ECE 3057: Architecture, Concurrency and Energy in Computation Fall 2019

Lecture 2: Instruction Set Architecture

David Anderson

School of Electrical and Computer Engineering
Georgia Institute of Technology

anderson@gatech.edu

Acknowledgment: Slides adapted from MIT 6.823 (Arvind and J. Emer). CMU 18-447 (O. Mutlu) and GT ECE 3057 (S. Yalamanchili, T. Krishna)

Topics

- Why do we need Instruction Set Architectures?
  - A Historical Perspective
- Elements of an ISA
- MIPS ISA

1823: Charles Babbage

A polynomial can be computed from difference tables

An example:

$f(n) = n^2+n+41$
$d1(n) = f(n) - f(n-1) = 2n$
$d2(n) = d1(n) - d1(n-1) = 2$

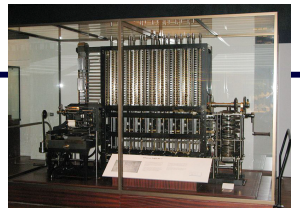
$d2(n) = 2$
$d1(n) = d1(n-1) + d2(n)$
$f(n) = f(n-1) + d1(n)$

Charles Babbage 1791-1871
Lucasian Professor of Mathematics, Cambridge University, 1827-1839
Father of the "Computer"

n	0	1	2	3	4 ...
d2(n)			2 →	2 →	2
d1(n)		2 →	4 →	6 →	8
f(n)	41 →	43 →	47 →	53 →	61

What computation unit needed? *an adder!*

Difference Engine



1823
- Babbage's paper is published

1834
- The paper is read by Scheutz & his son in Sweden

1842
- Babbage gives up the idea of building it; He is onto Analytic Engine!

1855
- Scheutz displays his machine at the Paris World Fair
- Can compute any 6th degree polynomial
- *Speed:* 33 to 44 32-digit numbers per minute!
  - *Scheutz machine is at the Smithsonian.*
  - *Difference Engine #2 – London science museum*
    - *copy at computer museum in CA*

Analytic Engine

The forerunner to the modern digital computer

- 1833: Babbage's paper was published
  - conceived during a hiatus in the development of the difference engine
- Inspiration: Jacquard Looms
  - looms were controlled by punched cards
    - The set of cards with fixed punched holes dictated the pattern of weave => ***program***
    - The same set of cards could be used with different colored threads => ***inputs***
- 1871: Babbage dies
  - The machine remains unrealized
    - It is not clear if the analytic engine could be built even today using only mechanical technology

Ada Byron aka Lady Lovelace

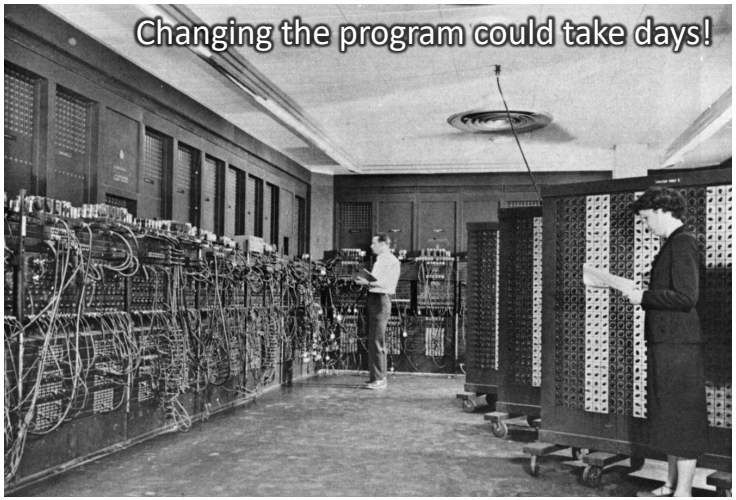
- Translated lectures of Luigi Menabrea who published notes of Babbage's lectures in Italy
- Lovelace considerably embellished notes and described how Analytical Engine could be used to calculate Bernoulli numbers.
  - The first program!

Ada Lovelace, 1815-1852
The first programmer

From Mechanical to Electrical

- **Harvard Mark I (1944)**
  - Proposed by Howard Aiken – Professor of Physics at Harvard and built by IBM Endicott laboratories
  - Essentially mechanical but had some electro-magnetically controlled relays and gears
    - Weighed *5 tons* and had *750,000* components
    - A synchronizing clock that beat every *0.015* seconds
  - Performance
    - 0.3 sec for addition, 6 seconds for Multiplication, 1 minute for sine
  - But -- broke down once a week!
- **ENIAC**
  - Designed and built by Eckert and Mauchly at the University of Pennsylvania during 1943-45
  - **The first, completely electronic, operational, general-purpose analytical calculator!**
    - 30 tons, 72 square meters, 200KW
    - Lights dimmed in Philadelphia when turned on.
  - Application: Ballistic Calculations
  - Performance
    - Read in 120 cards per minute, Addition 200 μs, Division 6 ms
    - 1000 times faster than Mark I
  - But frequent tube burnouts reduced accuracy

