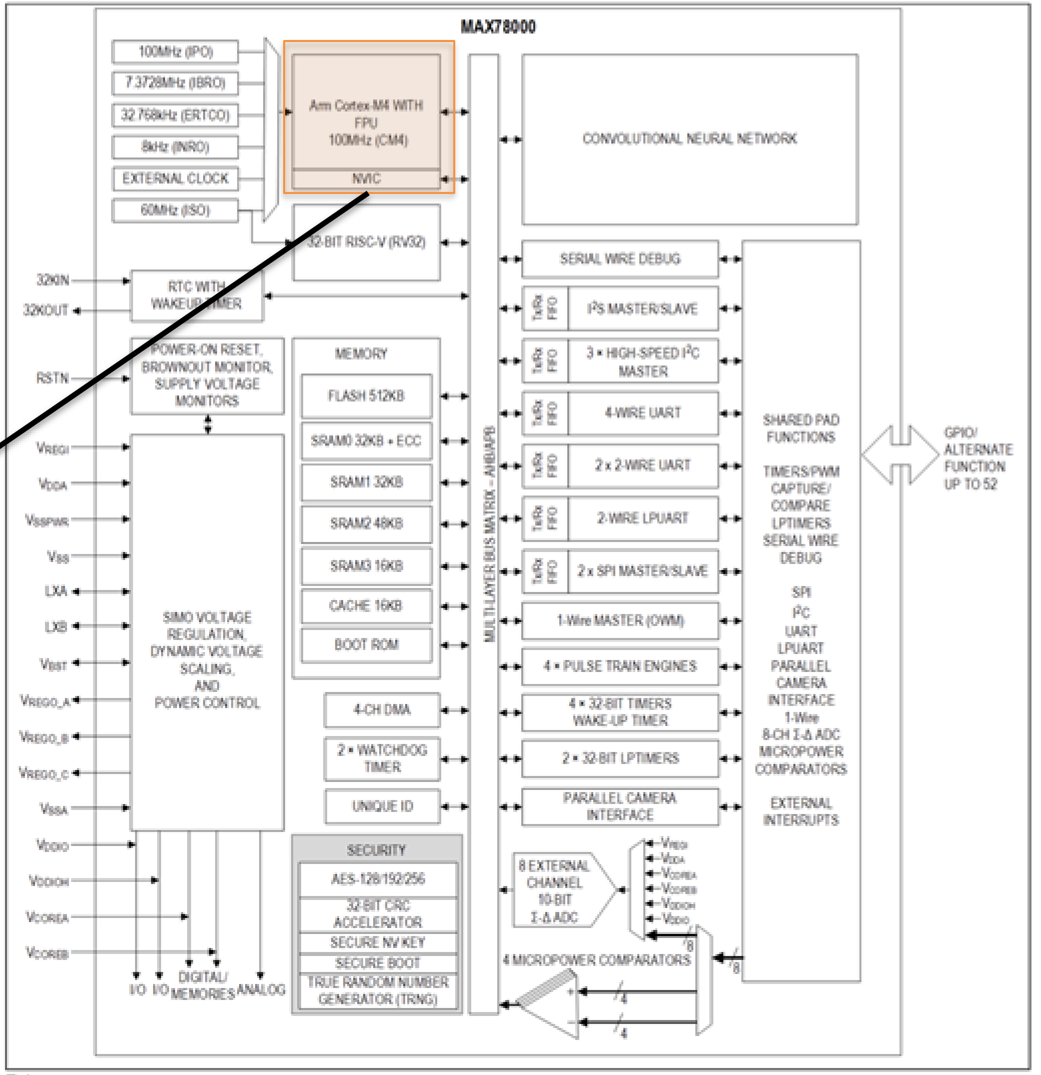
Implications of being area and energy constrained

- Performance / Watt >> than raw Performance
  - Latest designs are **5 µA/MHz** (this is the measure that matters for IoT!)
- Fewer general purpose registers (There are effectively 12)
  - Many of the smaller (16-bit) encodings can only access r0-r7
- Much slower core frequency (many in the 1-8 MHz, fastest M3's 48–200 MHz)
- Much simpler microarchitecture
  - In-order design
  - Limited parallelism
- Tightly coupled memory — No cache!
  - (well, a 3 word instruction cache)
  - **Just 1 cycle memory access penalty!**
    - i.e., `ldr` instruction takes 2 cycles, with no cache!
  - ***VERY* different than "applications" processors**
- Q: How might Amdahl's law explain tradeoffs in embedded MCUs?
  - Embedded processors are *duty cycled*, modern ones run ~0.1% of the time
  - **In embedded: Compute is not the bottleneck! New arch tradeoff opportunity!**

How is ML at the edge changing the edge?

- Hot new chip: [MAX78000]
  - 22 uA/MHz Cortex-M4
  - + RISC-V Co-Processor
  - + CNN accelerator
  - + *many* peripherals

In this whole chip, this part is the processor



There are many, many more deeply embedded processors than high performance general purpose processors

- 0% of processors in the world are “high-performance” processors
  - Seriously, the number of Intel Core XXX and AMD XXX are a *rounding error* compared to AVR, MIPS (yes, our MIPS), PIC, ARM Cortex M’s, etc
- **So why do we talk about the fancy machines?**

There are many, many more deeply embedded processors than high performance general purpose processors

- 0% of processors in the world are “high-performance” processors
  - Seriously, the number of Intel Core XXX and AMD XXX are a *rounding error* compared to AVR, MIPS (yes, our MIPS), PIC, ARM Cortex M’s, etc
- **So why do we talk about the fancy machines?**
  - **Thought experiment: Which gives you the most aggregate processing power:**
  - (Very) Coarse estimate: 1 trillion PIC-8’s in the world
    - Say, average 50 MHz, CPI of ~20 [for 32-bit math]
  - (Very) Coarse estimate: 120 billion ARM Cortex-M’s in the world
    - Say, average 24 MHz, CPI of ~1.25
  - (Very) Coarse estimate: 1 billion Intel Core i7’s in the world
    - Say, average 4 GHz, CPI of ~0.25

Okay, that was a fun aside, but show me the fancy fast stuff

- Step 1: When in doubt: **Don’t stop**, guess and guess well

Going faster by guessing **if** and and **where** we’re branching

BRANCH PREDICTORS

Dealing With Branch Hazards
(what we did earlier)

- Hardware
  - stall until you know which direction
  - reduce hazard through earlier computation of branch direction
  - guess which direction
    - assume not taken (easiest)
    - more educated guess based on history
      - (requires that you know it is a branch before it is even decoded!)
- Hardware/Software
  - nops
  - instructions that get executed either way (delayed branch).

Dealing With Branch Hazards
(How pretty much everything does it nowadays)

- Hardware
  - stall until you know which direction
  - reduce hazard through earlier computation of branch direction
  - guess which direction
    - assume not taken (easiest)
    - more educated guess based on history
      - (requires that you know it is a branch before it is even decoded!)

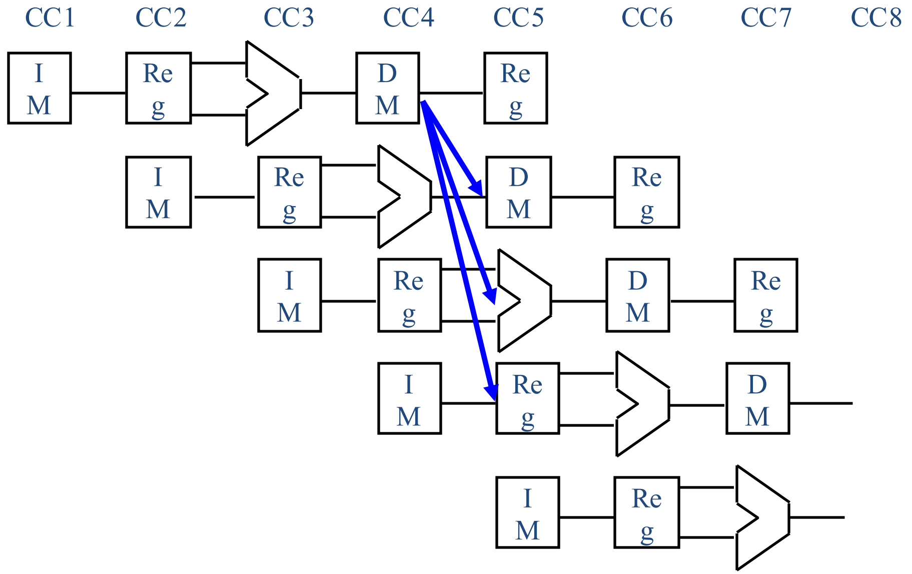
Now we'll do it this way!

- Hardware/Software
  - nops
  - instructions that get executed either way (delayed branch)

Assume Branch *Not Taken*

- works pretty well when you're right!

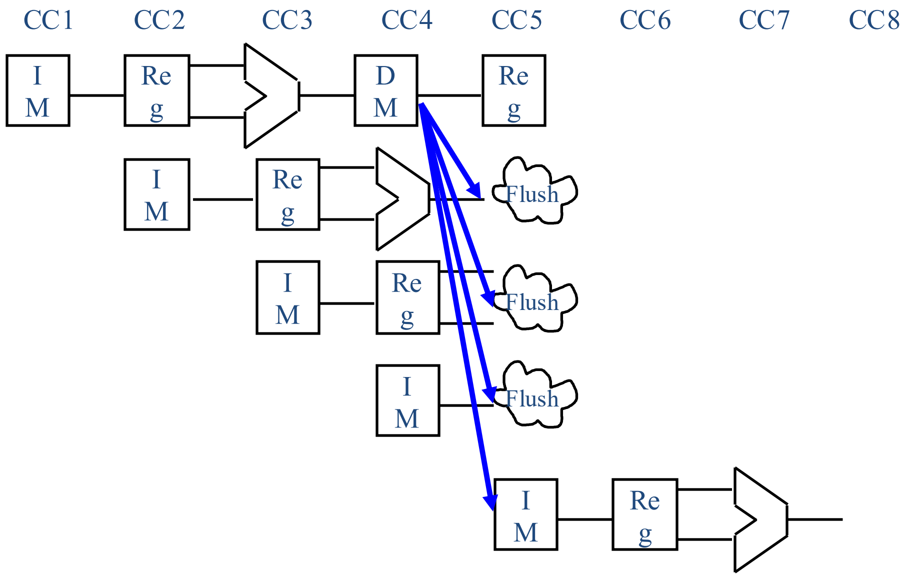
```
beq $4, $0, there

and $12, $2, $5

or ...

add ...

sw ...
```



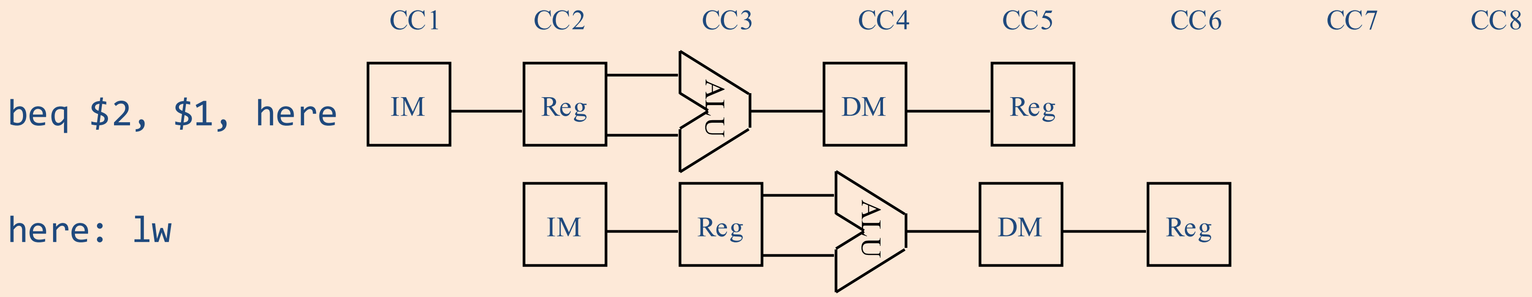
Assume Branch *Not Taken*

- *same performance as stalling* when you're wrong

```
beq $4, $0, there

and $12, $2, $5

or ...

add ...

there: sub $12, $4, $2
```



Branch Hazards – What if we predict taken instead?



Required information to predict Taken:

1. Whether an instruction is a branch (before decode)
2. The target of the branch
3. The actual outcome of the branch condition

	Required knowledge
A	2,3
B	1,2,3
C	1,2
D	2
E	None of the above

Branch Target Buffer (BTB)

aka, how to know it's a branch before you know it's a branch

- Keeps track of the PCs of recently seen branches and their targets.
- Consult during Fetch (in parallel with Instruction Memory read) to determine:
  - Is this a branch?
  - If so, what is the target

Branch Target Buffer (BTB)

aka, how to know it's a branch before you know it's a branch

- Keeps track of the PCs of recently seen branches and their targets.
- Consult during Fetch (in parallel with Instruction Memory read) to determine:
  - Is this a branch?
  - If so, what is the target

- **For the moment…**
  - Just remember that we need one of these to do all this prediction stuff
  - The BTB is taking care of the "only do prediction-y things if it's a branch"
  - We'll come back to how the BTB works in a bit

Branch Prediction

- Always assuming a branch is not taken is a crude form of *branch prediction*.
- What about loops that are *taken* 95% of the time?
  - we would like the option of assuming *not taken* for some branches, and assuming *taken* for others, depending on ???

Branch Prediction

- Historically, two broad classes of branch predictors:
  - **Static predictors** – for branch B, always make the same prediction.
  - **Dynamic predictors** – for branch B, make a new prediction every time the branch is fetched
- Tradeoffs?
- Modern CPUs all have sophisticated dynamic branch prediction.

Static Predictors

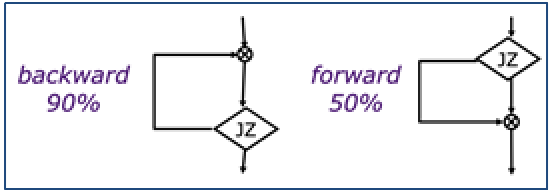
Decision about predicted direction <u>does not</u> depend on current machine state

- Always Not Taken
- Always Taken
- ________________
  - Hint: Simple rule for which branches likely always vs. never taken
- ________________
  - Hint: Compiler/ISA-supported approach, for programs you've run before

Static Predictors

Good enough??

- ______________
  - Hint: Simple rule for which branches likely always vs. never taken
- ______________
  - Hint: Compiler/ISA-supported approach, for programs you've run before

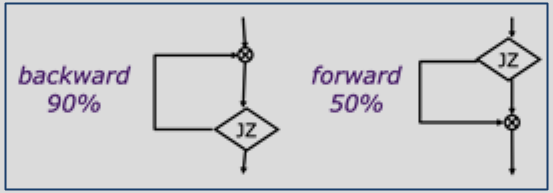


e.g., HP PA-RISC, Intel IA-64 typically reported compiler-selected as **~80% accurate**

Static Predictors

Good enough??

- ________________
  - Hint: Simple rule for which branches likely always vs. never taken
- ________________
  - Hint: Compiler/ISA-supported approach, for programs you've run before



e.g., HP PA-RISC, Intel IA-64 typically reported compiler-selected as **~80% accurate**

UltraSPARC-III
(Turn of the century machine, 1999)

◦ Mispredicted branch penalty of 6 cycles
◦ 4-wide issue

Branches are 15-20% of instructions…

Wasted work per branch @ 80% accuracy:

20% misprediction rate
× 6 stages
× 4-way issue
= **4.8 wasted instructions / branch**

Static Predictors

Are not nearly good enough!

- **State of the art predictors achieve 95–99.X% accuracy**
  - Note: As-always, this is highly workload dependent!

UltraSPARC-III
(Turn of the century machine, 1999)
◦ Mispredicted branch penalty of 6 cycles
◦ 4-wide issue

Branches are 15-20% of instructions…

Wasted work per branch @ 80% accuracy:

20% misprediction rate
× 6 stages
× 4-way issue
= **4.8 wasted instructions / branch**

Aside: Happening this year—
6th Championship Branch Prediction (CBP2025)

- https://ericrotenberg.wordpress.ncsu.edu/cbp2025/
  - Open-to-all competition to design the best branch predictor
    - Subject to realistic constraints, etc
- Last competition was in 2016!
  - Results: https://jilp.org/cbp2016/program.html
    - `For  8KB size:       4.2 MPKI  [~95.8% accurate]`
    - `For 64KB size:       3.3 MPKI  [~96.7% accurate]`
    - `  “unlimited”:  2.2~2.3 MPKI  [~97.8% accurate]`
  - These are on some of the “hardest to predict” workloads
    - On more regular workloads, 99.5%+ accuracy is commonplace

Dynamic Branch Prediction

- What information is available to make an intelligent prediction?

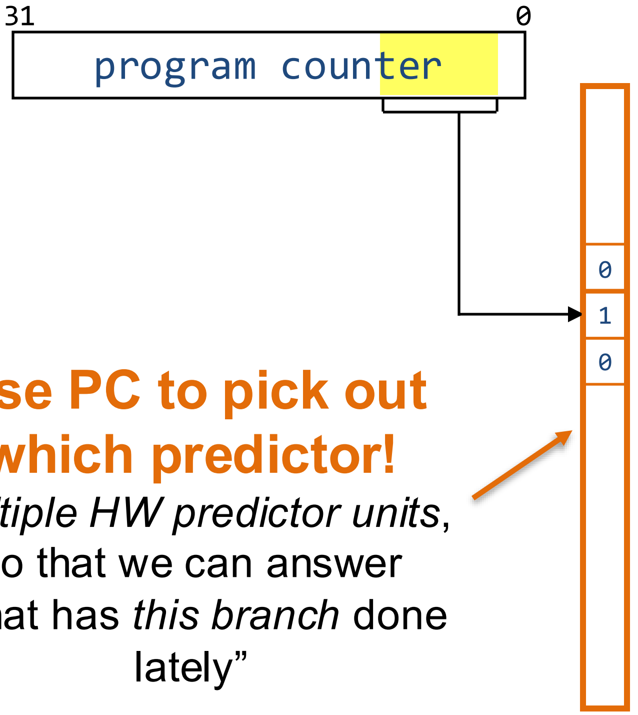
Branch Prediction: Simplest 1-bit predictor

```
for (i=0; i<10; i++) {
  ...
  ...
}
```

↓

```
...
...
add  $i, $i, #1
bne $i, #10, loop
```

Programs have many branches… usually need lots of predictors



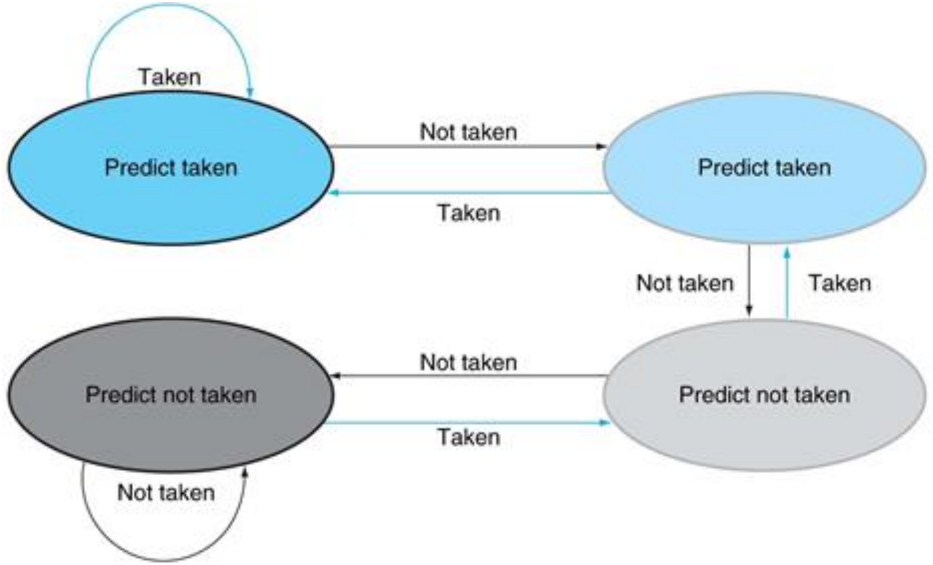
Use PC to pick out which predictor!

Multiple HW predictor units, so that we can answer "what has *this branch* done lately"

```
for (i=0; i<10; i++) {
   ...
   ...
}
```

```
...
...
add  $i, $i, #1
bne $i, #10, loop
```

Two-bit predictors give better loop prediction



This state machine also referred to as a *saturating counter*. It counts down (on *not takens*) to 00 or up (on *takens*) to 11, but does not wrap around.

```
for (i=0; i<10; i++) {
   ...
   ...
}
```

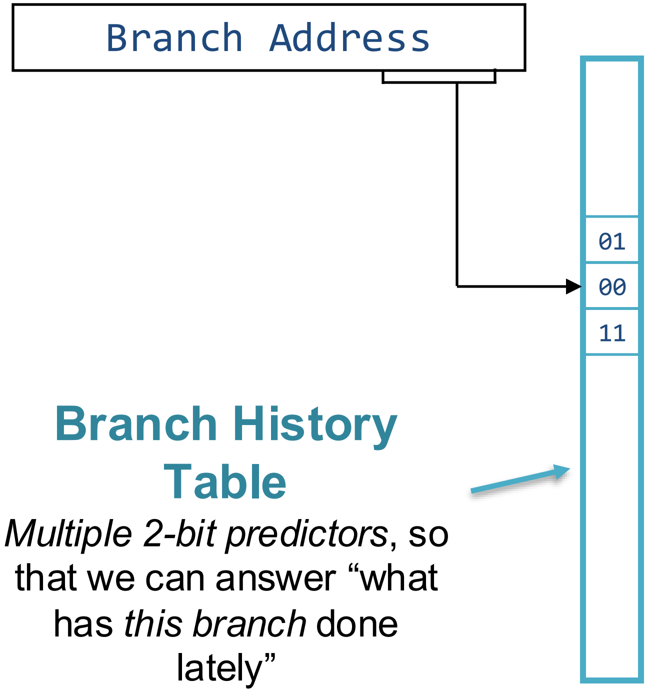
↓

```
...
...
add  $i, $i, #1
bne $i, #10, loop
```

Branch History Table
first introduced by the "[2-bit] bimodal predictor"*

- has limited size
- 2 bits by N (e.g. 4K)
- uses low bits of branch address to choose entry

- what about even/odd branch?



Branch History Table

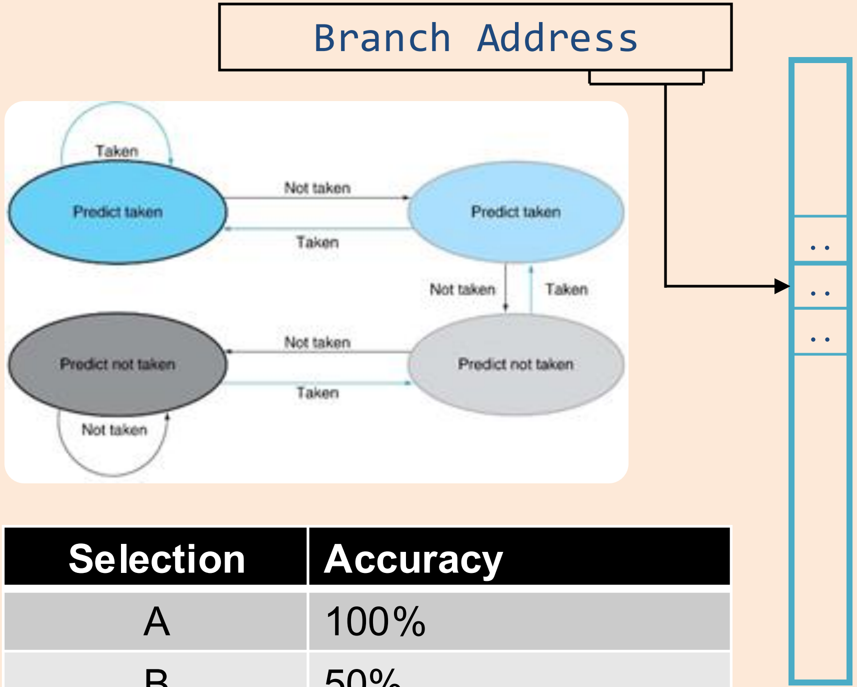
Multiple 2-bit predictors, so that we can answer "what has *this branch* done lately"

*For those who like the history, we're roughly around 1981 at this point: *A Study of Branch Prediction Strategies*, James E. Smith

2-bit bimodal predictor

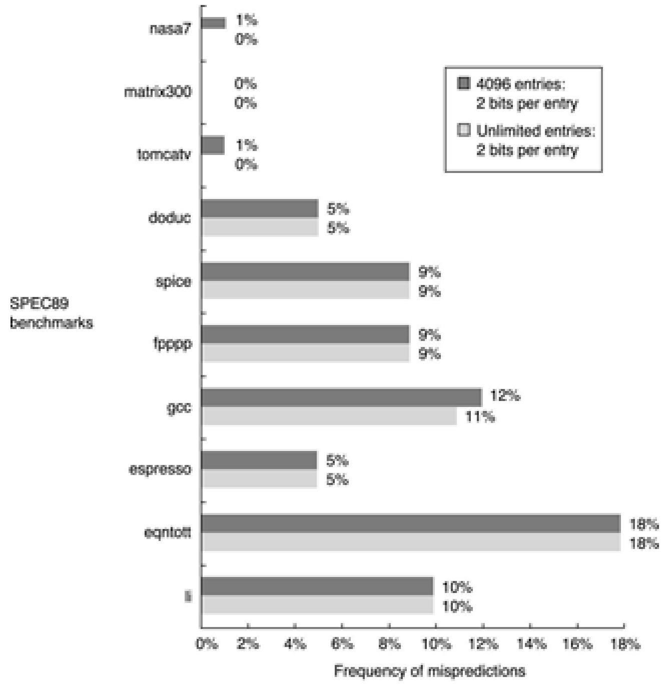
- For the following loop, what will be the **prediction accuracy** of the bimodal predictor for the **conditional branch that closes the loop**?

```
for (i=0; i< 2; i++)
  // loop runs twice per call
  {  z = …  }
```



Selection	Accuracy
A	100%
B	50%
C	0%
D	Maybe 0%, maybe 50%
E	other

2-bit bimodal misprediction rates



Is this good enough?

