CS3350B Computer Organization
Chapter 4: Instruction-Level Parallelism
Part 3: Beyond Pipelining

Iqra Batool

Department of Computer Science
University of Western Ontario, Canada

Monday March 11, 2024

Outline

Instruction-Level Parallelism (ILP)

Instruction-level parallelism involves executing multiple instructions at the same time.

- Instructions may simply overlap (pipelining) or,
- Instructions may be executed completely in parallel (**superscalar**).

There are many techniques which are used to provide ILP or to support ILP in achieving greater speed-up.

- Pipelining.
- Branch prediction.
- **Superscalar** execution.
- **Very Long Instruction Word** (VLIW).
- **Register renaming**.
- **Loop unrolling**.

Multiple Issue Processors

- A **multiple issue processor** issues (executes) multiple instructions within a clock cycle. (Aims for CPI < 1)
  - 3 types:
    - VLIW Processors.
    - Static Superscalar Processors (essentially same as VLIW).
    - Dynamic Superscalar Processors.

  through pipelining, CPI approaching 1 (but never 1)

- By their nature, all multiple issue processors have multiple execution units (ALUs) in their datapath.
- Depending on the type of multiple issue processor, other circuitry may also be duplicated or augmented.
- *Note:* multiple issue processors are not necessarily pipelined (these concepts are separate) but in reality pipelining is so good and multiple issue came after so all multiple issue processors are also pipelined.

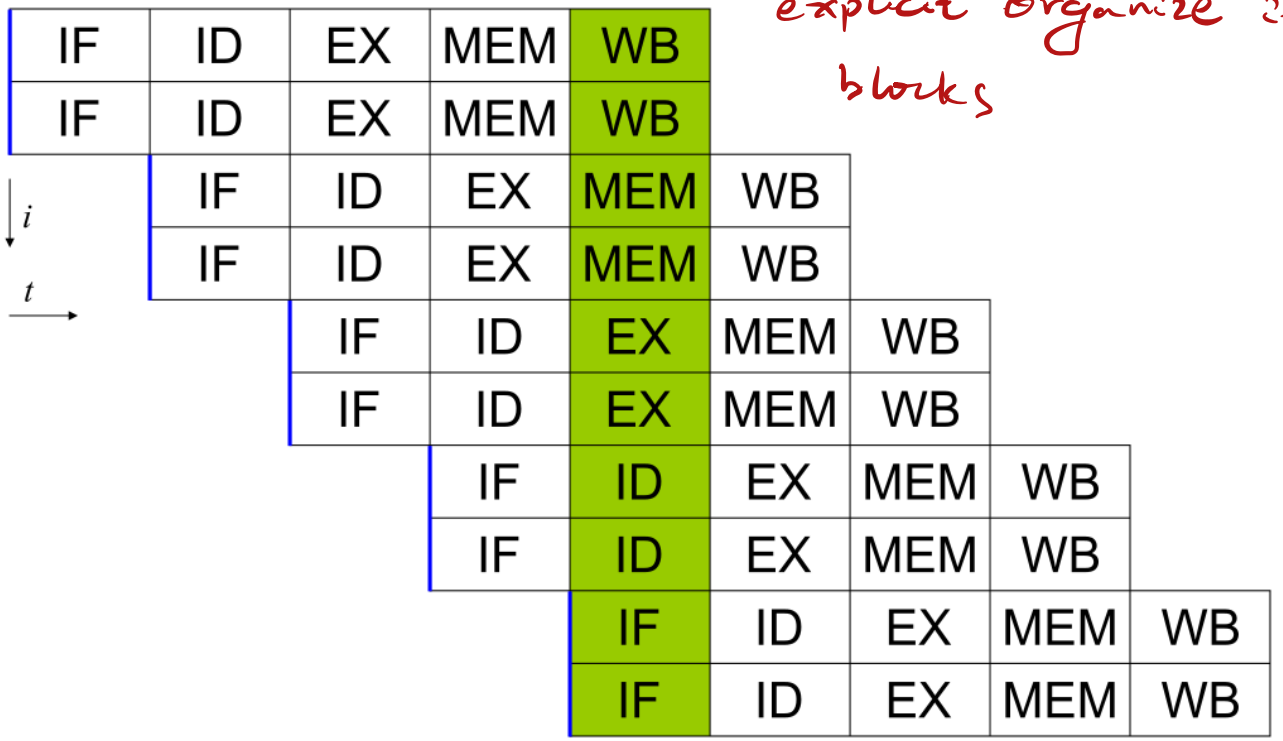
Static Superscalar Processors

multiple instruction execution together.

