Credits: Yousif (Intel), Kubi@UCB, Asanovic @ MIT, http://csep1.phy.ornl.gov

Multiple Issue Microprocessors Handout 15

November 11, 2004

Shoukat Ali

shoukat@umr.edu

UMR UNIVERSITY OF MISSOURI-ROLLA
The Name. The Degree. The Difference.

1

By The Way…

- recall: renaming – a method to eliminate WAW and WAR hazards
  - recall: RS can rename registers
  - recall: ROB can rename registers
  - new: physical registers can rename registers
    - a set of registers that HW is free to use to resolve WAW and WAR hazards
      - these registers are not visible to the programmer
      - registers in the register file are programmable visible registers and are called architectural or logical registers
- example
  - assume 8 logical, 8 physical registers

fetched stream of instructions

```
ADDD    F3, F4, F2
ADDD    F7, F5, F2
MULT    F7, F7, F3
ADDD    F1, F1, #1
SD      0(R2), F1
```

2

Renaming Using Physical Registers

fetched stream of instructions

ADDD F3, F4, F2
ADDD F7, F5, F2
MULT F7, F7, F3
ADDD F1, F1, #1
SD 0(R2), F1

logical → physical:

renamed stream of instructions

free physical registers:

p1, p2, p3, p4, p5, p6, p7, p8

3

Renaming Using Physical Registers

fetched stream of instructions

ADDD F3, F4, F2
ADDD F7, F5, F2
MULT F7, F7, F3
ADDD F1, F1, #1
SD 0(R2), F1

logical → physical:

F3 → p1

renamed stream of instructions

ADDD p1, F4, F2

free physical registers:

p2, p3, p4, p5, p6, p7, p8

4

Renaming Using Physical Registers

fetched stream of instructions

```
ADDD    F3, F4, F2
ADDD    F7, F5, F2
MULT    F7, F7, F3
ADDD    F1, F1, #1
SD       0(R2), F1
```

logical → physical:

F3 → p1
F7 → p2

free physical registers:

p3, p4, p5, p6, p7, p8

renamed stream of instructions

```
ADDD    p1, F4, F2
ADDD    p2, F5, F2
```

For each instruction

1) Read source register mappings from map table

2) Acquire new result physical register from free pool

3) Modify map table for new result register mapping

5

Renaming Using Physical Registers

fetched stream of instructions

```
ADDD    F3, F4, F2
ADDD    F7, F5, F2
MULT    F7, F7, F3
ADDD    F1, F1, #1
SD       0(R2), F1
```

logical → physical:

F3 → p1
F7 → p2

free physical registers:

p3, p4, p5, p6, p7, p8

renamed stream of instructions

```
ADDD    p1, F4, F2
ADDD    p2, F5, F2
MULT    F7, p2, p1
```

step 1

For each instruction

1) Read source register mappings from map table

2) Acquire new result physical register from free pool

3) Modify map table for new result register mapping

6

Renaming Using Physical Registers

fetched stream of instructions

```
ADDD    F3, F4, F2
ADDD    F7, F5, F2
MULT    F7, F7, F3
ADDD    F1, F1, #1
SD       0(R2), F1
```

logical → physical:

F3 → p1
F7 → p3

free physical registers:

p4,
p5, p6, p7, p8

renamed stream of instructions

```
ADDD    p1, F4, F2
ADDD    p2, F5, F2
MULT    p3, p2, p1
```

step 2

For each instruction

1) Read source register mappings from map table

2) Acquire new result physical register from free pool

3) Modify map table for new result register mapping

7

Renaming Using Physical Registers

fetched stream of instructions

```
ADDD    F3, F4, F2
ADDD    F7, F5, F2
MULT    F7, F7, F3
ADDD    F1, F1, #1
SD       0(R2), F1
```

logical → physical:

F3 → p1
F7 → p3
F1 → p4

free physical registers:

p5, p6, p7, p8

renamed stream of instructions

```
ADDD    p1, F4, F2
ADDD    p2, F5, F2
MULT    p3, p2, p1
ADDD    p4, F1, #1
```

For each instruction

1) Read source register mappings from map table

2) Acquire new result physical register from free pool

3) Modify map table for new result register mapping

8

Renaming Using Physical Registers

fetched stream of instructions

```
ADDD    F3, F4, F2
ADDD    F7, F5, F2
MULT    F7, F7, F3
ADDD    F1, F1, #1
SD      0(R2), F1
```

logical → physical:

F3 → p1
F7 → p3
F1 → p4

free physical registers:

p5, p6, p7, p8

renamed stream of instructions

```
ADDD    p1, F4, F2
ADDD    p2, F5, F2
MULT    p3, p2, p1
ADDD    p4, F1, #1
SD      0(R2), p4
```

For each instruction

1) Read source register mappings from map table

2) Acquire new result physical register from free pool

3) Modify map table for new result register mapping

9

Where Are We?