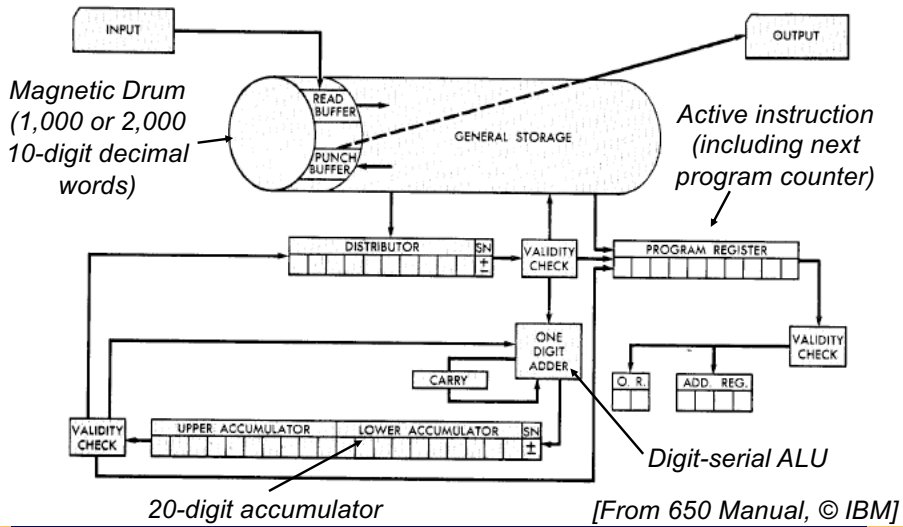
Electronic Numerical Integrator and Computer (ENIAC)



Electronic Discrete Variable Automatic Computer (EDVAC)

- ENIAC's programming system was external
  - Sequences of instructions were executed independently of the results of the calculation
  - Human intervention required to take instructions "out of order"
- EDVAC was designed by Eckert, Mauchly and von Neumann in 1944 to solve this problem
  - Solution was the **stored program computer**
    - "program can be manipulated as data"
      - The same storage can be used to store program and data
- First Draft of a report on EDVAC was published in 1945, but just had von Neumann's signature!
  - **Without a doubt the most influential paper in computer architecture**

The IBM 650 (1953-54)



Programmer's view of IBM 650

A drum machine with 44 instructions

Instruction: 60 1234 1009

"Load the contents of location 1234 into the *distribution*; put it also into the *upper accumulator*; set *lower accumulator* to zero; and then go to location 1009 for the next instruction."

- Programmer's view of the machine was inseparable from the actual hardware implementation
  - Good programmers optimized the placement of instructions on the drum to reduce latency!

Compatibility Problem at IBM

By early 60's, *IBM had 4 incompatible lines of computers!*

701	→	7094
650	→	7074
702	→	7080
1401	→	7010

Each system had its own

- Instruction set
- I/O system and Secondary Storage:
  magnetic tapes, drums and disks
- assemblers, compilers, libraries,...
- market niche
  business, scientific, real time, ...

⇒ *IBM 360*

IBM 360 : Design Premises
Amdahl, Blaauw and Brooks, 1964

Architecture of the IBM System/360

Abstract: The architecture* of the newly announced IBM System/360 features four innovations:

1. An approach to storage which permits and exploits very large capacities, hierarchies of speeds, read-only storage for microprogram control, flexible storage protection, and simple program relocation.

2. An input/output system offering new degrees of concurrent operation, compatible channel operation, data rates approaching 5,000,000 characters/second, integrated design of hardware and software, a new low-cost, multiple-channel package sharing main-frame hardware, new provisions for device status information, and a standard channel interface between central processing unit and input/output devices.

3. A truly general-purpose machine organization offering new supervisory facilities, powerful logical processing operations, and a wide variety of data formats.

4. Strict upward and downward machine-language compatibility over a line of six models having a performance range factor of 50.

The term **architecture here is used to describe the attributes of a system as seen by a programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the dataflow and controls, the logical design and the physical **implementation***

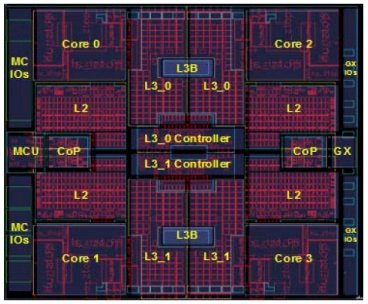
IBM 360: Initial Implementations (1964)

	Model 30	. . .	*Model 70*
Memory Capacity	8K - 64 KB		256K - 512 KB
Memory Cycle	2.0μs	...	1.0μs
Datapath	8-bit		64-bit
Circuit Delay	30 nsec/level		5 nsec/level
Registers	in Main Store		in Transistor
Control Store	Read only 1μsec		Dedicated circuits

- Six implementations (Models, 30, 40, 50, 60, 62, 70)
- 50X performance difference cross models
- ISA completely hid the underlying technological differences between various models.
- With minor modifications, IBM 360 ISA is still in use!

