Computer Architecture. Week 6

Muhammad Fahim, Alexander Tormasov

Innopolis University

m.fahim@innopolis.ru
a.tormasov@innopolis.ru

October 5, 2020

Topic of the lecture

- Instructions: Language of the Computer

Topic of the tutorial

- Yuri Panchul will deliver the tutorialc

Topic of the lab

- MIPS Instruction
- MARS

Content of the class

Finite State Machines (FSMs) (1/2)

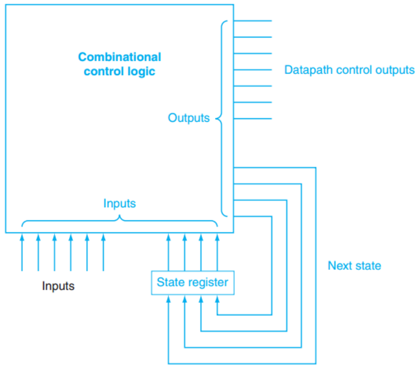
- A FSM is defined as:
  - A set of states S (circles), an initial state s_0 and a transition function that maps from the current input and current state to the output and the next state (arrows between states)
- Mathematical Representation
  - Consider
    - S = $s_0, s_1, \ldots s_{n-1}$ is a finite set of states.
    - I = $i_0, i_1, \ldots i_{k-1}$ is a finite set of input values.
    - O= $o_0, o_1, \ldots o_{m-1}$ is a finite set of output values.
  - **Definition**: A finite state machine is a function $F : (S \times I) \to (S \times O)$ that gets a sequence of input values $I_k \in$ I, k = 0,1,2, ... and it produces a sequence of output values $O_k \in$ O, k= 1,2, ... such that:

$$\mathbf{F}(s_k,\ i_k) = (s_{k+1},\ o_{k+1}),\ \text{k} = 0,1,2,\ \ldots$$

Finite State Machines (FSMs) (2/2)

- Function can be represented with a state transition diagram
- With combinational logic and registers, any FSM can be implemented in hardware
- State transitions are controlled by the clock
  - On each clock cycle the machine checks the inputs and generates a new state (could be same) and new output
- You will see FSMs in Discrete Math course as well.

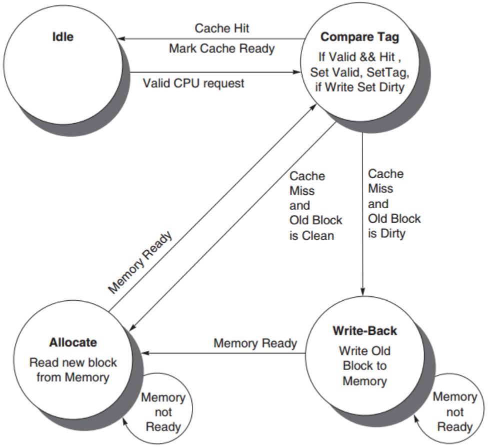
Hardware Implementation of FSM



- Finite-state machine controllers are typically implemented using a block of combinational logic and a register to hold the current state
- **NOTE:** Transition state can not be captured/observed and it is hidden inside transaction.

FSM Example I

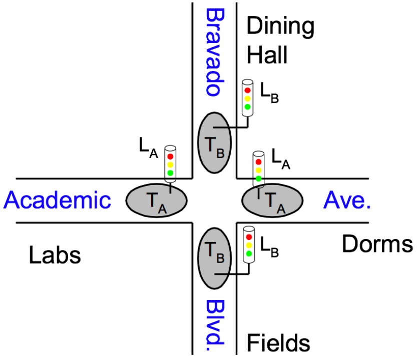
- Simple Cache Controller



FSM Example II (1/6)

Traffic light controller

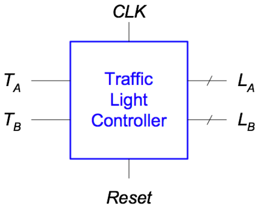
- **Traffic sensors**: T_A, T_B (TRUE when there's traffic)
- **Lights**: L_A, L_B

