

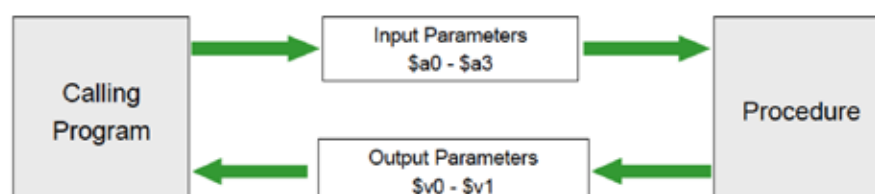
CSE2021 Computer Organization

Chapter 2: Part 2

Supporting Procedures

Procedure Calling

- Procedure (function) performs a specific task and return results to caller.
- Steps required
 1. Place parameters in registers
 2. Transfer control to procedure
 3. Acquire storage for procedure
 4. Perform procedure's operations
 5. Place result in register for caller



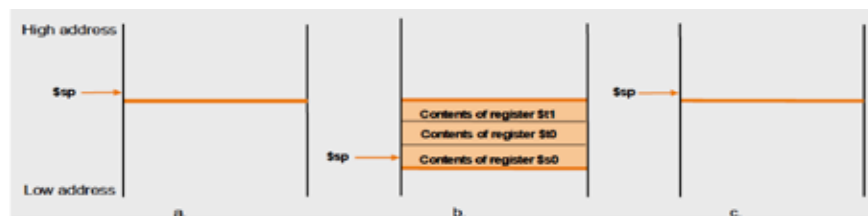
Procedure Calling

- Calling program
 - place parameters in registers \$a0 - \$a3
 - Transfer control to procedure
- Called procedure
 - Acquire storage for procedure, save values of required register in a stack \$sp
 - Perform procedure's operations, restore the values of registers that it used
 - Place result in register for caller \$v0 - \$v1
 - Return to place of call by returning to instruction whose address is saved in \$ra

Register Usage

- \$a0 – \$a3: arguments (reg's 4 – 7)
- \$v0, \$v1: result values (reg's 2 and 3)
- \$t0 – \$t9: temporaries
 - Can be overwritten by callee
- \$s0 – \$s7: saved
 - Must be saved/restored by callee
- \$gp: global pointer for static data (reg 28)
- \$sp: stack pointer (reg 29)
- \$fp: frame pointer (reg 30)
- \$ra: return address (reg 31)

Stack



- First-in-last-out queue
- Placing data onto the stack: push
- Removing data from the stack: pop
- Stack “grow” from higher addresses to lower addresses
 - Push values onto the stack by subtracting from the stack pointer
 - Pop values from the stack by adding to the stack pointer

Procedure Call Instructions

- Procedure call: jump and link
 - `jal ProcedureLabel`
 - Jumps to target address
 - Address of following instruction put in \$ra
 - \$ra is called the return address
- Procedure return: jump register
 - `jrr $ra`
 - Copies \$ra to program counter
 - Can also be used for computed jumps
 - e.g., for case/switch statements

Leaf Procedure Example

- Procedures that do not call others are called leaf procedure.
- C code:


```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

 - Arguments g, h, i, j in \$a0, \$a1, \$a2, \$a3
 - f in \$s0 (hence, need to save \$s0 on stack)
 - Result in \$v0

Leaf Procedure Example

- MIPS code for procedure:

leaf_example:	
addi \$sp, \$sp, -4	Save \$s0 on stack
sw \$s0, 0(\$sp)	
add \$t0, \$a0, \$a1	Procedure body
add \$t1, \$a2, \$a3	
sub \$s0, \$t0, \$t1	
add \$v0, \$s0, \$zero	Result
lw \$s0, 0(\$sp)	Restore \$s0
addi \$sp, \$sp, 4	
jr \$ra	Return

Leaf Procedure Example

- MIPS code for calling function:

```
main:
...
jal leaf_example
...
```

Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
 - Its return address
 - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Non-Leaf Procedure Example

- C code:

```
int fact (int n)
{
    if (n < 1) return 1;
    else return n * fact(n - 1);
}
```

- Argument n in \$a0
- Result in \$v0
- slt and slti instructions – compare two registers and sets a third register to 1

Examples: slt \$t0, \$s3, \$s4 # \$t0=1 if \$s3<\$s4

slti \$t0, \$s2, 10 # \$t0=1 if \$s2<10

Non-Leaf Procedure Example

- MIPS code:

fact:		
addi \$sp, \$sp, -8	# adjust stack for 2 items	
sw \$ra, 4(\$sp)	# save return address	
sw \$a0, 0(\$sp)	# save argument	
slti \$t0, \$a0, 1	# \$t0=1 if \$a0 < 1 (n<1)	
beq \$t0, \$zero, L1	# jump to L1 if \$t0=0(n>=1)	
addi \$v0, \$zero, 1	# if so, result is 1	
addi \$sp, \$sp, 8	# pop 2 items from stack*	
jr \$ra	# and return	
L1: addi \$a0, \$a0, -1	# else decrement n	
jal fact	# recursive call	
lw \$a0, 0(\$sp)	# restore original n	
lw \$ra, 4(\$sp)	# and return address	
addi \$sp, \$sp, 8	# pop 2 items from stack	
mul \$v0, \$a0, \$v0	# multiply to get result	
jr \$ra	# and return	

Note: \$a0 & \$ra do not change if n<1, so \$a0 & \$ra are not loaded before pop them

Non-Leaf Procedure Example: n=4 (1)

**Main call (\$a0)₁=4,
(\$ra)₁=return addr in main**

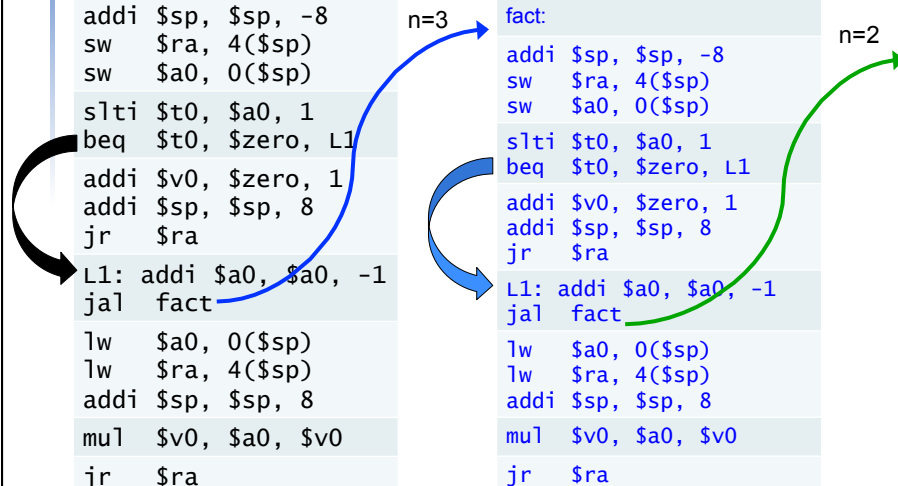
fact:

```
addi $sp, $sp, -8
sw   $ra, 4($sp)
sw   $a0, 0($sp)
slti $t0, $a0, 1
beq  $t0, $zero, L1
addi $v0, $zero, 1
addi $sp, $sp, 8
jr   $ra
L1: addi $a0, $a0, -1
jal  fact
lw   $a0, 0($sp)
lw   $ra, 4($sp)
addi $sp, $sp, 8
mul  $v0, $a0, $v0
jr   $ra
```

**fact1 call (\$a0)₂=3
(\$ra)₂=return addr in fact1**

fact:

```
addi $sp, $sp, -8
sw   $ra, 4($sp)
sw   $a0, 0($sp)
slti $t0, $a0, 1
beq  $t0, $zero, L1
addi $v0, $zero, 1
addi $sp, $sp, 8
jr   $ra
L1: addi $a0, $a0, -1
jal  fact
lw   $a0, 0($sp)
lw   $ra, 4($sp)
addi $sp, $sp, 8
mul  $v0, $a0, $v0
jr   $ra
```



Non-Leaf Procedure Example: n=4 (2)

**fact2 call (\$a0)₃=2,
(\$ra)₃=return addr in fact2**

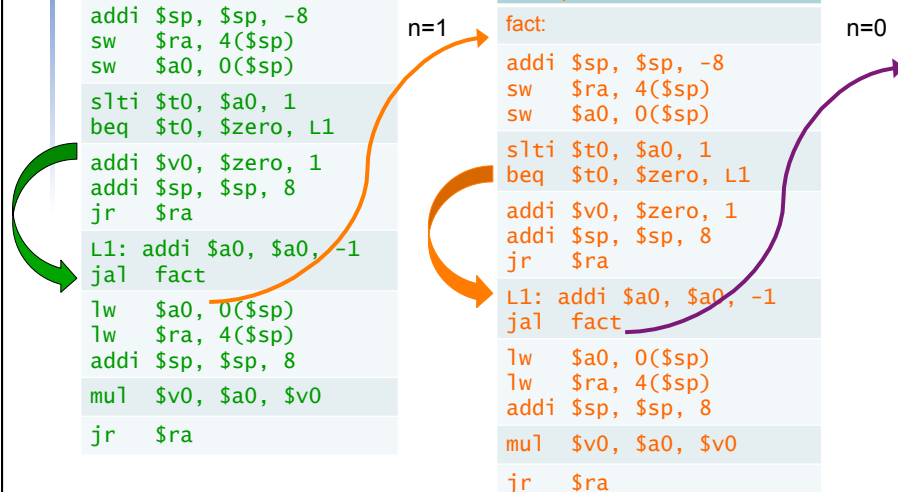
fact:

```
addi $sp, $sp, -8
sw   $ra, 4($sp)
sw   $a0, 0($sp)
slti $t0, $a0, 1
beq  $t0, $zero, L1
addi $v0, $zero, 1
addi $sp, $sp, 8
jr   $ra
L1: addi $a0, $a0, -1
jal  fact
lw   $a0, 0($sp)
lw   $ra, 4($sp)
addi $sp, $sp, 8
mul  $v0, $a0, $v0
jr   $ra
```

**fact3 call (\$a0)₄=1
(\$ra)₄=return addr in fact3**

fact:

```
addi $sp, $sp, -8
sw   $ra, 4($sp)
sw   $a0, 0($sp)
slti $t0, $a0, 1
beq  $t0, $zero, L1
addi $v0, $zero, 1
addi $sp, $sp, 8
jr   $ra
L1: addi $a0, $a0, -1
jal  fact
lw   $a0, 0($sp)
lw   $ra, 4($sp)
addi $sp, $sp, 8
mul  $v0, $a0, $v0
jr   $ra
```



Non-Leaf Procedure Example: n=4 (3)

fact4 call (\$a0)₄=0,
(\$ra)₅=return addr in fact4

fact:

```
addi $sp, $sp, -8
sw   $ra, 4($sp)
sw   $a0, 0($sp)
slti $t0, $a0, 1
beq  $t0, $zero, L1
addi $v0, $zero, 1
addi $sp, $sp, 8
jr   $ra
L1: addi $a0, $a0, -1
jal  fact
lw   $a0, 0($sp)
lw   $ra, 4($sp)
addi $sp, $sp, 8
mul  $v0, $a0, $v0
jr   $ra
```

fact3 call (\$a0)₄=1
(\$ra)₄=return addr in fact3

fact:

```
addi $sp, $sp, -8
sw   $ra, 4($sp)
sw   $a0, 0($sp)
slti $t0, $a0, 1
beq  $t0, $zero, L1
addi $v0, $zero, 1
addi $sp, $sp, 8
jr   $ra
L1: addi $a0, $a0, -1
jal  fact
lw   $a0, 0($sp)
lw   $ra, 4($sp)
addi $sp, $sp, 8
mul  $v0, $a0, $v0
jr   $ra
```

$\$v0=1$

$\$v0=1*($a0)_4$
 $=1$

Non-Leaf Procedure Example: n=4 (2)

fact2 call (\$a0)₃=2,
(\$ra)₃=return addr in fact2

fact:

```
addi $sp, $sp, -8
sw   $ra, 4($sp)
sw   $a0, 0($sp)
slti $t0, $a0, 1
beq  $t0, $zero, L1
addi $v0, $zero, 1
addi $sp, $sp, 8
jr   $ra
L1: addi $a0, $a0, -1
jal  fact
lw   $a0, 0($sp)
lw   $ra, 4($sp)
addi $sp, $sp, 8
mul  $v0, $a0, $v0
jr   $ra
```

$\$v0=1$

$\$v0=1*($a0)_3$
 $=2$

fact1 call (\$a0)₂=3
(\$ra)₂=return addr in fact1

fact:

```
addi $sp, $sp, -8
sw   $ra, 4($sp)
sw   $a0, 0($sp)
slti $t0, $a0, 1
beq  $t0, $zero, L1
addi $v0, $zero, 1
addi $sp, $sp, 8
jr   $ra
L1: addi $a0, $a0, -1
jal  fact
lw   $a0, 0($sp)
lw   $ra, 4($sp)
addi $sp, $sp, 8
mul  $v0, $a0, $v0
jr   $ra
```

$\$v0=2*($a0)_2$
 $=6$

Non-Leaf Procedure Example: n=4 (1)

**Main call (\$a0)₁=4,
(\$ra)₁=return addr in main**
 fact:
 addi \$sp, \$sp, -8
 sw \$ra, 4(\$sp)
 sw \$a0, 0(\$sp)
 slti \$t0, \$a0, 1
 beq \$t0, \$zero, L1
 addi \$v0, \$zero, 1
 addi \$sp, \$sp, 8
 jr \$ra
 L1: addi \$a0, \$a0, -1
 jal fact
 lw \$a0, 0(\$sp)
 lw \$ra, 4(\$sp)
 addi \$sp, \$sp, 8
 mul \$v0, \$a0, \$v0 # \$v0 = 6 * (\$a0)₁ = 24
 jr \$ra

\$v0=4

Register Summary

- The following registers are preserved on call
 - \$s0-\$s7, \$gp, \$sp, \$fp, and \$ra

Register Number	Mnemonic Name	Conventional Use	Register Number	Mnemonic Name	Conventional Use
\$0	zero	Permanently 0	\$24, \$25	\$t8, \$t9	Temporary
\$1	\$at	Assembler Temporary (reserved)	\$26, \$27	\$k0, \$k1	Kernel (reserved for OS)
\$2, \$3	\$v0, \$v1	Value returned by a subroutine	\$28	\$gp	Global Pointer
\$4-\$7	\$a0-\$a3	Arguments to a subroutine	\$29	\$sp	Stack Pointer
\$8-\$15	\$t0-\$t7	Temporary (not preserved across a function call)	\$30	\$fp	Frame Pointer
\$16-\$23	\$s0-\$s7	Saved registers (preserved across a function call)	\$31	\$ra	Return Address

CSE2021 Computer Organization

Communicating with People

Character Data

- Byte-encoded character sets
 - ASCII: 128 characters
 - 95 graphic, 33 control
 - Latin-1: 256 characters
 - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
 - Used in Java, C++ wide characters, ...
 - Most of the world's alphabets, plus symbols
 - UTF-8, UTF-16: variable-length encodings

ASCII Representation of Characters

Dec	Hex	Name	Char	Ctrl-char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0		Null	NUL	CTRL-@	32	20	Space	64	40	@	96	60	`
1		Start of heading	SOH	CTRL-A	33	21	!	65	41	A	97	61	a
2		Start of text	STX	CTRL-B	34	22	"	66	42	B	98	62	b
3		End of text	ETX	CTRL-C	35	23	#	67	43	C	99	63	c
4		End of xmit	EOT	CTRL-D	36	24	\$	68	44	D	100	64	d
5		Enquiry	ENQ	CTRL-E	37	25	%	69	45	E	101	65	e
6		Acknowledge	ACK	CTRL-F	38	26	&	70	46	F	102	66	f
7		Bell	BEL	CTRL-G	39	27	'	71	47	G	103	67	g
8		Backspace	BS	CTRL-H	40	28	(	72	48	H	104	68	h
9		Horizontal tab	HT	CTRL-I	41	29	)	73	49	I	105	69	i
10	0A	Line feed	LF	CTRL-J	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	VT	CTRL-K	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	FF	CTRL-L	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage feed	CR	CTRL-M	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	SO	CTRL-N	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	SI	CTRL-O	47	2F	/	79	4F	O	111	6F	o
16	10	Data line escape	DLE	CTRL-P	48	30	0	80	50	P	112	70	p
17	11	Device control 1	DC1	CTRL-Q	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	DC2	CTRL-R	50	32	2	82	52	R	114	72	r
19	13	Device control 3	DC3	CTRL-S	51	33	3	83	53	S	115	73	s
20	14	Device control 4	DC4	CTRL-T	52	34	4	84	54	T	116	74	t
21	15	Neg acknowledge	NAK	CTRL-U	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	SYN	CTRL-V	54	36	6	86	56	V	118	76	v
23	17	End of xmit block	ETB	CTRL-W	55	37	7	87	57	W	119	77	w
24	18	Cancel	CAN	CTRL-X	56	38	8	88	58	X	120	78	x
25	19	End of medium	EM	CTRL-Y	57	39	9	89	59	Y	121	79	y
26	1A	Substitute	SUB	CTRL-Z	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	ESC	CTRL-[	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	FS	CTRL-\	60	3C	<	92	5C	\	124	7C	
29	1D	Group separator	GS	CTRL-]	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	RS	CTRL-^	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	US	CTRL-~	63	3F	?	95	5F	_	127	7F	DEL

ASCII Characters

- American Standard Code for Information Interchange (ASCII).
- Most computers use 8-bit to represent each character. (Java uses Unicode, which is 16-bit).
- Strings are combination of characters.
- How to load a byte?
 - lb, lbu, sb for byte (ASCII)
 - lh, lhu, sh for half-word instruction (Unicode)

Byte/Halfword Operations

- Could use bitwise operations
- MIPS byte/halfword load/store
 - String processing is a common case
- `lb rt, offset(rs)` `lh rt, offset(rs)`
 - Sign extend to 32 bits in `rt`
- `lbu rt, offset(rs)` `lhu rt, offset(rs)`
 - `lbu` – load byte unsigned
 - Zero extend to 32 bits in `rt`
- `sb rt, offset(rs)` `sh rt, offset(rs)`
 - Store just rightmost byte/halfword

String Copy Example

- C code (copy string `y` to string `x`)
 - Null-terminated string

