Superscalar Architectures

COMP375 Computer Architecture and Organization

Summer Classes

Subject	Number	Sec	TITLE	CRN	format	Instructor
COMP	322	60	Internet Systems	30938	online	Bullock
COMP	385	30	Theory of Computing	31444	lecture	Kim
COMP	390	06A	Social Implication of Computer	30890	online	Bullock
COMP	450	06A	Operating Systems	31445	online	Carr
COMP	467	06A	Database Design	41214	online	Hinton
COMP	476	30	Networked Computer Systems	30892	online	Hinton
COMP	510	06A	Software Engineering	30891	online	Torres

Superscalar Processors

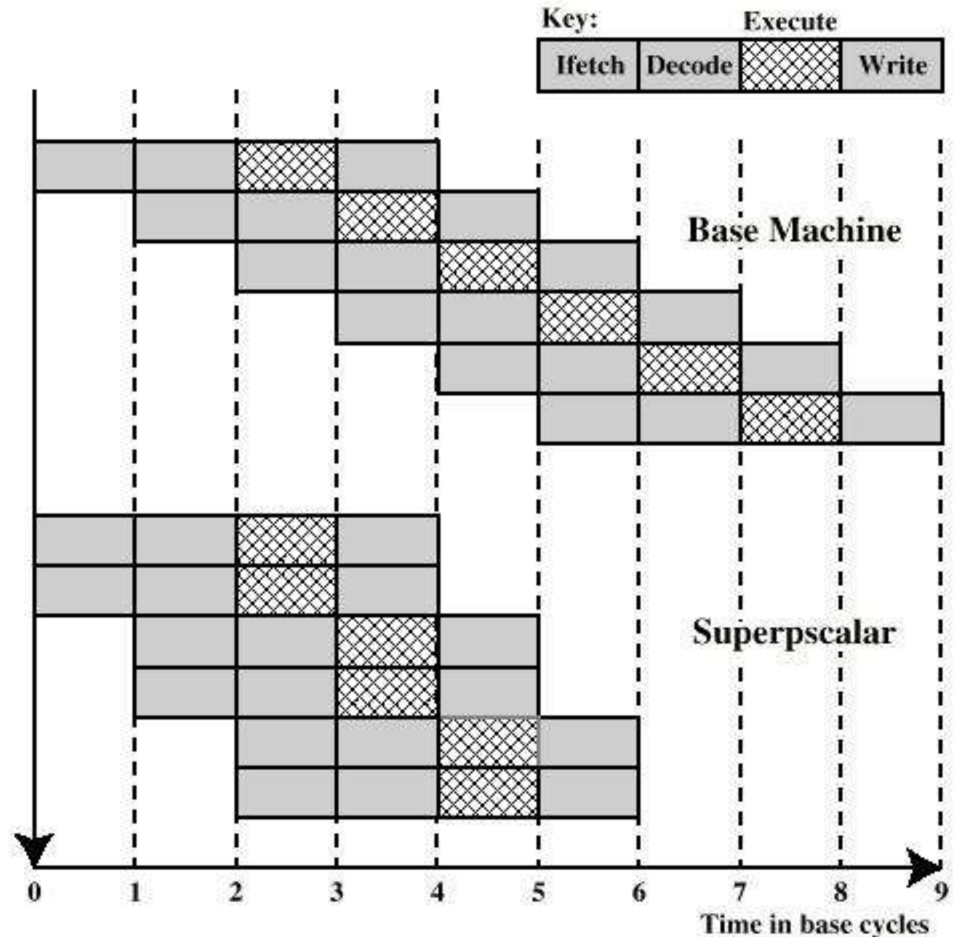
- Able to execute multiple instructions at a single time.
- Uses multiple ALUs and execution resources.
- Takes a sequential program and runs adjacent instructions in parallel if possible.
- The Pentium Pro and following Intel processors are superscalar as are many other modern processors.

Superscalar vs. Multiprocessor

- Superscalar machines execute regular sequential programs. The programmer is unaware of the parallelism.
- The programmer must explicitly code parallelism for multiprocessor systems.
- Simple instructions (arithmetic, load/store, conditional branch) can be initiated and executed independently
- Equally applicable to RISC & CISC

Superscalar Pipelining

Comparison of regular pipelining and superscalar pipelining



graphic from the textbook ©Pearson Prentice Hall

The superscalar processor described could run (at best)

1. same speed as simple processor at the same clock
2. 1.5 times as fast
3. twice as fast
4. three times as fast
5. faster than a speeding bullet

Instruction Level Parallelism

- As the processor is fetching and decoding instructions, it determines if they can be executed at the same time.
- Instructions executed in parallel must be hazard free.
- Some architectures, such as the Intel Itanium, explicitly define parallel instructions.