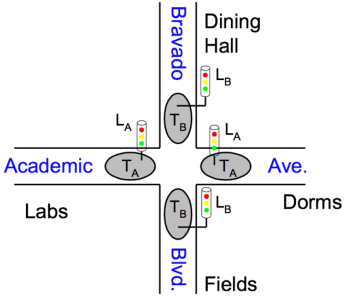


FSM Example II (2/6)

Traffic light controller

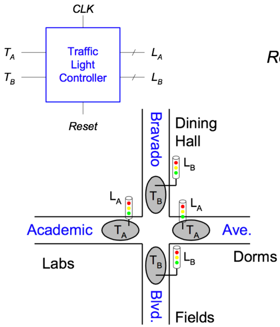
- **Inputs**: CLK, Reset, T_A, T_B
- **Outputs**: L_A, L_B

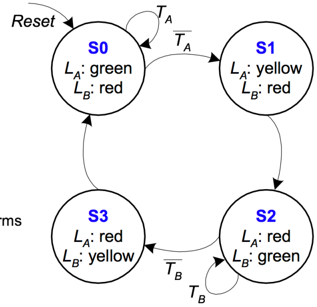




FSM Example II (3/6)

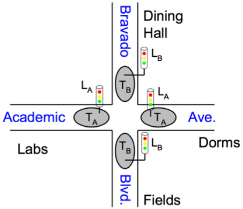
- **FSM**: Outputs labeled in each state
- **States**: Circles
- **Transitions**: Arcs



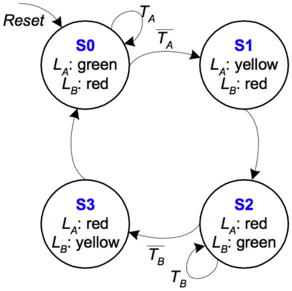


FSM Example II (4/6)

FSM State Transition Table



Current State	Inputs		Next State
S	T_A	T_B	S'
S0	0	X	S1
S0	1	X	S0
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0



FSM Example II (5/6)

FSM Encoded State Transition Table

Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

State	Encoding
S0	00
S1	01
S2	10
S3	11

$$S'_1 = S_1 \oplus S_0$$
$$S'_0 = \overline{S_1}\,\overline{S_0}\,\overline{T_A} + S_1\overline{S_0}\,\overline{T_B}$$

FSM Example II (6/6)

FSM Output Table

Current State		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

Output	Encoding
green	00
yellow	01
red	10

$$L_{A1} = S_1 \qquad L_{B1} = \overline{S_1}$$
$$L_{A0} = \overline{S_1} S_0 \qquad L_{B0} = S_1 S_0$$

Instruction Set

- To command a computer's hardware, you must speak its language.
- The words of a computer's language are called instructions, and its vocabulary is called an instruction set.
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

Classification of Instruction Sets

- The instruction sets can be classified by
  - Number of addresses (1,2,3...), or
  - Operand internal storage inside the CPU;
  - Number of explicit operands per instruction;
  - Operand location;
  - Operations;
  - Type and size of operands.
- The type of internal storage in the CPU is the most basic differentiation. The major choices are
  - Stack-based architecture
  - Accumulator-based architecture
  - Register-Set based architecture

Stack-based Architecture

- A stack machine has a stack as a part of the processor state
- No registers, No explicit operands in ALU
- Typical operations: push, pop, +, *, ...
- Instructions like + implicitly specify the top 2 elements of the stack as operands.
- For Example: The code segment for A = B + C * D

```
Push B;    Push C;    Push D
Mul
Add
Pop A
```

- Promoted in 60's and gain limited popularity. Examples are: Burroughs B5500/6500, HP3000/70
- **Advantages:** Short instructions and easy to write compiler
- **Disadvantages:** Inefficient code and stack is a bottleneck

Accumulator-based Architecture

- Single register A
- One explicit operand per instruction
- Two instruction types: op (A := A op *M) and load/store (*M := A)
- Short instructions possible, minimal internal state
- High memory traffic – many loads and stores

- For Example: The code segment for C = A + B

```
Load A
Add B
Store C
```

- The most popular early architecture: IBM 7090, DEC PDP-8, MOS 6502

Register-Set Based Architecture

- All operands are explicit either registers or memory locations
- General Purpose Registers (GPR)
- Allows fast access to temporary values
- Permits clever compiler optimization
- Reduce traffic to memory
- For Example: The code segment for C = A + B

```
load $t0, A
load $t1, B
add $s0, $t0, $t1
```

- The dominant architectures are CDC 6600, IBM360/370 (RX), PDP-11, 68000, all RISC machines, etc.