IBM 360: Forty-Six years later… zEnterprise196 Microprocessor

- 1.4 billion transistors, Quad core design
  - Up to 96 cores (80 visible to OS) in one multichip module
  - 5.2 GHz, IBM 45nm SOI CMOS technology
    - 64-bit virtual addressing
  - original 360 was 24-bit; 370 was a 31-bit extension
  - Superscalar, out-of-order
    - Up to 72 instructions in flight
  - Variable length instruction pipeline: 15-17 stages
  - Each core has 2 integer units, 2 load-store units and 2 floating point units
  - 8K-entry Branch Target Buffer
    - Very large buffer to support commercial workload
  - Four Levels of caches:
    - 64KB L1 I-cache, 128KB L1 D-cache
    - 1.5MB L2 cache per core
    - 24MB shared on-chip L3 cache
    - 192MB shared off-chip L4 cache



[September 2010]

Elements of an ISA

Elements of an ISA

- Instructions
  - Opcodes, Addressing Modes, Data Types
  - Instruction Types and Formats
  - Registers, Condition Codes
- Memory
  - Address space, Addressability, Alignment
  - Virtual memory management
- Call, Interrupt/Exception Handling
- Access Control, Priority/Privilege
- I/O: memory-mapped vs. instr.
- Task/thread Management
- Power and Thermal Management
- Multi-threading support, Multiprocessor support

Two Kinds of ISAs

- **Complex instruction Set Computing (CISC)**
  - An instruction does a lot of work, e.g. many operations
    - Insert in a doubly linked list
    - Compute FFT
    - String copy
- **Reduced Instruction Set Computing (RISC)**
  - An instruction does small amount of work, it is a primitive
    - Add
    - XOR
    - Shift

Complex vs Simple Instructions

- Advantages of Complex instructions
  - Denser encoding → smaller code size → better memory utilization, saves off-chip bandwidth, better cache hit rate (better packing of instructions)
  - Easier for assembly-language programming
  - Simpler compiler: no need to optimize small instructions as much (with caveats)
- Disadvantages of Complex Instructions
  - Larger chunks of work → compiler has less opportunity to optimize
  - More complex hardware → translation from a high level to control signals and optimization needs to be done by hardware
  - Compiler is limited in fine-grained optimizations it can do

Examples

- CISC
  - X86, PDP-11, VAX, …
- RISC
  - Alpha, MIPS, ARM, RISC-V
- Most Popular ISAs Today?
  - x86 vs ARM
    - x86 micro-ops → RISC-like micro instructions
  - RISC-V - an open-source ISA from Berkeley

Microarchitecture

- Implementation of an ISA under specific design constraints and goals
  - Constraints
    - Cost
    - Performance
    - Maximum power consumption
    - Energy consumption (battery life)
    - Availability
    - Reliability and Correctness
    - Time to Market
  - Goals
    - determined by the "Problem" space (application space)
    - the intended users/*market*
      - E.g., smartphone vs laptop vs HPC vs cloud

Microarchitecture

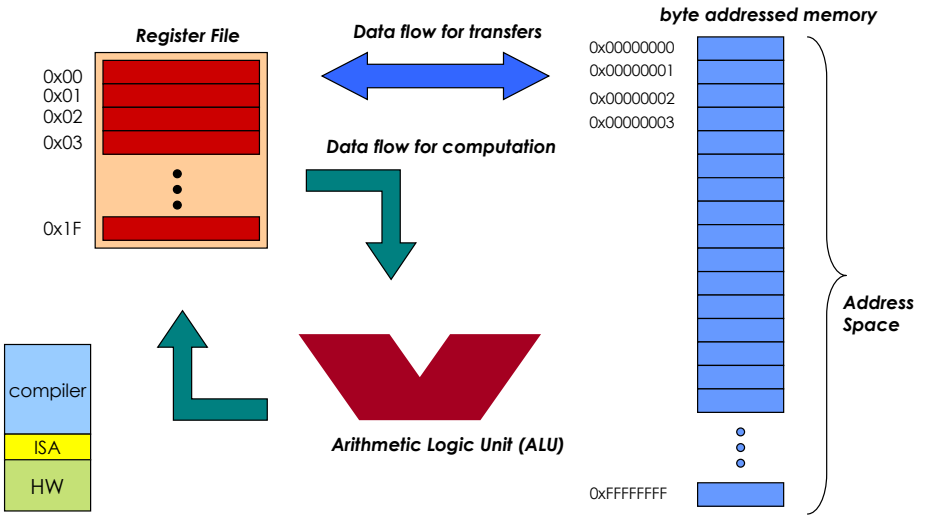
- Anything done in hardware without exposure to software
  - Pipelining
  - In-order versus out-of-order instruction execution
  - Memory access scheduling policy
  - Speculative execution
  - Superscalar processing (multiple instruction issue?)
  - Clock gating
  - Caching? Levels, size, associativity, replacement policy
  - Prefetching?
  - Voltage/frequency scaling?
  - Error correction?