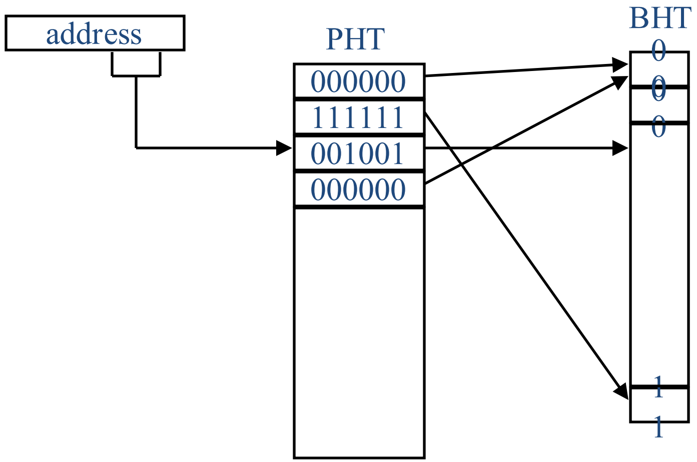
Can We Do Better?

- Can we get more information dynamically than just the recent bias of this branch?

Can We Do Better?

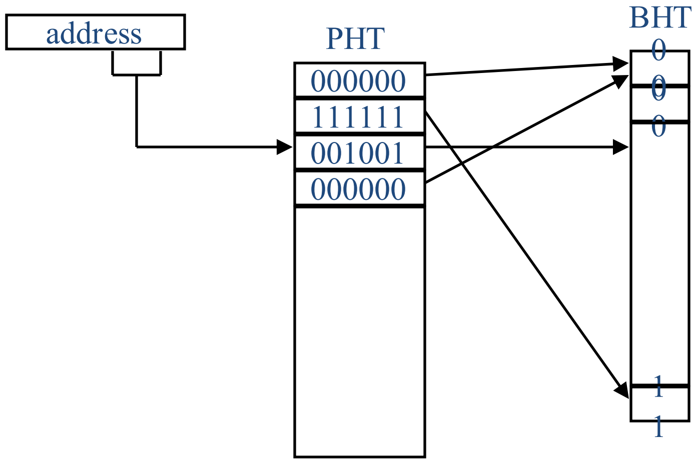
Yes: 2-level local predictor

- Can we get more information dynamically than just the recent bias of this branch?
- We can look at patterns (2-level *local* predictor) for a particular branch.
  - last eight branches 00100100, then it is a good guess that the next one is "1" (taken)



Can We Do Better?

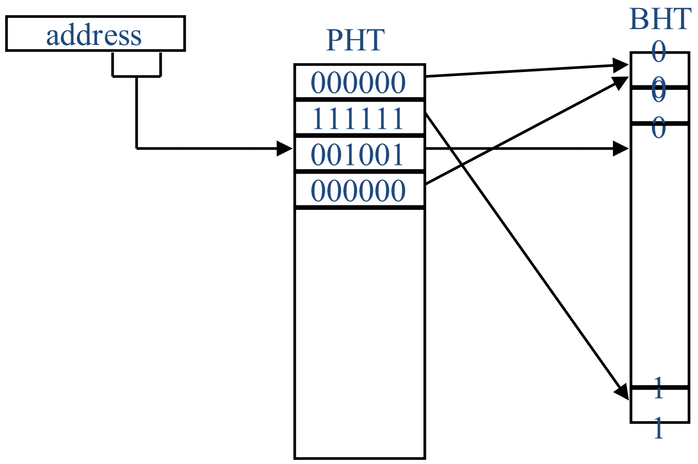
Yes: 2-level local predictor



- "2-level" → Two tables
- *Pattern History Table (PHT)*
  - Indexed by PC (branch address)
  - Width ~= Pattern Complexity
- *Branch History Table (BHT)*
  - Indexed by *pattern*
  - Same structure as used in the 2-bit bimodal, but different meaning!
  - No longer "what is this branch likely to do next", now, "what is likely to come next in this pattern"

Can We Do Better?

Yes: 2-level local predictor



- even / odd branch?

Can We Do Better?

- Can we get more information dynamically than just the recent bias of this branch?

Can We Do Better?

Yes: Correlating Predictor

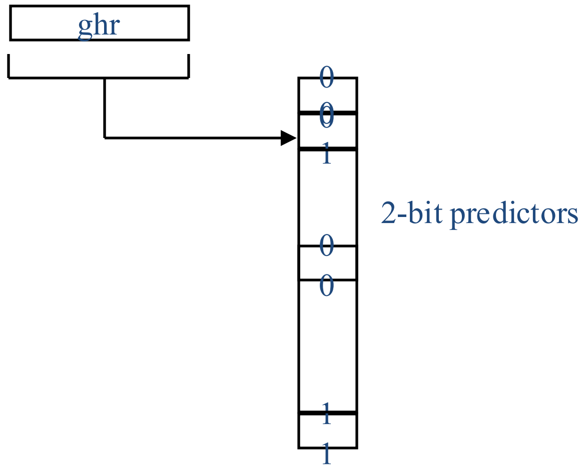
- Can we get more information dynamically than just the recent bias of this branch?
- *Correlating Branch Predictors* also look at other branches for clues

```
if (i == 0)
  ...
if (i > 7)
  ...
```

- Typically use two indexes
  - Global history register --> history of last m branches (e.g., 0100011)
  - branch address

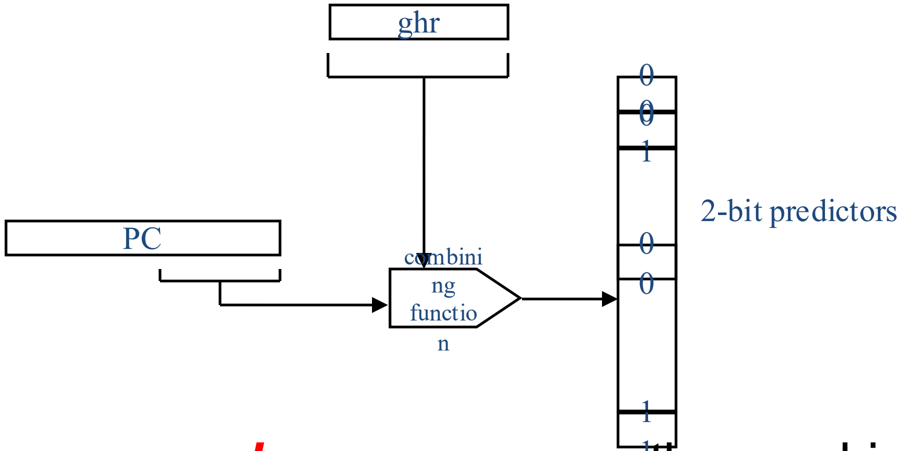
Correlating Branch Predictors

- The *global history register (ghr)* is a shift register that records the last *n* branches (of any address) encountered by the processor.
  - ”What does the pattern of *recent branching done* tell me?”



Two-level correlating branch predictors*

- Can use both the PC address and the GHR

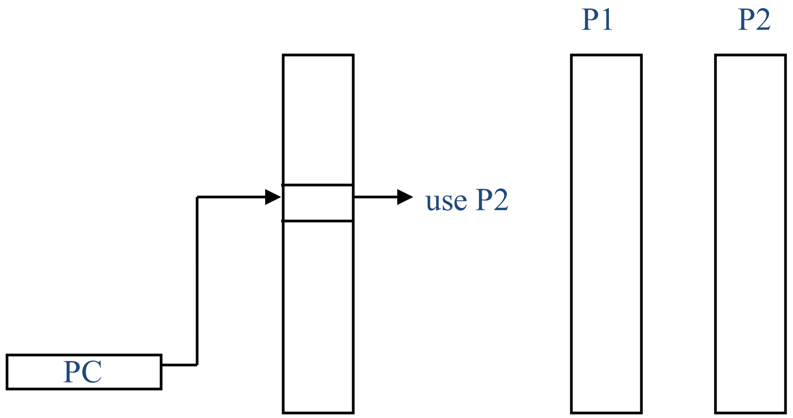


- Most common – ***gshare***: use xor as the combining function.

*For those who like the history, we're around 1993 now: McFarling, "Combining Branch Predictors," DEC WRL TR 1993

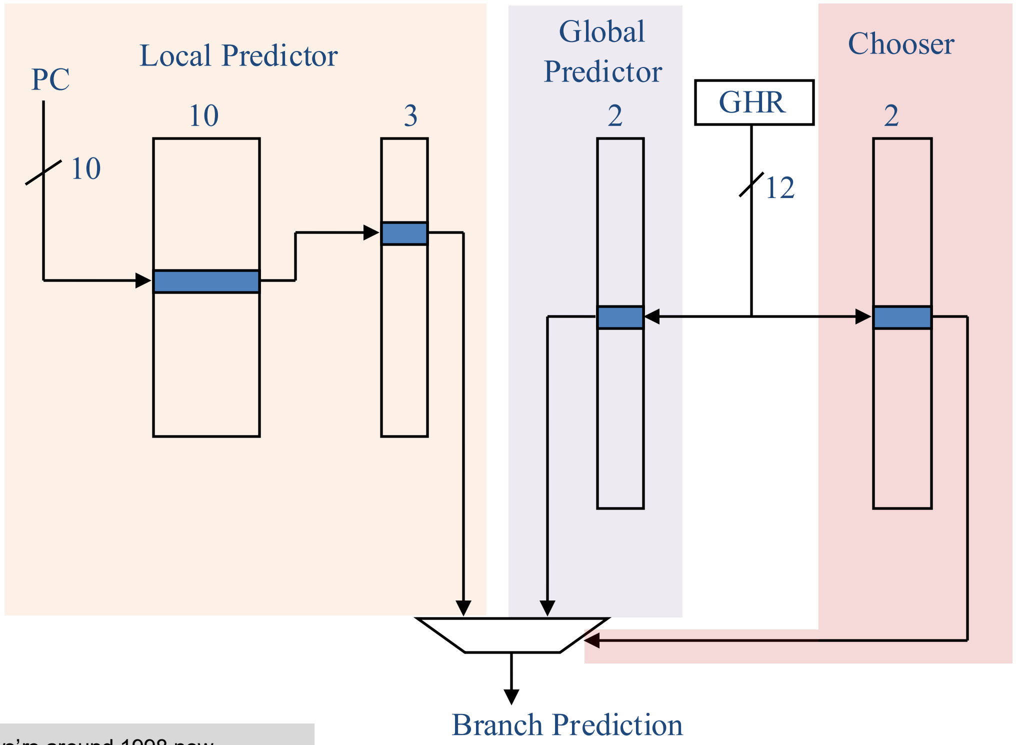
Are we happy yet????

- ***Combining choice branch predictors*** use multiple schemes and a voter to decide which predictor typically does better for *that* branch.



Compaq/Digital Alpha 21264*

Example of a real-world *combining choice branch predictor*



*For those who like the history, we're around 1998 now

Aliasing in Branch Predictors

- Branch predictors will always be of finite size, while code size is relatively unlimited.

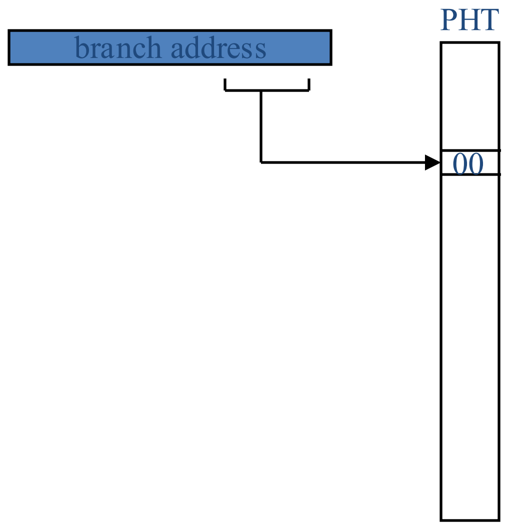
Aliasing in Branch Predictors

- Branch predictors will always be of finite size, while code size is relatively unlimited.
- What happens when (in the common case) there are more branches than entries in the branch predictor?

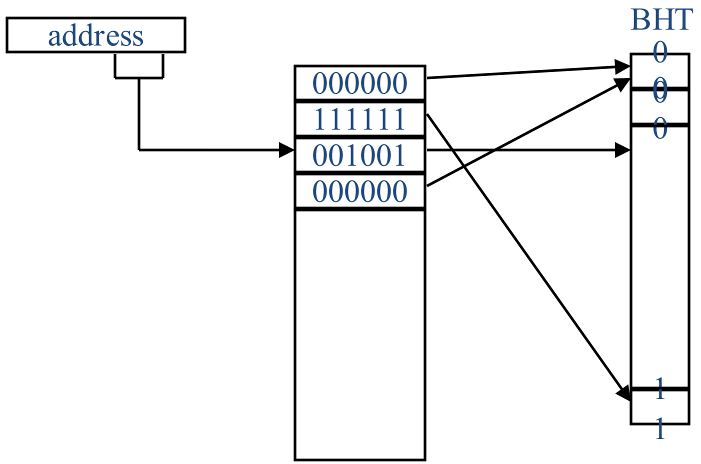
Aliasing in Branch Predictors

- Branch predictors will always be of finite size, while code size is relatively unlimited.
- What happens when (in the common case) there are more branches than entries in the branch predictor?
- We call these conflicts *aliasing*.
- In practice, we can have…
  - **negative aliasing** (when biases are different)
  - or **neutral aliasing** (biases same)
  - but positive aliasing is unlikely

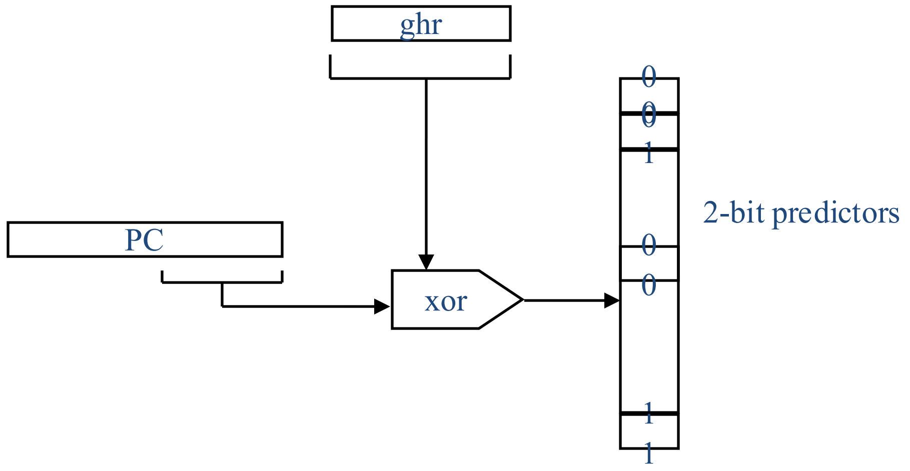
Bimodal aliasing



Local Predictor Aliasing



Gshare aliasing



How Close are We to Modern Branch Prediction?

Everything presented in detail in 141 is pre-2000 (!)

- Latest branch predictors significantly more sophisticated
- Use more advanced correlating techniques, larger structures, and even AI
  - Neural-network-based, “perceptron” branch predictors
    - *[Vintan, IJCNN ‘99][Jiminez, HPCA ‘01]*
    - High prediction accuracy, but more computation
    - Implemented in AMD Piledriver, AMD Ryzen
  - TAGE predictor
    - *TAgged GEometric predictor [Seznac, JILP ‘06]*
    - Keep multiple predictions with different history lengths
    - Partially tag predictions to avoid false matches
    - Only provide prediction on tag match
    - Rumored to be what Intel uses
    - Most recent winners of branch prediction competitions are some derivative of this

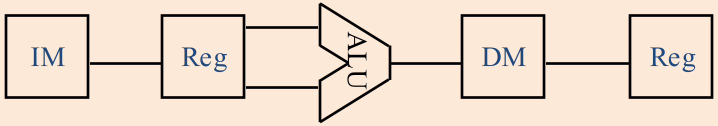
OKAY. So how many of these crazy branch predictor variations do I need to memorize for CSE 141??

- What I want you to know about branch predictors:
  - Why they are useful (why do we put *so much work* into making good ones)?
  - What info do predictors need to operate, and where do they get this info?
  - How the simpler ones work, specifically…
    - 1-bit predictor
    - 2-bit bimodal predictor
    - 2-level local predictor
  - What some of the ’additional tricks’ are, specifically…
    - What is a “Global History Register”?
    - What does a “combining function” do?
  - What problems can arise that confound prediction?
  - ***<u>Given a description</u>***, how to analyze novel branch predictor performance

The Other Half of Branch Prediction

- Remember from earlier….
  - Presupposes what two pieces of information are available at *fetch time*?
    -
    -
  - **Branch Target Buffer** supplies this information

Think back to our original, 5-stage MIPS machine… …and think about **jumps**, which don't have any messy prediction issues



- How many stalls/flushes are required for each of the following situations:
  - Assume control flow **resolves in decode**

	Jump Register, jumps use delay slot	**Jump Immediate,** jumps use delay slot	**Jump Register,** no delay slots	**Jump Immediate,** no delay slots
A	1	1	2	2
B	0	0	0-3	1
C	0-3	0-3	1-3	0-3
D	1-4	1-4	3	1-4
E	*None of the above*			

Jump Immediate, Jump Register – *with* delay slots

- What parts of our MIPS machine makes this stall-free? Hazard-free?

Jump Immediate, Register – with no delay slots

- What parts of this machine gets us to 1 stall / flush (which one, why?)

Can we eliminate the flush for [most] jumps without delay slots?

- (I mean, would I ask if we couldn't?)
- What is the difference between jump immediate and jump register here?

Wait, if a jump is just a ‘control flow operation’ that we always take, can’t we just re-use our branch support hardware (BTB?)

- We could, but there are some reasons it’s not a great idea
  - (why not?)
  - Waste of space
    - … not hard to predict whether a jump will be taken…
  - Aliasing
    - Lots of “taken” predictions...

What information do we need to mitigate different types of control flow hazards? How well can we do?

	Need to learn in instruction type before decode?	Need to record history of last destination?	Control flow change prediction accuracy?	Destination prediction accuracy?
Jump Immediate	Yes	Yes	100%	100%
Jump Register	Yes	Yes	100%	???
Branch	Yes	Yes	???	???

What is this about?

	Need to learn in instruction type before decode?	Need to record history of last destination?	Control flow change prediction accuracy?	Destination prediction accuracy?
Jump Immediate	Yes	Yes	100%	100%
Jump Register	Yes	Yes	100%	???
Jump Register to _________	Yes	No	100%	**~100%**
Branch	Yes	Yes	???	???

To support all these different needs, we build custom structures for each case [caveat: names vary!]

	Need to learn in instruction type before decode?	Need to record history of last destination?	Control flow change prediction accuracy?	Destination prediction accuracy?
Jump Immediate		**Target History Table**		
Jump Register		**Target History Table**		
JR to $ra		**Return Address Stack**		
Branch		**Target History Table + Branch History Table**		

The best way to keep track of all of this is to reason out what is needed to support various features

- What must ____________________, that handles **jump immediate**, look like?

The best way to keep track of all of this is to reason out what is needed to support various features

- What must ____________________, that handles **jump register**, look like?

The best way to keep track of all of this is to reason out what is needed to support various features

- What must ____________________, that handles **`jump to $ra`**, look like?

The best way to keep track of all of this is to reason out what is needed to support various features

- What must ___________________, that handles **branches**, look like?

Pulling it all back together: For our MIPS machine **without any delay slots**, but **with THT, RAS, and BHT…**

- Workload is 50% arithmetic, 5% jump immediate, 10% jump to GP register, 15% jump to $ra, and 20% branches.
  - Jumps to GP registers go to the same destination 90% of the time
  - Branches are predicted with 80% accuracy
  - Assume no aliasing
- What is the CPI?

Pulling it all back together: For our MIPS machine **without any delay slots**, but **with THT, RAS, and BHT…**

- Workload is 50% arithmetic, 5% jump immediate, 10% jump to GP register, 15% jump to $ra, and 20% branches.
  - Jumps to GP registers go to the same destination 90% of the time
  - Branches are predicted with 80% accuracy
  - Assume no aliasing
- What is the CPI?

A	B	C	D	E
< 1.0	>=1.0, < 1.04	>=1.04, < 1.08	>= 1.08, < 1.12	>= 1.12

Reviewing the branch predictors we have learned about

- Single-bit predictor
- Two-bit bimodal
- Two-level local

Rank the physical size of the following control hazard mitigation hardware elements (for a 32-bit machine)

i. 1024-entry THT
ii. 1024-entry BHT with 1-bit predictors
iii. 512-entry BHT with bimodal predictors
iv. 256-entry BHT and a 2-level local predictor with 7-bit patterns and 1-bit predictors

What are all these entries worth anyway?

- Assume the following branches are encountered in a loop such that each branch is seen once each loop
- If a machine has a 256-entry BHT with 1-bit predictors, what is the prediction accuracy for each branch?