Remarks on Instruction Sets

- While most early machines used stack or accumulator-style architectures
- All machines designed in the past twenty years use a register-set based architecture (except quantum).
- The reason for the registers are:
  - Faster than memory
  - Easier for a compiler to use
  - It can be used more effectively

Code Comparison

- Code sequence C = A + B for different instruction set architectures

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A Push B Add Pop C	Load A Add B Store C	Load R1, A Add R1, B Store C, R1	Load R1,A Load R2, B Add R3, R1, R2, Store C, R3

Comparison of ISAs

- **Stack-based Architecture**
  - **Pros.**: Simple model of expression evaluation. Good code density.
  - **Cons.**: A stack can't be randomly accessed. It makes it difficult to generate efficient code.
- **Accumulator-based Architecture**
  - **Pros.**: Minimizes internal state of machine. Short instructions
  - **Cons.**: Since accumulator is only temporary storage, memory traffic is high.
- **Register-Set based Architecture**
  - **Pros.**: Most general model for code generation
  - **Cons.**: All operands must be named, leading to longer instructions

Classification of ISAs by Complexity Metric

- An ISA may be classified by architectural complexity.
- **Complex Instruction Set Computer (CISC)**
  - It has many specialized instructions, some of which may only be rarely used in practical programs.
  - Example: IBM 370/168, VAX 11/780, Intel 80486
- **Reduced Instruction Set Computer (RISC)**
  - It simplifies the processor by efficiently implementing only the instructions that are frequently used in programs, while the less common operations are implemented as subroutines.
  - Example: SUN SPARC, Intel i86, MIPS R2000, R4000, MIPS also used in many embedded computers

CISC vs RISC

- Comparison of CISC and RISC

CISC	**RISC**
Multiple instruction sizes and format	Instruction of the same set with a few formats
More addressing modes	Fewer addressing modes
Complexity lies in microprogram	Complexity lies in compiler
Pipelining is difficult	Pipelining is easy
Emphasis on hardware	Emphasis on software

The MIPS Instruction Set Architecture

- Developed by John Hennessy and colleagues at Stanford in 1980's
- MIPS is a reduced instruction set computer (RISC) ISA developed by MIPS computer systems (now MIPS Technologies)
- Large share of embedded core market
- Applications in consumer electronics, network/storage equipment, cameras, printers, ...
- Used MIPS32 as an example throughout the course

- Guide to MIPS64 `https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MIPS_Architecture_MIPS64_InstructionSet_%20AFP_P_MD00087_06.05.pdf`
- MIPS is a commercial derivative of the original RISC-1 design from Berkely

A modern ISA – RISC-V

- **RISC-V** is an open standard instruction set architecture (ISA) based on established reduced instruction set computer (RISC) principles.

- Unlike most other ISA designs, the RISC-V ISA is provided under open source licenses that do not require fees to use.
  - **Designer**: University of California, Berkeley
  - **Bits**: 32, 64, 128
  - **Introducing Year**: 2010

- RISC-V is the fifth iteration of that architecture. **Two differences**:
  - It is more modern; it includes learning ideas that MIPS did not
  - It is more versatile expandable
- Book Link: `https://tinyurl.com/y6brwos8`

Operations of the Computer Hardware

- Every computer must be able to perform arithmetic
- Add and subtract consists of three operands
- Two sources and one destination

$$add\ \ a,\ b,\ c \qquad \#\ a\ gets\ b\ +\ c$$

- All arithmetic operations have this form
- **Design Principle 1:** Simplicity favours regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Arithmetic Example