MIPS ISA

Readings

- Useful Reference listing all MIPS instructions and encodings:
  - http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html
- Chapter 2
  - 2.1, Figure 2.1, 2.2 – 2.7
  - 2.9, Figure 2.15 (2.9.1 in online text)
  - 2.10, 2.16, 2.17
- Appendix A9, A10

Instruction Set Architecture (ISA)

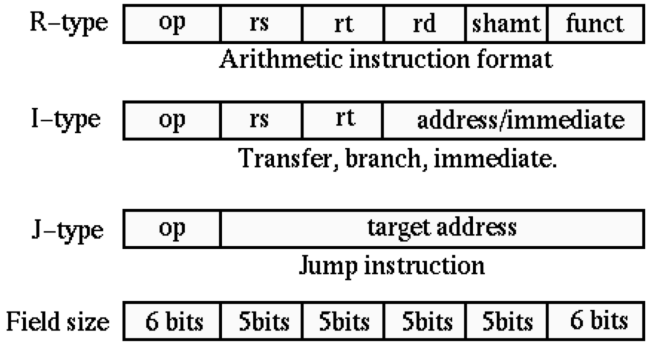


MIPS ISA

- Arithmetic and Logical Instructions
- Review memory organization
- Memory (data movement) instructions
- Control flow instructions
- Procedure/Function calls
- Program assembly, linking, & encoding

Not being covered

MIPS ISA



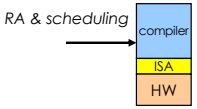
Useful Reference listing all MIPS instructions and encodings:
http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html

MIPS ISA

- Arithmetic and Logical Instructions
- Review memory organization
- Memory (data movement) instructions
- Control flow instructions
- Procedure/Function calls
- Program assembly, linking, & encoding

Arithmetic and Logical Instructions

- Operands must be registers
  - only 32 registers provided
- All memory accesses accomplished via loads and stores to/from registers
  - A common feature of RISC processors
- Compiler associates variables with registers
  - Some registers are not visible to the programmer
    - Program counter
    - Status register
    - ……



MIPS Programmer Visible Registers

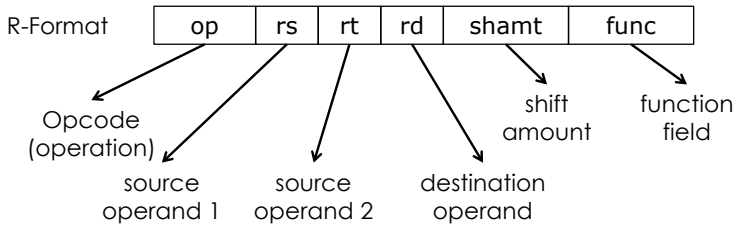
Register	Names	Usage by Software Convention
$0	$zero	Hardwired to zero
$1	$at	Reserved by assembler
$2 - $3	$v0 - $v1	Function return result registers
$4 - $7	$a0 - $a3	Function passing argument value registers
$8 - $15	$t0 - $t7	Temporary registers, caller saved
$16 - $23	$s0 - $s7	Saved registers, callee saved
$24 - $25	$t8 - $t9	Temporary registers, caller saved
$26 - $27	$k0 - $k1	Reserved for OS kernel
$28	$gp	Global pointer
$29	$sp	Stack pointer
$30	$fp	Frame pointer
$31	$ra	Return address (pushed by call instruction)
$hi	$hi	High result register (remainder/div, high word/mult)
$lo	$lo	Low result register (quotient/div, low word/mult)

Logical Operations

- Instructions for bitwise manipulation
- Useful for extracting and inserting groups of bits in a word
  - e.g.,: masking, inverting, …

Operation	C	Java	MIPS
Shift left	<<	<<	`sll`
Shift right	>>	>>>	`srl`
Bitwise AND	&	&	`and, andi`
Bitwise OR	|	|	`or, ori`
Bitwise NOT	~	~	`nor`

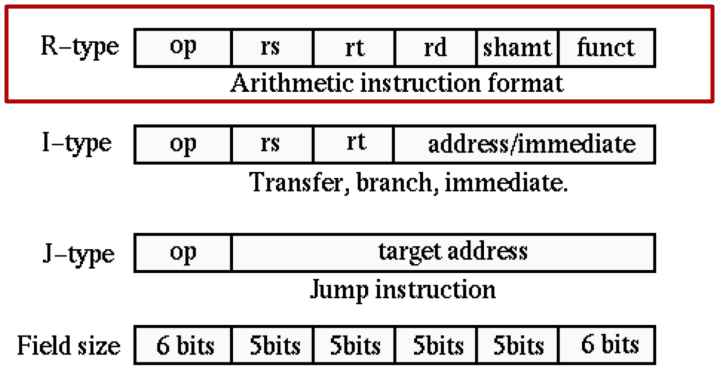
Encoding: Instruction Format



- Instructions, like registers and words of data, are also 32 bits long
  - Example: `add $t0, $s1, $s2`
  - registers have numbers, `$t0=9, $s1=17, $s2=18`