Inst Addr	Branch Pattern
0x400	T T T T
0x600	T N T N
0x800	N N N N

	0x400	0x600	0x800
A	100%	0%	100%
B	0%	0%	0%
C	100%	50%	100%
D	33%	33%	33%
E	None of these		

Control Flow Prediction Key Points

- Mispredicts are ***expensive***, especially in deeper pipelines
- Predictors must answer three things correctly to avoid misprediction:
  1. Is the instruction **currently being fetched** a jump or a branch?
     - Must know the answer based on instruction address, *not address contents*!
  2. If so, are we likely to take **change control flow***?
     - *Go somewhere other than PC+4
  3. If so, **where is it going to take us**?
- The best predictions combine multiple sources of information

These interleave a bit, so a quick preview of all the ideas before diving in

OVERVIEW OF ADVANCED CONCEPTS

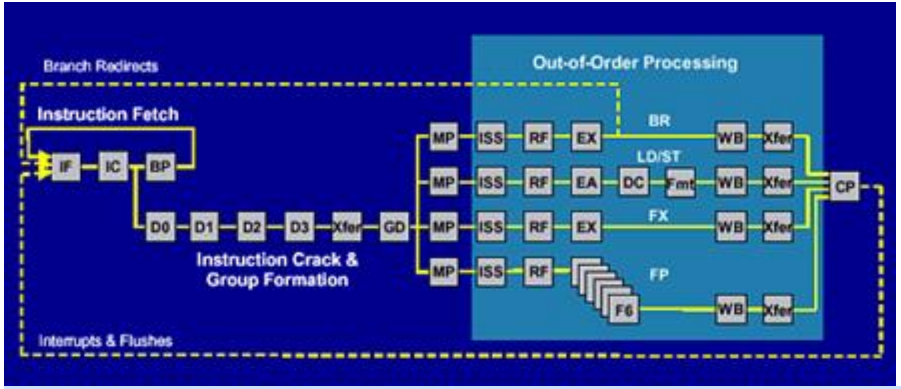
Parallelism in Today’s Most Advanced Processors

Modern pipelines are not fundamentally different than the techniques we discussed, they’re just…

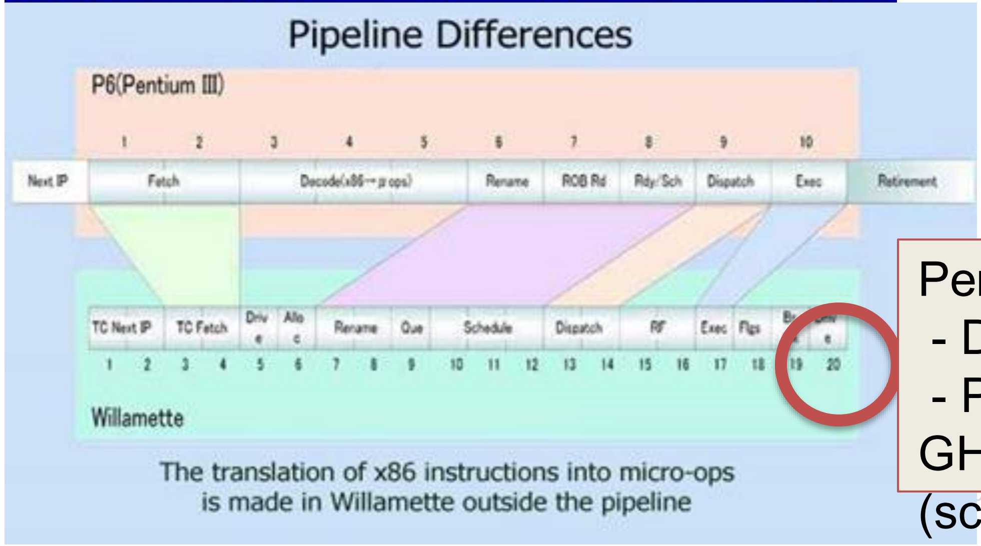
- …Deeper! (more stages)
- …And combined with
  - superscalar execution
  - out-of-order execution

Deeper Pipelines

- Power 4

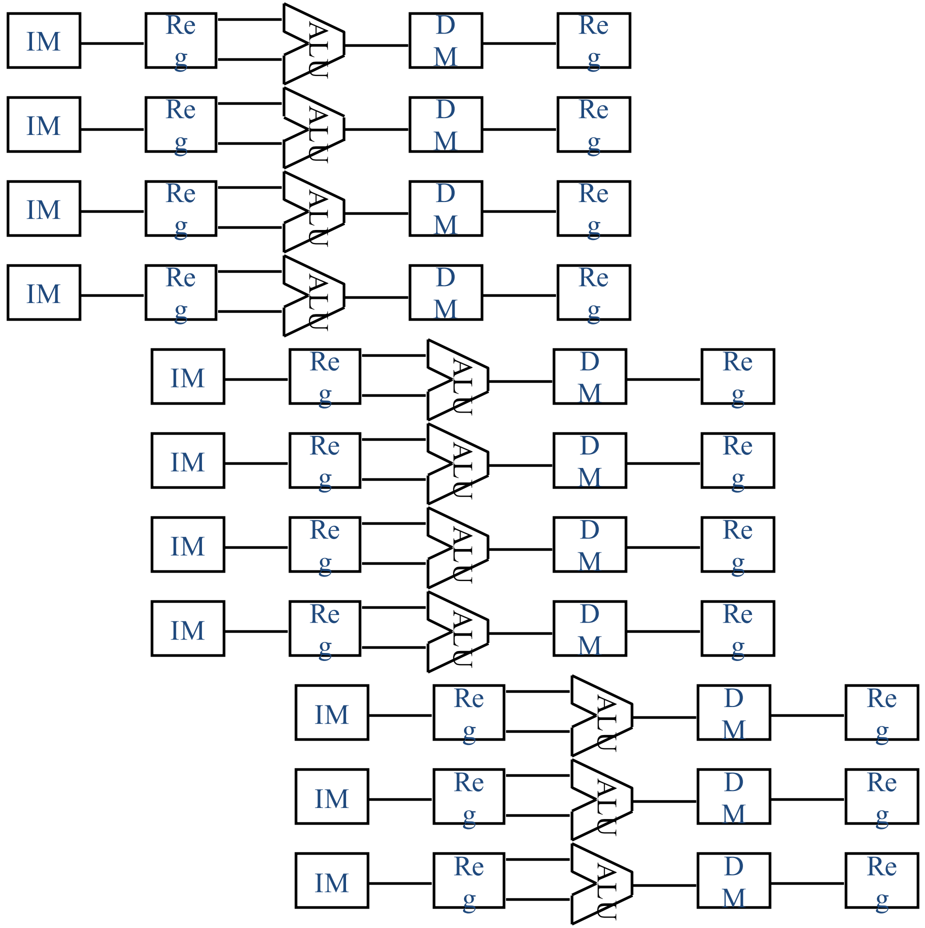


- Pentium 3
- Pentium 4

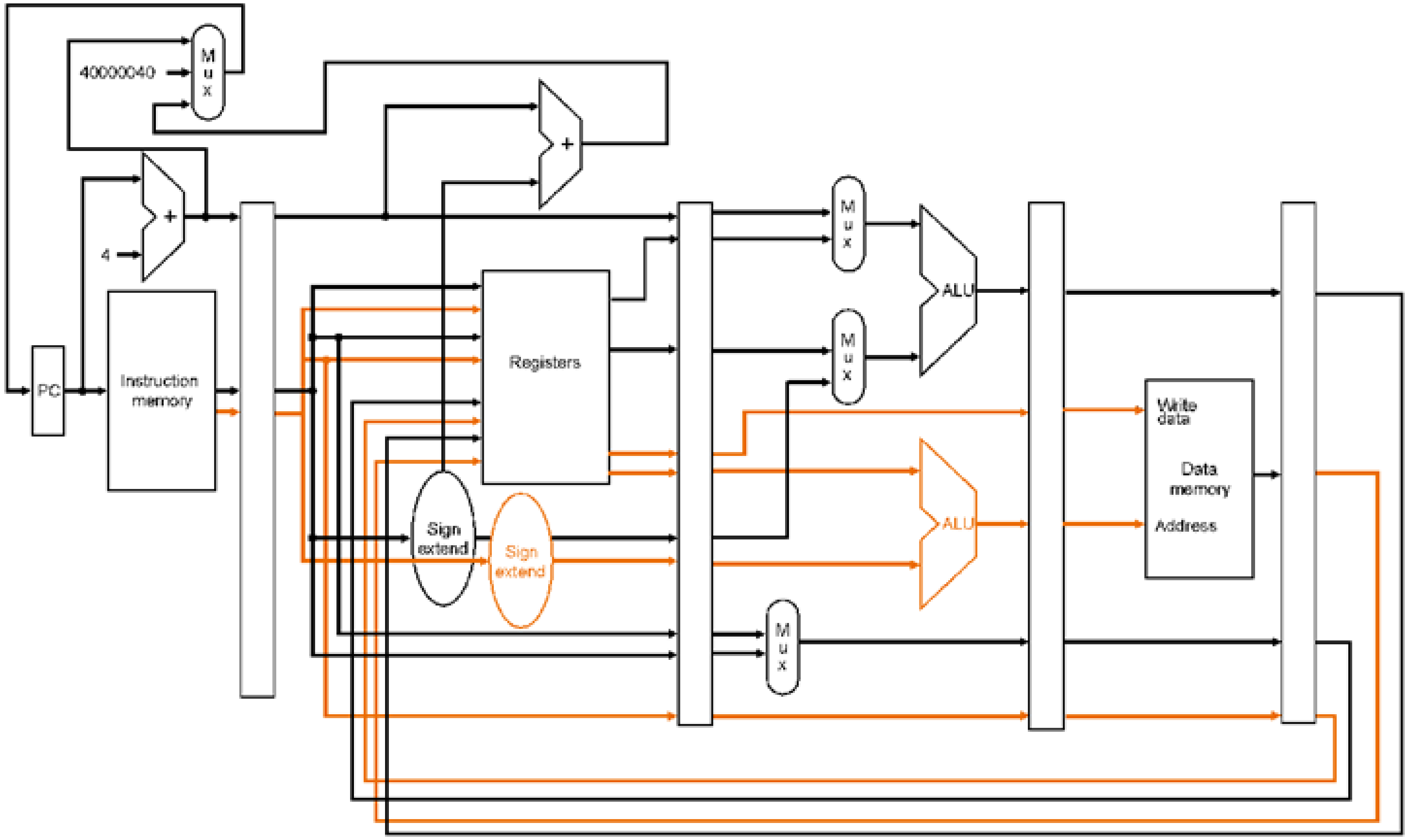


Pentium 4 "Prescott"
- Deeper still: 31 stages!
- Planned for up to 5 GHz operation! (scrapped)

Superscalar Execution

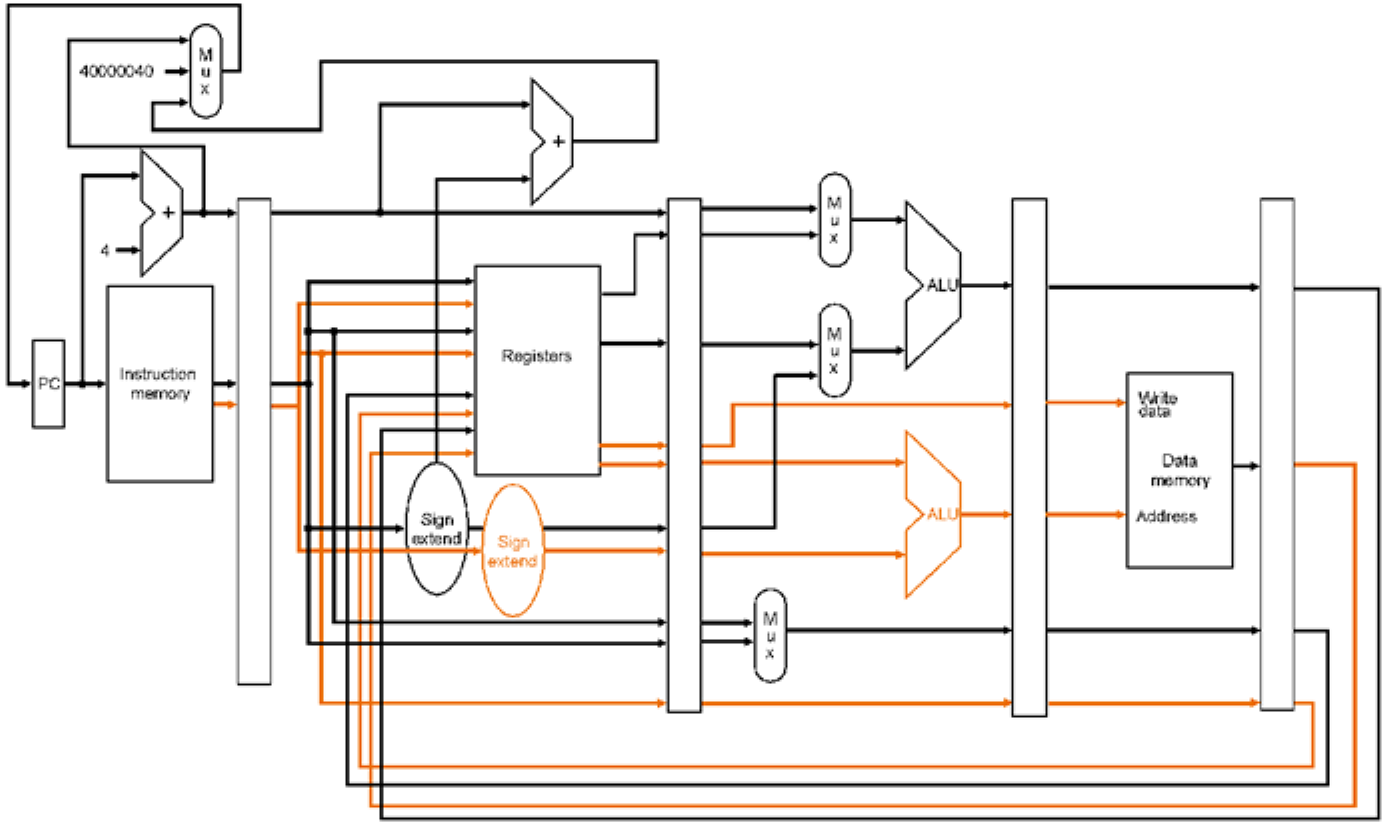


A modest superscalar MIPS — What can this do in parallel?



A	Any two instructions
B	Any two *independent* instructions
C	An arithmetic instruction and a memory instruction
D	Any instruction and a memory instruction
E	None of the above

A modest superscalar MIPS



- what can this machine do in parallel?
- **what other logic is required?**
- Represents earliest superscalar technology
  - (eg, circa early 1990s)

Superscalar Execution

- To execute four instructions in the same cycle, we must find four independent instructions

Superscalar Execution

- To execute four instructions in the same cycle, we must find four independent instructions
- If the four instructions fetched are *guaranteed by the compiler* to be independent, this is a *VLIW* machine

Superscalar Execution

- To execute four instructions in the same cycle, we must find four independent instructions
- If the four instructions fetched are guaranteed by the compiler to be independent, this is a *VLIW* machine
- If the four instructions fetched are only executed together if hardware confirms that they are independent, this is an *in-order superscalar* processor.

Superscalar Execution

- To execute four instructions in the same cycle, we must find four independent instructions
- If the four instructions fetched are guaranteed by the compiler to be independent, this is a *VLIW* machine
- If the four instructions fetched are only executed together if hardware confirms that they are independent, this is an *in-order superscalar* processor.
- If the hardware actively finds four (not necessarily consecutive) instructions that are independent, this is an *out-of-order superscalar* processor.

Superscalar Execution

- To execute four instructions in the same cycle, we must find four independent instructions
- If the four instructions fetched are guaranteed by the compiler to be independent, this is a *VLIW* machine
- If the four instructions fetched are only executed together if hardware confirms that they are independent, this is an *in-order superscalar* processor.
- If the hardware actively finds four (not necessarily consecutive) instructions that are independent, this is an *out-of-order superscalar* processor.
- **What do you think are the tradeoffs?**

Superscalar Scheduling

Assume `in-order, 2-issue, ld-store followed by integer`

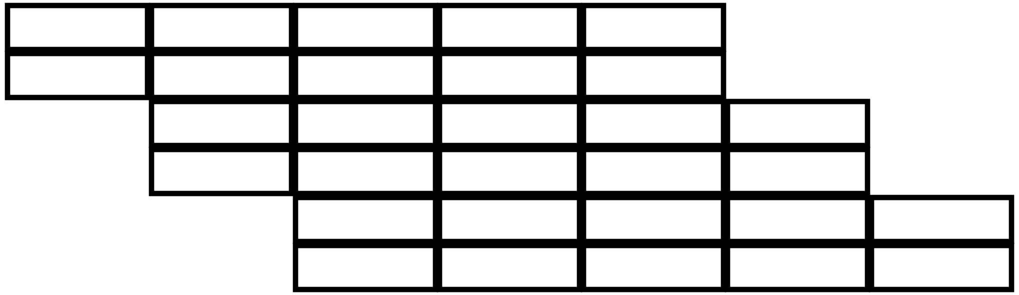
```
        lw   $6,   36($2)
        add  $5,  $6, $4
        lw   $7, 1000($5)
        sub  $9, $12, $5
```

Assume `4-issue, in-order, any combination (VLIW?)`

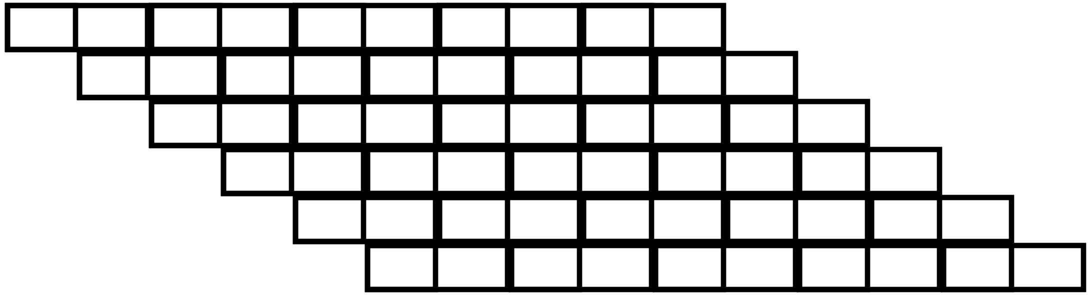
```
        lw   $6, 36($2)
        add  $5,  $6, $4
        lw   $7, 1000($5)
        sub  $9, $12, $5
        sw   $5,  200($6)
        add  $3,  $9, $9
        and $11,  $7, $6
```

When does each instruction **begin execution** (i.e., enter the **X** stage)?

Superscalar vs. superpipelined



(multiple instructions in the same stage, same clock rate as scalar)



(more total stages, faster clock rate)

Finding ways to **execute** instructions at the same time

INSTRUCTION LEVEL PARALLELISM

What is ILP?

- The characteristic of a program that certain instructions are ***independent***, and can potentially be ***executed in parallel***.
- Any mechanism that creates, identifies, or exploits the independence of instructions, allowing them to be executed in parallel.

What is ILP?

- The characteristic of a program that certain instructions are ***independent***, and can potentially be ***executed in parallel***.
- Any mechanism that creates, identifies, or exploits the independence of instructions, allowing them to be executed in parallel.

- Why do we want/need ILP?
  - In a superscalar architecture?
  - What about a scalar architecture?

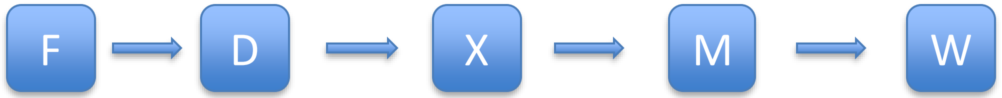
Where do we find ILP?

- In basic blocks?
  - 15-20% of (dynamic) instructions are branches in typical code
- Across basic blocks?
  - how?

```
for (i=1; i<=1000; i++)
    x[i] = x[i] * s
```

Simplest hardware for ILP

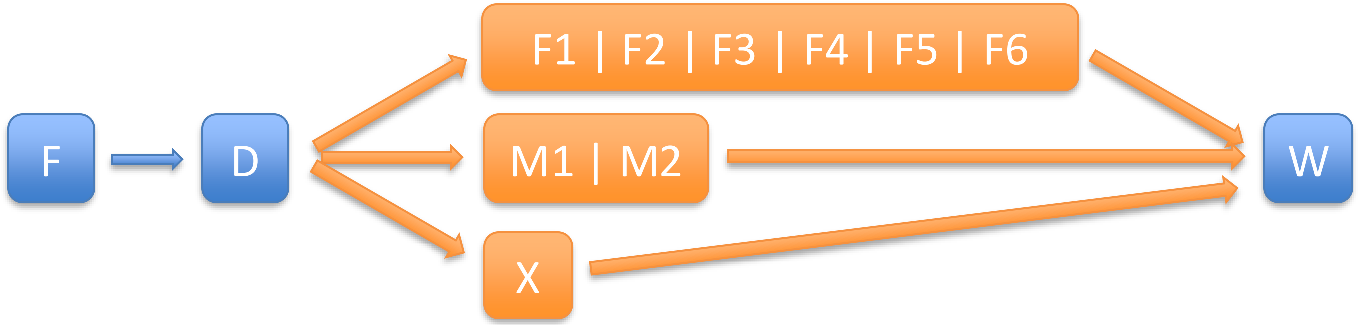
- Take our 5-stage, in-order, scalar pipeline and add parallel *eXecution*
- How/where?
  - Different types of execution units
  - Idea: Don't let long-running executions block *independent* work



Simplest hardware for ILP

Operation	***Latency (stalls)***
FP Mult	*6 (5)*
LD	*2 (1)*
Int ALU	*1 (0)*

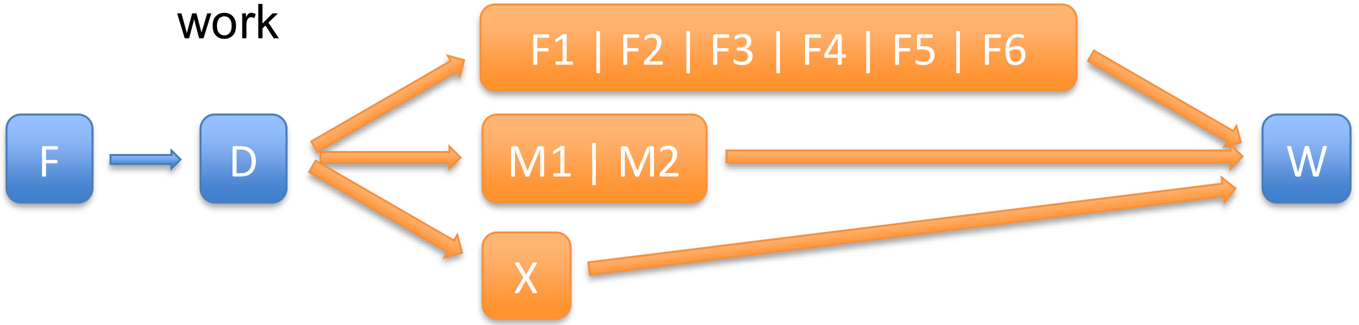
- Take our 5-stage, in-order, scalar pipeline and add parallel *eXecution*
- How/where?
  - Different types of execution units (three for now)
  - Idea: Don't let long-running executions block *independent* work



Operation	***Latency (stalls)***
FP Mult	*6 (5)*
LD	*2 (1)*
Int ALU	*1 (0)*

Simplest hardware for ILP

- Take our 5-stage, **in-order**, scalar pipeline and add parallel *eXecution*
- How/where?
  - Different types of execution units (three for now)
  - Idea: Don't let long-running executions block *independent* work



New structural hazards?

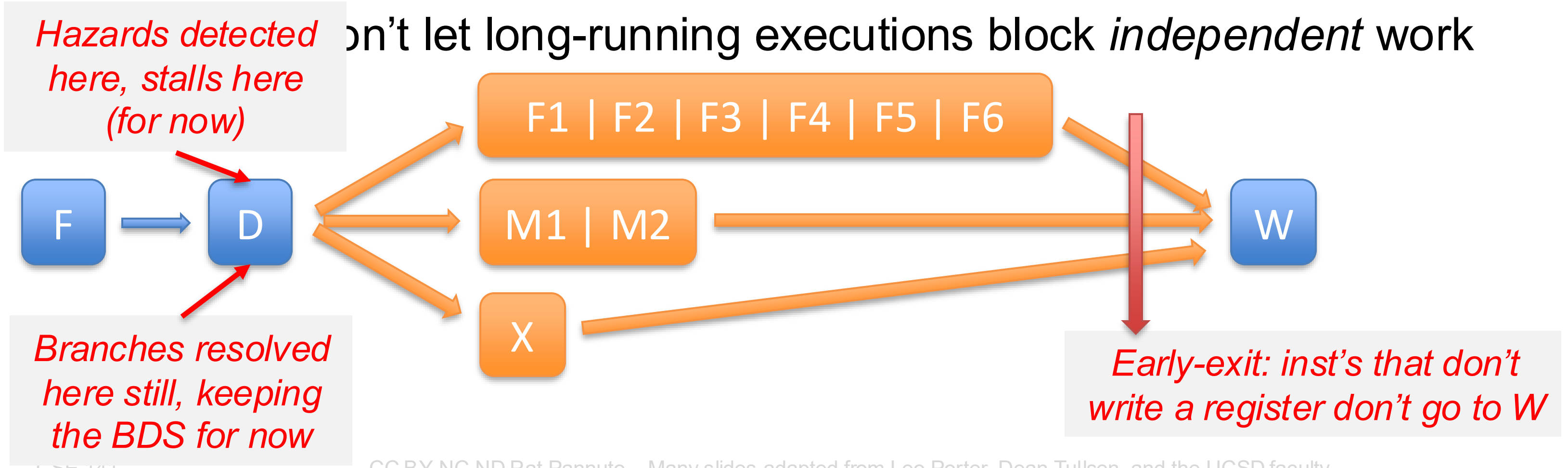
Can we eliminate any structural hazards??

Where to stall?

Simplest hardware for ILP

Operation	*Latency (stalls)*
FP Mult	*6 (5)*
LD	*2 (1)*
Int ALU	*1 (0)*

- Take our 5-stage, **in-order**, scalar pipeline and add parallel *eXecution*
- How/where?
  - Different types of execution units (three for now)
  - on't let long-running executions block *independent* work


Hazards detected here, stalls here (for now)
F
D
F1 | F2 | F3 | F4 | F5 | F6
M1 | M2
X
W
Branches resolved here still, keeping the BDS for now
Early-exit: inst's that don't write a register don't go to W

How do we expose ILP?

- by moving instructions around.
- How??
  - software
  - Hardware

- Today we're going to talk about both software and hardware techniques. In modern processors, we exploit both. But the software techniques are particularly interesting because they each have corresponding hw techniques.

Exposing ILP in software

- instruction scheduling (changes ILP within a basic block)
- loop unrolling (allows ILP across iterations by putting instructions from multiple iterations in the same basic block)
- Others (trace scheduling, software pipelining) we'll talk about later…

A sample loop

Where are the dependencies and stalls?

```
Loop:    LD       F0,0(R1)          ;F0 <= array element, R1 is pointer
                  MULD      F4,F0,F2  ;multiply scalar in F2
                  SD        F4, 0(R1) ;store result
                  ADDI      R1,R1,8   ;increment pointer 8B (DW)
                  SEQ  R3, R1, R2     ;R2 <= (R1 == R2)
                  BNEZ      R3,Loop   ;branch R3!=zero
                  NOP                 ;delayed branch slot
```

Operation	***Latency (stalls)***
FP Mult	*6 (5)*
LD	*2 (1)*
Int ALU	*1 (0)*

A latency of X means:

If instruction A begins ex (issues) in cycle Y and instruction B depends on a value produced by A, B can begin ex (issue) in cycle X+Y.

Baseline Performance

Hazard detect in decode, allow 'early-exit'

```
Loop: LD    F0, 0(R1)
      MULD  F4, F0, F2
      SD    0(R1), F4
      ADDI  R1, R1, 8
      SEQ   R3, R1, R2
      BNEZ  R3, Loop
      NOP
```

Operation	***Latency (stalls)***
FP Mult	***6 (5)***
LD	***2 (1)***
Int ALU	***1 (0)***

Cycles/iteration?

CPI?