- when we studied pipelines, we were naively happy at the prospect of getting one instruction executed every clock cycle
- then we discovered that pipeline might have to be stalled for data and hazards
  - 2 cycle stall per data hazard
- then we realized that maybe compiler could re-order the code to avoid stalls – that was static scheduling
- then it dawned upon us that hardware could also re-order the code to avoid stalls – that was dynamic scheduling
  - dyn scheduling: in order issue, o3 execution
    - using either Thornton's scoreboard or Tomasulo's algorithm
  - why would we want HW, instead of compiler, to re-order code?
    - memory dependencies can take an unpredictable number of cycles, so cannot expect compiler to know how many independent instructions to put after a load

CPI = 1

CPI = A
#stg >A> 1

A > CPI
CPI >= 1

A > CPI
CPI >= 1

10

More: Where Are We?

- we were really happy with Tomasulo's algorithm but then we realized that it can let imprecise exceptions occur
- so we fixed up Tom's alg by putting in a re-order buffer that would still allow in order issue and o3 execution, but would also enforce in order committing (aka retiring, graduation)
- then we realized that re-order buffer could also be used to carry out speculative execution e.g., after a branch prediction, load speculation, way prediction, etc.
  - ROB allows in order issue, speculative execution, but in order committing

11

Still More: Where Are We?

- now we can design a microprocessor that can do o3 execution, can have precise interrupts, and can execute speculatively (with roll back if speculation is wrong)
  - we have really created a beast that wants more and more instructions to execute
    - how do you feed that beast?
- we need to have an instruction fetch unit that can fetch more than one instruction at a time from cache and an instruction issue unit that can issue multiple instructions at a time
- now multiple instructions will be completed at the same time
  - CPI < 1 (or IPC, instructions per cycle > 1)

CPI < 1!

12

Multiple Issue Processors

Multiple Issue Processors

- a single-issue processor issues ONE instruction every clock cycle
  - **at best**, CPI can be 1
- a multiple-issue processor can issue *k* (k > 1) instructions in each cycle
  - at its best, CPI can be as small as 1/*k*
    - or IPC can be as large as *k*
- two basic types of multiple-issue processors
  - superscalar processors
  - VLIW processors
    - VLIW = very long instruction word

Superscalar Processors

- superscalar processor:
  - pipeline receives a traditional sequential stream of instructions
    - k instructions are fetched every cycle (k is typically 2-6)
  - data and structural hazards are checked for by the HW
  - depending on the hazards, 0 to k instructions are issued per clock
    - dynamic issue

CC1	CC2	CC3	CC4	CC5	CC6	CC7
IF	ID	EX	WB			
IF	ID	EX	WB			
	IF	ID	EX	WB		
	IF	ID	EX	WB		
		IF	ID	EX	WB	
		IF	ID	EX	WB	

15

VLIW Processors

- VLIW processor:
  - pipeline receives a non-traditional stream of instructions
    - compiler does extensive re-ordering of the instructions, checking for data and structural hazards
    - compiler packs *k* instructions in one instruction packet
      - sometimes some of these instructions are just NOPs because there is not enough parallelism
  - pipeline receives a stream of instruction packets
  - all instructions in a packet are issued every clock cycle
    - static issue

CC1	CC2	CC3	CC4	CC5	CC6	CC7
IF	**ID**	EX	WB			
		EX	WB			
	IF	**ID**	EX	WB		
			EX	WB		
		IF	**ID**	EX	WB	
				EX	WB	

16

Superscalar Versus VLIW

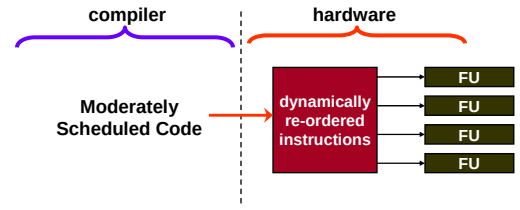
- difference is really in who does the dependence checks and structural hazard checks
- VLIW processors expect dependency-free code on each cycle whereas superscalar processors must figure out dependencies using hardware
- VLIW expects parallel instructions to be provided to them
  - VLIW = EPIC = explicitly parallel instruction computers
    - example: Intel/HP Itanium microprocessor
- superscalar just gets regular instruction stream and "parallelizes" iinstructions itself using HW dependence checking
  - Alpha 21264, Intel P4