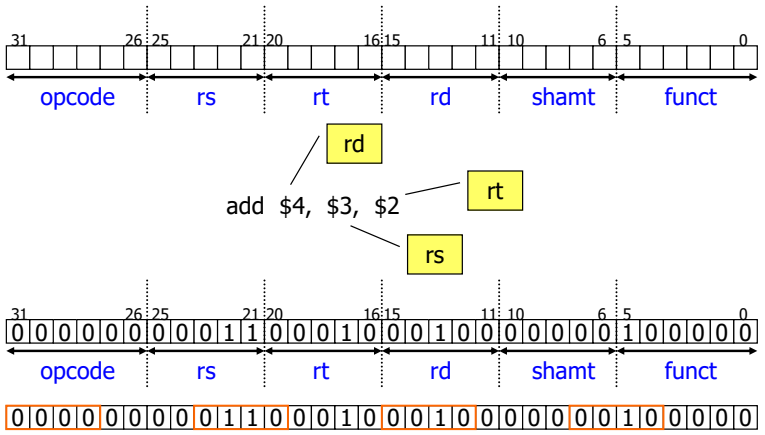
Opcodes on page A-50 (Figure 7.10.2)
Encodings – Section A10 (7.10)

MIPS ISA



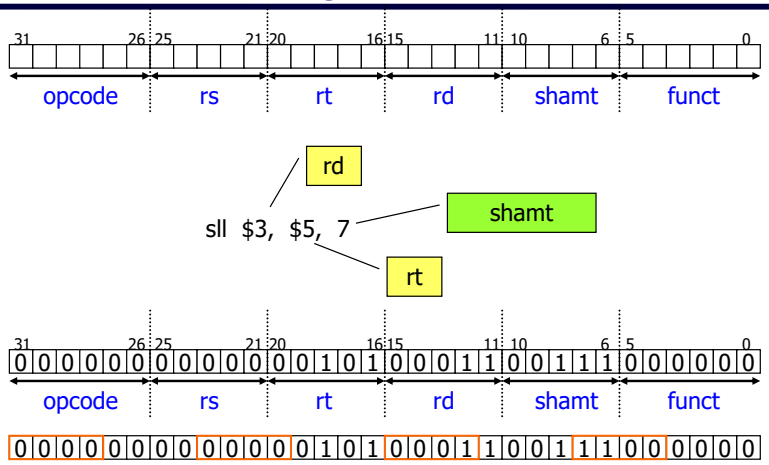
Useful Reference listing all MIPS instructions and encodings:
http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html

R-Type Encoding Example: ADD



Encoding = 0x00622020

R-Type Encoding Example: SLL



Encoding = 0x000519C0

MIPS ISA

- Arithmetic and Logical Instructions
- Review memory organization
- Memory (data movement) instructions
- Control flow instructions
- Procedure/Function calls
- Program assembly, linking, & encoding

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.

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
4	8 bits of data
5	8 bits of data
6	8 bits of data
...	

Memory Organization

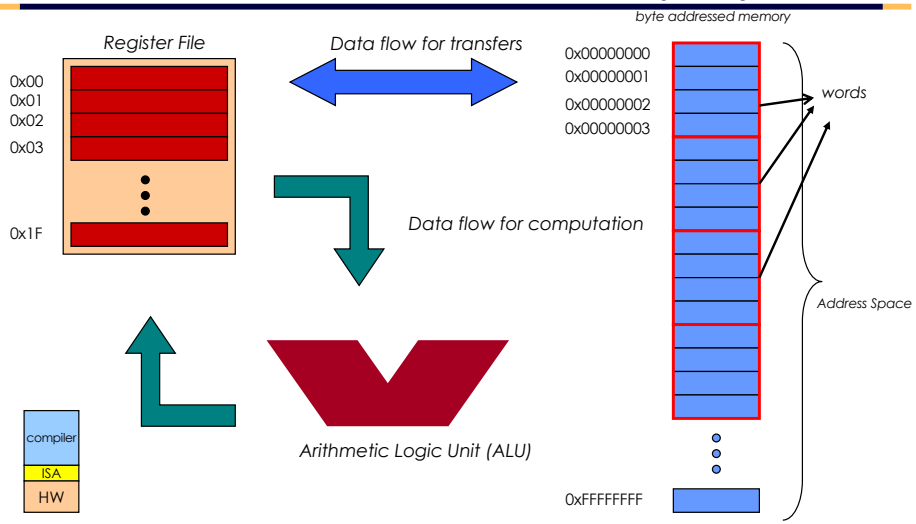
- Bytes are nice, but most data items use larger "words"
  - MIPS provides LW/LB and SW/SB instructions
- For MIPS, a word is 32 bits or 4 bytes.