Superscalar Hazards

- Data Dependency – Instructions executed in parallel cannot depend upon the results of the other instruction.

```
Add  R1, R2
Sub  R3, R1
```
Needs correct R1

- These instructions cannot be executed in parallel.

Superscalar Resource Hazards

- Superscalar processors must have multiple execution resources. The number of resources (such as ALUs) limit the parallelism.

Superscalar Control Hazards

- Control hazards impact the pipelining of both simple and superscalar processors.
- The impact of a control hazard is greater with a superscalar processor because multiple instructions may need to be discarded.

Compiler Support

- An intelligent compiler can reorder the instructions so that adjacent instructions do not create data hazards.
- Compilers for explicitly superscalar processors generate bundles of instructions that are executed in parallel.

RISC Processors

- Modern processors take advantage of the architectural advances learned over the past decades
- Most new processors (i.e. Intel Itanium or ARM) are RISC processors

Design Alternatives

- **CISC** – Complex Instruction Set Computer
  - Pentium is the most popular example
- **RISC** – Reduced Instructions Set Computer
  - PowerPC, MIPS, SPARC, Intel Itanium, ARM
- No precise definition. The Intel Pentium 4 borrows many design ideas from RISC.

Evolution of CISC Designs

- Motivation to efficiently use expensive resources
  - Processor
  - Memory
- High density code
  - Complex instructions
    - Hardware complexity is handled by ***microprogramming***
    - Microprogramming is also helpful to
      - Reduce the impact of memory access latency
      - Offers flexibility, multiple members of the same family
  - Tailored to high-level language constructs

Simple Instructions

- Simple instructions are preferred
  - Complex instructions are mostly ignored by compilers
  - The Intel Pentium has several instructions that appear to be designed for Cobol programs.
  - Most of the instructions used are very simple.
- An implementation that supports complex instructions slows the execution of simple instructions.

Instruction Frequencies

	Dynamic Occurrence		Memory-Reference Weighted	
	Pascal	C	Pascal	C
Assign	45%	38%	14%	15%
Loop	5%	3%	33%	26%
Call	15%	12%	44%	45%
IF	29%	43%	7%	13%
other	6%	1%	2%	1%

Cache vs. Microcode

- Simple instructions can be implemented without microcode
- Cache memory is about as fast as microcode memory.
- Executing several simple instructions to perform something takes about as long as executing a complex instruction requiring lots of microcode.

Addressing Modes

- Complex addressing modes lead to variable length instructions
  - Leads to inefficient instruction decoding and scheduling
- Complex addressing modes varies the time required to fetch an operand.
  - Reduces pipeline efficiency