- C code

$$f = (g + h) - (i + j);$$

- MIPS code

$$\begin{array}{ll} add\ t0,\ g,\ h & \#\ temp\ \ t0 = g + h \\ add\ t1,\ i,\ j & \#\ temp\ \ t1 = i + j \\ sub\ f,\ t0,\ t1 & \#\ f = t0 - t1 \end{array}$$

Operands of the Computer Hardware

- Arithmetic instructions use register operands
- MIPS has a 32 × 32-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a “word”
- Assembler names
  - $t0, $t1, ..., $t9 for temporary values
  - $s0, $s1, ..., $s7 for saved variables
- **Design Principle 2:** Smaller is faster
  - Smaller number of bits, gates and transistors

Operands of the Computer Hardware

- C code

$$f \ = \ (g \ + \ h) \ - \ (i \ + \ j);$$

  - f, ..., j in \$s0, ..., \$s4

- Compiled MIPS code

$$\begin{gathered} add\ \$t0,\ \$s1,\ \$s2 \\ add\ \$t1,\ \$s3,\ \$s4 \\ sub\ \$s0,\ \$t0,\ \$t1 \end{gathered}$$

Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Accessing word should be a multiple of 4 bytes align
  - Accessing the half-word should be 2 bytes align
- MIPS architecture processors can be configured to run either in big or in little endian mode.

Byte Ordering

- Jonathan Swift's Gulliver's Travels
- Big Endian: Least significant byte has highest address
- Little Endian: Least significant byte has lowest address
- Example: Word x value 0x01234567, with address &x is 0x100
- Big Endian

	0x100	0x101	0x102	0x103			
	01	23	45	67			

- Little Endian

	0x100	0x101	0x102	0x103			
	67	45	23	01			

- For little-endian, the assembly language instructions that work with different length numbers (1, 2, 4 bytes) proceed in the same way by first picking up the least significant byte, at address base+0 and going towards the most significant byte

Memory Operand Example

- C code

$$g \ = \ h \ + \ A[8];$$

  - g in \$s1, h in \$s2 and base address of A in \$s3

- Compiled MIPS code

$$lw \quad \$t0,\ 32(\$s3)$$
$$add \quad \$s1,\ \$s2,\ \$t0$$

  - Index 8 requires offset of 32
  - In MIPS, words must start at addresses that are multiples of 4
  - This requirement is called an alignment restriction
  - Why? Because alignment leads to faster data transfers

Registers vs. Memory

- Registers are faster to access than memory
- 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 optimization is important!

Constant or Immediate Operands (1/2)

- Constant data specified in an instruction

$$addi\quad \$s3,\ \$s3,\ 4$$

- No subtract immediate instruction
  - Just use a negative constant

$$addi\quad \$s2,\ \$s1,\ -1$$

- **Design Principle 3:** Make the common case fast
  - Avoid memory access

Constant or Immediate Operands (2/2)

- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - For Example: move between registers

$$add \quad \$t2,\ \$s1,\ \$zero$$

Representing Instructions

- Instructions are encoded in binary called machine code
- MIPS instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, ...
  - Regularity!
- Register numbers
  $t0 – $t7 are reg's 8 – 15
  $t8 – $t9 are reg's 24 – 25
  $s0 – $s7 are reg's 16 – 23

MIPS Register Names and Conventions

Register Number	**Name**	**Usage**
\$0	\$zero	constant
\$1	\$at	assembler temporary
\$2–\$3	\$v0–\$v1	function return values
\$4–\$7	\$a0–\$a3	function arguments
\$8–\$15	\$t0–\$t7	temporaries
\$16–\$23	\$s0–\$s7	saved temporaries
\$24–\$25	\$t8–\$t9	more temporaries
\$26–\$27	\$k0–\$k1	reserved for OS kernel
\$28	\$gp	global pointer
\$29	\$sp	stack pointer
\$30	\$fp	frame pointer
\$31	\$ra	return address

MIPS R-format Instructions

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- **Instruction fields**
    - op: operation code (opcode)
    - rs: first source register number
    - rt: second source register number
    - rd: destination register number
    - shamt: shift amount only for shift instructions (00000 for now)
    - funct: function code (extends opcode)
- Example in coming tutorial session.