Baseline Performance

Hazard detect in decode, allow 'early-exit'

			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Loop:	LD	F0, 0(R1)	F	D	M1	M2	W													
	MULD	F4, F0, F2		F	D	D	F1	F2	F3	F4	F5	F6	W							
	SD	0(R1), F4			F	F	D	D	D	D	D	D	M1	M2						
	ADDI	R1, R1, 8					F	F	F	F	F	F	D	X	W					
	SEQ	R3, R1, R2											F	D	X	W				
	BNEZ	R3, Loop												F	D					
	NOP														F	D				
	LD	F0, 0(R1)														F	…			

Operation	***Latency (stalls)***
FP Mult	***6 (5)***
LD	***2 (1)***
Int ALU	***1 (0)***

Cycles/iteration?

CPI?

Baseline Performance

Faster analysis using latency analysis

Operation	***Latency (stalls)***
FP Mult	***6 (5)***
LD	***2 (1)***
Int ALU	***1 (0)***

```
                                    Cycle Issued
Loop:  LD     F0, 0(R1)               1
       stall *1                       2
       MULD   F4, F0, F2              3 [+1, F0]
       stall *5                       4,5,6,7,8
       SD     0(R1), F4               9 [+5, F4]
       ADDI   R1, R1, 8              10
       SEQ    R3, R1, R2             11
       BNEZ   R3, Loop               12
       NOP                           13
```

Why is this (#) measure useful?

It tells you the number of slots you have to fill, aka the number of <u>independent</u> instructions you have to find, to avoid any stalls!

Instruction Scheduling

Operation	***Latency (stalls)***
FP Mult	***6 (5)***
LD	***2 (1)***
Int ALU	***1 (0)***

```
Loop: LD    F0, 0(R1)

      MULD  F4, F0, F2
      SD    0(R1), F4

      ADDI  R1, R1, 8

      SEQ   R3, R1, R2
      BNEZ  R3, Loop

      NOP
```

Instruction Scheduling

Operation	***Latency (stalls)***
FP Mult	*6 (5)*
LD	*2 (1)*
Int ALU	*1 (0)*

```
Loop: LD    F0, 0(R1)
      MULD  F4, F0, F2
      SD    0(R1), F4
      ADDI  R1, R1, 8
      SEQ   R3, R1, R2
      BNEZ  R3, Loop
      NOP
```

```
Loop: LD    F0, 0(R1)
      ADDI  R1, R1, 8
      MULD  F4, F0, F2
      SEQ   R3, R1, R2
      BNEZ  R3, Loop
      SD    -8(R1), F4
```

Operation	*Latency (stalls)*
FP Mult	*6 (5)*
LD	*2 (1)*
Int ALU	*1 (0)*

Instruction Scheduling

```
Loop: LD    F0, 0(R1)
      MULD  F4, F0, F2
      SD    0(R1), F4
      ADDI  R1, R1, 8
      SEQ   R3, R1, R2
      BNEZ  R3, Loop
      NOP
```

```
Loop: LD    F0, 0(R1)
      ADDI  R1, R1, 8
      MULD  F4, F0, F2
      SEQ   R3, R1, R2
      BNEZ  R3, Loop
      SD    -8(R1), F4
```

Cycles/iteration?

Assume scalar, in-order processor.

Instruction Scheduling

Operation	Latency (stalls)
FP Mult	6 (5)
LD	2 (1)
Int ALU	1 (0)

```
Loop: LD     F0, 0(R1)
      MULD   F4, F0, F2
      SD     0(R1), F4
      ADDI   R1, R1, 8
      SEQ    R3, R1, R2
      BNEZ   R3, Loop
      NOP
```

```
Loop: LD     F0, 0(R1)        1
      ADDI   R1, R1, 8        2
      MULD   F4, F0, F2       3
      SEQ    R3, R1, R2       4
      BNEZ   R3, Loop         5
      SD     -8(R1), F4       9 [5 stalls from cycle 3;
                                 SEQ,BNEZ cover two of them;
                                 so need to add three stalls;
                                 nominally cycle 6 + 3 stalls → 9]

      LD     F0, 0(R1)       10
      …
```

Cycles/iteration?

Assume scalar, in-order processor.

Loop Unrolling

```
Loop: LD    F0, 0(R1)

      ADDI  R1, R1, 8
      MULD  F4, F0, F2
      SEQ   R3, R1, R2
      BNEZ  R3, Loop

      SD    -8(R1), F4
```

Why unroll loops?

We can't get any more scheduling win in a basic block this *small*

```
Loop: LD    F0, 0(R1)

      ADDI  R1, R1, 8
      MULD  F4, F0, F2
      SEQ   R3, R1, R2
      BNEZ  R3, Loop

      SD    -8(R1), F4
```

$\}$ Need to find more instructions that can go here to hide the MULD latency

Loop Unrolling
Step 1: Stamp out a second copy

```
Loop: LD    F0, 0(R1)

      ADDI  R1, R1, 8
      MULD  F4, F0, F2
      SEQ   R3, R1, R2
      BNEZ  R3, Loop

      SD    -8(R1), F4
```

```
Loop: LD    F0, 0(R1)
      ADDI  R1, R1, 8
      MULD  F4, F0, F2
      SEQ   R3, R1, R2
      BNEZ  R3, Loop
      SD    -8(R1), F4
      LD    F0, 0(R1)
      ADDI  R1, R1, 8
      MULD  F4, F0, F2
      SEQ   R3, R1, R2
      BNEZ  R3, Loop
      SD    -8(R1), F4
```

Loop Unrolling

Step 2: Grow the basic block, eliminate redundant instructions

```
Loop: LD    F0, 0(R1)

      ADDI  R1, R1, 8
      MULD  F4, F0, F2
      SEQ   R3, R1, R2
      BNEZ  R3, Loop

      SD    -8(R1), F4
```

$\Rightarrow$

```
Loop: LD    F0, 0(R1)
      ADDI  R1, R1, 8
      MULD  F4, F0, F2
      ~~SEQ   R3, R1, R2~~
      ~~BNEZ  R3, Loop~~
      SD    -8(R1), F4
      LD    F0, 0(R1)
      ADDI  R1, R1, 8
      MULD  F4, F0, F2
      SEQ   R3, R1, R2
      BNEZ  R3, Loop
      SD    -8(R1), F4
```

What assumption is this making?

Operation	***Latency (stalls)***
FP Mult	*6 (5)*
LD	*2 (1)*
Int ALU	*1 (0)*

Loop Unrolling

How's performance doing now?

```
Loop: LD    F0, 0(R1)
      MULD  F4, F0, F2
      SD    0(R1), F4
      LD    F0, 8(R1)
      ADDI  R1, R1, 16
      MULD  F4, F0, F2
      SEQ   R3, R1, R2
      BNEZ  R3, Loop
      SD    -8(R1), F4
```

Cycles/"iteration"?

Careful: How to compare with prior loop performance fairly??

Loop Unrolling

Step 3: Repeat instruction scheduling???

```
Loop: LD    F0, 0(R1)
      MULD  F4, F0, F2
      SD     0(R1), F4
      LD    F0, 8(R1)
      ADDI  R1, R1, 16
      MULD  F4, F0, F2
      SEQ   R3, R1, R2
      BNEZ  R3, Loop
      SD    -8(R1), F4
```

Loop Unrolling

Step 3: Repeat instruction scheduling???

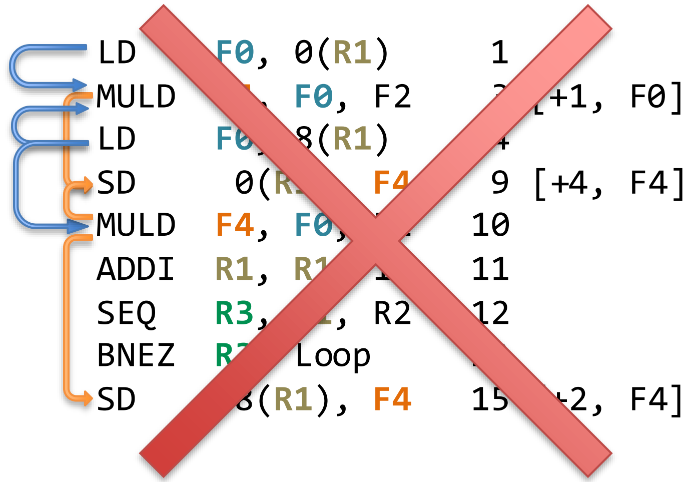
```
Loop: LD    F0, 0(R1)
      MULD  F4, F0, F2
      SD     0(R1), F4

      LD    F0, 8(R1)
      ADDI  R1, R1, 16
      MULD  F4, F0, F2
      SEQ   R3, R1, R2
      BNEZ  R3, Loop
      SD    -8(R1), F4
```

```
Loop:     LD    F0, 0(R1)      1
          MULD  F4, F0, F2     3 [+1, F0]
          LD    F0, 8(R1)      4
          SD     0(R1), F4     9 [+4, F4]
          MULD  F4, F0, F2    10
          ADDI  R1, R1, 16    11
          SEQ   R3, R1, R2    12
          BNEZ  R3, Loop      13
          SD    -8(R1), F4    15 [+2, F4]
```

Loop Unrolling

Step 3: ~~Repeat instruction scheduling???~~ — Make instruction scheduling easier…

```
Loop: LD    F0, 0(R1)
      MULD  F4, F0, F2
      SD     0(R1), F4

      LD    F0, 8(R1)
      ADDI  R1, R1, 16
      MULD  F4, F0, F2
      SEQ   R3, R1, R2
      BNEZ  R3, Loop
      SD    -8(R1), F4
```



```
Loop:
      LD    F0, 0(R1)          1
      MULD  [illegible], F0, F2   [illegible] [+1, F0]
      LD    F0, 8(R1)          4
      SD     0(R[illegible]  F4   9 [+4, F4]
      MULD  F4, F0, [illegible]   10
      ADDI  R1, R1[illegible]     11
      SEQ   R3, [illegible], R2   12
      BNEZ  R[illegible] Loop
      SD    [illegible]8(R1), F4  15 [illegible]2, F4]
```

Register Renaming

Differentiate ***data* dependencies** and ***name* dependencies**

```
Loop: LD    F0, 0(R1)
      MULD  F4, F0, F2
      SD     0(R1), F4
      LD    F0, 8(R1)
      ADDI  R1, R1, 16
      MULD  F4, F0, F2
      SEQ   R3, R1, R2
      BNEZ  R3, Loop
      SD    -8(R1), F4
```

```
Loop: LD    F0, 0(R1)
      MULD  F4, F0, F2
      SD     0(R1), F4
      LD    ??, 8(R1)
      ADDI
      MULD
      SEQ
      BNEZ
      SD
```

Register Renaming

Differentiate ***data*** **dependencies** and ***name*** **dependencies**

```
Loop: LD    F0, 0(R1)
      MULD  F4, F0, F2
      SD     0(R1), F4
      LD    F0, 8(R1)
      ADDI  R1, R1, 16
      MULD  F4, F0, F2
      SEQ   R3, R1, R2
      BNEZ  R3, Loop
      SD    -8(R1), F4
```

```
Loop: LD     F0, 0(R1)
      MULD   F4,  F0, F2
      SD     0(R1), F4
      LD    F10, 8(R1)
      ADDI   R1,  R1, 16
      MULD  F14, F10, F2
      SEQ    R3,  R1, R2
      BNEZ   R3, Loop
      SD    -8(R1), F14
```

Register Renaming

Differentiate *data* dependencies and *name* dependencies

Operation	*Latency (stalls)*
FP Mult	*6 (5)*
LD	*2 (1)*
Int ALU	*1 (0)*

```
Loop: LD     F0, 0(R1)
      MULD   F4,  F0, F2
      SD     0(R1), F4
      LD    F10, 8(R1)
      ADDI   R1,  R1, 16
      MULD  F14, F10, F2
      SEQ    R3,  R1, R2
      BNEZ   R3, Loop
      SD    -8(R1), F14
```

Cycles/“iteration”?

Register Renaming

Enables better scheduling [doesn't change perf on its own]

Operation	Latency (stalls)
FP Mult	6 (5)
LD	2 (1)
Int ALU	1 (0)

```
Loop: LD      F0, 0(R1)
      MULD    F4,  F0, F2
      SD      0(R1), F4

      LD     F10, 8(R1)
      ADDI    R1,  R1, 16
      MULD   F14, F10, F2
      SEQ     R3,  R1, R2
      BNEZ    R3, Loop
      SD     -8(R1), F14
```

Cycles/"iteration" for best-possible schedule? [I got 12, can you beat me?]

What is limiting you…? What could you do to help?

Compiler Perspectives on Code Movement

- Remember: *dependencies* are a property of code, whether or not it is a HW *hazard* depends on the given pipeline.
- Compiler must respect (*True*) Data dependencies (RAW)
  - Easy to determine for registers (fixed names)
  - Hard for memory:
    - Does 100(R4) = 20(R6)?
    - From different loop iterations, does 20(R6) = 20(R6)?
- False dependences (WAR and WAW) can sometimes be overcome.

Compiler Perspectives on Code Movement

- Compilers must also preserve *control dependence*
- Example

```
if (c1)
    I1;
if (c2)
    I2;
```

- `I1` is control dependent on `c1`
- `I2` is control dependent on `c2` but not on `c1`.

Compiler Perspectives on Code Movement

- Two (obvious) constraints on control dependences:
  - An instruction that is *control dependent* on a branch cannot be moved before the branch such that its execution is no longer controlled by the branch.
  - An instruction that is not *control dependent* on a branch cannot be moved to after the branch such that its execution is controlled by the branch.
- Control dependencies relaxed to get parallelism; as long as we get same effect if preserve order of exceptions and data flow

Code Motion

- Can be done in SW or HW
- Why SW?
- Why HW?

Code Motion

- Can be done in SW or HW
- Why SW?
- Why HW?

- Also, like software, we'd like the following capabilities in our hardware code motion.
  - Ability to move instructions across branches
  - Ability to overcome (or ignore) false dependences
  - Both easier in hardware

HW Schemes: Instruction Parallelism

- Why in HW at run time?
  - Works when can’t know dependence until run time
    - Variable latency
    - Control dependent data dependence
  - Can schedule differently every time through the code.
  - Compiler simpler
  - **Code for one machine runs well on another**
- Key idea: Allow instructions behind stall to proceed

```
DIVD        F0, F2,  F4
ADDD       F10, F0,  F8
SUBD  F12, F8, F14
```

  - Enables out-of-order execution => out-of-order completion

First HW ILP Technique: Out-of-order Issue/Dynamic Scheduling

- Problem – need to get stalled instructions out of the ID stage, so that subsequent instructions can begin execution.

First HW ILP Technique: Out-of-order Issue/Dynamic Scheduling

- Problem – need to get stalled instructions out of the ID stage, so that subsequent instructions can begin execution.
- Must separate detection of structural hazards from detection of data hazards

First HW ILP Technique: Out-of-order Issue/Dynamic Scheduling

- Problem – need to get stalled instructions out of the ID stage, so that subsequent instructions can begin execution.
- Must separate detection of structural hazards from detection of data hazards
- Must split ID operation into two:
  - Issue (decode, check for structural hazards)
  - Read operands (read operands when NO DATA HAZARDS)

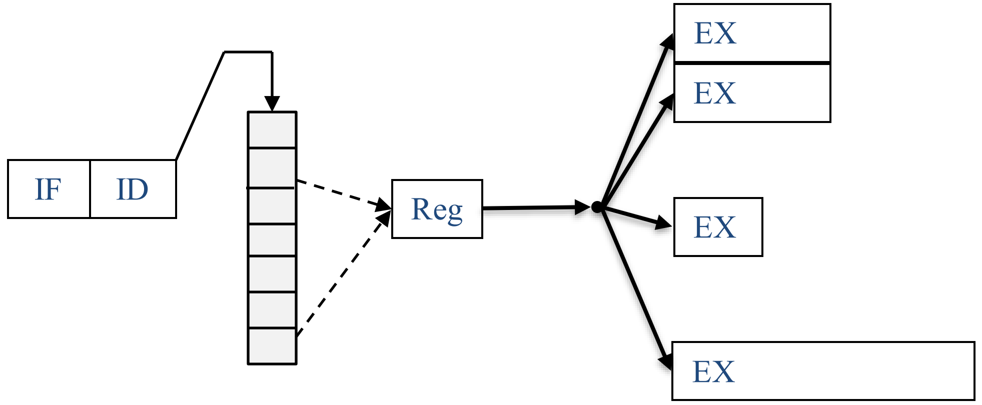
First HW ILP Technique: Out-of-order Issue/Dynamic Scheduling

- Problem – need to get stalled instructions out of the ID stage, so that subsequent instructions can begin execution.
- Must separate detection of structural hazards from detection of data hazards
- Must split ID operation into two:
  - **Dispatch** (decode, check for structural hazards)
  - Read operands (read operands when NO DATA HAZARDS)
  - Otherwise, one stalled (for data) instruction would cause all others to back up behind the ID stage.
  - i.e., must be able to dispatch even when a data hazard exists

First HW ILP Technique: Out-of-order Issue/Dynamic Scheduling

- Problem – need to get stalled instructions out of the ID stage, so that subsequent instructions can begin execution.
- Must separate detection of structural hazards from detection of data hazards
- Must split ID operation into two:
  - Dispatch (decode, check for structural hazards)
  - Read operands (read operands when NO DATA HAZARDS)
  - Otherwise, one stalled (for data) instruction would cause all others to back up behind the ID stage.
  - i.e., must be able to dispatch even when a data hazard exists
- instructions *dispatch* in-order, but proceed to EX out-of-order
  - (terminology highly inconsistent. Common is dispatch->issue. A few say issue->dispatch. Book treats dispatch/issue as synonyms, and doesn't really have a term for "proceed to EX")

For example…



Could also have reg read before the queue – we’ll look at a couple different options.

Dynamic Scheduling by hand

in-order out-of-order

```
DIVD   F0, F2, F4  (10 cycles)
ADDD  F10, F0, F8  (4 cycles)
SUBD  F12, F8,F14  (4 cycles)
ADDD  F20, F2, F3  …
MULTD F13,F12, F2  (6 cycles)
ADDD   F4, F1, F3  …
ADDD   F5, F4,F13  …
```

(assume pipelined FP ADD units)

Dynamic Scheduling by hand

```
                                          in-order         out-of-order
DIVD   F0, F2,  F4 (10 cycles)              1                1
ADDD  F10, F0,  F8 (4 cycles)              11               [2]->11
SUBD  F12, F8, F14 (4 cycles)              12                3
ADDD  F20, F2,  F3  …                      13                4
MULTD F13,F12,  F2 (6 cycles)              16               [5]-> 7
ADDD   F4, F1,  F3  …                      17                6
ADDD   F5, F4, F13  …                      18               {7}--scalar--> 8

(assume pipelined FP ADD units)
```

Dynamic Scheduling (*aka,* Out-of-Order Scheduling) on our original MIPS?

- Issues (begins execution of) an instruction as soon as all of its dependences are satisfied, even if prior instructions are stalled. `(assume 2-issue, any combination)`

```
lw   $6,   36($2)
add  $5,  $6, $4
lw   $7, 1000($5)
sub  $9, $12, $8
sw   $5,  200($6)
add  $3,  $9, $9
and $11,  $5, $6
```

Compiler? Exceptions?

- Does out-of-order execution in the hardware make compilation harder or easier?
- Does out-of-order execution in the hardware make exception handling harder or easier?

ILP: Key Points

- You can find, create, and exploit Instruction Level Parallelism in SW or HW
- Loop level parallelism is usually easiest to see
- Dependencies exist in a program, and become hazards if HW cannot resolve
- SW dependencies/compiler sophistication determine if compiler can/should unroll loops
- SW code motion is limited by lack of runtime knowledge of dependencies (esp. memory), latencies (esp. memory), and control flow.

HW-based ILP in a real machine—what do the details look like?

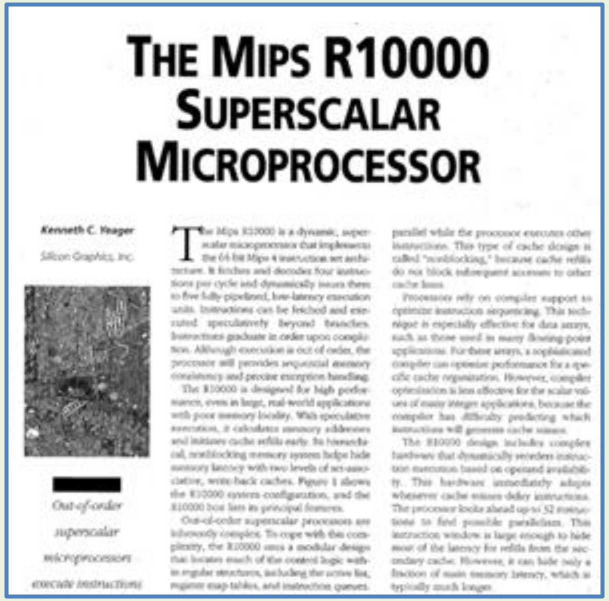
MIPS R10K: DYNAMIC SCHEDULING

Dynamic Scheduling Key Points

- Dynamic scheduling is code motion in HW.
- Dynamic scheduling can do things SW scheduling (static scheduling) cannot.
- Register renaming eliminates WAW, WAR dependencies.
- To get cross-iteration parallelism, we need to eliminate WAW, WAR dependencies.

MIPS R10K Whitepaper

- Driving Factors
- Challenges
- Key features of R10K?
- What are the structures that allow
  - Register renaming?
  - In-order commit?
  - Correct handling of branch mispredicts/exceptions



THE MIPS R10000 SUPERSCALAR MICROPROCESSOR

Kenneth C. Yeager

Silicon Graphics, Inc.

Out-of-order superscalar microprocessors execute instructions

The Mips R10000 is a dynamic, superscalar microprocessor that implements the 64-bit Mips 4 instruction set architecture. It fetches and decodes four instructions per cycle and dynamically issues them to five fully-pipelined, low-latency execution units. Instructions can be fetched and executed speculatively beyond branches. Instructions graduate in order upon completion. Although execution is out of order, the processor still provides sequential memory consistency and precise exception handling.

The R10000 is designed for high performance, even in large, real-world applications with poor memory locality. With speculative execution, it calculates memory addresses and initiates cache refills early. Its hierarchical, nonblocking memory system helps hide memory latency with two levels of set-associative, write-back caches. Figure 1 shows the R10000 system configuration, and the R10000 box lists its principal features.

Out-of-order superscalar processors are inherently complex. To cope with this complexity, the R10000 uses a modular design that locates much of the control logic within regular structures, including the active list, register map tables, and instruction queues.

parallel while the processor executes other instructions. This type of cache design is called "nonblocking," because cache refills do not block subsequent accesses to other cache lines.

Processors rely on compiler support to optimize instruction sequencing. This technique is especially effective for data arrays, such as those used in many floating-point applications. For these arrays, a sophisticated compiler can optimize performance for a specific cache organization. However, compiler optimization is less effective for the scalar values of many integer applications, because the compiler has difficulty predicting which instructions will generate cache misses.

The R10000 design includes complex hardware that dynamically reorders instruction execution based on operand availability. This hardware immediately adapts whenever cache misses delay instructions. The processor looks ahead up to 32 instructions to find possible parallelism. This instruction window is large enough to hide most of the latency for refills from the secondary cache. However, it can hide only a fraction of main memory latency, which is typically much longer.

