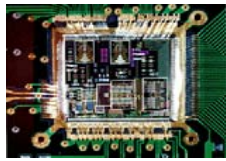
Lecture 2: Instructions: Language of the Computers

CSCE5610 Computer System Architecture
CSCE4610 Computer Architecture

Instructor: Saraju P. Mohanty, Ph. D.

NOTE: The figures, text etc included in slides are borrowed from various books, websites, authors pages, and other sources for academic purpose only. The instructor does not claim any originality.

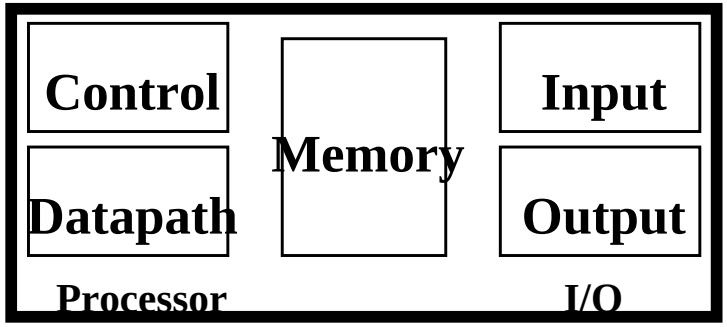


Review

Computer Organization & Architecture:

- Computer Architecture = ISA + machine organization
- Processor = Datapath + Controller
- All computers consist of five components:

  (1) Datapath (2) Control (3) Memory (4) Input device; and (5) Output device

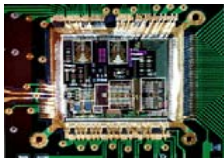


What are Instructions?

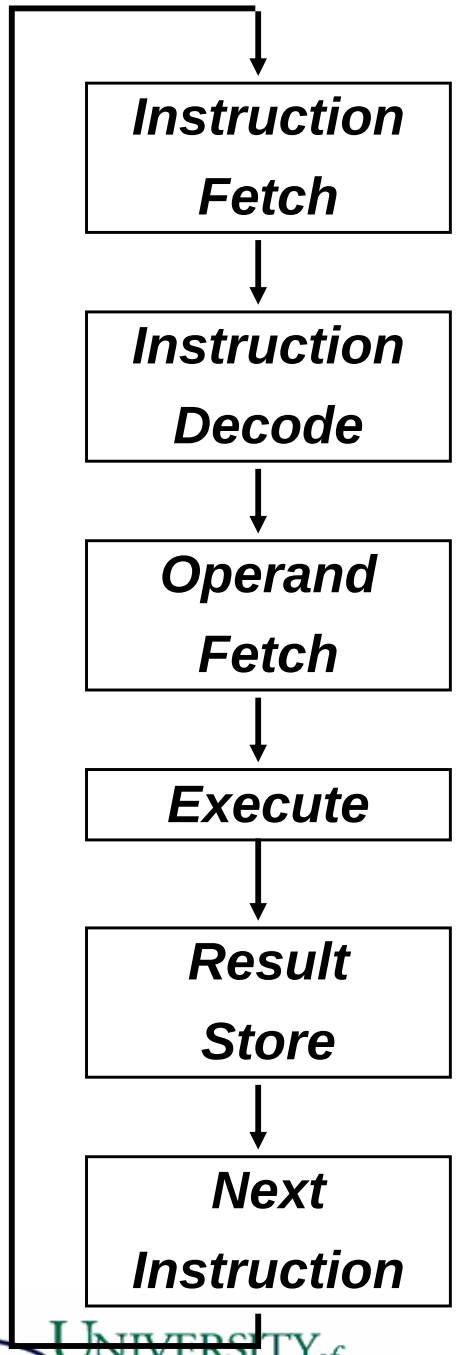
- Language of the Machine
- More primitive than higher level languages
  e.g., no sophisticated control flow
- Very restrictive
  e.g., MIPS Arithmetic Instructions
- We'll be working with the MIPS instruction set architecture
  - similar to other architectures developed since the 1980's
  - used by NEC, Nintendo, Silicon Graphics, Sony

Design goals:

- *maximize performance*
- *minimize cost*
- *reduce design time*

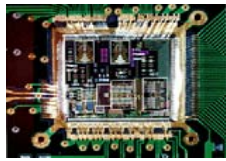


Instruction Set Architecture: What Must be Specified?



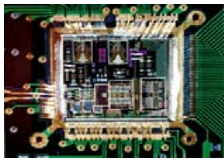
- Instruction Format or Encoding
  - how is it decoded?
- Location of operands and result
  - where other than memory?
  - how many explicit operands?
  - how are memory operands located?
  - which can or cannot be in memory?
- Data type and Size
- Operations
  - what are supported
- Successor instruction
  - jumps, conditions, branches

- fetch-decode-execute is implicit!



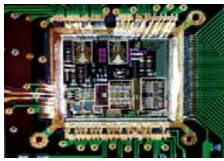
Instruction Categories in MIPS Processor

- Arithmetic
- Logical
- Data Transfer
- Conditional Branch
- Unconditional Branch



Design Principles

- Instruction complexity is only one variable
  - lower instruction count vs. higher CPI / lower clock rate
- Design Principles:
  - simplicity favors regularity
  - smaller is faster
  - make the common case fast
  - good design demands compromise
- Instruction set architecture
  - a very important abstraction indeed!