0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data

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?

Instruction Set Architecture (ISA)

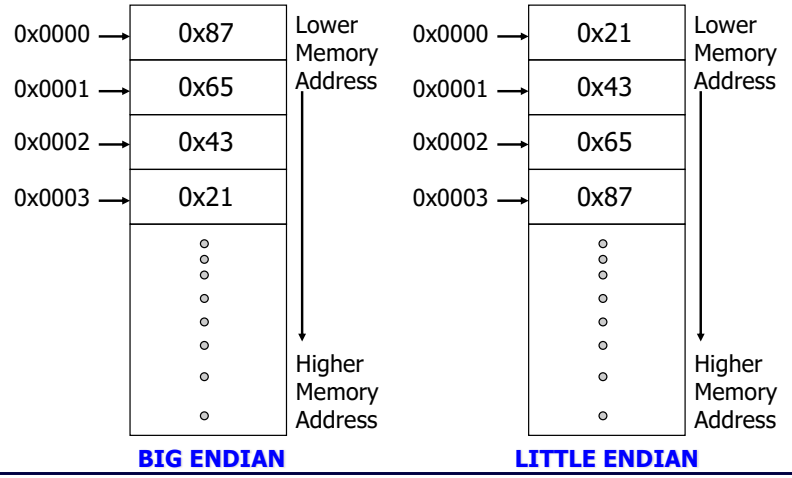


Endianness [defined by Danny Cohen 1981]

- Byte ordering — How is a multiple byte data word stored in memory
- Endianness (from Gulliver's Travels)
  - Big Endian
    - Most significant byte of a multi-byte word is stored at the lowest memory address
    - e.g. Sun Sparc, PowerPC
  - Little Endian
    - Least significant byte of a multi-byte word is stored at the lowest memory address
    - e.g. Intel x86
- Some embedded & DSP processors would support both for interoperability
  - Most MIPS architectures (except R8000) supported both

Example of Endian

Store 0x87654321 at address 0x0000, byte-addressable

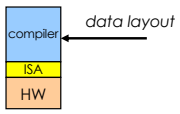


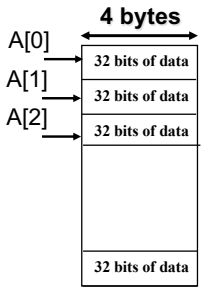
MIPS ISA

- Arithmetic and Logical Instructions
- Review memory organization
- Memory (data movement) instructions
- Control flow instructions
- Procedure/Function calls
- Program assembly, linking, & encoding

Memory Instructions

- Load and Store Instructions
  - Load: lb, lui, lw
  - Store: sb, sw





C code:
```
long A[100];
A[9] = h + A[8];
```

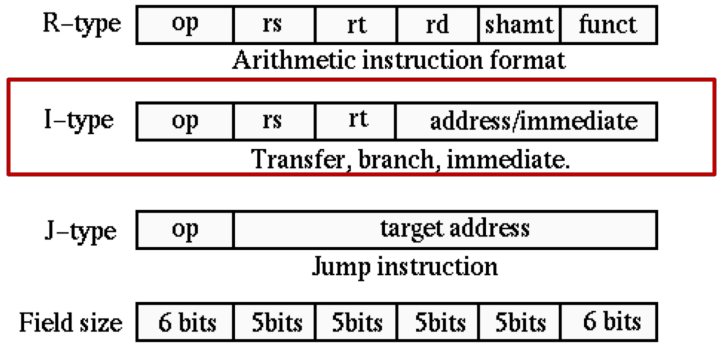
MIPS code:
```
lw  $t0, 32($s3) #load word
add $t0, $s2, $t0
sw  $t0, 36($s3)
```

index base register

Registers vs Memory

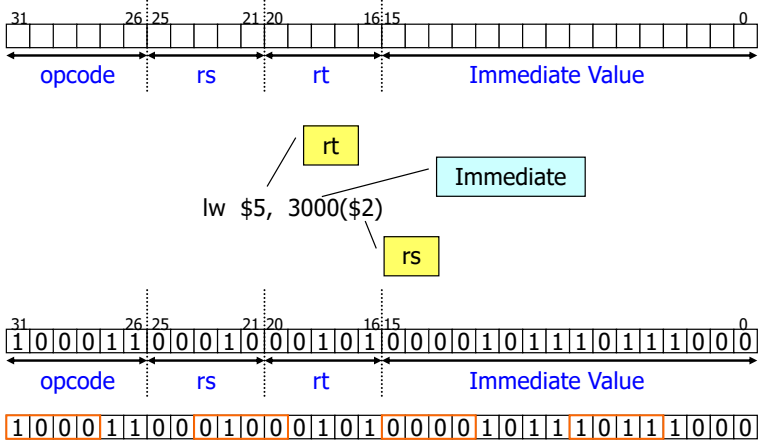
- Registers are faster to access than memory
  - Why? *Smaller is faster*
  - Rationale for the Memory Hierarchy
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register usage optimization is important!