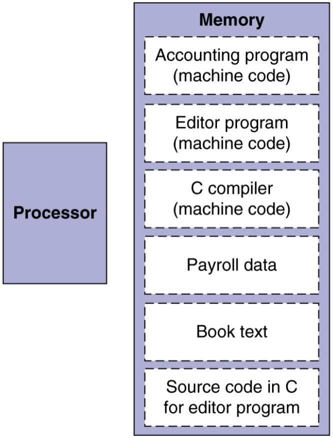
MIPS I-format Instructions

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: -2^{15} to $+2^{15}-1$
  - Address: offset added to base address in rs
- **Design Principle 4:** Good design demands good compromises
  - The compromise represented by the MIPS design, was to make all the instructions the same length,thereby requiring different instruction formats. For example: R-format, I-format and J-format.
  - Keep formats as similar as possible

Stored Program Computers

- The Big Picture (von Neumann architecture)


Processor
Memory
Accounting program (machine code)
Editor program (machine code)
C compiler (machine code)
Payroll data
Book text
Source code in C for editor program

Stored Program Computers

- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, ...
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	MIPS
Shift left logical	«	«	sll
Shift right logical	»	»>	srl
Bitwise AND	&	&	and, andi
Bitwise OR	|	|	or, ori
Bitwise NOT	∼	∼	nor

- Useful for extracting and inserting groups of bits in a word

Shift Operations

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- shamt: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - sll by i bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - srl by i bits divides by 2^i (unsigned only)
- Shift: arithmetic vs logical
  - left shift close to be identical (except exceptions)
  - Shift right different in sign bit which is used as filler

Example of Shift Operations (1/2)

- Shift Left Logical

Original pattern	10100111
Resulting pattern	01001110

```
sll  d, s, shft     # $d gets the bits in $s
                    # shifted left logical
                    # by shft positions
```

- A logical left shift by n bits can be used as a fast means of multiplying by 2^n

Example of Shift Operations (2/2)

- Shift Right Logical

Original pattern	1010 0111
Resulting pattern	01010 011

```
srl  d, s, shft     # $d gets the bits in $s
                    # shifted right logical
                    # by shft positions
```

- A logical right shift by n bits can be used as a fast means of division by 2^n
- For arithmetic shift resulting pattern will be: 11010 011
- NOTE: Bitwise AND, OR and NOT are simple and will be discussed in today's tutorial.

Branch Instructions

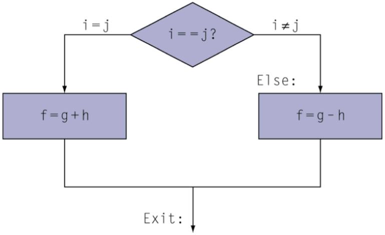
Conditional/Unconditional

- j L1
  - unconditional jump to instruction labeled L1
- beq rs, rt, L1
  - if (rs == rt) branch to instruction labeled L1
- bne rs, rt, L1
  - if (rs != rt) branch to instruction labeled L1

Compiling If Statements

- C Code

```
if (i == j)
    f = g + h;
else
    f = g - h;
```



- MIPS Code

```
bne $s3,$s4,Else          # go to Else if i != j
add $s0,$s1,$s2           # f = g + h (skipped if i != j)
j Exit                    # go to Exit
Else:sub $s0,$s1,$s2      # f = g - h (skipped if i = j)
Exit:
```

Compiling Loop Statements

- C Code

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

- MIPS Code

```
Loop: sll $t1,$s3,2        # Temp reg $t1 = i * 4
add $t1,$t1,$s6            # $t1 = address of save[i],
                           # The base of save in $s6
lw $t0,0($t1)              # Temp reg $t0 = save[i]
bne $t0,$s5, Exit          # go to Exit if save[i] != k
addi $s3,$s3,1             # i = i + 1
j     Loop                 # go to Loop
Exit:
```