Modern Architectures

- Alpha 21264+, MIPS R10K+, Pentium 4 use an ***instruction queue***.
- They use explicit ***register renaming***.
  - Registers are not read until instruction issues (begins execution).
  - Register renaming ensures no conflicts.

```
Div  R5, R4, R2
Add  R7, R5, R1
Sub  R5, R3, R2
Lw   R7, 100(R5)
```

→

R1	PR23
R2	PR02
R3	PR17
R4	PR45
R5	PR13
R6	PR20
R7	PR30
…	

Modern Architectures

- Alpha 21264+, MIPS R10K+, Pentium 4 use an ***instruction queue***.
- They use explicit ***register renaming***.
  - Registers are not read until instruction issues (begins execution).
  - Register renaming ensures no conflicts.

```
Div    R5, R4, R2
Add    R7, R5, R1
Sub    R5, R3, R2
Lw     R7, 100(R5)
```

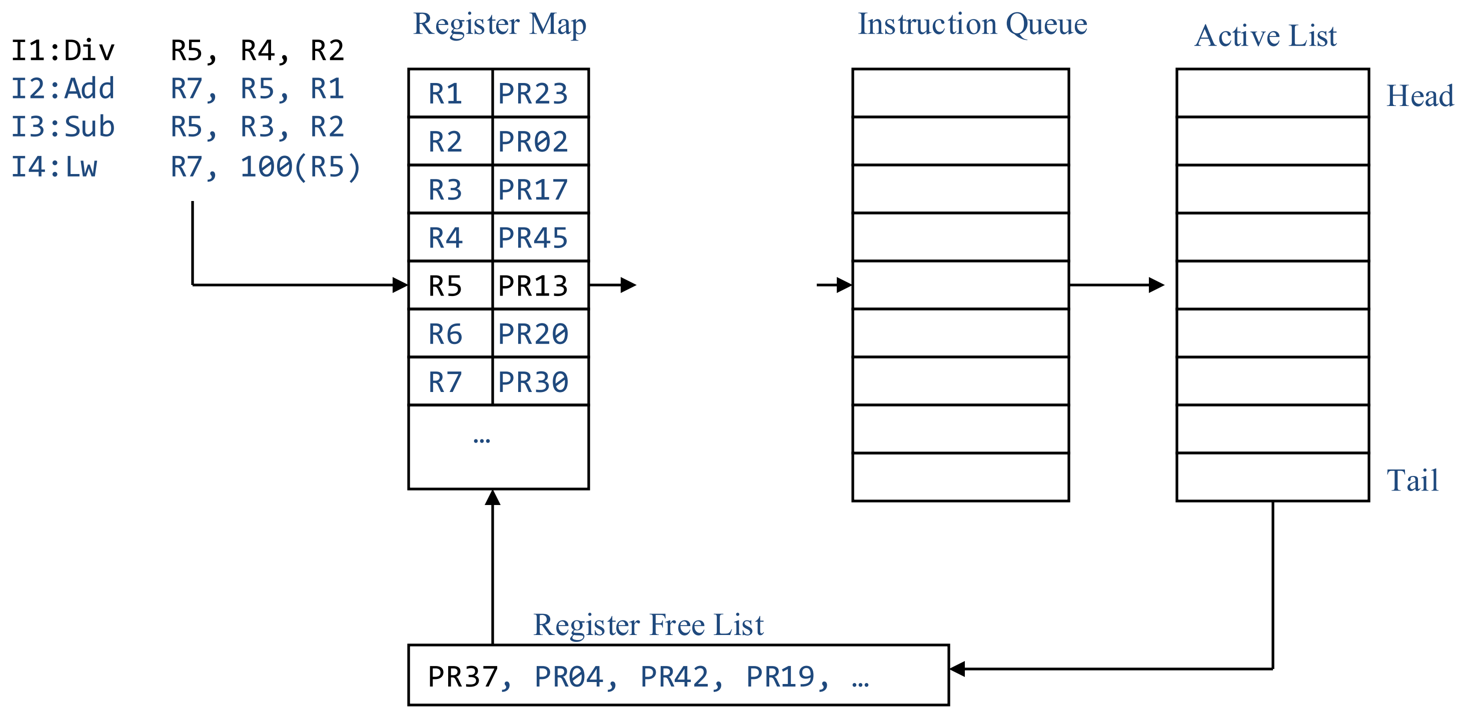
→

```
Div    PR37, PR45, PR02
Add    PR04, PR37, PR23
Sub    PR42, PR17, PR02
Lw     PR19,    100(PR42)
```

R1	PR23
R2	PR02
R3	PR17
R4	PR45
R5	PR13
R6	PR20
R7	PR30
…	

MIPS R10000, some detail

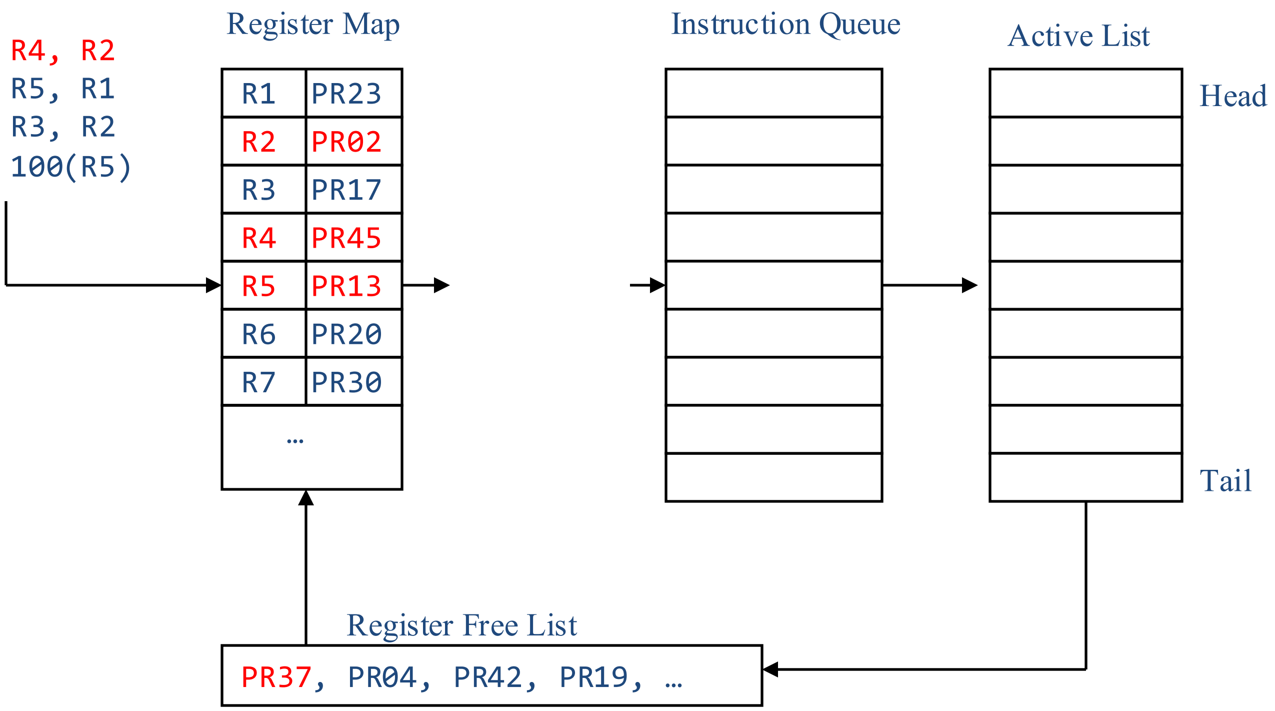
This machine supports issuing <u>up to</u> 4 instructions / cycle! (We'll look at them one by one though)



Active list – maintains original instruction order, determines when a physical register can be freed.

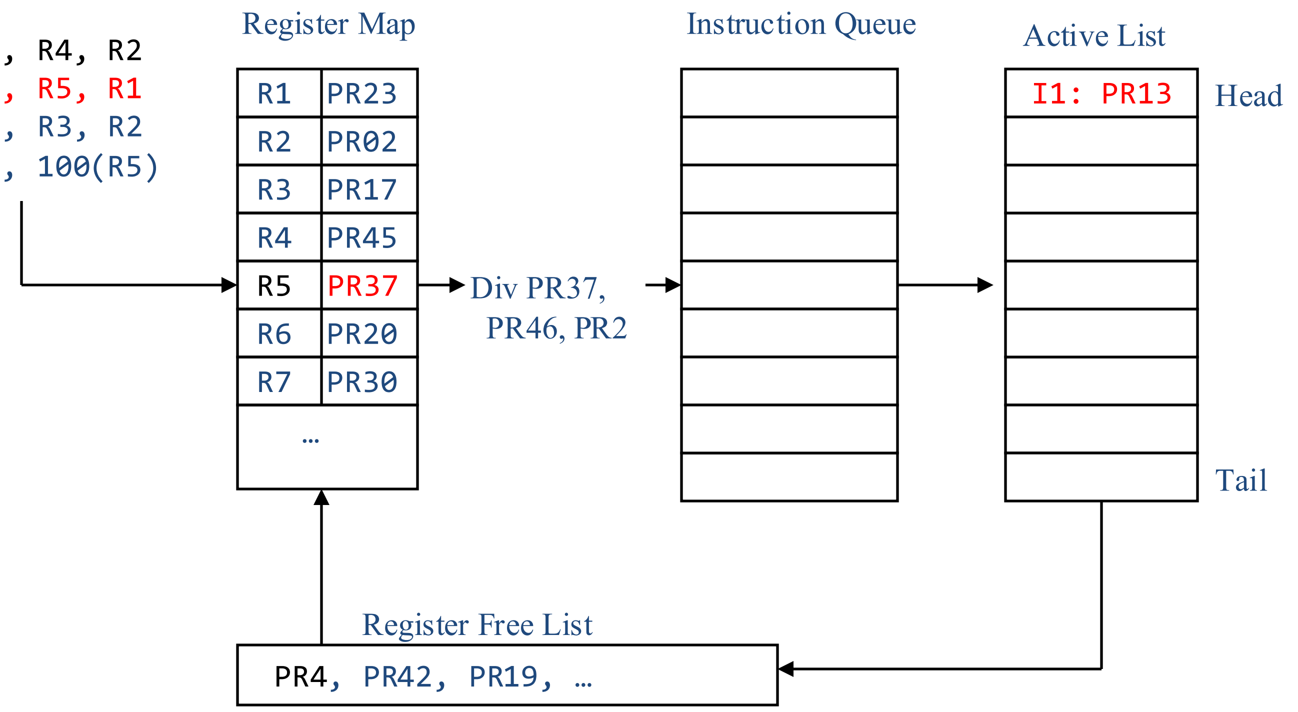
MIPS R10000, some detail

```
I1:Div    R5, R4, R2
I2:Add    R7, R5, R1
I3:Sub    R5, R3, R2
I4:Lw     R7, 100(R5)
```



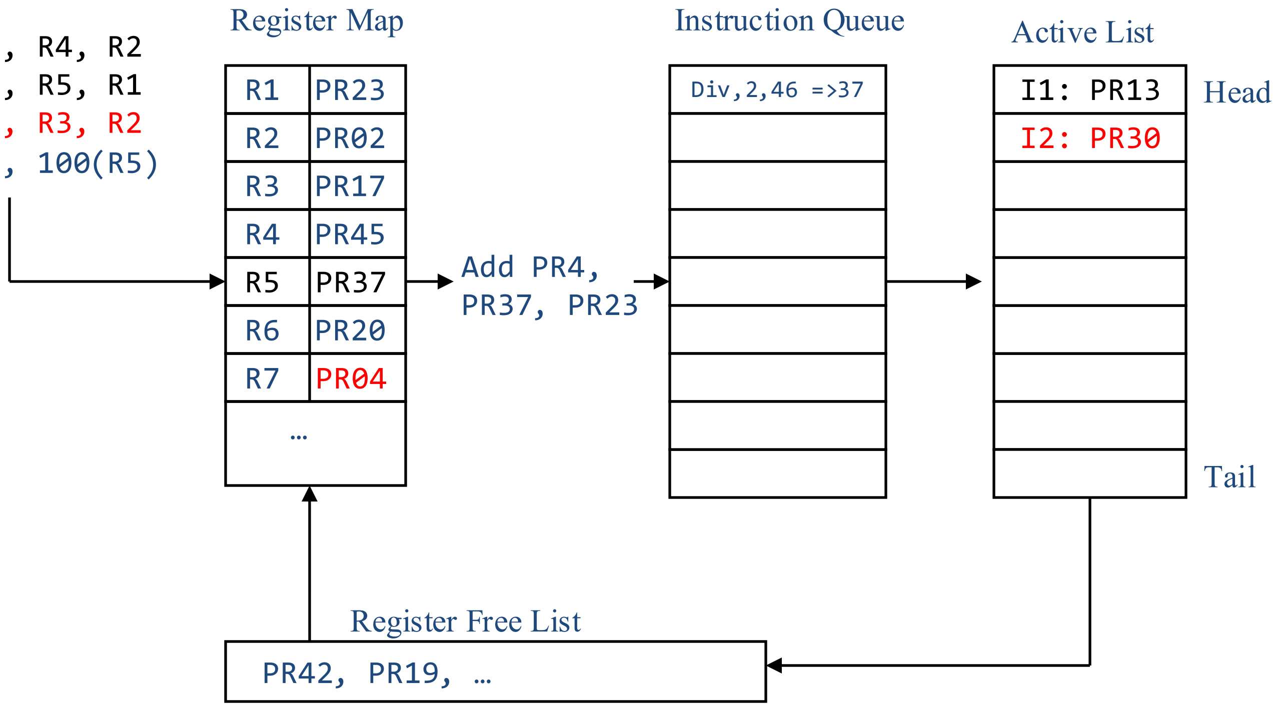
MIPS R10000, some detail

```
I1:Div    R5, R4, R2
I2:Add    R7, R5, R1
I3:Sub    R5, R3, R2
I4:Lw     R7, 100(R5)
```

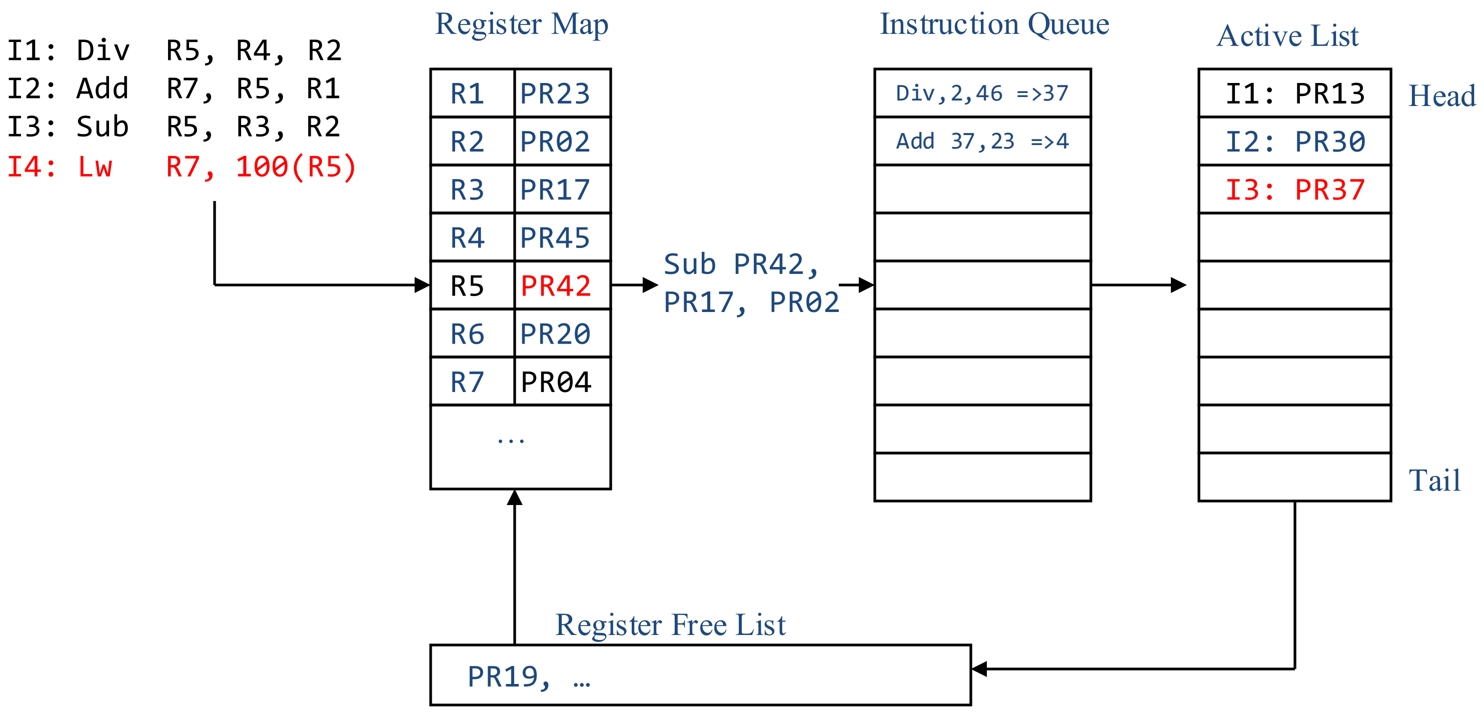


MIPS R10000, some detail

```
I1:Div    R5, R4, R2
I2:Add    R7, R5, R1
I3:Sub    R5, R3, R2
I4:Lw     R7, 100(R5)
```

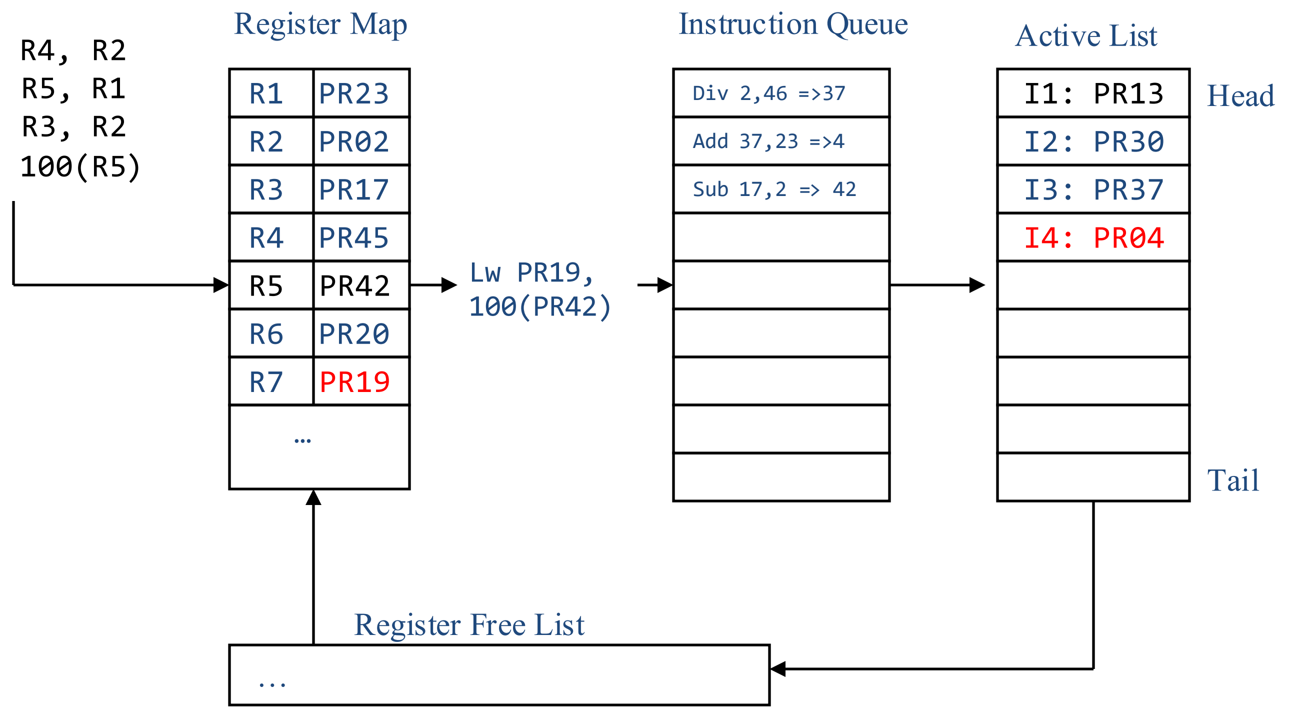


MIPS R10000, some detail



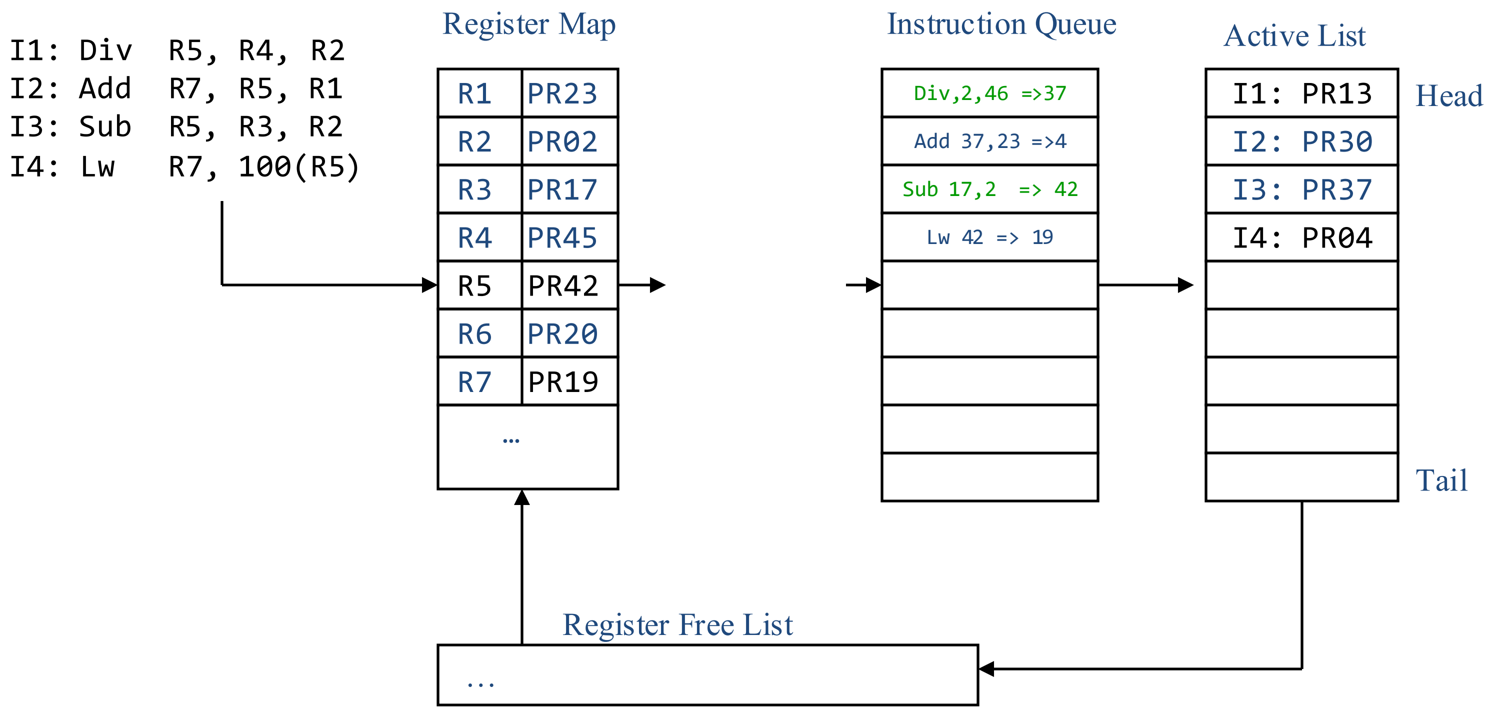
MIPS R10000, some detail

```
I1: Div  R5, R4, R2
I2: Add  R7, R5, R1
I3: Sub  R5, R3, R2
I4: Lw   R7, 100(R5)
```



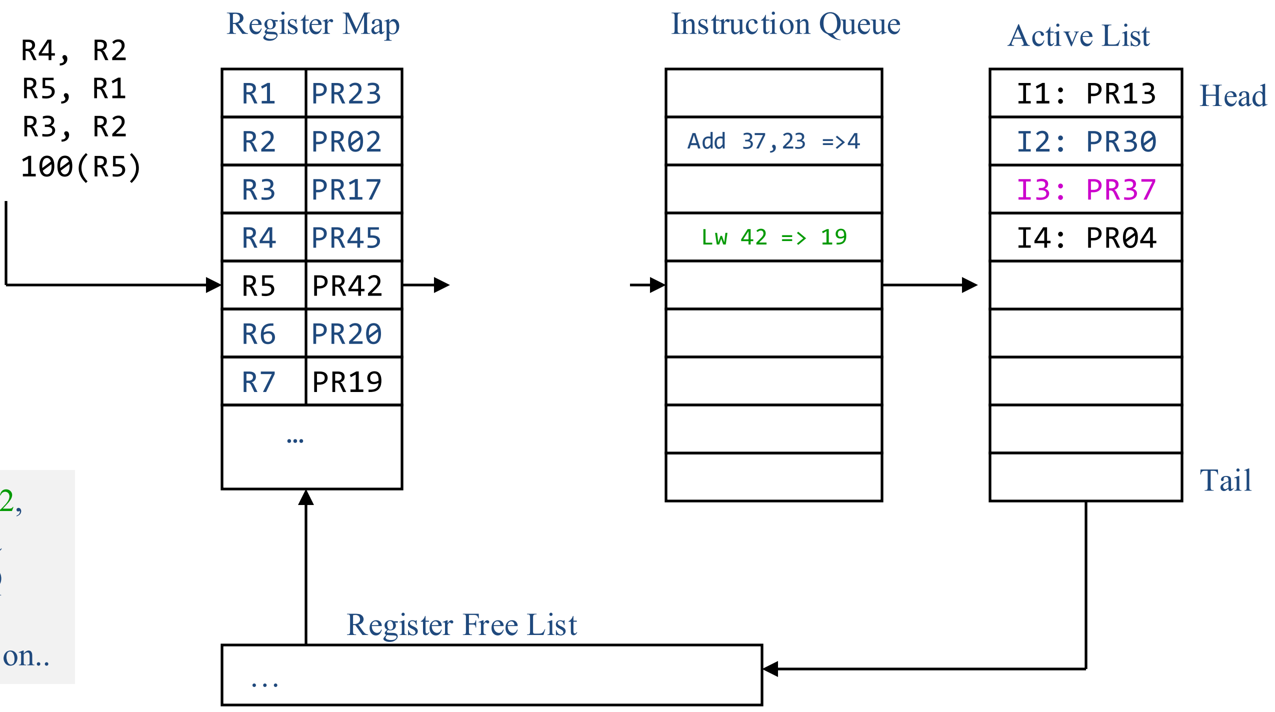
MIPS R10000, some detail

```
I1: Div  R5, R4, R2
I2: Add  R7, R5, R1
I3: Sub  R5, R3, R2
I4: Lw   R7, 100(R5)
```



MIPS R10000, some detail

```
I1: Div  R5, R4, R2
I2: Add  R7, R5, R1
I3: Sub  R5, R3, R2
I4: Lw   R7, 100(R5)
```

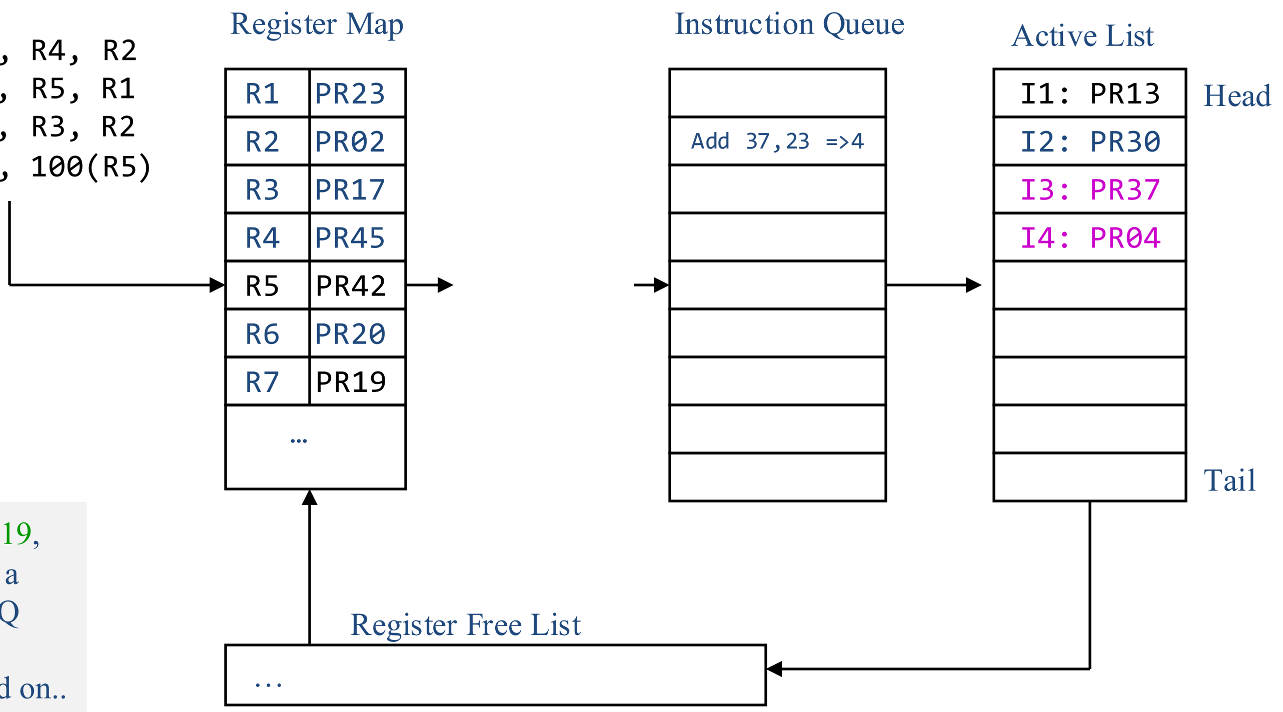


I3, producing register 42, completes, broadcasts a completion signal to IQ

I1 is still being worked on..