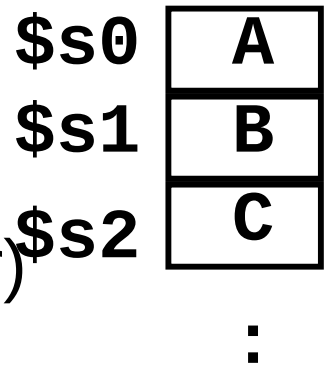


MIPS Arithmetic

- All instructions have 3 operands
- Operand order is fixed (destination first)
- Example:

C code: A= B + C

MIPS code: add $s0, $s1, $s2
(associated with variables by compiler)



Note: (1) "$s0" represents a register

(2) Variables A, B, C are stored in registers $s0, $s1, and $s2 respectively

MIPS Arithmetic

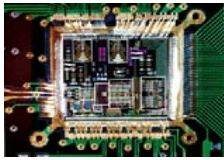
- Design Principle: simplicity favors regularity. Why?
- Of course this complicates some things...

C code: A = B + C + D;
E = F - A;

MIPS code: add $t0, $s1, $s2
add $s0, $t0, $s3
sub $s4, $s5, $s0

Note: register $t0, $t1 are temporary registers

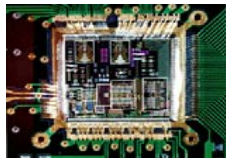
- Operands must be registers, only 32 registers provided
- Design Principle: smaller is faster. Why?



General Purpose Registers (GPRs) Dominate

° 1975-1995 all machines use general purpose registers

° Advantages of registers
  - registers are faster than memory
  - registers are easier for a compiler to use
  - registers can hold variables
    - memory traffic is reduced, so program is speeded up (since registers are faster than memory)
    - code density improves (since register named with fewer bits than memory location)

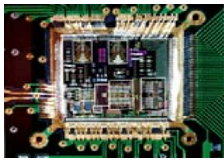


Registers vs. Memory

- In MIPS processor, arithmetic instructions operands must be registers
- Only 32 registers provided
- Compiler associates variables with registers
- What about programs with lots of variables?

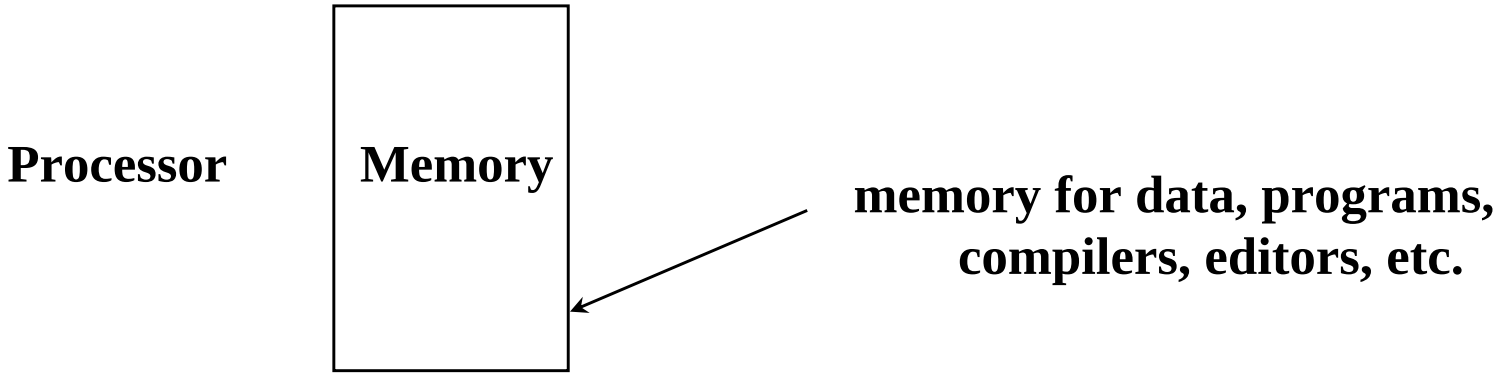
  Solution: *Spilling Registers*

  Excessive variables are stored in Memory and moved from memory to register file by *load* and *store* instructions.



Stored Program Concept

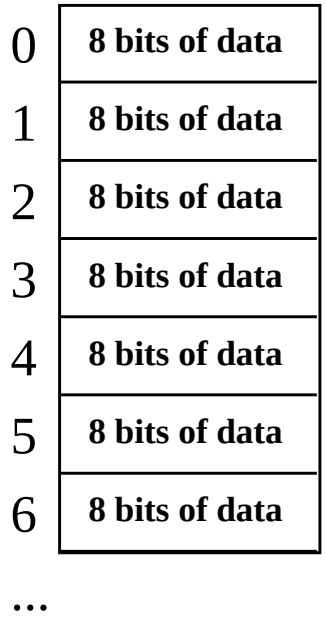
- Instructions are bits
- Programs are stored in memory
  — to be read or written just like data



- Fetch & Execute Cycle
  - Instructions are fetched and put into a special register
  - Bits in the register "control" the subsequent actions
  - Fetch the “next” instruction and continue

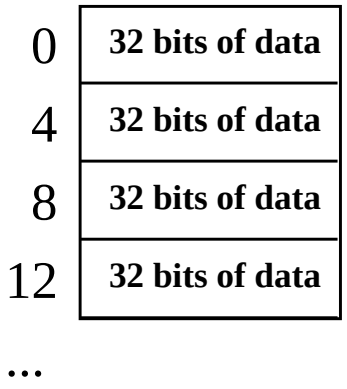
Memory Organization

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- "Byte addressing" means that the index points to a byte of memory.