17

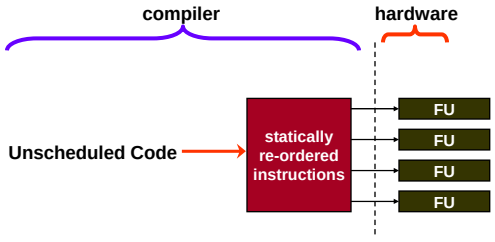
Superscalar Versus VLIW

Superscalar:

Alpha 21264, Pentium III/4, MIPS R10K, HP PA 8500, IBM Power 2, IBM RS64III



VLIW:
Itanium



18

Details of Issue Stage in Superscalar

- assume a 4-way superscalar
- instruction fetch unit fetches four instructions
  - called an "issue packet"
- instruction fetch unit examines each of the four instructions ***in program order***
  - an instruction X is not issued if X would cause a structural hazard or a data hazard with
    - either an earlier instruction already in execution
    - or an instruction earlier in this issue packet
- so in a given cycle, as many as four instructions may be issued
  - or as few as _____

19

More Details of Issue Stage in Superscalar

- logically issue checks are done in program order
- in practice, all instructions in issue packet are examined at once
  - hazard checks done in parallel

- above issue checks are complex
  - issue stage might need more time than a single clock cycle
  - most superscalar processors devote two cycles to issue stage
  - issue stage is split and pipelined
    - one approach:
      - first issue stage: detect dependences b/w instr in packet
      - second issue stage: detect dependences b/w instr selected in first issue stage and instr already in execution

20

A Two-Issue Superscalar Processor

CC1	CC2	CC3	CC4	CC5	CC6	CC7
IF	ID	EX	WB			
IF	ID	EX	WB			
	IF	ID	EX	WB		
	IF	ID	EX	WB		
		IF	ID	EX	WB	
		IF	ID	EX	WB	

- above assumes that two instructions fetched in any cycle are totally independent of each other
- what if the instructions depend on each other?

CC1	CC2	CC3	CC4	CC5	CC6	CC7
IF	ID	EX	WB	WB		
IF	ID					
	IF	ID	EX	WB		
	IF	ID	EX	WB		
		IF	ID	EX	WB	
		IF	ID	EX	WB	
			IF	ID	EX	WB

21

New Complexity

CC1	CC2	CC3	CC4	CC5	CC6	CC7
IF	ID	EX	WB			
IF	ID	EX	WB			
	IF	ID	EX	WB		
	IF	ID	EX	WB		
		IF	ID	EX	WB	
		IF	ID	EX	WB	

- IF: need to fetch more than one instruction
- ID: need highly multi-ported register file and/or replicated register file
- MEM: need multi-ported data cache or need to run data cache at a speed higher than that of processor
- WB: need many result buses/bypasses/register file write ports

22

Two Types of Superscalar Processors

- call instructions in an issue packet, N1, N2, N3, N4
  - say N1 has a dependency on an earlier issued in-execution instr
  - say N2 has no dependency on any instruction
  - can N2 be issued before N1?

- if no: statically scheduled, dynamically issued (static superscalar)
  - all re-ordering is done by compiler
  - instructions are dynamically issued
    - when HW is issuing instructions, it does full dependence check
    - when instr X is hazardous, X and all instr after X are stopped from issuing

- if yes: dynamically scheduled, dynamically issued (dynamic superscalar)
  - some re-ordering is done by compiler and some by HW
  - instructions are dynamically issued and re-ordered
    - when HW is issuing instructions, it does full dependence check
    - any independent instr is issued

23

Just For Comparison: VLIW

- what is the answer to prev slide question for a VLIW processor?
- i.e., can N2 be issued before N1?

- above is a trick question – all instr in a packet are independent with respect to each other or previously issued in-execution instr

- just for completion: statically scheduled, statically issued (VLIW)
  - instructions are statically scheduled
    - before execution, re-ordering done by compiler
  - instructions are static issue
    - when HW is issuing instructions, it does not do ANY dependence check

24

Need of Branch Prediction for Superscalar

- branches arrive N times faster in an N-way SS
- need for ambitious scheduling
  - in simple pipelines, a branch would cause 1 cycle stall, i.e. next instr cannot be issued without stalling for a cycle
  - in N-way SS pipeline, a branch could stall {(N-1) + N} next instructions
  - this motivates the need to develop excellent branch prediction schemes PLUS speculative execution methods (ROB)

25