MIPS R10000, some detail

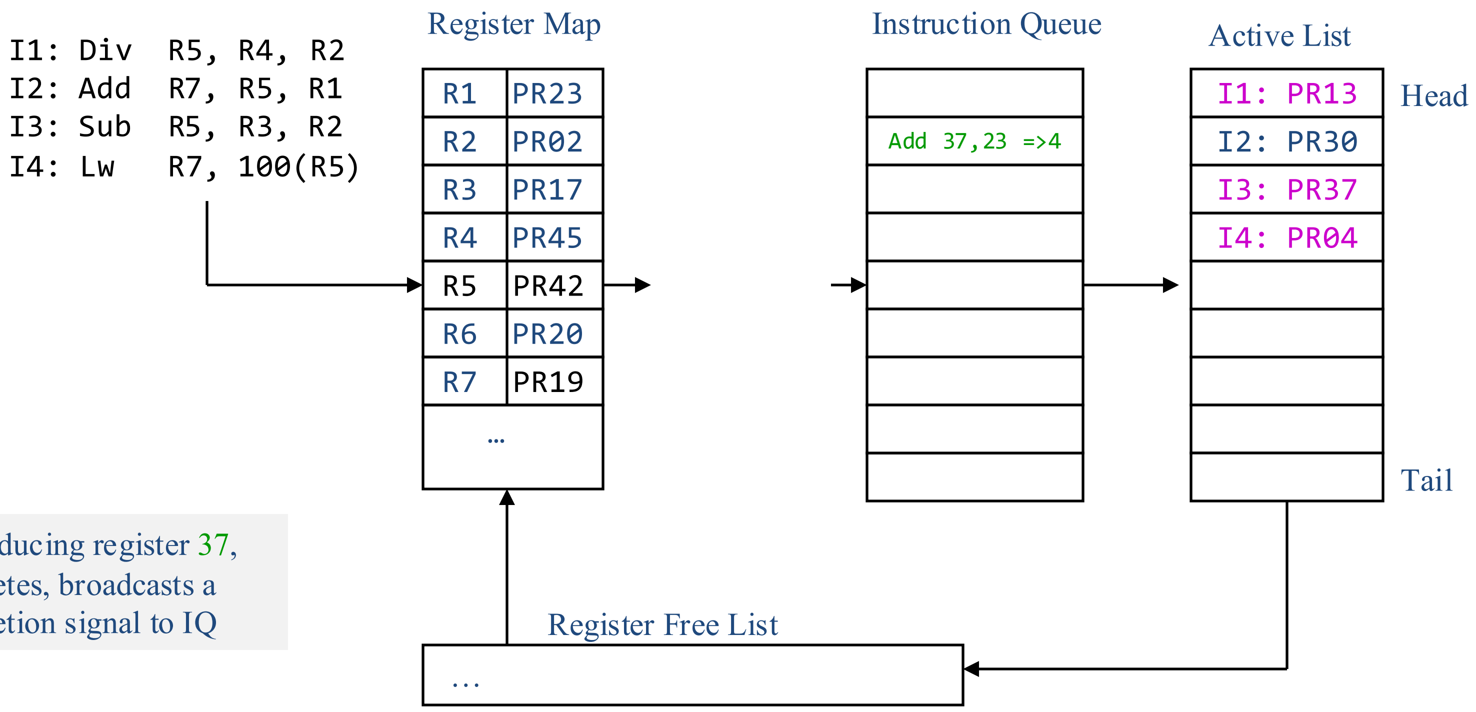
```
I1: Div  R5, R4, R2
I2: Add  R7, R5, R1
I3: Sub  R5, R3, R2
I4: Lw   R7, 100(R5)
```



I4, producing register 19, completes, broadcasts a completion signal to IQ

I1 is still being worked on..

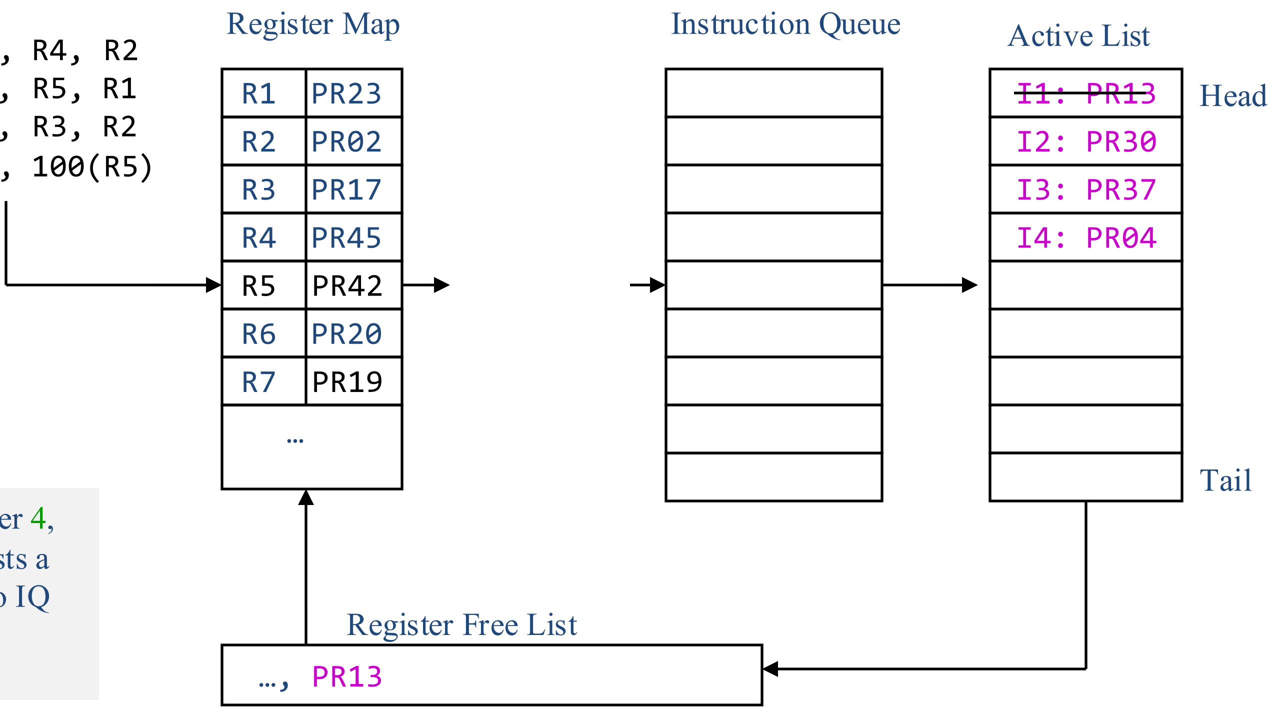
MIPS R10000, some detail



I1, producing register 37, completes, broadcasts a completion signal to IQ

MIPS R10000, some detail

```
I1: Div  R5, R4, R2
I2: Add  R7, R5, R1
I3: Sub  R5, R3, R2
I4: Lw   R7, 100(R5)
```

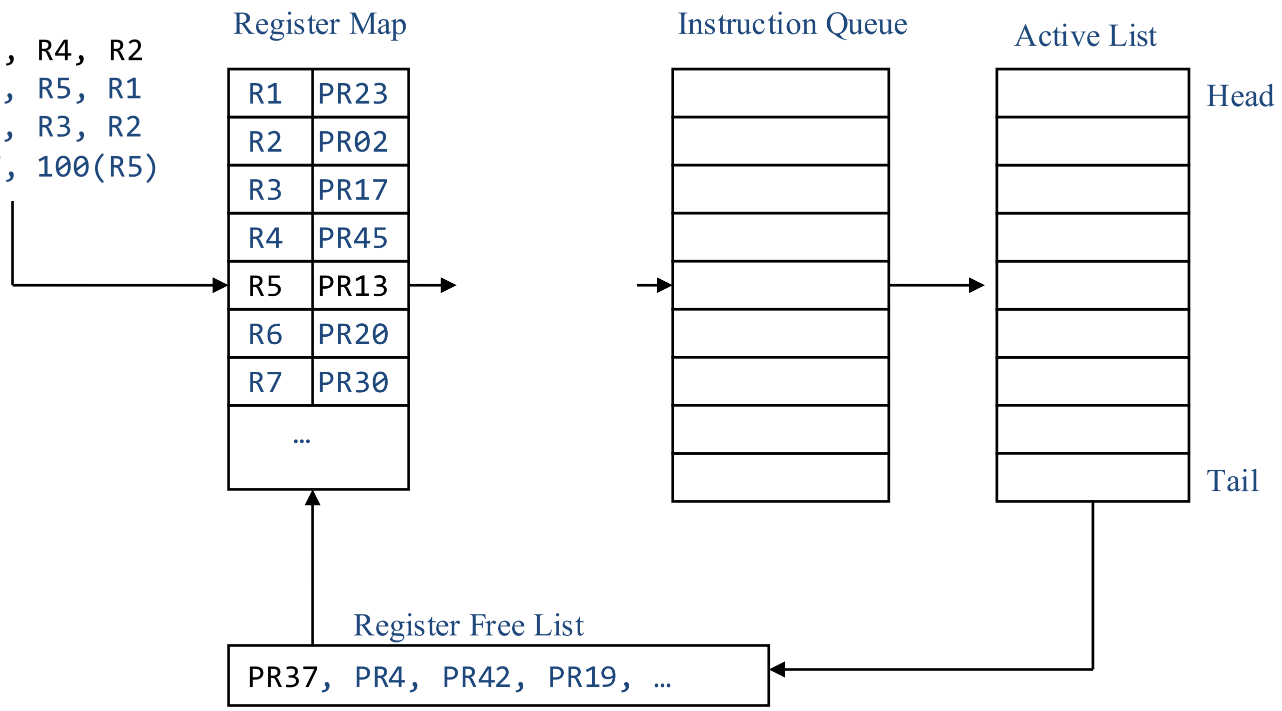


I2, producing register 4, completes, broadcasts a completion signal to IQ

I1 commits.

MIPS R10000, some detail

```
I1: Div  R5, R4, R2
I2: Add  R7, R5, R1
I3: Sub  R5, R3, R2
I4: Lw   R7, 100(R5)
```



Active list – maintains original instruction order, determines when a physical register can be freed.

MIPS R10000

- How do you handle branch mispredicts?

- Any other questions about MIPS R10000?

- Any observations?

Try this: (assume loads take 12 cycles)

```
lw          r1, 1000(r5)
add         r6,  r7,  r9
sub         r4,  r2,  r1
lw          r8, 2000(r4)
addi        r5,  r5,  #8
addi       r10, r10,  #1
bne        r10, r11, (top of loop)
nop
```

```
lw          r1, 1000(r5)
add         r6,  r7,  r9
sub         r4,  r2,  r3
lw          r8, 2000(r4)
add         r5,  r1,  r6
addi       r10, r10,  #1
bne        r10, r11, (top of loop)
nop
```

Assume long-running loop. Assume large instruction queue. Almost identical programs.

Which will run faster on an OOO machine?

Generalizing concepts for **parallel** execution of instructions → **CPI's < 1.0**

SUPERSCALAR

Pipeline Performance

- CPI = 1.0 + BSPI + FPSPI + LdSPI

Branch prediction attacks this

Out-of-order execution, instruction scheduling attack these

Now what?

- CPI = 1.0 + BSPI + FPSPI + LdSPI

Branch prediction attacks this

Out-of-order execution, instruction scheduling attack these

Multiple Instruction Issue

or

The insts go marching two by two, hurrah, hurrah...

Getting CPI < 1: Issuing Multiple Instructions/Cycle

- Superscalar
  - variable number of instructions issued each cycle
  - parallelism detected in hardware
- Very Long Instruction Words (VLIW)
  - fixed number of instructions issued each cycle
  - parallelism scheduled by the compiler
  - IA-64 (Itanium)

Early Superscalar

- Very conservative – given that they already had integer functional units/reg file, and fp units/reg file, they decided they could execute two instructions/cycle with little extra hardware.

Getting CPI < 1: Issuing Multiple Instructions/Cycle

- Early attempt – Superscalar MIPS:
  - 2 instructions, 1 FP & 1 anything else, in-order
  - Fetch 64-bits/clock cycle; Int on left, FP on right
  - Can only issue 2nd instruction if 1st instruction issues

Type	*Pipe*	*Stages*						
Int. instruction		IF	ID	EX	MEM	WB		
FP instruction		IF	ID	EX	MEM	WB		
Int. instruction			IF	ID	EX	MEM	WB	
FP instruction			IF	ID	EX	MEM	WB	
Int. instruction				IF	ID	EX	MEM	WB
FP instruction				IF	ID	EX	MEM	WB

Superscalar MIPS Implications

- More ports for FP registers to do FP load & FP op in a pair
- 1 cycle load delay expands to 3 instructions in SS
  - instruction in right half can’t use it, nor instructions in next slot
- Branch hazard also expands to 3

- Let’s revisit our favorite loop….

Unrolled Loop that Minimizes Stalls for Scalar

```
1 Loop:  LD    F0,0(R1)
2        LD    F6,-8(R1)
3        LD    F10,-16(R1)
4        LD    F14,-24(R1)
5        ADDD  F4,F0,F2
6        ADDD  F8,F6,F2
7        ADDD  F12,F10,F2
8        ADDD  F16,F14,F2
9        SD    0(R1),F4
10       SD    -8(R1),F8
11       SD    -16(R1),F12
12       SUBI  R1,R1,#32
13       BNEZ  R1,LOOP
14       SD    8(R1),F16       ; 8-32 = -
24
```

LD to ADDD: 2 Cycle
ADDD to SD: 3 Cycles

14 clock cycles, or 3.5 per iteration

Loop Unrolling and Superscalar

	Integer instruction	*FP instruction*	*Clock cycle*
Loop:	LD F0,0(R1)	1	
	LD F6,-8(R1)	2	
	LD F10,-16(R1)	ADDD F4,F0,F2	3
	LD F14,-24(R1)	ADDD F8,F6,F2	4
	LD F18,-32(R1)	ADDD F12,F10,F2	5
	SD 0(R1),F4	ADDD F16,F14,F2	6
	SD -8(R1),F8	ADDD F20,F18,F2	7
	SD -16(R1),F12	8	
	SD -24(R1),F16	9	
	SUBI R1,R1,#40	10	
	BNEZ R1,LOOP	11	
	SD -32(R1),F20	12	

- Unrolled 5 times to avoid delays
- 12 clocks, or 2.4 clocks per iteration

Superscalar Flexibility

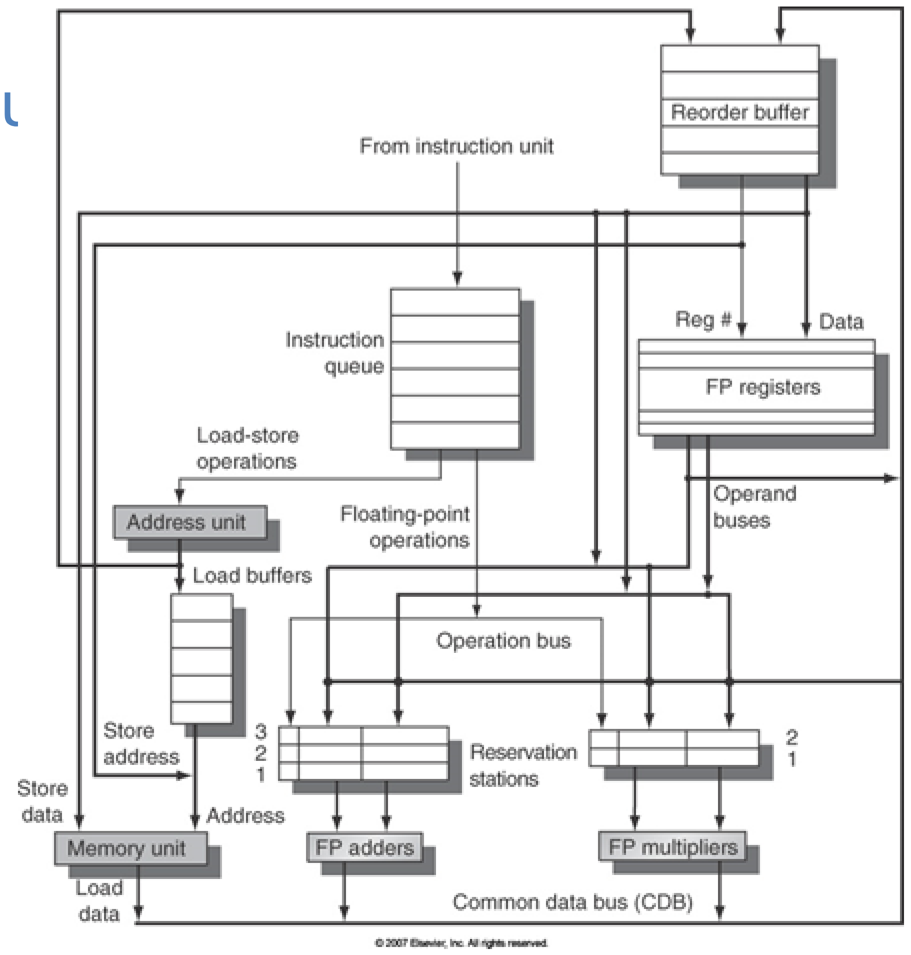
- We quickly got frustrated with superscalar machines that limited what instructions could be scheduled together
  - E.g., 1 int, 1 fp, int at even address, fp at odd address.
- Thus, a modern 4-wide superscalar might be able to do, in a single cycle
  - Up to 4 integer adds
  - Up to 2 (maybe 3?) loads/stores
  - A couple fp adds or multiplies
  - At most 1 fp divide
  - Several branches
- Out-of-order execution makes this much easier, as well, because the instruction grouping (each cycle) is determined by the hw scheduler, not the fetch unit (compiler)

Dynamic Scheduling and Superscalar

- Dependencies stop instruction dispatch in In-order SS
- Code compiled for scalar pipeline will run poorly on SS
  - May want code to vary depending on how superscalar (i.e., recompile for each target pipeline)
- Simple approach: Combine dynamic scheduling (e.g., Tomasulo) with the ability to fetch and issue multiple instructions simultaneously
  - requires multiple CDBs (on Tomasulo/RSs)
  - can complicate updating of register bookkeeping if instructions are dependent, but can be done

Superscalar Dynamic Issu

- Issues/complications?



Performance of Dynamic SS

Iteration no.	*Instructions*	*Dispatch*	*Executes*	*Writes result clock-cycle number*
1	LD F0,0(R1)	1	2	4
1	ADDD F4,F0,F2		1	
1	SD 0(R1),F4	2		
1	SUBI R1,R1,#8		3	
1	BNEZ R1,LOOP		4	
2	LD F0,0(R1)	5		
2	ADDD F4,F0,F2		5	
2	SD 0(R1),F4	6		
2	SUBI R1,R1,#8		7	
2	BNEZ R1,LOOP		8	

4+ clocks per iteration (bottleneck?)

Branches, Decrements still take 1 clock cycle

Addd 4 cycle "latency" (in execute 3 cycles, write result 4th, use 5th)

Ld 3 cycle "latency" (in execute 2 cycles, write result 3rd, use 4th)

Performance of Dynamic SS

Iteration no.	*Instructions*	*Dispatch*	*Executes*	*Writes result clock-cycle number*
1	LD F0,0(R1)	1	2	4
1	ADDD F4,F0,F2		1	
1	SD 0(R1),F4	2		
1	SUBI R1,R1,#8		3	
1	BNEZ R1,LOOP		4	
2	LD F0,0(R1)	5		
2	ADDD F4,F0,F2		5	
2	SD 0(R1),F4	6		
2	SUBI R1,R1,#8		7	
2	BNEZ R1,LOOP		8	

Conservative assumption – can only dispatch 2 if they read from different register files.

4+ clocks per iteration (bottleneck?)

Branches, Decrements still take 1 clock cycle

Addd 4 cycle "latency" (in execute 3, write result 4th, use 5th)

Ld 3 cycle "latency" (in execute 2, write result 3rd, use 4th)

Performance of Dynamic SS

Iteration no.	*Instructions*	*Dispatch*	*Executes*	*Writes result clock-cycle number*
1	LD F0,0(R1)	1	2	4
1	ADDD F4,F0,F2		1	
1	SD 0(R1),F4	2		
1	SUBI R1,R1,#8		3	
1	BNEZ R1,LOOP		4	
2	LD F0,0(R1)	5		
2	ADDD F4,F0,F2		5	
2	SD 0(R1),F4	6		
2	SUBI R1,R1,#8		7	
2	BNEZ R1,LOOP		8	

4+ clocks per iteration (bottleneck?)

Branches, Decrements still take 1 clock cycle

Addd 4 cycle "latency" (in execute 3 cycles, write result 4th, use 5th)

Ld 3 cycle "latency" (in execute 2 cycles, write result 3rd, use 4th)

Limits of Superscalar

- While Integer/FP split is simple for the HW, get CPI of 0.5 only for programs with:
  - Exactly 50% FP operations
  - No hazards
- If more instructions issue at same time, greater difficulty of decode and issue
  - Even 2-scalar => examine 2 opcodes, 6 register specifiers, & decide if 1 or 2 instructions can issue

- In modern superscalar processors, register renaming logic, forwarding logic is quadratic in issue width.
- Register file is complex (4 issue requires 8 read ports and 4 write ports)
- Out-of-order instruction wakeup and scheduling also very complex.
- These are the reasons superscalar essentially stopped at 4.