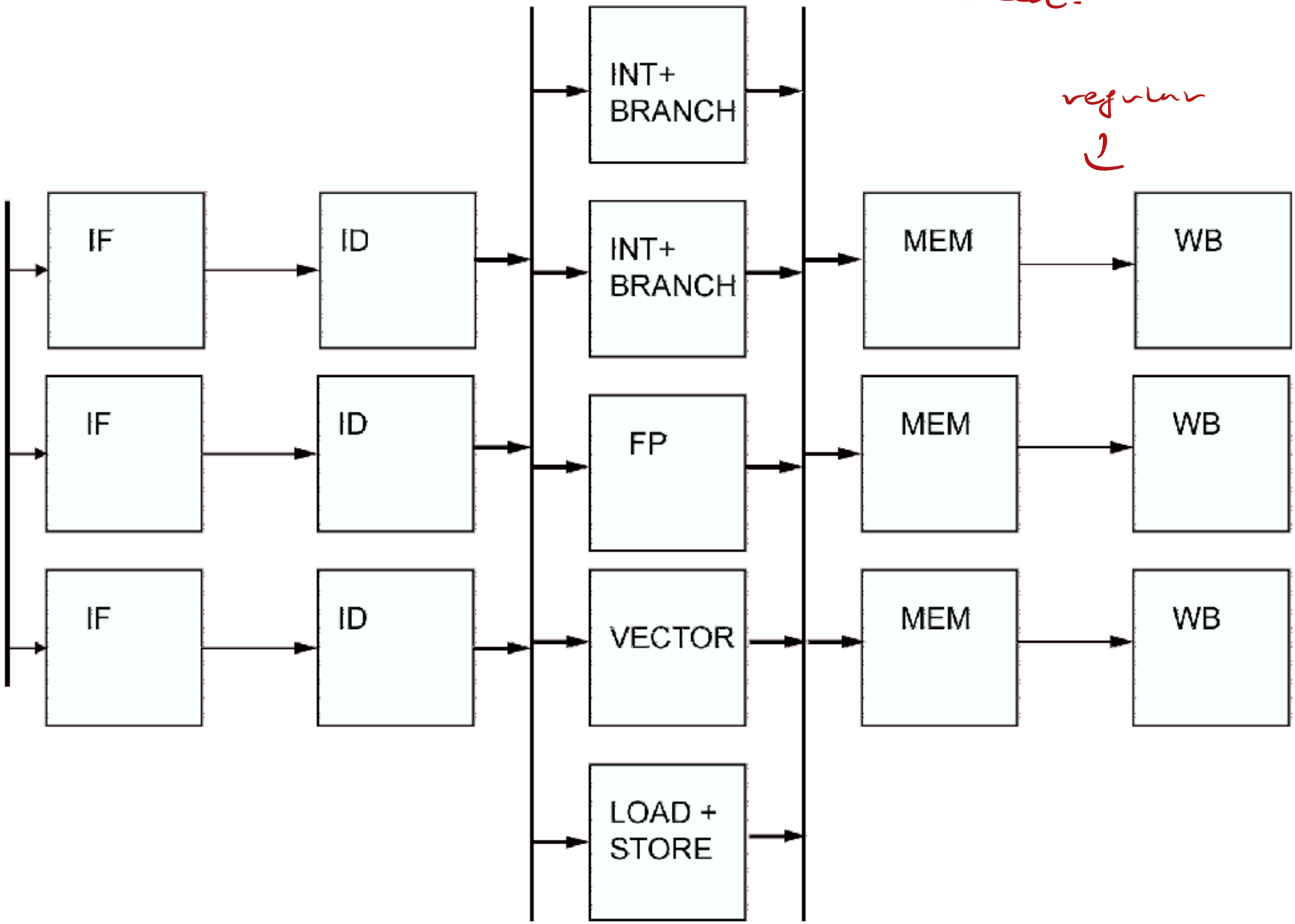
- The name static implies the **code scheduling** is done by compiler.
- Basically side-by-side datapaths simultaneously executing instructions.
- Compiler handles dependencies and hazards and scheduling code so that instructions on different datapaths don't conflict.
- Near identical to VLIW so we'll skip the details.

explicit organize it into blocks

duplicated datapath.



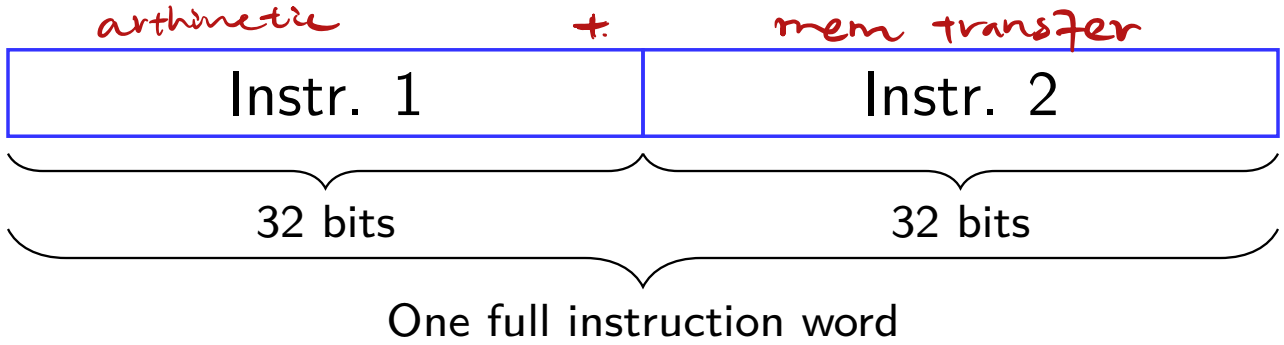
Static Superscalar Pipeline



Outline

VLIW Processors (1/2)

- VLIW processors have *very long instruction words*.
- Essentially, multiple instructions are encoded within a single (long) instruction memory word called an **issue packet**.
- The instructions which can be packed together are limited. Usually only one `lw`/`sw`, only one branch, rest arithmetic.
- In this case instructions word size ≠ data memory word size.
- Simplest scheme: just concatenate multiple instructions together.
- Ex: Two 32-bit instrs. together in a single 64-bit instruction word.