Microcode and L1 cache are located in the

1. BIOS, RAM
2. CPU chip, CPU chip
3. CPU chip, RAM
4. BIOS, CPU chip

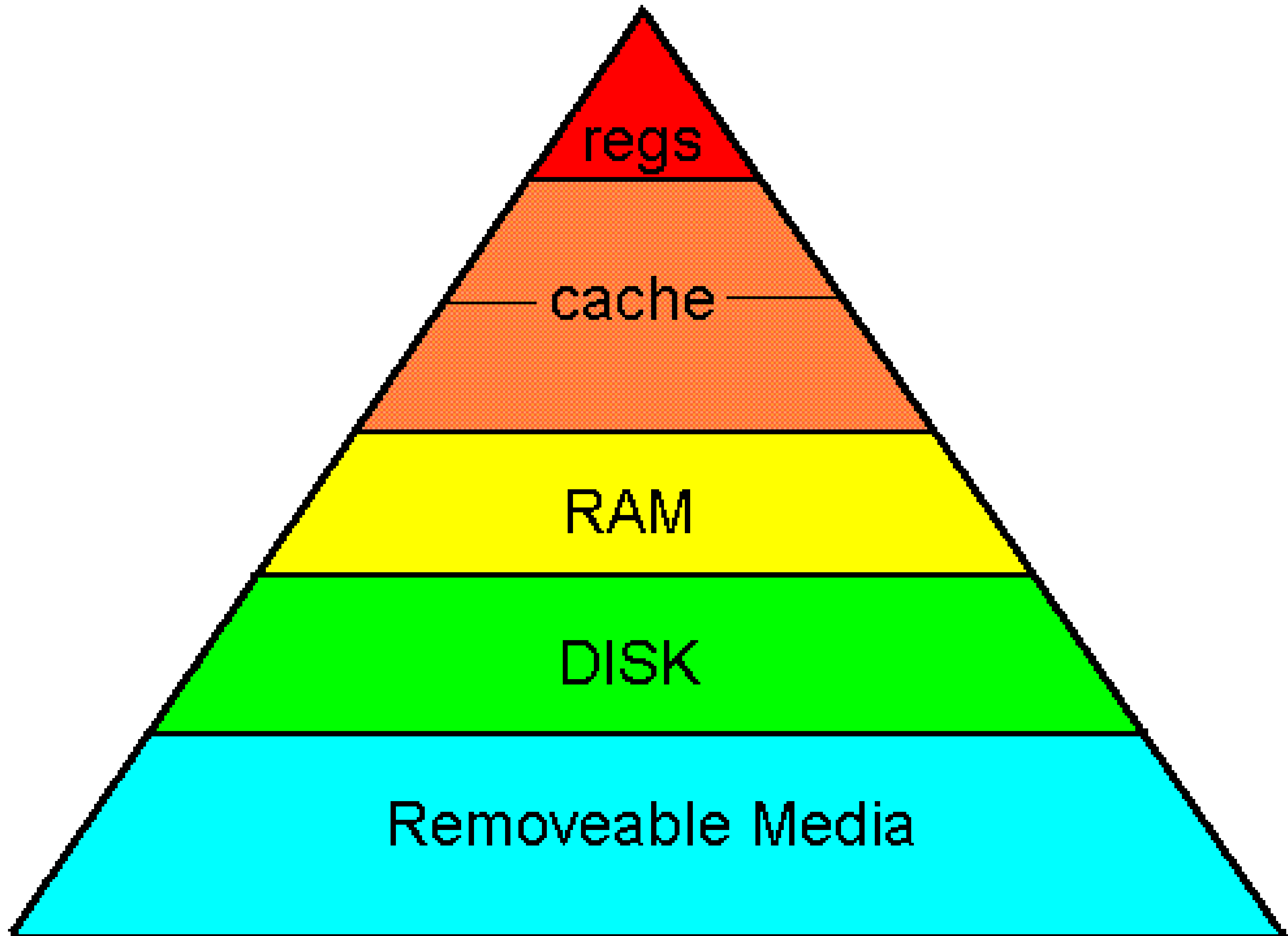
Register Operands

- Almost all RISC instructions use register operands.
- Usually only one Load and one Store instruction access RAM.
- Many RISC systems use three register operands

```
ADD  R1,R2,R3
```

- Avoids data hazard of putting the result back in the source register.

Memory Hierarchy



Function Calls

- Most function calls have few arguments
  - Only 1.25% of the calls have more than 6 arguments
- Most functions have few local variables
  - More than 93% have less than 6 local scalar variables
- Function call/return: ~15% of HLL statements
  - Constitute 31–33% of machine instructions
  - Generate nearly half (45%) of memory references

Avoiding the Stack

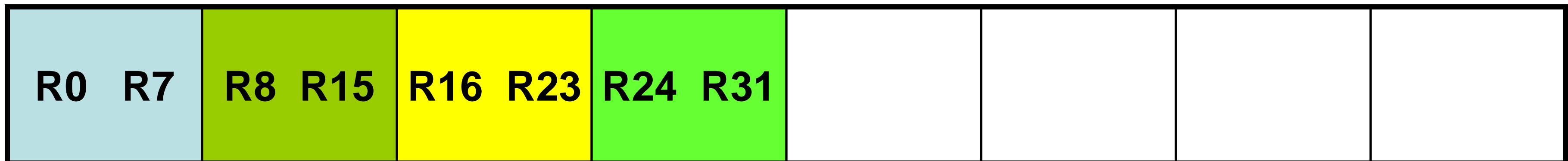
- Stack operations are memory intensive.
- Many RISC processors avoid using a stack and use registers instead.
- Arguments are passed in the registers.
- The function’s return address is stored in a register.
- Registers are used to hold local variables.

Register Stacks

- Many RISC processors have a large number of registers, not all of which are visible at any one time.
- The mapping of register X to a hardware register changes when a function is called.