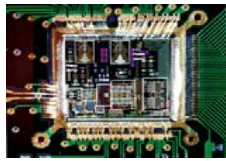
Memory Organization

- Bytes are nice, but most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes.



Registers hold 32 bits of data

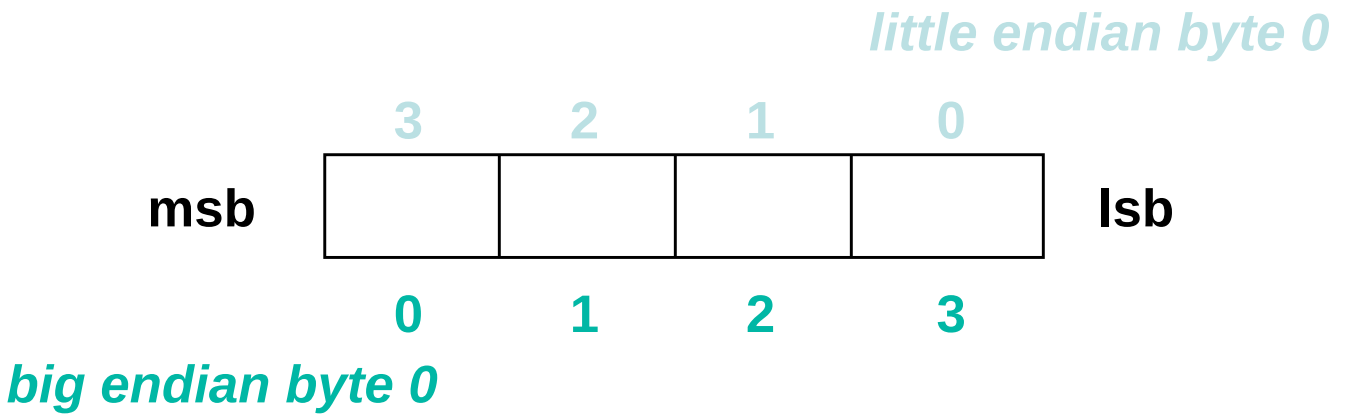
- 2^{32} bytes with byte addresses from 0 to $2^{32}-1$
- 2^{30} words with byte addresses 0, 4, 8, ... $2^{32}-4$
- Words are aligned
  i.e., what are the least 2 significant bits of a word address?