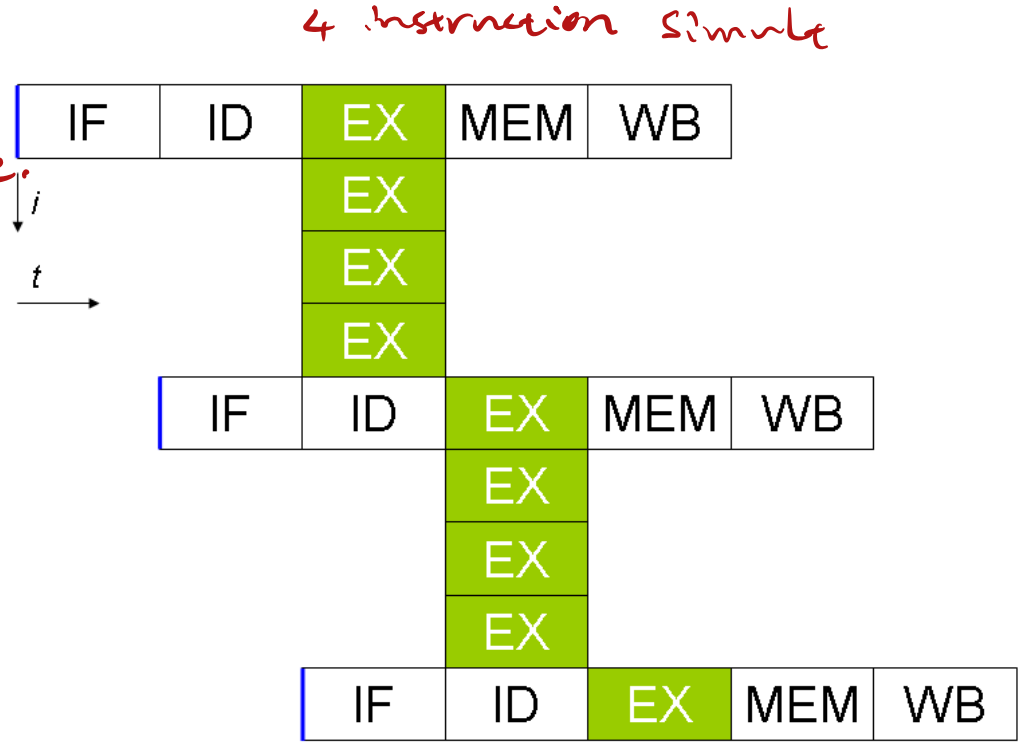


VLIW Processors (2/2)