```
void strcpy (char x[], char y[])
{ int i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

 - Addresses of `x`, `y` in `$a0`, `$a1`
 - `i` in `$s0`

String Copy Example

- MIPS code:

strcpy:		
addi	\$sp, \$sp, -4	# adjust stack for 1 item
sw	\$s0, 0(\$sp)	# save \$s0
add	\$s0, \$zero, \$zero	# i = 0
L1:	add \$t1, \$s0, \$a1	# addr of y[i] in \$t1
	lbu \$t2, 0(\$t1)	# \$t2 = y[i]
	add \$t3, \$s0, \$a0	# addr of x[i] in \$t3
	sb \$t2, 0(\$t3)	# x[i] = y[i]
	beq \$t2, \$zero, L2	# exit loop if y[i] == 0
	addi \$s0, \$s0, 1	# i = i + 1
	j L1	# next iteration of loop
L2:	lw \$s0, 0(\$sp)	# restore saved \$s0
	addi \$sp, \$sp, 4	# pop 1 item from stack
	jr \$ra	# and return

32-bit Constants

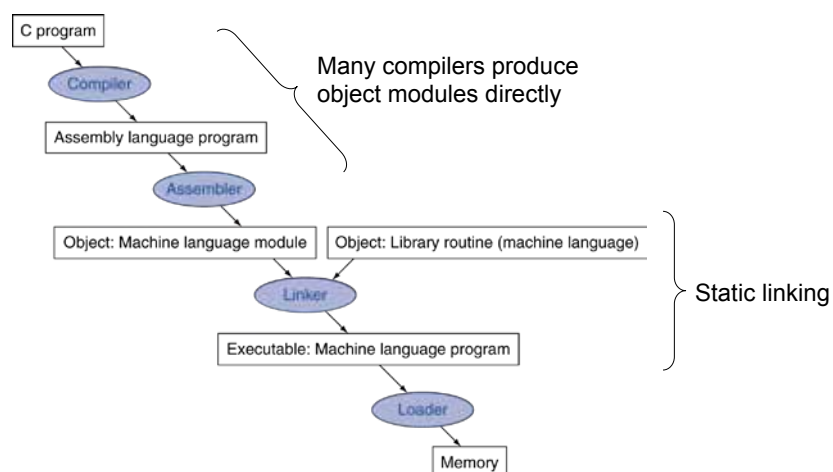
- Most constants are small
 - 16-bit immediate is sufficient
- For the occasional 32-bit constant
 - Use lui (load upper immediate)
 - lui rt, constant
 - Copies 16-bit constant to left 16 bits of rt
 - Clears right 16 bits of rt to 0