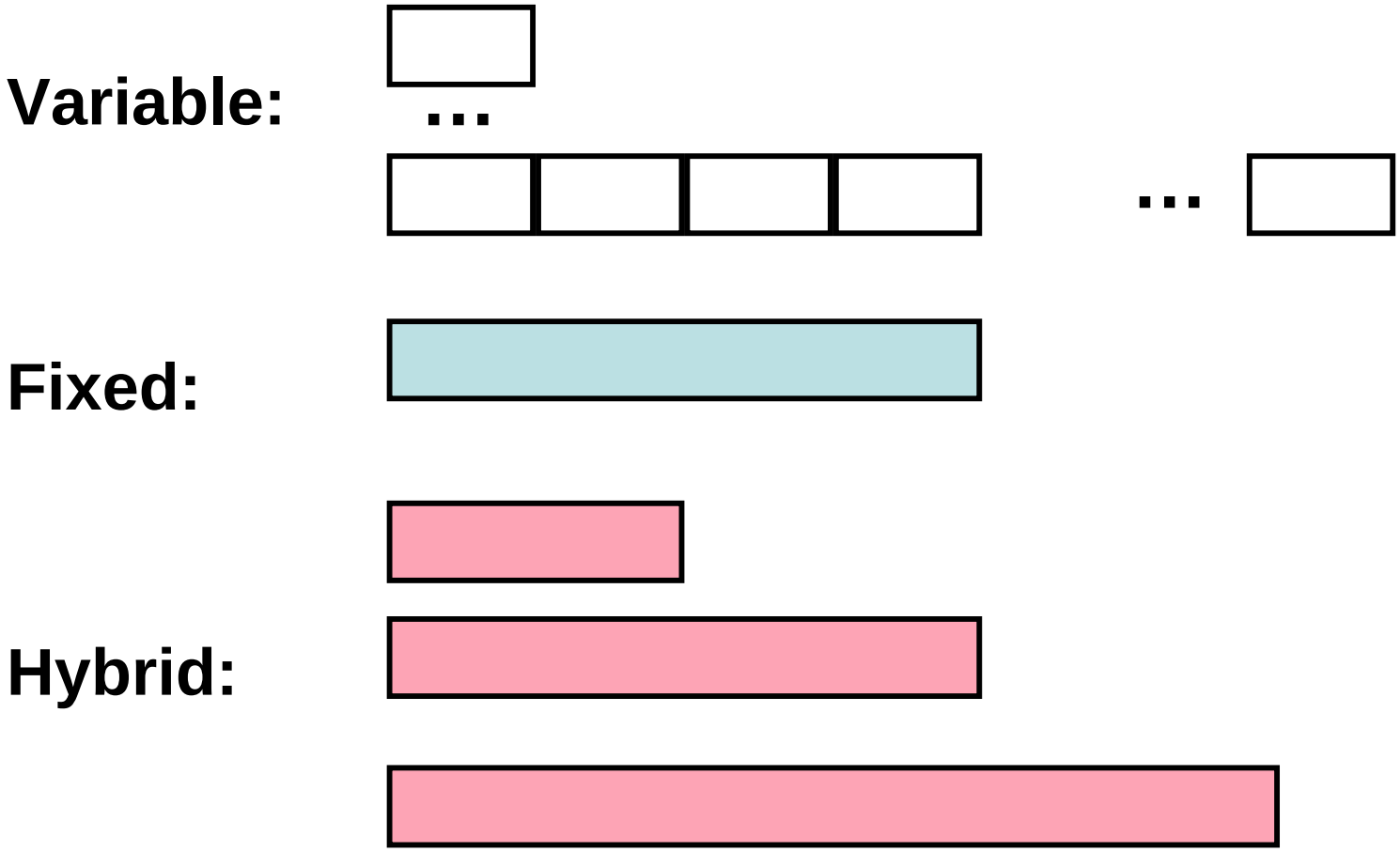
Addressing Objects: Endianess

- Big Endian: address of most significant byte
  IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

- Little Endian: address of least significant byte
  Intel 80x86, DEC Vax, DEC Alpha (Windows NT)

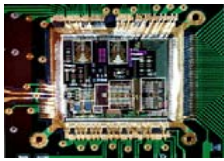


Generic Examples of Instruction Format Widths



Summary of Instruction Formats

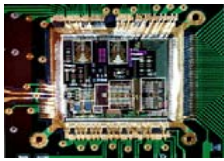
- If code size is most important, use variable length instructions
- If performance is most important, use fixed length instructions
- Recent embedded machines (ARM, MIPS) added optional mode to execute subset of 16-bit wide instructions (Thumb, MIPS16); per procedure decide performance or density



Load & Store Instructions by Example

- Load and store instructions are used for data movement between memory and registers in the register file
- Example:
  C code: A[8] = h + A[8];
  MIPS code: lw $t0, 32($s3) # $t0 = A[8]
  add $t0, $s2, $t0# $t0 = h + $t0
  sw $t0, 32($s3) # A[8] = $t0

  Note: (1) lw = load word, sw = store word
  (2) $t0 is a temporary register that accumulates the final result
  (3) Register $s2 holds variable “h”
  (4) Register $s3 is the index register that holds the start address of the array A I.e the location where array A starts..
- Store word has destination last
- Remember arithmetic operands are registers, not memory!



Our First Example

- Can we figure out the code?

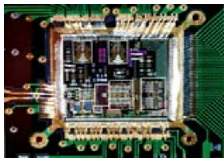
C Code:

```
swap(int v[], int k);
{ int temp;
        temp = v[k]
        v[k] = v[k+1];
        v[k+1] = temp;
}
```

MIPS Code:

Note: (1) Registers used: $2, $4, $5, $15, $16, $31
(Remember: Only 32 registers in MIPS processor)
(2) muli = multiply immediate instruction
(3) jr = jump return instruction
(4) Array v starts at location 0 in the memory
(5) Register $4 = Base address of array v
(6) Register $5 = variable k

```
swap:
        muli $2, $5, 4
        add     $2, $4, $2
        lw      $15, 0($2)
        lw      $16, 4($2)
        sw      $16, 0($2)
        sw      $15, 4($2)
        jr      $31
```

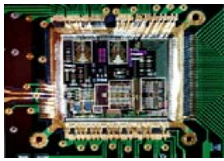



So far we've learned:

- MIPS
  - — loading words but addressing bytes
  - — arithmetic on registers only

- Instruction — Meaning

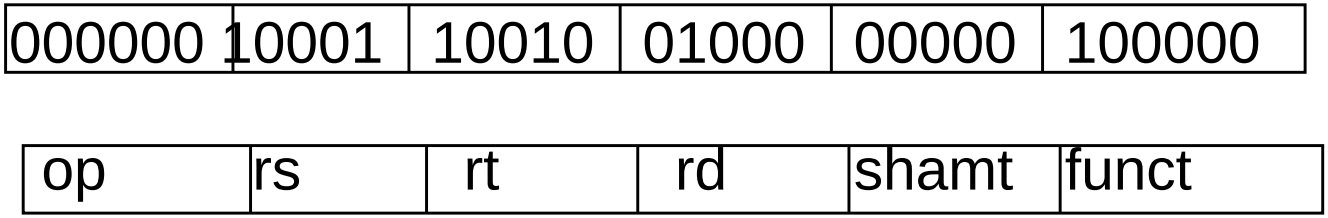
Instruction	Meaning
add $s1, $s2, $s3	$s1 = $s2 + $s3
sub $s1, $s2, $s3	$s1 = $s2 – $s3
lw $s1, 100($s2)	$s1 = Memory[$s2+100]
sw $s1, 100($s2)	Memory[$s2+100] = $s1



Machine Language

- Instructions, like registers and words of data, are also 32 bits long
  - Example: add $t0, $s1, $s2
  - registers have numbers, t0=8, s1=17, s2=18

- Instruction Format (R-type):



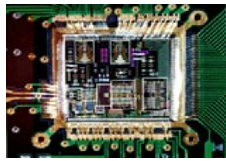
op: operation of the instruction

rs: the first register source operand

rt: the second register source operand

shamt: shift amount (we will look at this later..)

funct: function; this field selects the variant of the operation in the op field



Machine Language

- Consider the load-word and store-word instructions,
  - What would the regularity principle have us do?
  - New principle: Good design demands a compromise

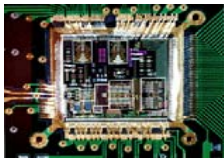
- Introduce a new type of instruction format
  - I-type for data transfer instructions
  - other format was R-type for register

Example: lw $t0, 32($s2)

35	18	9	32

op	rs	rt	16 bit number

- Where's the compromise?