Superscalar Key Points

- Only way to get CPI < 1 is multiple instruction issue
- SS requires duplicated hardware, more dependence checking
  - Prohibitively difficult to go more than, say, 4-wide
- Without duplication of functional units, will see limited improvement
- SS combined with dynamic scheduling can be powerful

Letting **software schedule hardware** to get **CPI’s < 1.0**

VERY LONG INSTRUCTION WORD (VLIW)

Getting CPI < 1: Issuing Multiple Instructions/Cycle

- Superscalar
  - variable number of instructions issued each cycle
  - parallelism detected in hardware
- Very Long Instruction Words (VLIW)
  - fixed number of instructions issued each cycle
  - parallelism scheduled by the compiler
  - IA-64 (Itanium)

VLIW Processors

- Very Long Instruction Word
- N-wide VLIW issues packets of N instructions simultaneously. Compiler guarantees independence of those N instructions.

add r5, r4, r1	multd f6, f4, f2	lw r2, 0(r7)	sub r8, r6, r1	beqz r9, label
sub r11, r5, r1	addd f8, f0, f6	sw r5, 8(r7)	nop	beqz r2, l2

Loop Unrolling in VLIW

Memory reference 1	*Memory reference 2*	*FP operation 1*	*FP op. 2*	*Int. op/ branch*	*Clock*	
LD F0,0(R1)	LD F6,-8(R1)				1	
LD F10,-16(R1)	2	LD F14,-24(R1)				
LD F18,-32(R1) F8,F6,F2	3	LD F22,-40(R1)		ADDD F4,F0,F2		ADDD
LD F26,-48(R1)	4		ADDD F12,F10,F2		ADDD F16,F14,F2	
		ADDD F20,F18,F2		ADDD F24,F22,F2		5
SD 0(R1),F4	SD -8(R1),F8	ADDD F28,F26,F2				6
SD -16(R1),F12	7	SD -24(R1),F16				
SD -32(R1),F20 R1,R1,#48	8	SD -40(R1),F24				SUBI
SD -0(R1),F28				BNEZ R1,LOOP		9

- Unrolled 7 times to avoid delays
- 7 results in 9 clocks, or 1.3 clocks per iteration
- Need more registers in VLIW

Traditional VLIW vs Intel IA64

- Traditional VLIW (multiflow, transmeta)
  - Hard to fill large VLIW
    - Lack of ILP
    - Poor match of instruction stream vs VLIW slots.
  - Lots of NOPs
- Intel IA64 (Itanium)
  - More flexible groupings of instructions
  - Only 3-wide (but multiple-issue)
    - so up to 6 ops per cycle for initial Itanium
  - Can collapse multiple groups into 1 3-op (not parallel) instruction, so fewer NOPs

Limits to Multi-Issue Machines

- 1 branch in 5 instructions => how to keep a 5-way VLIW busy?
- Latencies of units => many operations must be scheduled
  - Need about (ALU Pipeline Depth) x (No. Functional Units) of independent operations to keep machines busy
- Instruction mix may not match hardware mix
  - need duplicate FUs, increased flexibility
- Increase ports to Register File (VLIW example needs 6 read and 3 write for Int. Reg. & 6 read and 4 write for FP reg)
- Increase ports to memory
- Decoding/renaming SS and impact on clock rate

Superscalar vs. VLIW

- Superscalar Positives

- VLIW Positives

More pipeline more better?

HW CASE STUDY: PENTIUM 4

A pipeline case study

- Intel Pentium 4
  - The apex of OoO processors – subsequent processors backtracked wrt pipeline depth.
  - However, later architectures have larger instruction windows (instruction queues, ROB)

Pentium 4 microarchitecture

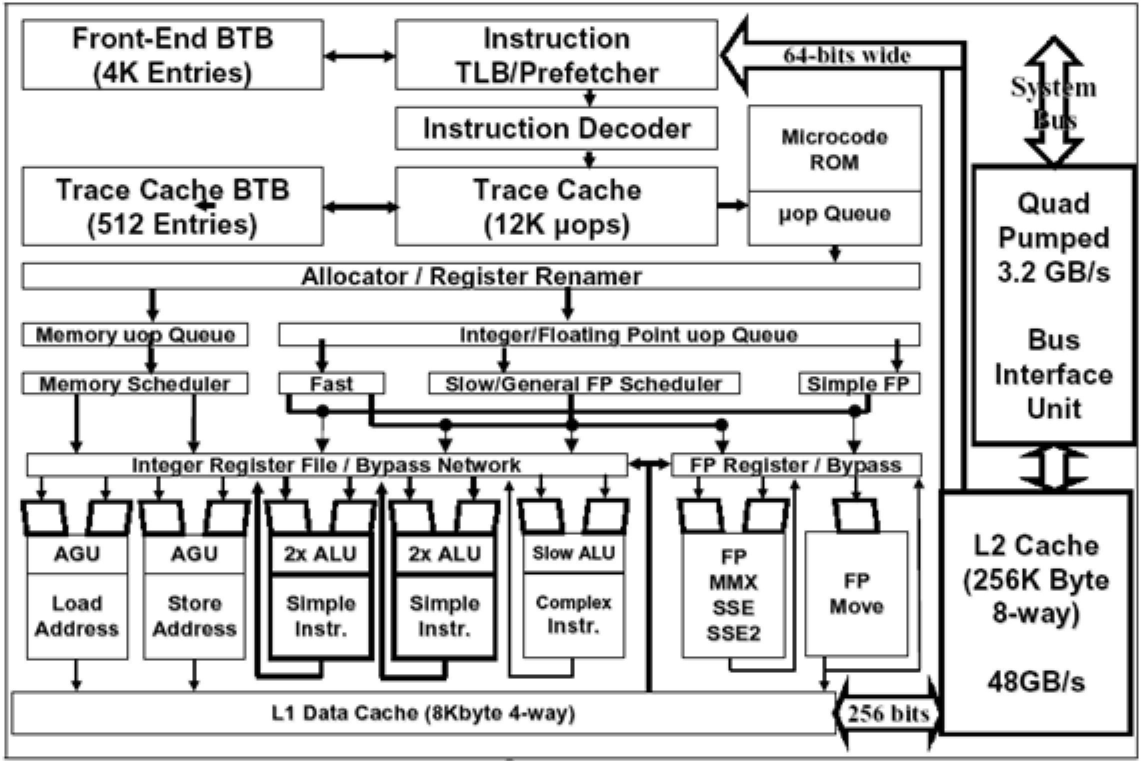
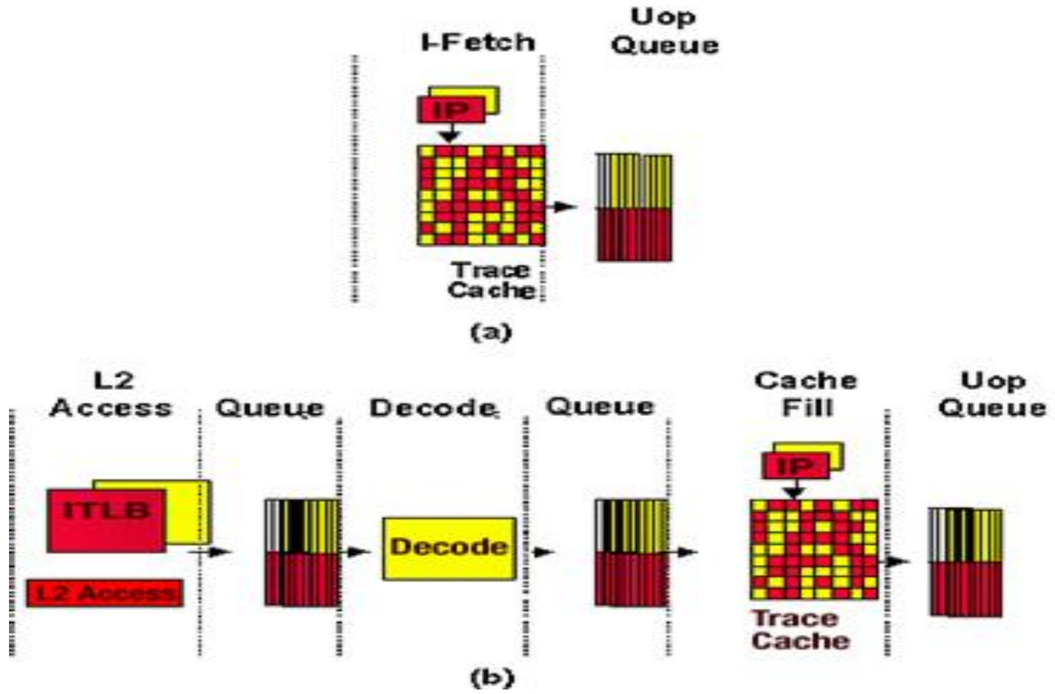
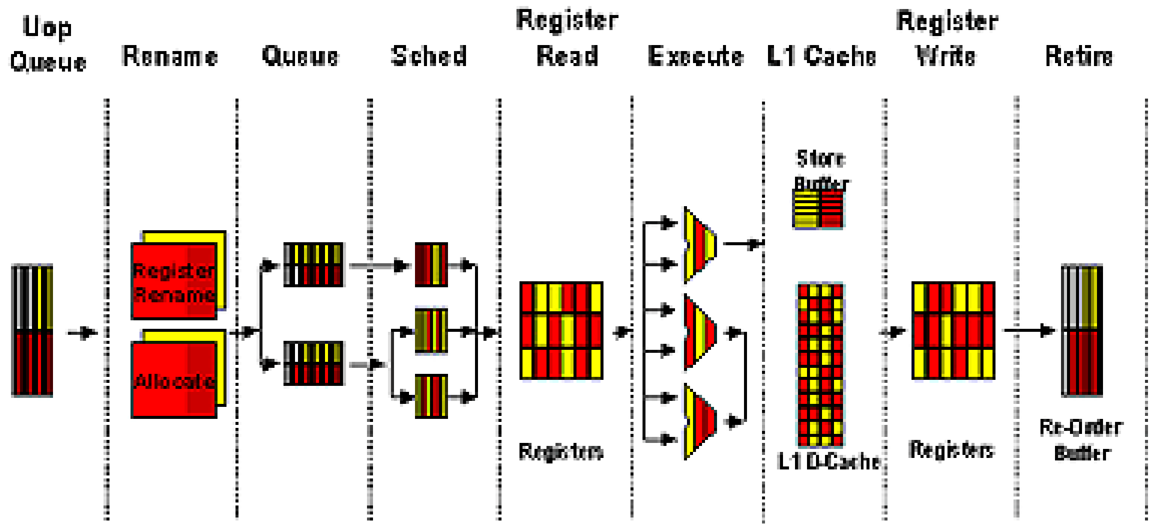


Figure 4: Pentium® 4 processor microarchitecture

Pentium 4 front-end



Pentium 4 back-end



- *126 in-flight* instructions (ROB/active list size)

Pentium 4 Summary

- 20-stage pipeline
- IA32 (x86) ISA translated to RISC-like µops
- µops stored in trace cache
- Decode/retire 3 µops/cycle
- Execute 6 µops/cycle
- Dynamically scheduled
- Explicit Register Renaming
- Simultaneous Multithreading (hyper-threading)

More **reorder-buffer → more better**

HW CASE STUDY: MODERN CORES

Modern (Pre-Multicore) Processors

- Pentium II, III – 3-wide superscalar, out-of-order, 14 integer pipeline stages
- Pentium 4 – 3-wide superscalar, out-of-order, simultaneous multithreading, 20+ pipe stages
- AMD Athlon, 3-wide ss, out-of-order, 10 integer pipe stages
- AMD Opteron, similar to Athlon, with 64-bit registers, 12 pipe stages, better multiprocessor support.
- Alpha 21164 – 2-wide ss, in-order, 7 pipe stages
- Alpha 21264 – 4-wide ss, out-of-order, 7 pipe stages
- Intel Itanium – 3-operation VLIW, 2-instruction issue (6 ops per cycle), in-order, 10-stage pipeline

More Recent Developments – Multicore Processors

- IBM Power 4, 5, 6, 7
  - Power 4 dual core
  - Power 5 and 6, dual core, 2 simultaneous multithreading (SMT) threads/core
  - Power7 4-8 cores, 4 SMT threads per core
- Sun Niagara
  - 8 cores, 4 threads/core (32 threads).
  - Simple, in-order, scalar cores.
- Sun Niagara 2
  - 8 cores, 8 threads/core.
- Intel Quad Core Xeon
- AMD Quad Core Opteron
- Intel Nehalem, Ivy Bridge, Sandy Bridge, Haswell, Skylake, …(Core i3, i5, i7, etc.)
  - 2 to 8 cores, each core SMT (2 threads)
- AMD Phenom II
  - 6 cores, not multithreaded
- AMD Zen
  - 4-8 (mainstream, but up to 32) cores, 2 SMT threads/core, superscalar (6 micro-op/cycle)

Intel SkyLake

- Up to 4 cores (CPUs)
- Each core can have 224 uncommitted instructions in the pipeline
    - Up to 72 loads
    - Up to 56 stores
  - 97 unexecuted instructions in the pipeline waiting to be scheduled
  - Has 180 physical integer registers (used via register renaming)
  - Has 168 physical floating point registers
  - Executes up to 4 (?) micro-ops/cycle (think RISC instructions)
  - Has a 16-cycle branch hazard

- (note—Intel now hiding more and more architectural details)

Intel SkyLake

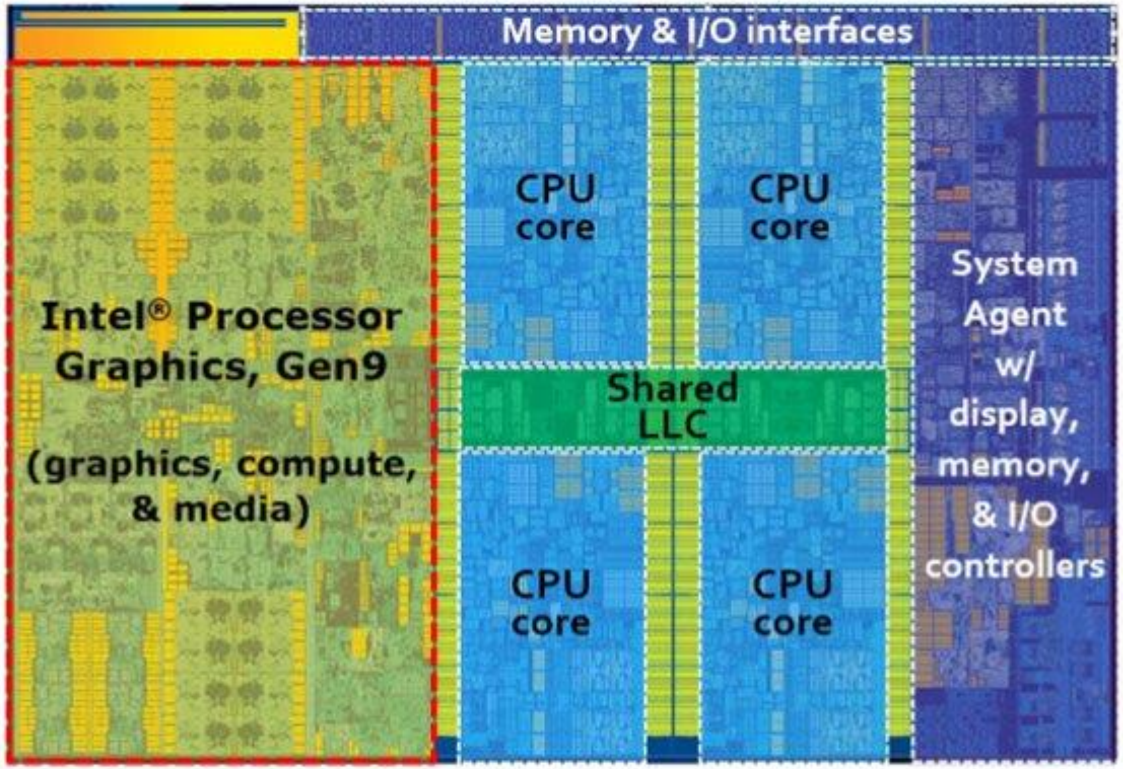

Memory & I/O interfaces
Intel® Processor Graphics, Gen9
(graphics, compute, & media)
CPU core
CPU core
Shared LLC
CPU core
CPU core
System Agent w/ display, memory, & I/O controllers

Figure 1: Architecture components layout for an Intel® Core™ i7 processor 6700K for desktop systems. This SoC contains 4 CPU cores, outlined in blue dashed boxes. Outlined in the red dashed box, is an Intel® HD Graphics 530. It is a one-slice instantiation of Intel processor graphics gen9 architecture.

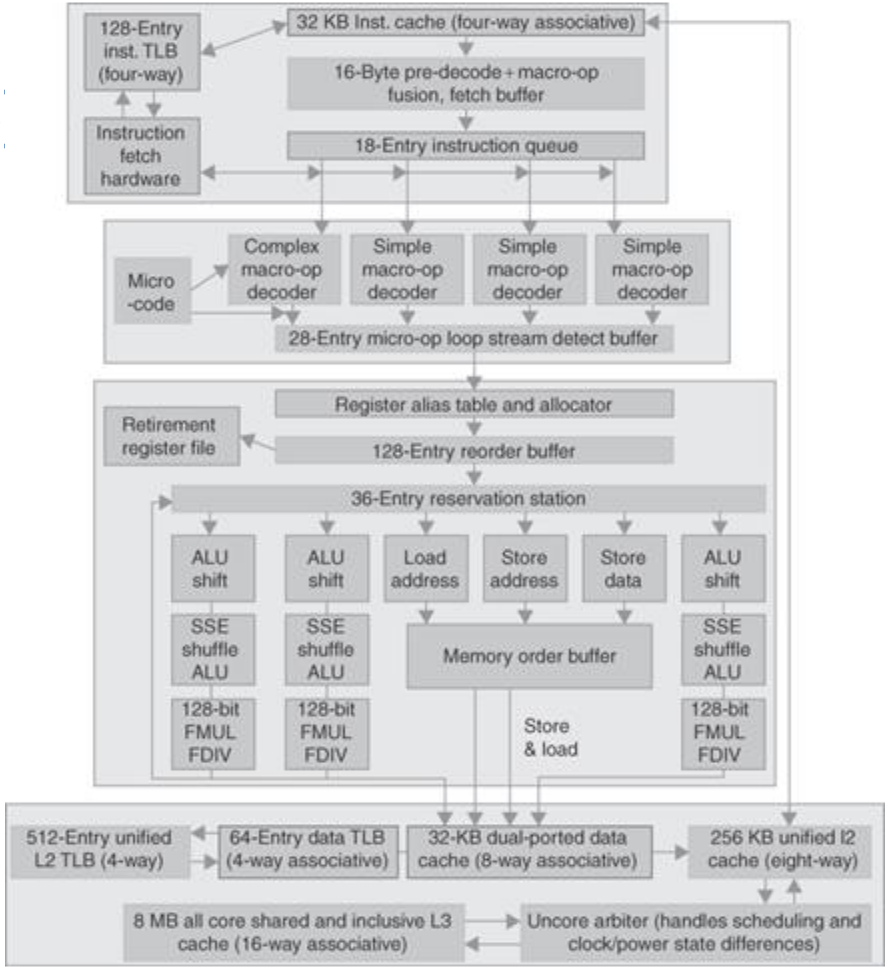
Intel Nehalem (Core I7) [2008

The Intel Core i7 pipeline structure shown with the memory system components.

The total pipeline depth is 14 stages, with branch mispredictions costing 17 cycles.

There are 48 load and 32 store buffers.

The six independent functional units can each begin execution of a ready micro-op in the same cycle.


128-Entry inst. TLB (four-way)
32 KB Inst. cache (four-way associative)
16-Byte pre-decode + macro-op fusion, fetch buffer
Instruction fetch hardware
18-Entry instruction queue
Micro -code
Complex macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
28-Entry micro-op loop stream detect buffer
Register alias table and allocator
Retirement register file
128-Entry reorder buffer
36-Entry reservation station
ALU shift
ALU shift
Load address
Store address
Store data
ALU shift
SSE shuffle ALU
SSE shuffle ALU
Memory order buffer
SSE shuffle ALU
128-bit FMUL FDIV
128-bit FMUL FDIV
Store & load
128-bit FMUL FDIV
512-Entry unified L2 TLB (4-way)
64-Entry data TLB (4-way associative)
32-KB dual-ported data cache (8-way associative)
256 KB unified l2 cache (eight-way)
8 MB all core shared and inclusive L3 cache (16-way associative)
Uncore arbiter (handles scheduling and clock/power state differences)

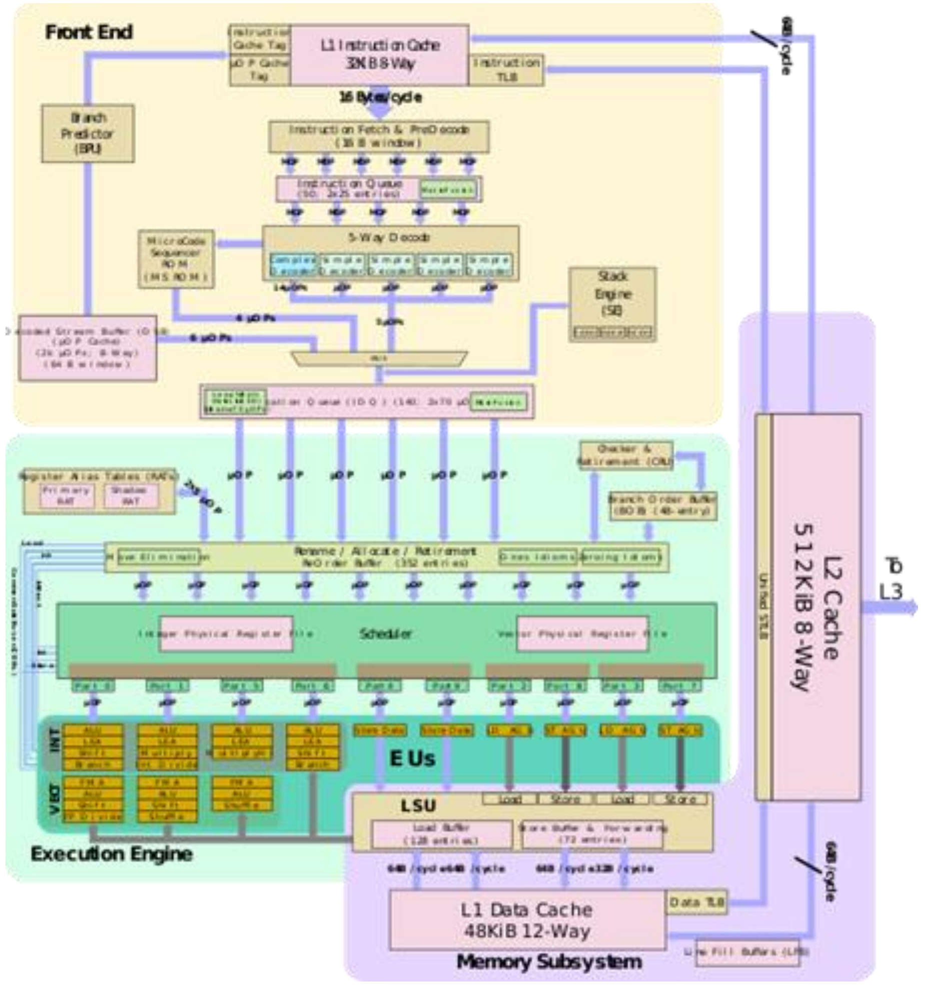
Intel SkyLake [2015]

- Up to 4 cores (CPUs)
- Each core can have 224 uncommitted instructions in the pipeline
    - Up to 72 loads
    - Up to 56 stores
  - 97 unexecuted instructions in the pipeline waiting to be scheduled
  - Has 180 physical integer registers (used via register renaming)
  - Has 168 physical floating point registers
  - Executes up to 4 micro-ops/cycle (think RISC instructions)
  - Has a 16-cycle branch hazard

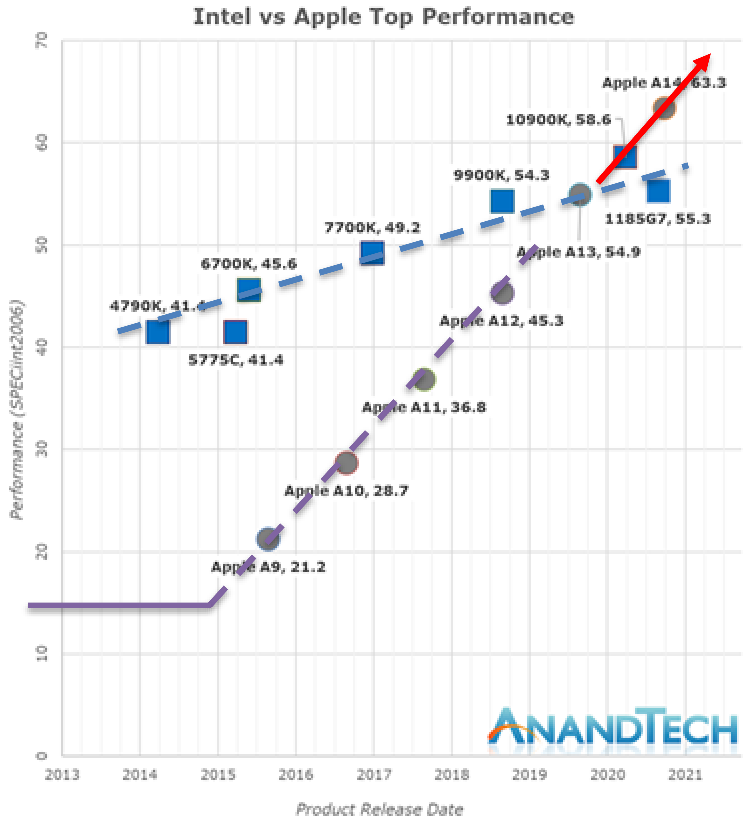
Intel IceLake/SunnyCove [2019]