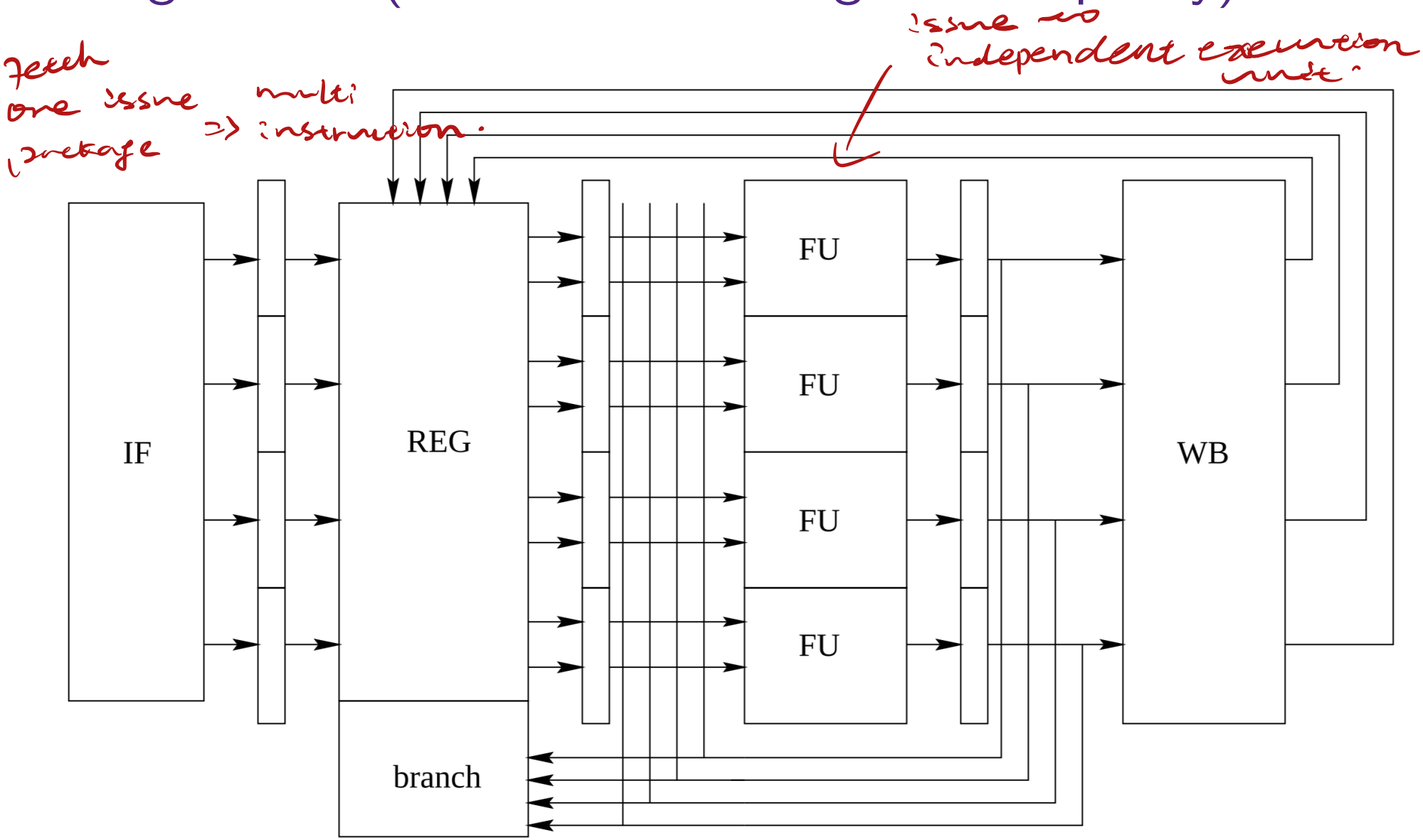
In a VLIW pipeline:

- One IF unit fetches a single long word encoding multiple instrs.
- ID stage must handle multiple simultaneous decodes and register reads.
- In EX stage, each instr. is **issued** to a different execution unit (ALU).

- Only one data memory to read/write from! *There is a limitation on which kinds of instructions can be executed simultaneously.*
- In the WB stage the register file must handle multiple writes (to different registers, obviously).



4-Stage VLIW (without MEM stage for simplicity)

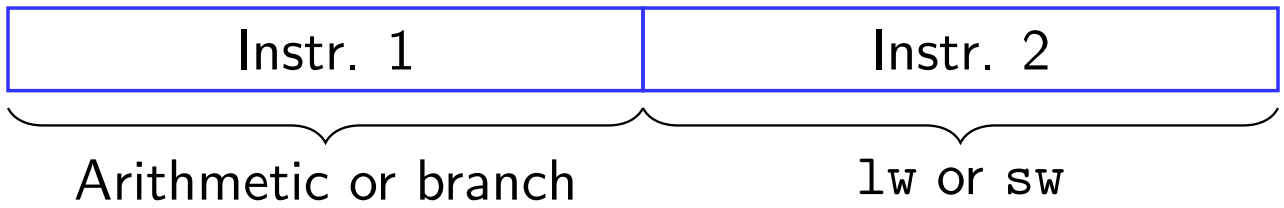


M. Oskin et al. Exploiting ILP in page-based intelligent memory. In *ACM/IEEE International Symposium on MICRO-32*, Proceedings, pages 208-218, 1999.

A VLIW Example (1/3)

Consider a 2-issue extension of MIPS.

- The first slot of the issue packet must be an arithmetic (R- or I-type) instruction or a branch.
- The second slot of the issue packet must be `lw` or `sw`.
- If compiler cannot find an instruction, insert `nop`.
  - ↳ Much like load-delay slot or branch-delay slot.



A VLIW Example (2/3)

read after write

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

WAR dependency

```
for (int i=n; i>0; --i) {
    A[i] += s2;
}
```

Need to schedule code for 2-issue.

- Instructions in same issue packet must be independent.
- Assume perfect branch prediction.
- Load-use and RAW dependencies still need to be handled.
  - ↳ But, assume all possible datapath optimizations (forwarding).

A VLIW Example (3/3)

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

	ALU or branch	Data transfer	CC
loop:	nop	lw $t0, 0($s1)	1
	addi $s1, $s1, -4	nop	2
	addu $t0, $t0, $s2	nop	3
	bne $s1, $0, loop	sw $t0, 4($s1)	4

- Scheduled 5 instructions in 4 cycles. In the limit, CPI approaches 0.8.
- nops don't count towards performance.
- Sometimes when scheduling code you need to adjust offsets.

Outline

Loop Unrolling : 3 benefits.