MIPS ISA



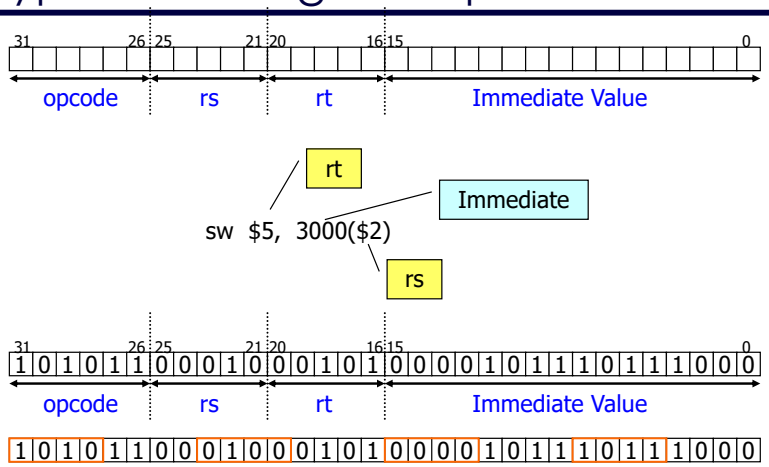
Useful Reference listing all MIPS instructions and encodings:
http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html

I-Type Encoding Example: LW



Encoding = 0x8C450BB8

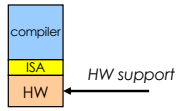
I-Type Encoding Example: SW



Encoding = 0xAC450BB8

What are Immediate values?

- Small constants are used quite frequently (50% of operands)
  e.g., A = A + 5;
  B = B + 1;
  C = C - 18;
- Solutions?
  - put 'typical constants' in memory and load them.
  - create hard-wired registers (like $zero) for constants like one.
  - Use immediate values
- Make Common Case Fast!
  - Small constants (<16-bit) are common
  - Immediate operand avoids a load instruction



Immediate Operands

- MIPS Instructions:

```
addi $29, $29, 4
slti $8, $18, 10
andi $29, $29, 6
ori $29, $29, 4
```

- No subtract immediate instruction
  - Just use a negative constant

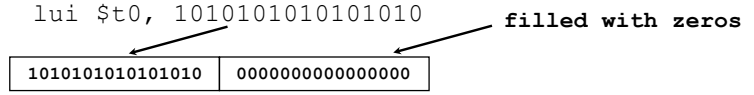
```
addi $s2, $s1, -1
```

- Hardwired values useful for common operations
  - E.g., move between registers

```
add $t2, $s1, $zero
```

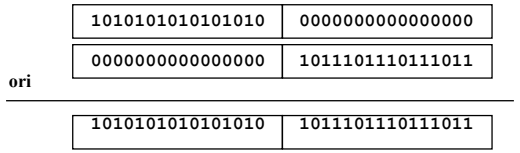

What about larger constants?

- We'd like to be able to load a 32 bit constant into a register
- Must use two instructions, new "load upper immediate" instruction



- Then must get the lower order bits right, i.e.,

```
ori $t0, $t0, 1011101110111011
```

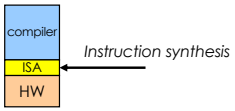


Pseudo Instructions

- Assembly instructions implemented with a mix of MIPS instructions

```
li $t0, 0xAAAABBBB

=>  lui $t0, 0xAAAA
    ori $t0, $t0, 0xBBBB
```




```
move $t1, $t2

=> add $t1, $t2, $zero
```


MIPS ISA

- Arithmetic and Logical Instructions
- Review memory organization
- Memory (data movement) instructions
- Control flow instructions
- Procedure/Function calls
- Program assembly, linking, & encoding

Control

- 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: ....
```


Control

- MIPS unconditional branch instructions:

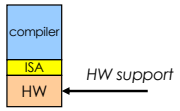
```
j  label
```

- Example:

Assembler calculates address

```
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:...
```

- *Can you build a simple loop?*



Compiling Loop Statements

- C code:

```
while (save[i] == k) i += 1;
```

  - i in $s3, k in $s5, address of save in $s6
- Compiled MIPS code:

```
Loop: sll  $t1, $s3, 2 #multiply by 4
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```


Encoding: Branches & Jumps

- Instructions:

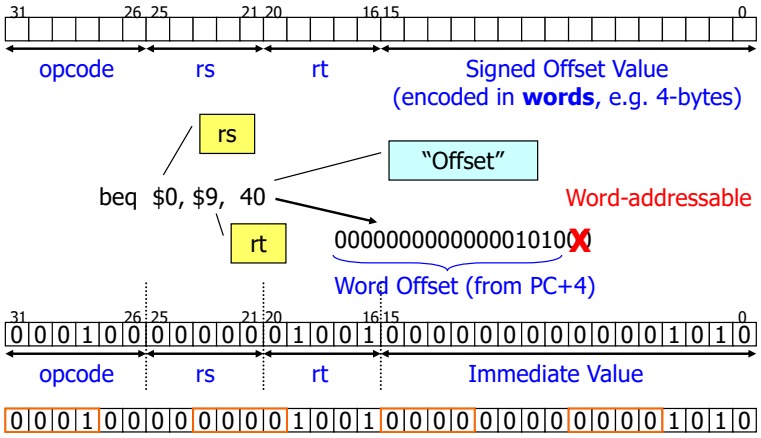
`bne $t4,$t5,Label`	Next instruction is at Label if `$t4` ≠ `$t5`
`beq $t4,$t5,Label`	Next instruction is at Label if `$t4` = `$t5`
`j Label`	Next instruction is at Label

- Formats:

Opcodes on page A-50 (Figure 7.10.2)
Encodings – Section A10 (7.10)

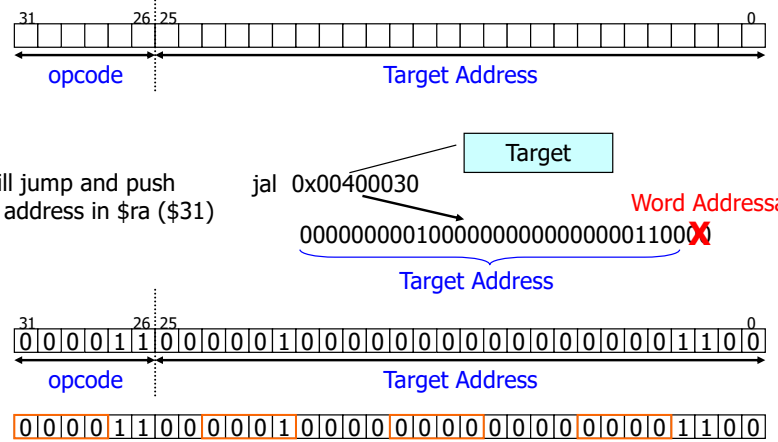
I	op	rs	rt	16 bit address
J	op	26 bit address		

I-Type Encoding Example: BEQ/BNE



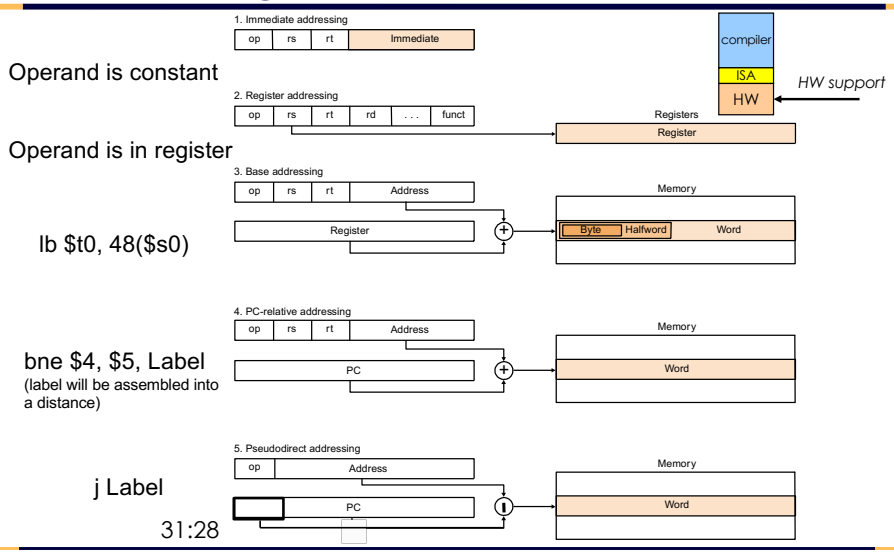
Encoding = 0x1009000A

J-Type Encoding Example: JAL



Encoding = 0x0C10000C

Addressing Modes



How far can you branch?

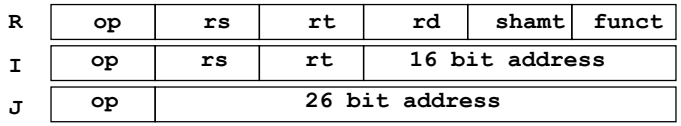
- BEQ/BNE
  - 2^16 words relative to PC+4
    - Most branches are local
  - What if branch is too far to encode?
    - Assembler rewrites code

```
        beq $s0,$s1, L1
                ↓
L1:     bne $s0,$s1, L2
        j L1
L2:     …
```

- JR/JAL
  - 2^26 words within boundary starting from PC[31:28]

Summary: MIPS ISA

- Simple instructions all 32 bits wide
- Very structured
- Only three instruction formats



- Rely on compiler to achieve performance

Opcodes on page A-50 (Figure 7.10.2)
Encodings – Section A10 (7.10)