- Up to 4 cores (client) or 40 (servers)
- Each core can have 352 uncommitted instructions in the pipeline
    - Up to 128 loads
    - Up to 72 stores
  - 160 unexecuted instructions in the pipeline waiting to be scheduled
  - Executes up to 5 micro-ops/cycle (think RISC instructions)

Intel IceLake/SunnyCove [2019]


Front End
L1 Instruction Cache 32KiB 8-Way
Instruction TLB
16 Bytes/cycle
Branch Predictor (BPU)
Instruction Fetch & PreDecode (16 B window)
Instruction Queue
5-Way Decode
MicroCode Sequencer ROM (MS ROM)
Stack Engine (SE)
4 µOPs
6 µOPs
5 µOPs
Rename / Allocate / Retirement ReOrder Buffer (352 entries)
Scheduler
Integer Physical Register File
Vector Physical Register File
EUs
INT
VECT
Execution Engine
LSU
Load
Store
Load Buffer (128 entries)
L1 Data Cache 48KiB 12-Way
Data TLB
Memory Subsystem
L2 Cache 512KiB 8-Way
To L3
64B/cycle

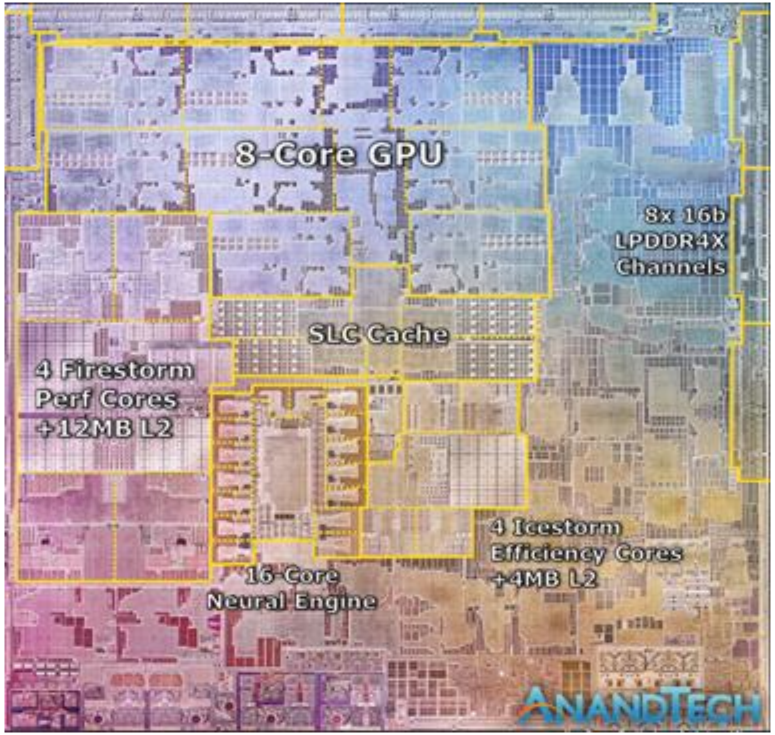
What do we know about the Apple M1? We can learn from the A14 (the M1 ***may*** be a rebranded, lightly enhanced A14)



This part implies they must be doing something different

This part makes a lot of sense for a new player

(Really this section should be what does Andrei Frumusanu know about the M1 – the AnandTech writeup is pretty good)

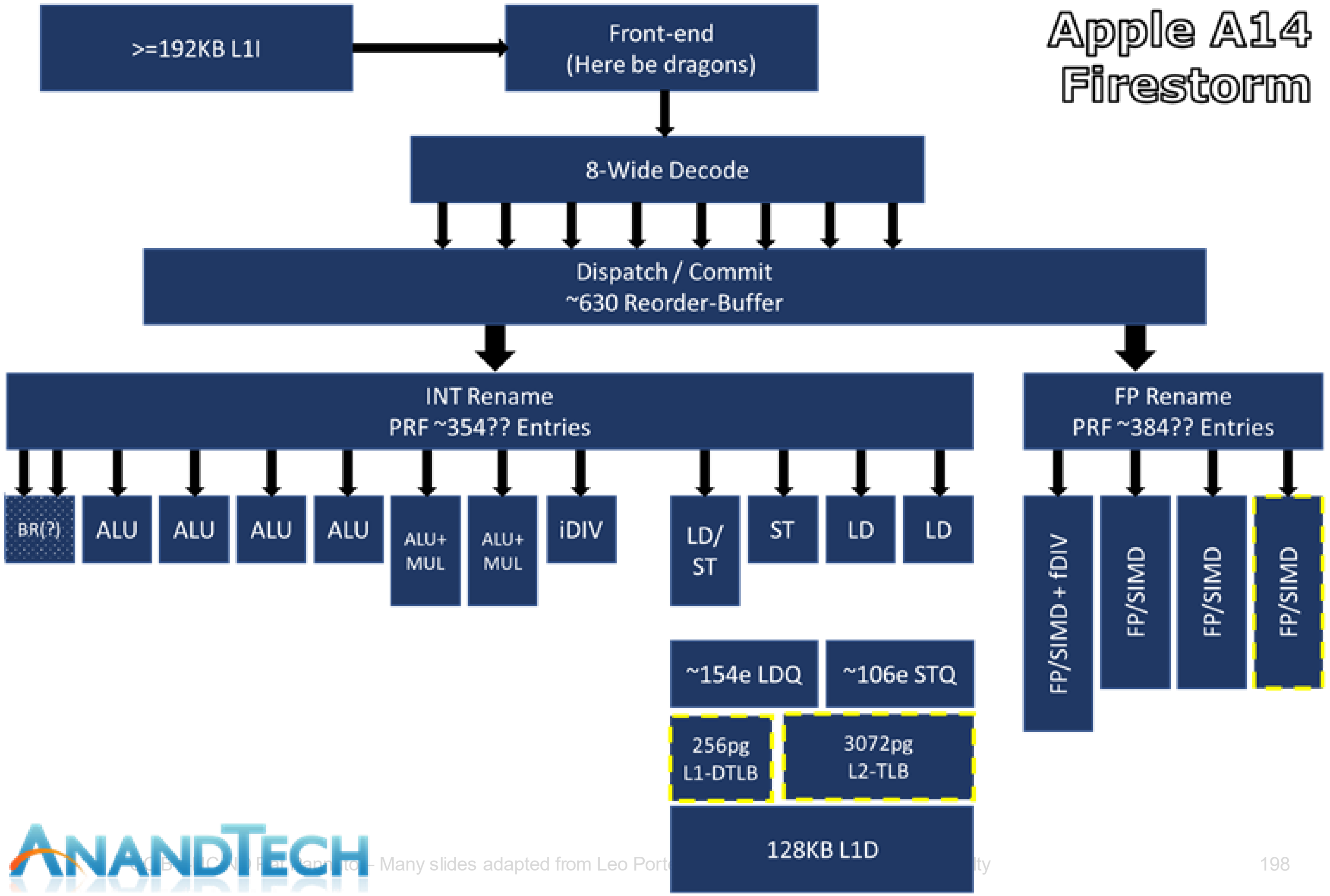


- 12 MB L2 cache [**<u>this is huge</u>**]
  - C.f. Intel Tiger Lake @ 1.25*4 = 5MB
  - C.f. Intel Cooper Lake @ 1*28 = 28MB
    - For $13,000
- Massive ILP
  - 8-wide instruction issue [SMT actual unclear]
  - C.f. Intel's 1+4 [CISC limitation??]
  - C.f. Samsung 6-wide [also ARM]
- Truly massive OoO window
  - **~630 instructions in flight??**
  - C.f. Intel Willow Cove at 352
  - C.f. AMD Zen3 at 256

Much more here: <u>https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive/2</u>

The Internet's Educated Guess

Apple A14 Firestorm



CSE 141 – Many slides adapted from Leo Port ty 198

What does all this HW get us?

BACK TO REALITY & REAL CODE

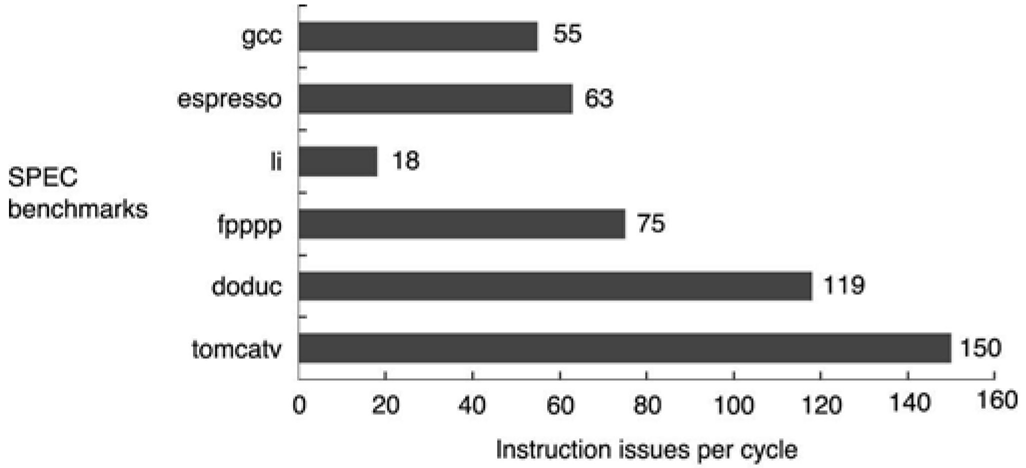
ILP in real code

- A number of years ago, a bunch of studies were done to identify how much inherent ILP could be found in real code (irrespective of the hardware).
- The point was to understand what features restricted our ability to exploit parallelism (since there was a huge difference between inherent parallelism [~100x] and exploited parallelism [2x?])
- Those are fairly obsolete now (workloads not that relevant anymore), but impact can still be seen:
  - Very large window sizes
  - Importance of hardware memory disambiguation

Less important in modern, memory-safe languages — *but* vast majority of code is not written in those

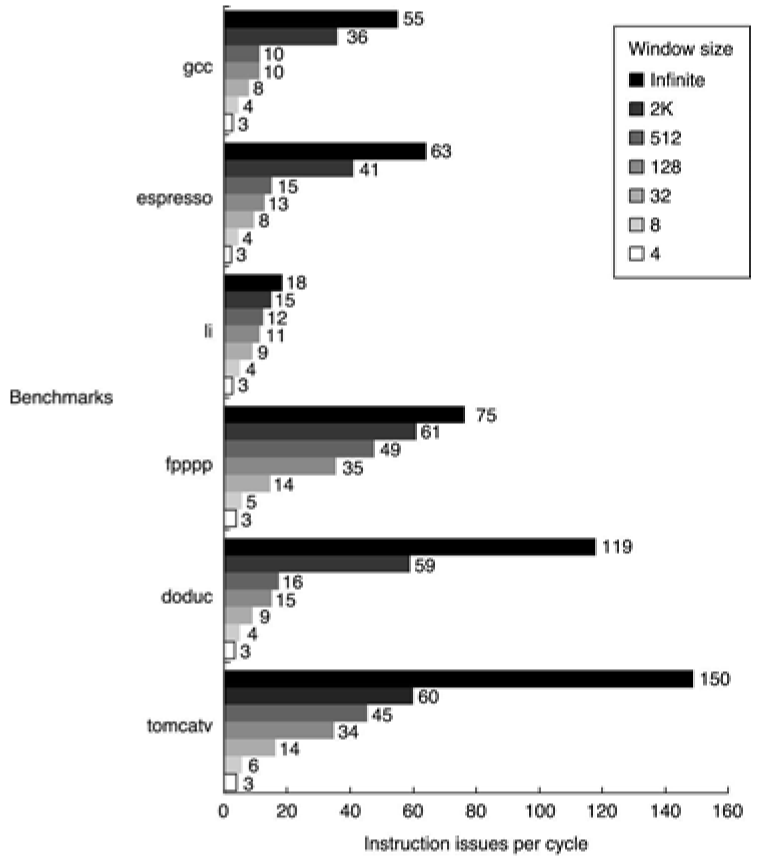
Even Python -> cPython…

ILP in real code

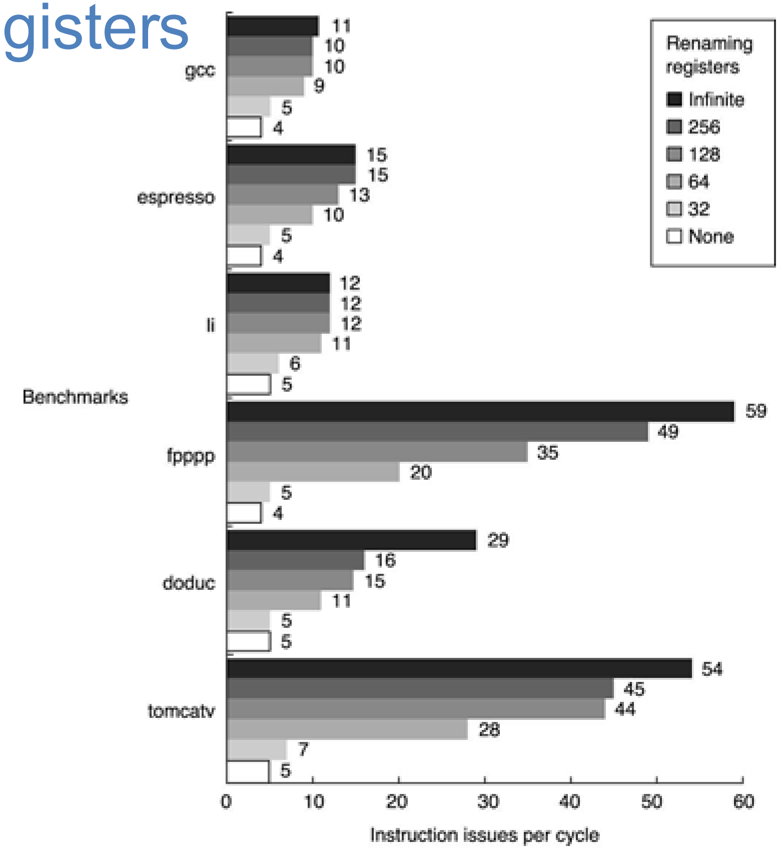


- Based on all kinds of ideal assumptions.
- Further limited by:
  - realistic branch prediction
  - finite renaming registers/scheduling window
  - imperfect alias analysis for memory operations

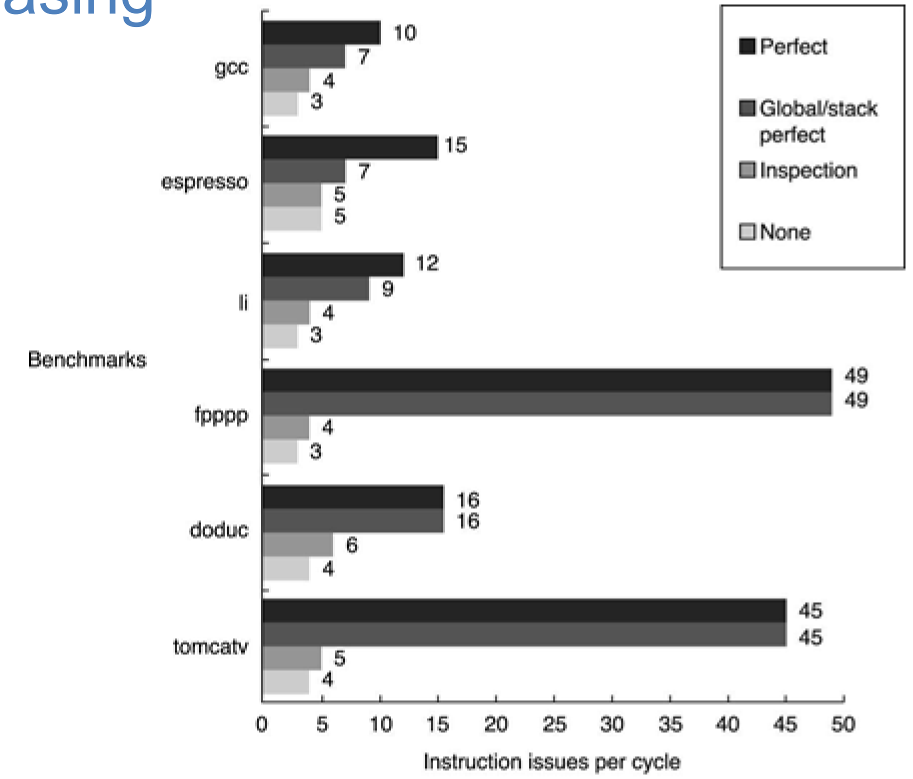
Window Size


Window size
Infinite
2K
512
128
32
8
4
Benchmarks
gcc
55
36
10
10
8
4
3
espresso
63
41
15
13
8
4
3
li
18
15
12
11
9
4
3
fpppp
75
61
49
35
14
5
3
doduc
119
59
16
15
9
4
3
tomcatv
150
60
45
34
14
6
3
0
20
40
60
80
100
120
140
160
Instruction issues per cycle

Renaming Registers



Memory Aliasing



A quick survey of other techniques

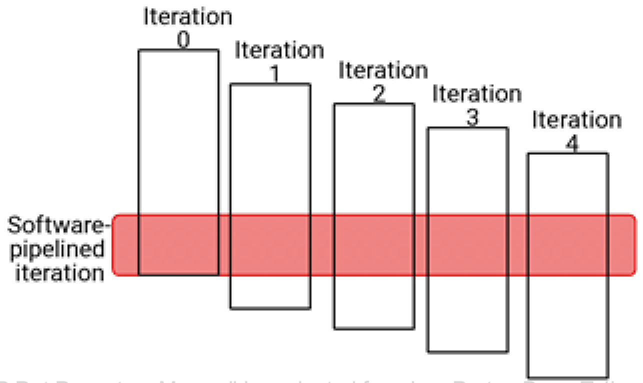
HOW DO WE FIND/EXPOSE MORE ILP?

Exposing More ILP

- These techniques (in purple) were originally motivated by VLIW, which needs tons of ILP to work at all – but useful for superscalar/dynamic/speculative processors, as well.
- Software Techniques
  - Software Pipelining
  - Trace Scheduling
- Hardware/Software Techniques (in use)
  - Predicated execution
  - Simultaneous Multithreading (but we'll talk about this later)
- Some other hardware/sw techniques (not in use)
  - Value prediction
  - Thread level speculation

Technique 1: Compiler support for ILP: Software Pipelining

- Observation: if iterations from loops are independent, then can get ILP by taking instructions from different iterations
- Problems with loop unrolling: not regular (overlaps some iterations but not others), inflates code, quickly exhausts all registers.
- Software pipelining: reorganizes loops so that each iteration (of the new loop) is made from instructions chosen from different iterations of the original loop
- Goal: get the effect of massive loop unrolling without the massive code expansion.



SW Pipelining Example

Unrolled 3 times

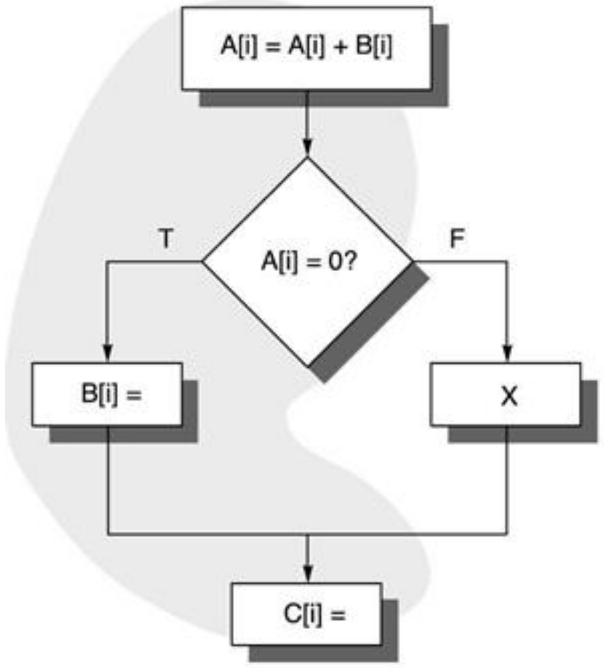
```
1   LD    F0,0(R1)
2   ADDD  F4,F0,F2
3   SD    0(R1),F4

4   LD    F6,-8(R1)
5   ADDD  F8,F6,F2
6   SD    -8(R1),F8

7   LD    F10,-16(R1)
8   ADDD  F12,F10,F2
9   SD    -16(R1),F12
10  SUBI  R1,R1,#24
11  BNEZ  R1,LOOP
```

Software Pipelined

```
         LD    F0,0(R1)
         ADDD  F4,F0,F2
         LD    F0,-8(R1)
1LP:SD         0(R1),F4;     Stores M[i]
2        ADDD  F4,F0,F2;     Adds to M[i-1]
3        LD    F0,-16(R1);   loads M[i-2]
4        SUBI  R1,R1,#8
5        BNEZ  R1,LP
         SD    0(R1),F4
         ADDD  F4,F0,F2
         SD    -8(R1),F4
```

Technique 2: Compiler Support for ILP: Trace Scheduling

- Creates long basic blocks by finding long paths in the code



Trace Scheduling

- Parallelism across IF branches vs. LOOP branches
  - (loop unrolling and sw pipelining address loop branches)
- Two steps:
  - *Trace Selection*
    - Find likely sequence of basic blocks (trace) of (statically predicted) long sequence of straight-line code
  - *Trace Compaction*
    - Squeeze trace into few VLIW instructions
    - Need bookkeeping code in case prediction is wrong

HW Support for More ILP

- Predication – Trading off branch hazards and control flow constraints for increased instruction bandwidth
- Value Prediction
- Thread Level Speculation

Technique 3:
Predication: HW support for More ILP

- Avoid branch prediction by turning branches into *conditionally executed instructions*: (aka predicated instructions)

  add c, a, b (x)   => if (x) then c = a + b else NOP

  – If false, then neither store result nor cause exception

```
                    ld     F2, 0(R2)
                    addd   F4, F2, F0
                    multd  F6, F4, F4
                    beqz   R3, go_on
                    nop
                    addd   F10, F0, F8
                    addi   R2, R2, #8
go_on:      addi    R2, R2, #8
                    bnez   F6, loop
```

Predicated Execution

- Drawbacks to conditional instructions
  - Still takes a clock & alu even if "annulled"
  - Stall if condition evaluated late
  - Complex conditions reduce effectiveness; condition becomes known late in pipeline
  - requires more operands! Often only available as conditional move.
- Advantages
  - eliminate prediction, misprediction
  - longer basic blocks, ...

- Critical technology for VLIW, sw pipelining. Why?

Predication in Real Processors

- Expanded ISA of Alpha, MIPS, PowerPC, SPARC have conditional move
- PA-RISC can annul any following instr
- Intel IA64 can predicate any instruction (even have multiple predicates).
  - Have a full register file just to hold 1-bit predicates.
- Intel IA32 has conditional moves
- ARM (A-series) has full support for predication.
  - Any instruction can be conditional on a set of flags
    - Zero, carry, greater-than, overflow, etc.
    - Flags set by previous arithmetic instructions
- RISC-V – no support for predication!

Technique 4: Value Prediction

- Once we…
  - eliminate false dependences through register renaming
  - have large instruction windows (instruction queues, ROBs, etc.)
- …then our primary problem becomes true dependencies
  - Some of them large (long latencies)
- Can we execute faster than true dependencies allow?

Value Prediction

- Value prediction allows the speculative completion of a stalled or long-latency operation by guessing the result of the computation.
- This allows subsequent instructions to execute before the predicted instruction completes.
- Recovery from misprediction is not that different from a branch mispredict
  - E.g., if we have a reorder buffer, all speculative values are easily flushed before they reach the register file.

Lipasti (1996), Sazeides (1997)

Value Prediction

- Can be particularly helpful with long-latency loads.

```
ld      R5, 1000(R1)

 (300 cycles)
```

- We also showed that sometimes we can get the same effect predicting short (e.g., 1-cycle) instructions.

```
dadd     R7, R5, R2
ld       R9, 0(R7)

(300 cycles)
```

Value Prediction

- Early work showed surprising quantities of predictable instructions (sometimes 70-80%)
  - Loads of highly redundant structure (e.g., sparse matrix with lots of zeroes)
  - Loads of constants
  - Induction variables (constant stride – 1,2,3,…)
  - …

- Need a prediction mechanism
  - Table of last values, stride delta, confidence counter

ILP Recap

- Parallelism is critical to modern computer system performance
  - Both at a **fine level (instruction level parallelism)**
  - And a **coarse level (thread level parallelism)**
- Mechanisms that create, or expose ILP:
  - loop unrolling, software pipelining, code motion
- Mechanisms that allow the machine to exploit ILP:
  - pipelining, superscalar, dynamic scheduling, speculative execution, and more advanced speculative techniques.

Advanced Pipelining – Key Points

- Scalar* pipelining aims to get **CT lower** while **keeping CPI close to 1**.
  - Reminder: Various hazards are what keeps CPI from being 1 in practice
  - *[fancy word for non-parallel]
- To improve performance, we must reduce cycle time (superpipelining) or get CPI below 1 (superscalar, VLIW).
  - What are the costs / problems of pipelining too deeply?
  - When does better CT no longer improve ET?
- Modern processors are fast because they work on *many hundred* instructions at once
- Simple pipelines are valuable when raw performance is less important
  - Specialization can be more efficient, but only if you know workload!