- Compilers use **loop unrolling** to *expose more parallelism*. (1)
- Essentially, body of loop is replaced with multiple copies of itself (still in a loop).
- Avoids unnecessary branch delays, can more effectively schedule code and fill load-use slots. (2)
- Ex: 4-time loop unrolling. Notice `i += 4` in unrolled code. (3) and we don't have to explore conditions that frequently

```
int i = 0;
for (i; i < n; i++) {
    A[i] += 10;
}
```

```
int i = 0;
for (i; i < n; i += 4) {
    A[i] += 10;
    A[i+1] += 10;
    A[i+2] += 10;
    A[i+3] += 10;
}
```

Unrolling MIPS

```
loop:   lw $t0, 0($s1)
        addu $t0, $t0, $s2
        sw $t0, 0($s1)
        addi $s1, $s1, -4
        bne $s1, $0, loop
```

```
for (int i=n; i>0; --i) {
    A[i] += s2;
}
```

1 increment & 1 branching cst

```
loop: lw   $t0,0($s1)  # $t0=array element
      lw   $t1,-4($s1) # $t1=array element
      lw   $t2,-8($s1) # $t2=array element
      lw   $t3,-12($s1)# $t3=array element
      addu $t0,$t0,$s2 # add scalar in $s2
      addu $t1,$t1,$s2 # add scalar in $s2
      addu $t2,$t2,$s2 # add scalar in $s2
      addu $t3,$t3,$s2 # add scalar in $s2
      sw   $t0,0($s1)  # store result
      sw   $t1,-4($s1) # store result
      sw   $t2,-8($s1) # store result
      sw   $t3,-12($s1)# store result
      addi $s1,$s1,-16 # decrement pointer
      bne  $s1,$0,loop   # branch if $s1 != 0
```

lw 4

add 4

sw 4

update counter by 1 big step.

- Notice, loop body is not exactly copied 4 times.
- Static register renaming: $t0 becomes $t1, $t2, $t3 for successive loops.
- Can now easily reschedule code and fill load-delay slots.
- Much fewer branch instr. and branch delay slots.

Combining Loop Unrolling and VLIW

Same 2-issue extension of MIPS. First slot is ALU, second slot is data transfer. Datapath has all forwarding and possible optimizations.

Remember:

- Need one instruction between load and use.
- Insert `nop` if no instruction possible

	ALU or branch	Data transfer	CC
loop:	addi $s1,$s1,-16	lw $t0,0($s1)	1
	nop	lw $t1,12($s1) #-4	2
	addu $t0,$t0,$s2	lw $t2,8($s1) #-8	3
	addu $t1,$t1,$s2	lw $t3,4($s1) #-12	4
	addu $t2,$t2,$s2	sw $t0,16($s1) #0	5
	addu $t3,$t3,$s2	sw $t1,12($s1) #-4	6
	nop	sw $t2,8($s1) #-8	7
	bne $s1,$0,loop	sw $t3,4($s1) #-12	8

- Scheduled 14 instructions in 8 cycles. In the limit, CPI $\rightarrow$ 0.57..