Instructions for Control flow

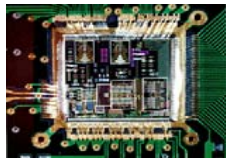
- Decision making instructions
  - alter the control flow,
  - i.e., change the "next" instruction to be executed

- MIPS conditional branch instructions:

```
bne $t0, $t1, Label
beq $t0, $t1, Label
```

- Example: if (i==j) h = i + j;

```
        bne $s0, $s1, Label
        add $s3, $s0, $s1
Label: ....
```

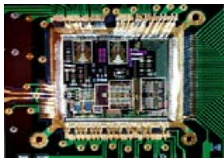
Unconditional Branch: jump instruction

- MIPS unconditional branch instructions:
  j label
- Jump Instruction Format:

op	26 bit address

- Example:

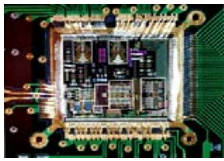
```
if (i!=j)                beq $s4, $s5, Lab1
    h=i+j;               add $s3, $s4, $s5
else                     j Lab2
    h=i-j;               Lab1:  sub $s3, $s4, $s5
                         Lab2:  ...
```

So far:

Instruction	Meaning
add $s1,$s2,$s3	$s1 = $s2 + $s3
sub $s1,$s2,$s3	$s1 = $s2 – $s3
lw $s1,100($s2)	$s1 = Memory[$s2+100]
sw $s1,100($s2)	Memory[$s2+100] = $s1
bne $s4,$s5,L	Next instr. is at Label if $s4 != $s5
beq $s4,$s5,L	Next instr. is at Label if $s4 = $s5
j Label	Next instr. is at Label
- Formats:

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

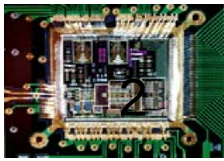


Control Flow

- We have: beq, bne, what about Branch-if-less-than?
- New instruction:

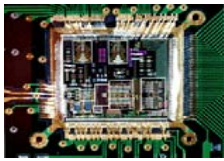
```
                                    if  $s1 < $s2 then
                                       $t0 = 1
        slt $t0, $s1, $s2           else
                                       $t0 = 0
```

- Can use this instruction to build "blt $s1, $s2, Label"
  — can now build general control structures
- Note that the assembler needs a register to do this,
  — there are policy of use conventions for registers



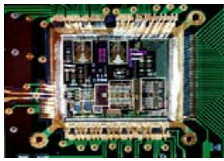
Assembly Language vs. Machine Language

- Assembly provides convenient symbolic representation
  - much easier than writing down numbers
  - e.g., destination first
- Machine language is the underlying reality
  - e.g., destination is no longer first
- Assembly can provide 'pseudoinstructions'
  - e.g., “move $t0, $t1” exists only in Assembly
  - would be implemented using “add $t0,$t1,$zero”
- When considering performance you should count real instructions



Other Issues

- Things we are not going to cover
  - support for procedures
  - linkers, loaders, memory layout
  - stacks, frames, recursion
  - manipulating strings and pointers
  - interrupts and exceptions
  - system calls and conventions
- We've focused on architectural issues
  - basics of MIPS assembly language and machine code
  - we'll build a processor to execute these instructions.



Overview of MIPS

- simple instructions all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

- rely on compiler to achieve performance
  — what are the compiler's goals?
- help compiler where we can

Addresses in Branches and Jumps

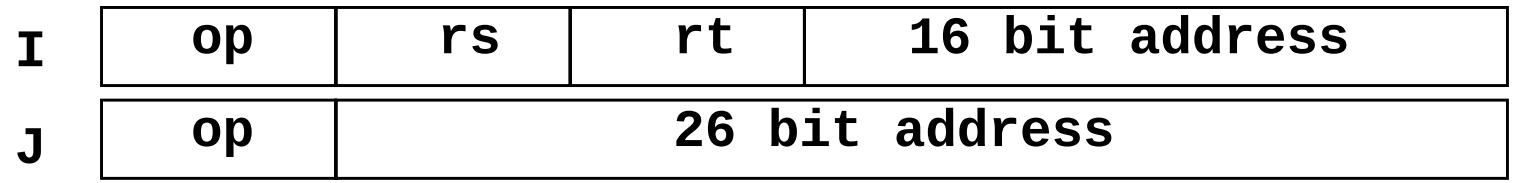
- Instructions:

  bne $t4,$t5,Label, Next instruction is at Label if $t4 !=$t5

  beq $t4,$t5,Label, Next instruction is at Label if $t4 = $t5

  j Label Next instruction is at Label

- Formats:



- Addresses are not 32 bits

  — How do we handle this with load and store instructions?