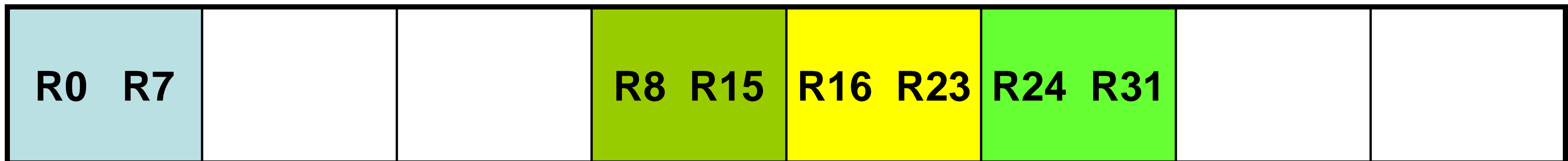
Before a Function Call

- Assume the assembly language programmer sees 32 registers.
- Before a function call, arguments and the return address are put in registers R24 to R31.



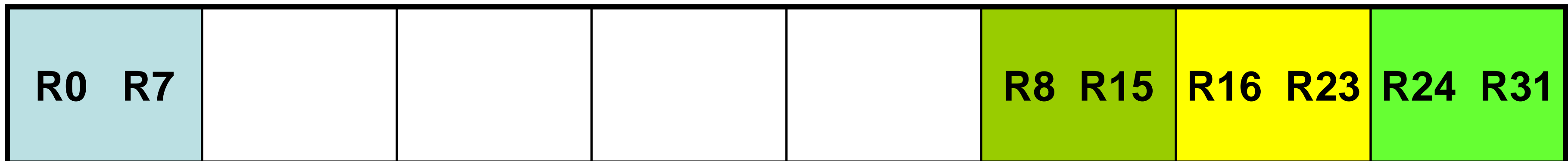
After a Function Call

- After a function call, the input arguments and the return address are available in registers R8 to R15.
- R16 to R23 are used for local variables.
- R24 to R31 contain arguments to next function



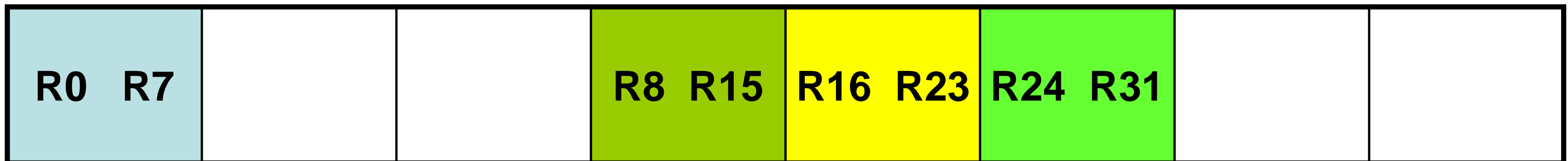
After another Function Call

- After another function call, the input arguments and the return address are again available in registers R8 to R15.
- Return values are also put in R8 to R15 upon function return.



After Function Return

- After the function return, the return values are available in registers R24 to R31.



A difference between RISC and CICS is

1. CISC functions might push the registers on the stack (i.e. **`pusha`**)
2. RISC functions use fewer parameters.
3. CISC save parameters in the microcode store
4. RISC processors do not use call instructions

RISC Design Principles

- Simple operations
  - Simple instructions that can execute in one cycle
- Register-to-register operations
  - Only load and store operations access memory
  - Rest of the operations on a register-to-register basis
- Simple addressing modes
  - A few addressing modes (1 or 2)

RISC Design Principles

- Large number of registers
  - Needed to support register-to-register operations
  - Minimize the procedure call and return overhead
- Fixed-length instructions
  - Facilitates efficient instruction execution
- Simple instruction format
  - Fixed boundaries for various fields

RISC Design Principle

- Start an instruction every cycle

- Simple, fixed length instructions are easy to pipeline.
- Only two instruction have memory operands all other operands are in registers.
- Delayed branches

Example Differences

	CISC		RISC
	VAX 11/780	Intel 486	MIPS R4000
# instructions	303	235	94
Addr. modes	22	11	1
Inst. size (bytes)	2-57	1-12	4
GP registers	16	8	32

chart © Dandamudi

RISC Traits

- Pipelined
- Simple uniform instructions
- Few instructions
- No microcode
- Few addressing modes
- Load/Store architecture
- Many identical general purpose registers
- Sliding register stack
- Delayed branches
- Fast

CICS advantages include:

1. Sliding register stack
2. Instructions designed to match high level language features
3. Load/Store architecture
4. Large microcode memory