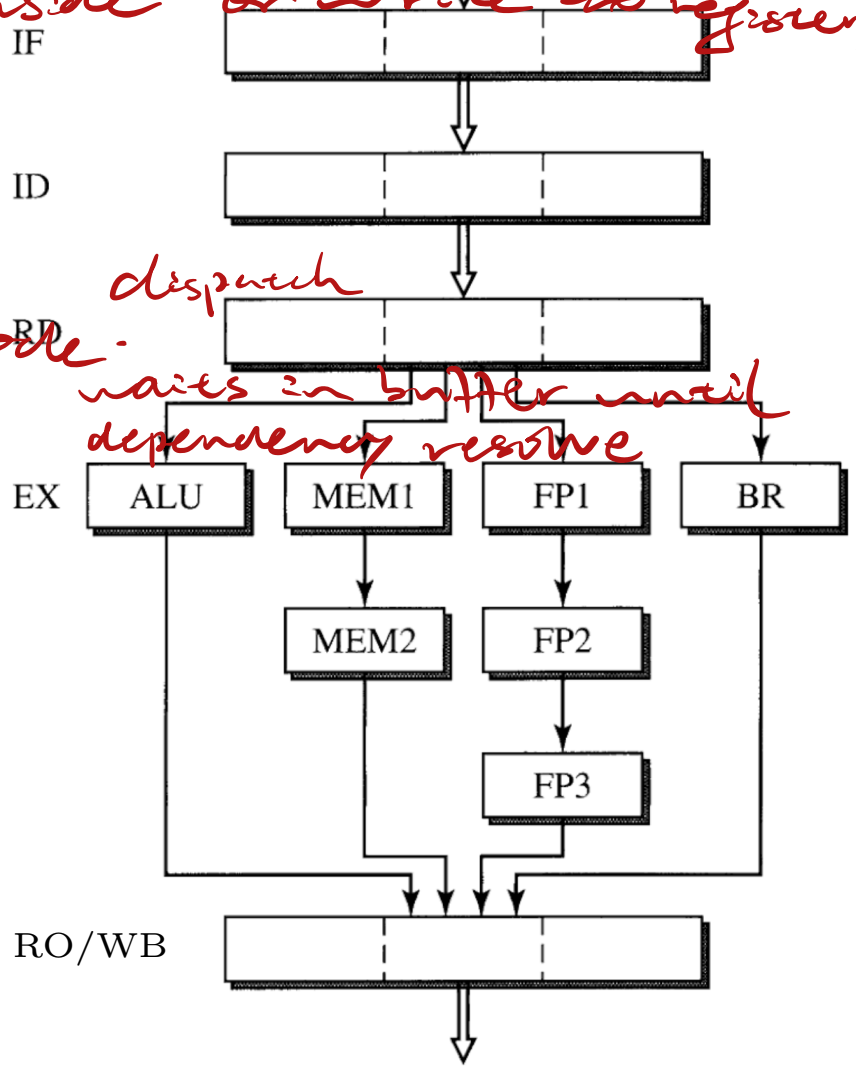
Outline

Dynamic Superscalar Processors

- Dynamic in the name implies that the hardware handles code scheduling.
- Because of the dynamic nature **out-of-order execution** can occur.
  - ↳ Instructions are actually executed in a different order than they are fetched.
- This scheme allows for different types of execution paths which all take a different amount of time:
  - ↳ Ex: Normal ALU, Floating point unit, Memory load/store path.
- This scheme is particularly good at overcoming stalls due to cache misses and other dependencies.
- However, hardware becomes *much* more complex than static schemes.

A Dynamic Superscalar Pipeline (1/2)

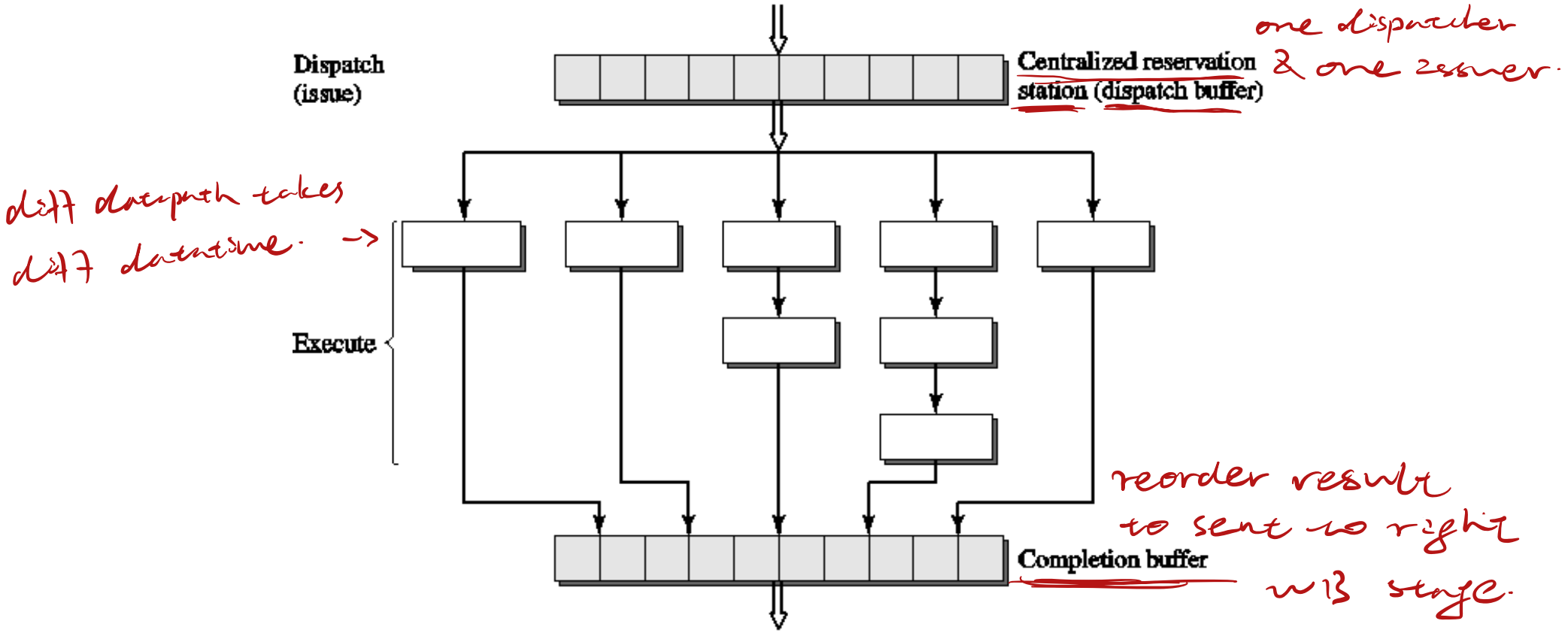
- Fetch one instr. per cycle as normal.
- "Pre-decode" instr. and add to **instruction buffer** in **RD** (dispatch) stage.
  - ↳ Out-of-order execution ⇒ must wait for updated values of operands.
- Once instr. operands are ready, **dispatcher** sends instr. to one of many execution units. This is where out-of-order execution is introduced.
- **Issue**: add instr. to buffer
- **Dispatch**: send instr. to exec. unit



Note: Dispatch/issue naming sometimes reversed (IBM vs Intel/AMD); what is important is that there are two steps between fetch and execution.