Addresses in Branches

- Instructions:

  bne $t4,$t5,Label,Next instruction is at Label if $t4!=$t5
  beq $t4,$t5,Label,Next instruction is at Label if $t4=$t5

- Formats:

- Could specify a register (like lw and sw) and add it to address
  - use Instruction Address Register (PC = program counter)
  - most branches are local (principle of locality)
- Jump instructions just use high order bits of PC
  - address boundaries of 256 MB

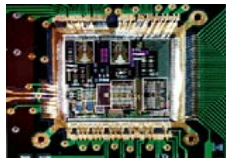
To summarize:

MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

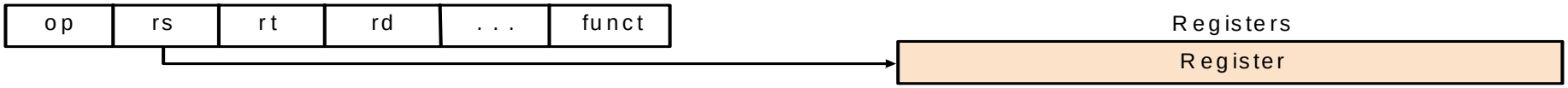
Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
Data transfer	load word	`lw $s1, 100($s2)`	`$s1 =` Memory[`$s2` + 100]	Word from memory to register
	store word	`sw $s1, 100($s2)`	Memory[`$s2` + 100] = `$s1`	Word from register to memory
	load byte	`lb $s1, 100($s2)`	`$s1 =` Memory[`$s2` + 100]	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	Memory[`$s2` + 100] = `$s1`	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	`beq $s1, $s2, 25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1, $s2, 25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1, $s2, $s3`	if (`$s2 < $s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if (`$s2 < 100`) `$s1` = 1; else `$s1` = 0	Compare less than constant
Uncondi-tional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call