lui \$s0, 61	0000 0000 0111 1101	0000 0000 0000 0000
ori \$s0, \$s0, 2304	0000 0000 0111 1101	0000 1001 0000 0000

CSE2021 Computer Organization

Translating and Starting a Program

Translation Hierarchy for C



Translation

- Assembler (or compiler) translates program into machine instructions
- Linker produces an executable image
- Loader load from image file on disk into memory

CSE2021 Computer Organization

SPIM Simulator

SPIM Simulator

- SPIM is a software simulator that runs assembly language programs
- SPIM is just MIPS spelled backwards
- SPIM can read and immediately execute assembly language files
- Two versions for different machines
 - Unix: xspim(used in lab), spim
 - PC/Mac: QtSpim
- Resources and Download
 - <http://spimsimulator.sourceforge.net>

System Calls in SPIM

- SPIM provides a small set of system-like services through the system call (syscall) instruction.
- Format for system calls
 - Place value of input argument in \$a0
 - Place value of system-call-code in \$v0
 - syscall

System Calls

Example: print a string

```
.data
    str:
        .ascii "answer is:"
.text
    addi $v0,$zero,4
    la $a0, str
    #pseudoinstruction
    syscall
```

Service	System Call Code	Arguments	Result
print_int	1	\$a0 = integer	
print_float	2	\$f12 = float	
print_double	3	\$f12 = double	
print_string	4	\$a0 = string	
read_int	5		integer (in \$v0)
read_float	6		float (in \$v0)
read_double	7		double (in \$v0)
read_string	8	\$a0 = buffer, \$a1 = length	
sbrk	9	\$a0 = amount	address (in \$v0)
exit	10		
print_character	11	\$a0 = character	
read_character	12		character (in \$v0)
open	13	\$a0 = filename, \$a1 = flags, \$a2 = mode	file descriptor (in \$v0)
read	14	\$a0 = file descriptor, \$a1 = buffer, \$a2 = count	bytes read (in \$v0)
write	15	\$a0 = file descriptor, \$a1 = buffer, \$a2 = count	bytes written (in \$v0)
close	16	\$a0 = file descriptor	0 (in \$v0)
exit2	17	\$a0 = value	

Reading

- Read Appendix A.9 for SPIM

CSE2021 Computer Organization

Other Instruction Sets

ARM & 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-bit flat	32-bit flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Integer Registers	15 × 32-bit	31 × 32-bit
Input/output	Memory mapped	Memory mapped

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...
 - AMD64 (2003): extended architecture to 64 bits
 - EM64T – Extended Memory 64 Technology (2004)
 - AMD64 adopted by Intel (with refinements)
 - Added SSE3 instructions
 - Intel Core (2006)
 - Added SSE4 instructions, virtual machine support
 - AMD64 (announced 2007): SSE5 instructions
 - Intel declined to follow, instead...
 - Advanced Vector Extension (announced 2008)
 - Longer SSE registers, more instructions
- If Intel didn't extend with compatibility, its competitors would!
 - Technical elegance ≠ market success

Concluding Remarks

- Design principles
 1. Simplicity favors regularity
 2. Smaller is faster
 3. Make the common case fast
 4. Good design demands good compromises
- Layers of software/hardware
 - Compiler, assembler, hardware
- MIPS: typical of RISC ISAs
 - c.f. x86

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, or, nor, andi, ori, sll, srl	12%	4%
Cond. Branch	beq, bne, slt, slti, sltiu	34%	8%
Jump	j, jr, jal	2%	0%

Acknowledgement

- The slides are adapted from Computer Organization and Design, 5th Edition, by David A. Patterson and John L. Hennessy, 2013, published by MK (Elsevier)