A Dynamic Superscalar Pipeline (2/2)

- **Before** WB stage, a **reorder buffer** or **completion stage** makes sure instructions are in-order before writing results back (a.k.a **committing**).
  - ↳ Out-of-order execution means WAR and WAW dependencies matter.
- Finally, instructions are **retired** and "exit" the datapath.



A **centralized reservation station** is both a dispatcher and issuer.

Pros and Cons of Dynamic Scheduling

- Compiler is only so good at scheduling code. Data hazards are hard to resolve. (statically)
  - ↳ Compiler only sees pointers but hardware sees actual memory addresses.
- Dynamic scheduling overcomes unpredictable stalls (cache misses) but requires complex circuitry. buffers, etc.
- Dynamic scheduling more aggressively overlaps instructions since operand values are read and then queued in reservation stations.
- Instructions are fetched in-order, executed out-of-order, and then committed/retired in-order and sequentially.
- *Flynn's bottleneck*: can only retire as many instr. per cycle as are fetched.
  - ↳ Superscalar machines usually augmented with fetching *multiple* instructions at once.

fetching instruction limit performance of superscalar processor.

Comparing Multiple Issue Processors (1/2)

VLIW

- Static scheduling.
- In-order execution.
- Single IF unit but many EX units.
- Instructions packed together in issue packet by compiler.

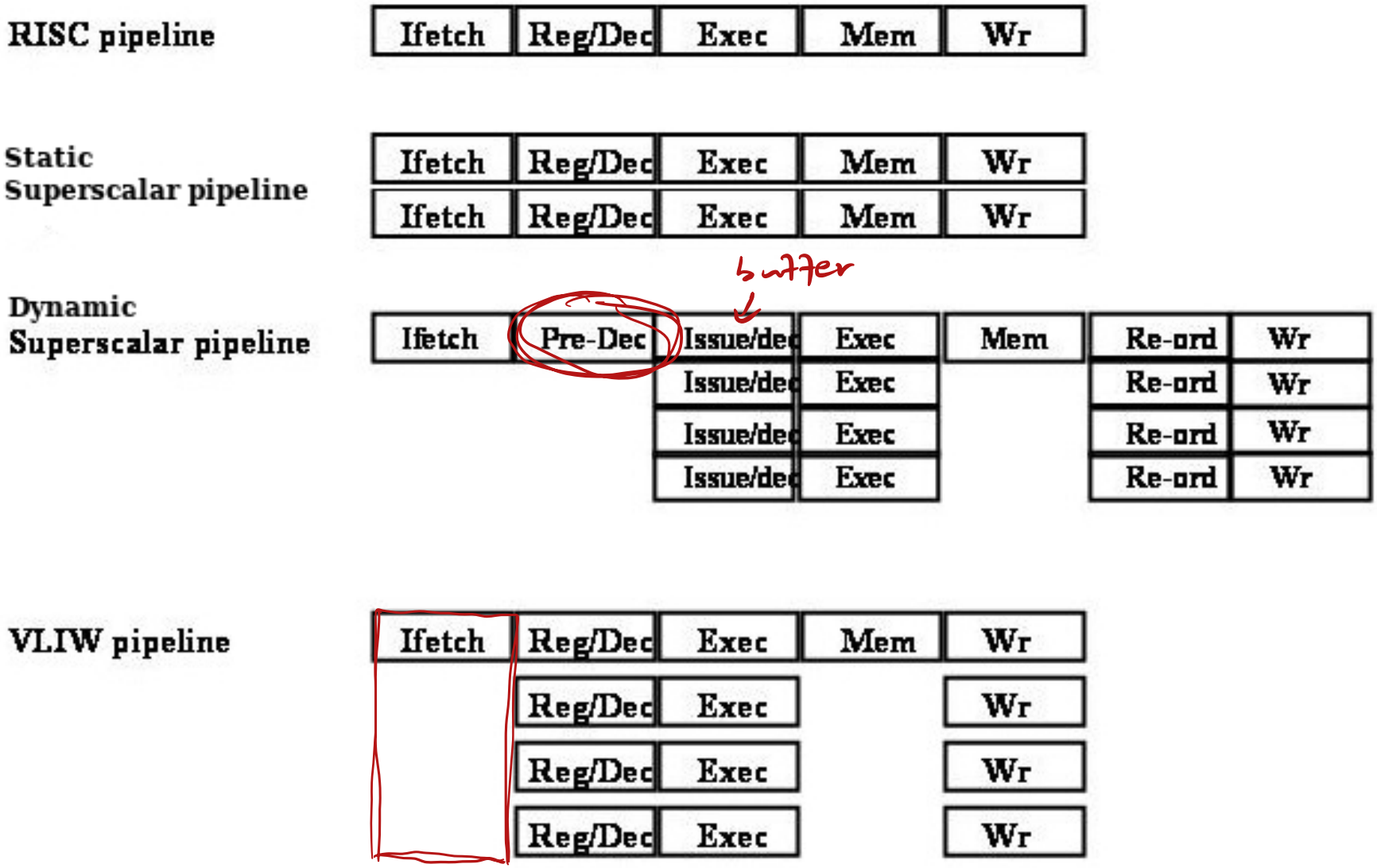
Static SS

- Static scheduling.
- In-order execution.
- Many IF units (or one IF fetching multiple instr.) and many EX units.
- Compiler explicitly schedules each "route" through the datapath.