Addressing in Branches

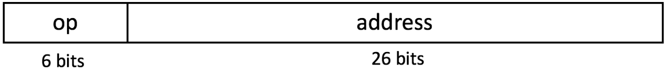
- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- PC-relative addressing
  - This mode can be used to load a register with a value stored in program memory
  - Target address = PC + offset $\times$ 4
  - PC already incremented by 4 by this time

Addressing in Jumps

- Jump (j and jal) targets could be anywhere in text segment
  - Encode full address in instruction



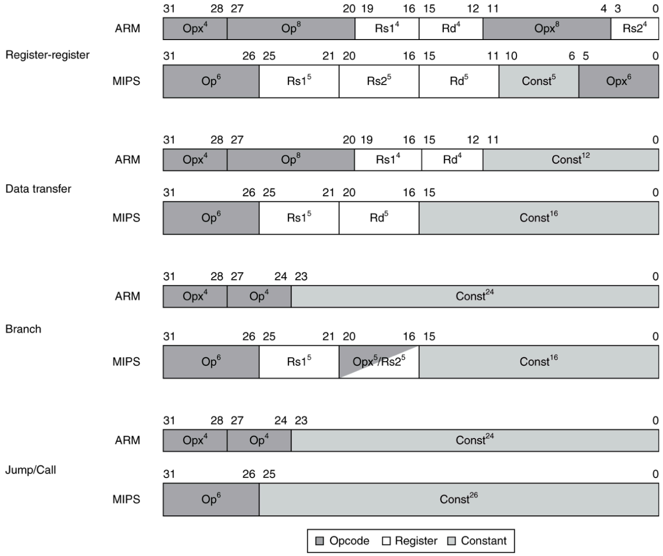
- (Pseudo) Direct jump addressing
  - It takes the upper four bits of the program counter, concatenated with the 26 bits of the direct address from the instruction, concatenated with two bits of 0 0

ARM and MIPS Similarities

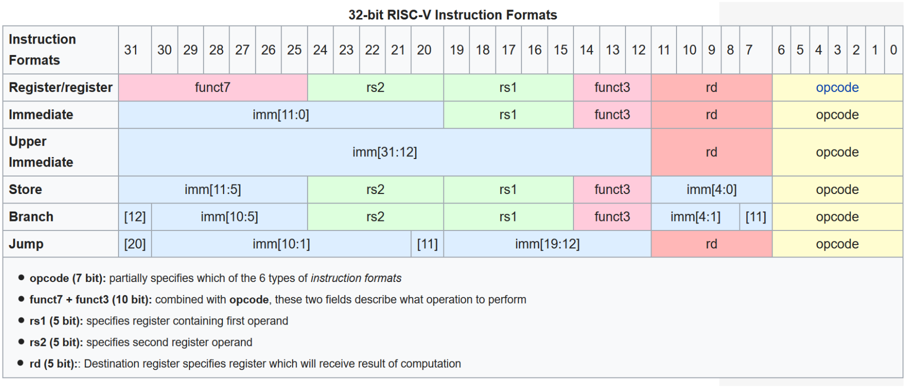
- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

	ARM	MIPS
Date Announced	1985	1985
Instruction Size	32 bits	32 bits
Address Space	32 bits	32 bits
Data Alignment	Aligned	Aligned
Data Addressing Modes	9	3
Registers	15 * 32 bits	31 * 32 bits
Input/Output	Memory Mapped	Memory Mapped

Instruction Encoding (ARM vs MIPS)



Instruction Set (RISC-V)



32-bit RISC-V Instruction Formats

Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Upper Immediate	imm[31:12]																				rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

- **opcode (7 bit):** partially specifies which of the 6 types of *instruction formats*
- **funct7 + funct3 (10 bit):** combined with **opcode**, these two fields describe what operation to perform
- **rs1 (5 bit):** specifies register containing first operand
- **rs2 (5 bit):** specifies second register operand
- **rd (5 bit)::** Destination register specifies register which will receive result of computation

The Intel x86 ISA

Evolution with backward compatibility

- 8080 (1974): 8-bit microprocessor
  - Accumulator, plus 3 index-register pairs
- 8086 (1978): 16-bit extension to 8080
  - Complex instruction set (CISC)
- 8087 (1980): floating-point coprocessor
  - Adds FP instructions and register stack
- 80286 (1982): 24-bit addresses, MMU
  - Segmented memory mapping and protection
- 80386 (1985): 32-bit extension (now IA-32)
  - Additional addressing modes and operations
  - Paged memory mapping as well as segments

The Intel x86 ISA

Further evolution...