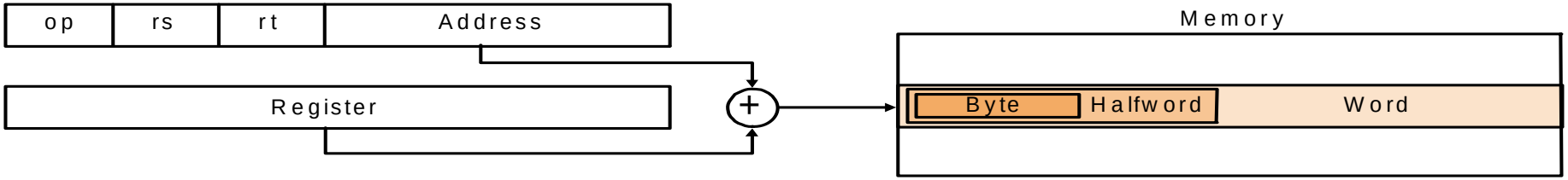
Addressing Modes

1. Immediate addressing

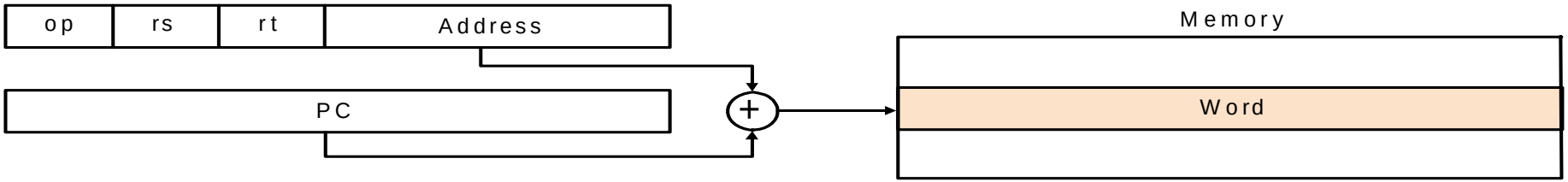
2. Register addressing



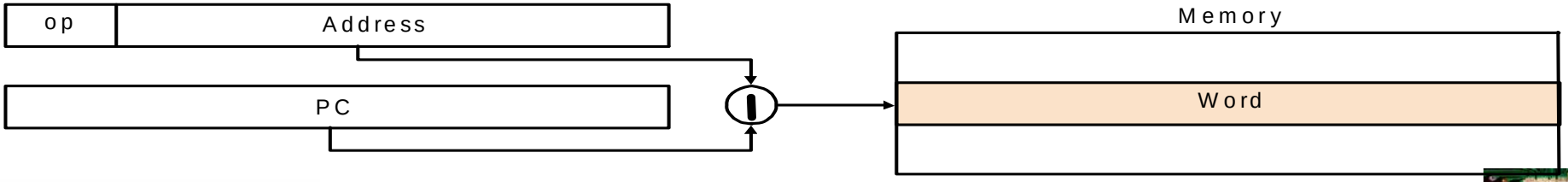
3. Base addressing

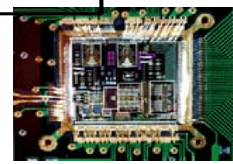


4. PC-relative addressing



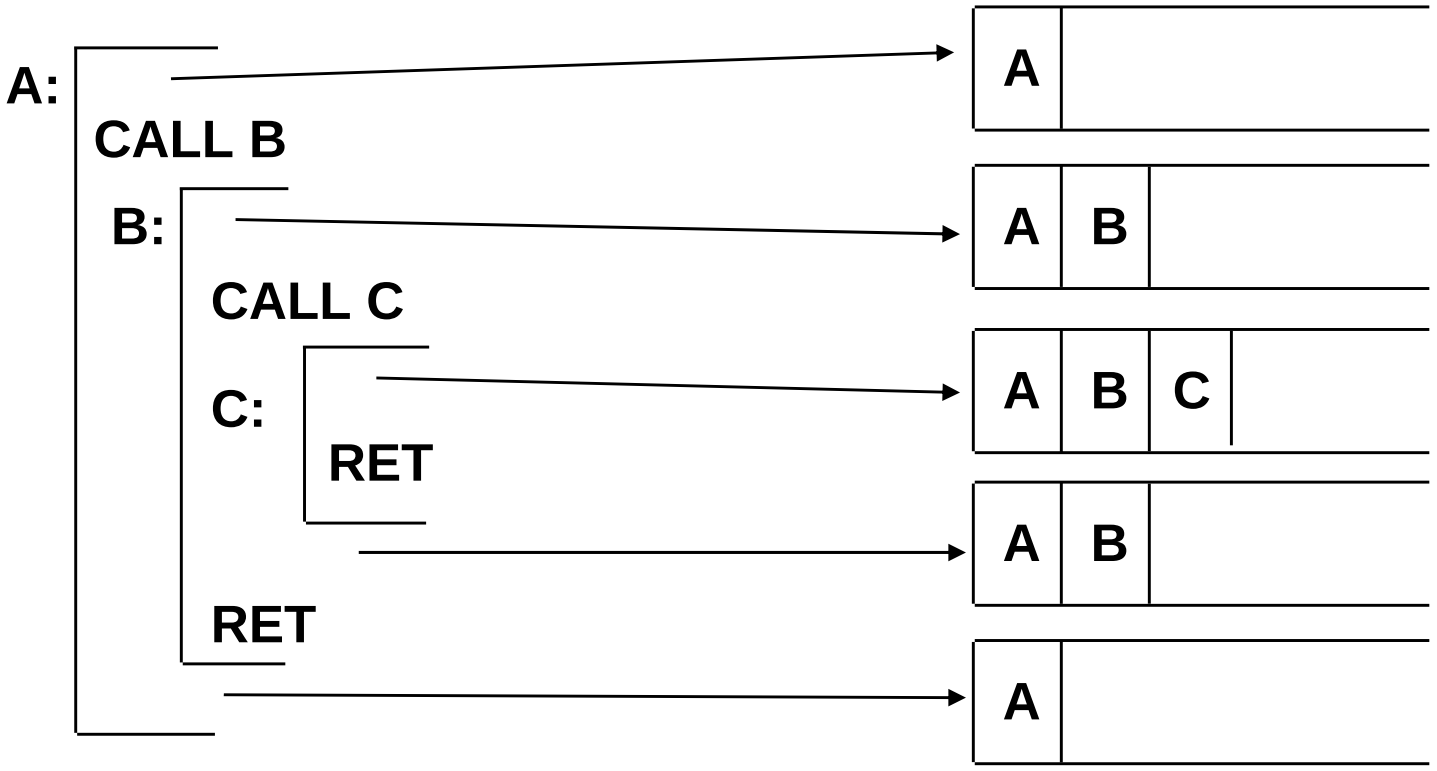
5. Pseudodirect addressing





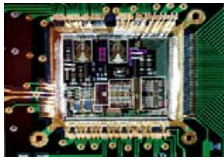
Calls: Why Are Stacks So Great?

Stacking of Subroutine Calls & Returns and Environments:



Some machines provide a memory stack as part of the architecture (e.g., VAX)

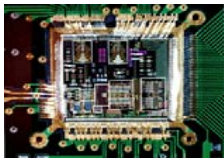
Sometimes stacks are implemented via software convention (e.g., MIPS)



MIPS: Software conventions for Registers

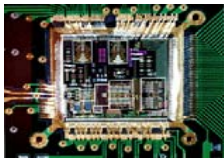
0	zero	constant 0
1	at	reserved for assembler
2	v0	expression evaluation &
3	v1	function results
4	a0	arguments
5	a1	
6	a2	
7	a3	
8	t0	temporary: caller saves
. . .		(callee can clobber)
15	t7	
16	s0	callee saves
. . .		(caller can clobber)
23	s7	
24	t8	temporary (cont'd)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	Pointer to global area
29	sp	Stack pointer
30	fp	frame pointer
31	ra	Return Address (HW)

Plus a 3-deep stack of mode bits.



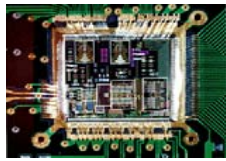
Alternative Architectures

- Design alternative:
  - provide more powerful operations
  - goal is to reduce number of instructions executed
  - danger is a slower cycle time and/or a higher CPI
- Sometimes referred to as “RISC vs. CISC”
  - virtually all new instruction sets since 1982 have been RISC
  - VAX: minimize code size, make assembly language easy *instructions from 1 to 54 bytes long!*



80x86

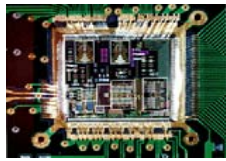
- 1978: The Intel 8086 is announced (16 bit architecture)
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits, +instructions
- 1985: The 80386 extends to 32 bits, new addressing modes
- 1989-1995: The 80486, Pentium, Pentium Pro add a few instructions (mostly designed for higher performance)
- 1997: MMX is added
  "This history illustrates the impact of the "golden handcuffs" of compatibility
  "adding new features as someone might add clothing to a packed bag"
  "an architecture that is difficult to explain and impossible to love"



A dominant architecture: 80x86

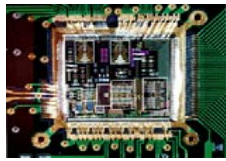
- See your textbook for a more detailed description
- Complexity:
  - Instructions from 1 to 17 bytes long
  - one operand must act as both a source and destination
  - one operand can come from memory
  - complex addressing modes
    e.g., “base or scaled index with 8 or 32 bit displacement”
- Saving grace:
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow

“what the 80x86 lacks in style is made up in quantity, making it beautiful from the right perspective”



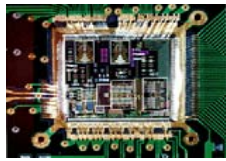
Summary: Salient features of MIPS I

- **32-bit fixed format inst** (3 formats)
- **32 32-bit GPR** (R0 contains zero) and 32 FP registers (and HI LO)
  - partitioned by software convention
- **3-address, reg-reg arithmetic instr.**
- **Single address mode for load/store:** base+displacement
  - no indirection, scaled
- **16-bit immediate plus LUI**
- **Simple branch conditions**
  - compare against zero or two registers for =,°
  - no integer condition codes
- **Delayed branch**
  - execute instruction after the branch (or jump) even if the branch is taken (Compiler can fill a delayed branch with useful work about 50% of the time)



Summary: Instruction set design (MIPS)

- Use general purpose registers with a load-store architecture: YES
- Provide at least 16 general purpose registers plus separate floating-point registers: 31 GPR & 32 FPR
- Support basic addressing modes: displacement (with an address offset size of 12 to 16 bits), immediate (size 8 to 16 bits), and register deferred; : YES: 16 bits for immediate, displacement (disp=0 => register deferred)
- All addressing modes apply to all data transfer instructions : YES
- Use fixed instruction encoding if interested in performance and use variable instruction encoding if interested in code size : Fixed
- Support these data sizes and types: 8-bit, 16-bit, 32-bit integers and 32-bit and 64-bit IEEE 754 floating point numbers: YES
- Support these simple instructions, since they will dominate the number of instructions executed: load, store, add, subtract, move register-register, and, shift, compare equal, compare not equal, branch (with a PC-relative address at least 8-bits long), jump, call, and return: YES, 16b
- Aim for a minimalist instruction set: YES



Data Types

Bit: 0, 1

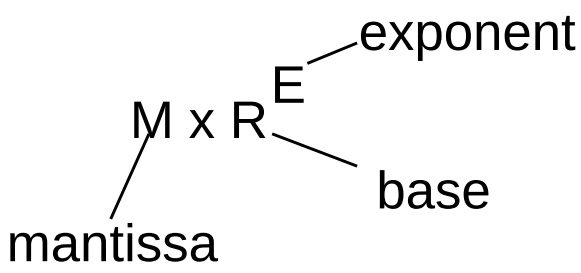
Bit String: sequence of bits of a particular length
- 4 bits is a nibble
- 8 bits is a byte
- 16 bits is a half-word
- 32 bits is a word
- 64 bits is a double-word

Character:
- ASCII 7 bit code

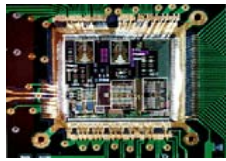
Decimal:
- digits 0-9 encoded as 0000b thru 1001b
- two decimal digits packed per 8 bit byte

Integers:
- 2's Complement

Floating Point:
- Single Precision
- Double Precision
- Extended Precision

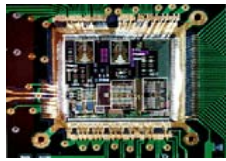


How many +/- #'s?
Where is decimal pt?
How are +/- exponents represented?



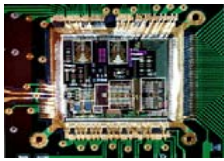
Compilers and Instruction Set Architectures

- Ease of compilation
  - orthogonality: no special registers, few special cases, all operand modes available with any data type or instruction type
  - completeness: support for a wide range of operations and target applications
  - regularity: no overloading for the meanings of instruction fields
  - streamlined: resource needs easily determined
- Register Assignment is critical too
  - Easier if lots of registers



Summary of Compiler Considerations

- Provide at least 16 general purpose registers plus separate floating-point registers,
- Be sure all addressing modes apply to all data transfer instructions,
- Aim for a minimalist instruction set.



Summary: Evaluating Instruction Sets?

Design-time metrics:

- ° Can it be implemented, in how long, at what cost?
- ° Can it be programmed? Ease of compilation?

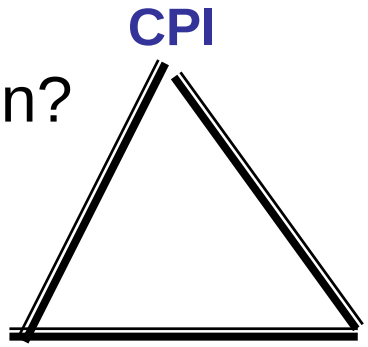
Static Metrics:

- ° How many bytes does the program occupy in memory?

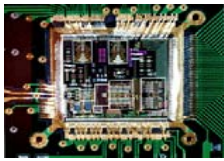
Dynamic Metrics:

- ° How many instructions are executed?
- ° How many bytes does the processor fetch to execute the program?
- ° How many clocks are required per instruction?
- ° How "lean" a clock is practical?

Best Metric: Time to execute the program!



NOTE: this depends on instructions set, processor organization, and compilation techniques.



Architecture Styles..

According to the operand(s) locations..

- Accumulator-style

One of the operands is in an implicit register known as accumulator

- Load-store architecture
  Both operands must be in the registers

- Register-memory
  One operand in register, the other in Memory

- Memory-Memory
  Both operands can be in Memory

- Stack-style
  Stack is used to evaluate expressions