Dynamic SS

- Dynamic scheduling.
- Out-of-order exec.
- Single IF unit but many EX units.
- IF unit might fetch multiple instr. per cycle.

Comparing Multiple Issue Pipelines (2/2)



Tapani Ahonen, University of Tampere

Outline

↑ improve dynamic schedule.

Writer After X Dependencies

Out-of-order execution causes hazards beyond RAW dependencies.

- WAR dependency – write a value after it is read by a previous instruction.
- Ex: Cannot write to $t1 until `addi` has read its value of $t1.

```
addi  $t0, $t1, 2
add   $t1, $t3, $0
```

- WAW dependency – write a value after its destination has been written to by a previous instruction. Needed to maintain consistent values for future instructions.
- Ex: If `add` executed before `addi` then value of $t1 incorrect.

```
addi  $t1, $t4, 12
add   $t1, $t3, $0
```

False Dependencies

- Some dependencies only exist due to re-use of the same variable/register.

```
t0 = A[4];
t0 = t0 + 3;
A[4] = t0;
t0 = A[64];
t0 = t0 + 5;
A[64] = t0;
```

we get dependency.

WAR dependency

```
t0 = A[4];
t0 = t0 + 3;
A[4] = t0;
t1 = A[64];
t1 = t1 + 5;
A[64] = t1;
```

use a new variable to avoid this problem.

- In out-of-order execution, the last 3 instructions cannot execute before the first 3 instructions, because of WAR dependency between `A[4] = t0` and `t0 = A[64]`.
- Renaming removes false dependency.

Register Renaming

- **Register renaming** is both a static and dynamic technique used to help superscalar pipelines and can fix WAR and WAW dependencies.
- Code is modified so that every destination is replaced with a unique "logical" destination (sometimes called value name).
- For every instruction which writes to a register, create a new value name for that destination.
  - ↳ Reservation stations provide a hardware buffer for storing logical destinations.
  - ↳ For dynamic renaming, hardware maintains a mapping from register names to logical destinations to find true operand values for incoming instructions.

Ex:

```
add $t6, $t0, $t2
sub $t4, $t2, $t0
xor $t0, $t6, $t2
and $t2, $t2, $t6
```

write to a logical instead. And have to come last.

RAW: $t6 in add and xor.
WAR: $t0 in sub and xor.
WAR: $t0 in add and xor.
WAR: $t2 in and and add.
WAR: $t2 in sub and and.
WAR: $t2 in sub and and.

Register Renaming Example

rename registers to logical destination.

Instr.	$t0	$t2	$t4	$t6	Renamed Instr.
Initially	V0	V1	V2	V3	—
`add $t6, $t0, $t2`				V4	add V4, V0, V1
`sub $t4, $t2, $t0`			V5		sub V5, V1, V0
`xor $t0, $t6, $t2`	V6				xor V6, V4, V1
`and $t2, $t2, $t6`		V7			add V7, V1, V4

XOR have to read updated value of t6, t2 is updated in V4.

current value of V2 => t1.

most current t6 => V4.

this is a new destination to store this value.

Register Renaming Example

Instr.	$t0	$t2	$t4	$t6	Renamed Instr.
Initially	V0	V1	V2	V3	—
`add $t6, $t0, $t2`				V4	`add V4, V0, V1`
`sub $t4, $t2, $t0`			V5		`sub V5, V1, V0`
`xor $t0, $t6, $t2`	V6				`xor V6, V4, V1`
`and $t2, $t2, $t6`					`add ??, V1, ??`

Register Renaming Example

Instr.	$t0	$t2	$t4	$t6	Renamed Instr.
Initially	V0	V1	V2	V3	—
`add $t6, $t0, $t2`				V4	`add V4, V0, V1`
`sub $t4, $t2, $t0`			V5		`sub V5, V1, V0`
`xor $t0, $t6, $t2`	V6				`xor V6, V4, V1`
`and $t2, $t2, $t6`					`add ??, V1, ??`

Instr.	$t0	$t2	$t4	$t6	Renamed Instr.
Initially	V0	V1	V2	V3	—
`add $t6, $t0, $t2`				V4	`add V4, V0, V1`
`sub $t4, $t2, $t0`			V5		`sub V5, V1, V0`
`xor $t0, $t6, $t2`	V6				`xor V6, V4, V1`
`and $t2, $t2, $t6`		V7			`add V7, V1, V4`

Summary

- Multiple issue processors execute multiple instructions simultaneously for CPI < 1.
- VLIW uses statically-scheduled issue packets.
- Loop unrolling exposes more parallelism *and* removes some branching overhead.
- Dynamic superscalar processors combat against unexpected stalls (cache misses) by allowing for out-of-order execution.
- Register renaming fixes WAR and WAW dependencies.
- RAW dependencies are the only true dependency and still must be accounted for in scheduling.