- i486 (1989): pipelined, on-chip caches and FPU
  - Compatible competitors: AMD, Cyrix, ...
- Pentium (1993): superscalar, 64-bit datapath
  - Later versions added MMX (Multi-Media eXtension) instructions
  - The infamous FDIV bug
- Pentium Pro (1995), Pentium II (1997)
  - New microarchitecture (see Colwell, The Pentium Chronicles)
- Pentium III (1999)
  - Added SSE (Streaming SIMD Extensions) and associated registers
- Pentium 4 (2001)
  - New microarchitecture
  - Added SSE2 instructions

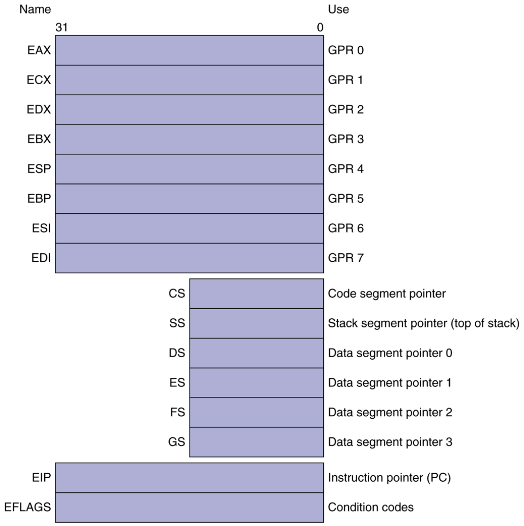
The Intel x86 ISA

And further...

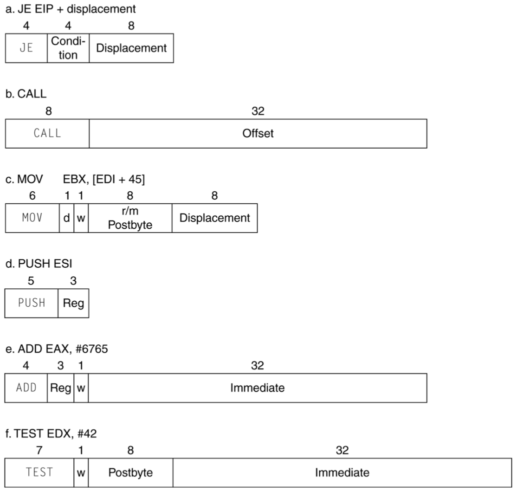
- AMDD64/x64 (2003): extended architecture to 64 bits
- EM64T – Extended Memory 64 Technology (2004)
- Intel Core (2006)
- AMD64 (announced 2007)
- Advanced Vector Extension (announced 2008)

If Intel didn't extend with compatibility, its competitors would
Technical elegance   market success

Basic x86 Registers



x86 Instruction Encoding



Note: Subset of instructions

Concluding Remarks

- Measure MIPS instruction executions in benchmark programs
  - Consider making the common case fast
  - Consider compromises

Instruction Class	**MIPS Examples**	**SPEC2006 Int**	**SPEC2006 FP**
Arithmetic	add, sub, addi	16%	48%
Data Transfer	lw, sw, lb, lbu, lh, lhu, sb, lui	35%	36%
Logical	and, andi, or, ori, sll, srl, nor	12%	4%
Condition, Branch	bne, beq, slt, slti, sltiu	34%	8%
Jump	j, jal, jr	2%	0%

Summary

Design Principles

1. Simplicity favors regularity
2. Smaller is faster
3. Make the common case fast
4. Good design demands good compromises