Lecture 5

Instruction Set Architecture(3)

Today's Topic

- Recap:
  - More control instructions
  - Procedure call
- Today's topic:
  - MIPS addressing
  - Translating and starting a program
  - a C sort example
  - Other popular ISAs

MIPS Addressing

- Addressing: how the instructions identify the operands of the instruction.
- MIPS Addressing mode:
  - Immediate addressing addi $s0, $s1, 5
  - Register addressing add $s0, $s1, $s2
  - Base/Displacement addressing lw $s0, 0($s1)
  - PC-relative addressing bne $s0, $s1, EXIT
  - Pseudo-direct addressing j EXIT

Immediate Addressing

- For instructions including immediate
  - E.g. addi, subi, andi, ori

op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

- Most constants are small
  - 16-bit immediate is sufficient

addi \$s0, \$a0,

- For the occasional 32-bit constant

  lui rt, constant

  - Copies 16-bit constant to left 16 bits of rt
  - Clears right 16 bits of rt to 0

lui \$s0,

lui \$s0, 61	0000 0000 0011 1101	0000 0000 0000 0000
ori \$s0, \$s0, 2304	0000 0000 0011 1101	0000 1001 0000 0000

Register Addressing

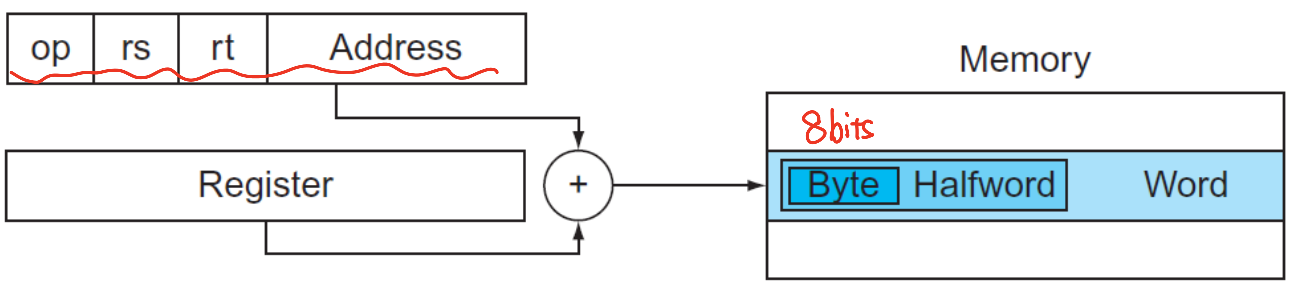
- Using register as the operand
- E.g. add, addi, sub, subi, lw, …

op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

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

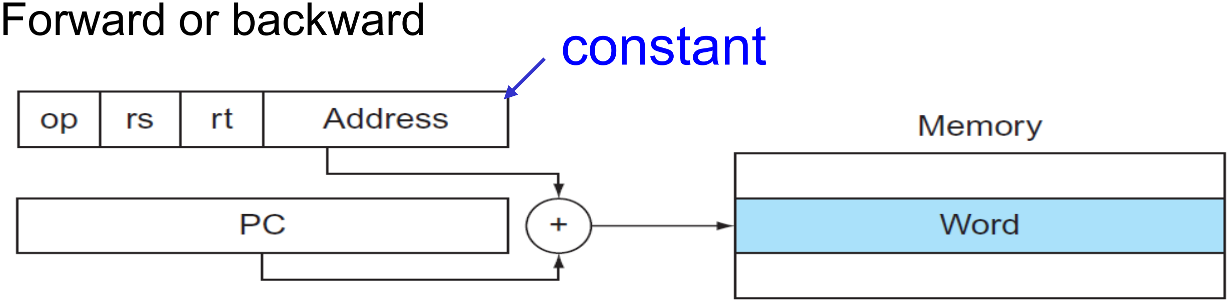
Base/displacement Addressing

- the operand is at the memory, whose address is the sum of a register and a constant lw/lh/lb/sw/sh/sb
  - e.g. lw $s0, 4($s1)



Branch Addressing (PC-relative addressing)

- Branch instructions specify
  - Opcode, two registers, target address
  - e.g. beq $s0 $s1 label
- Most branch targets are near branch
  - Forward or backward



- PC-relative addressing
  - Target address = PC + constant × 4
  - PC already incremented by 4 by this time

PC+4 constant x4

Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

Loop

00010

Code				Address						
Loop:	sll	$t1, $s3, 2		80000	0	0	19	9	4	0
	add	$t1, $t1, $s6		80004	0	9	22	9	0	32
	lw	$t0, 0($t1)		80008	35	9	8	0		
	bne	$t0, $s5, Exit		80012	5	8	21	2		
	addi	$s3, $s3, 1		80016	8	19	19	1		
	j	Loop		80020	2	20000				
Exit:	…			80024						

half word

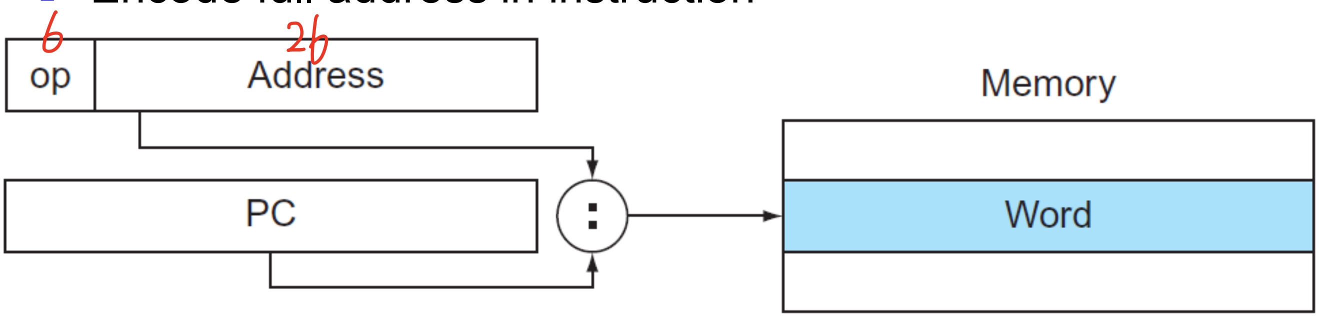
0X - - - - - - - -

byte

word

Jump Addressing (Pseudo-direct addressing)

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



- (Pseudo)Direct jump addressing
  - Target address = $\underbrace{PC31\ldots28}_{4}$: (address × 4)

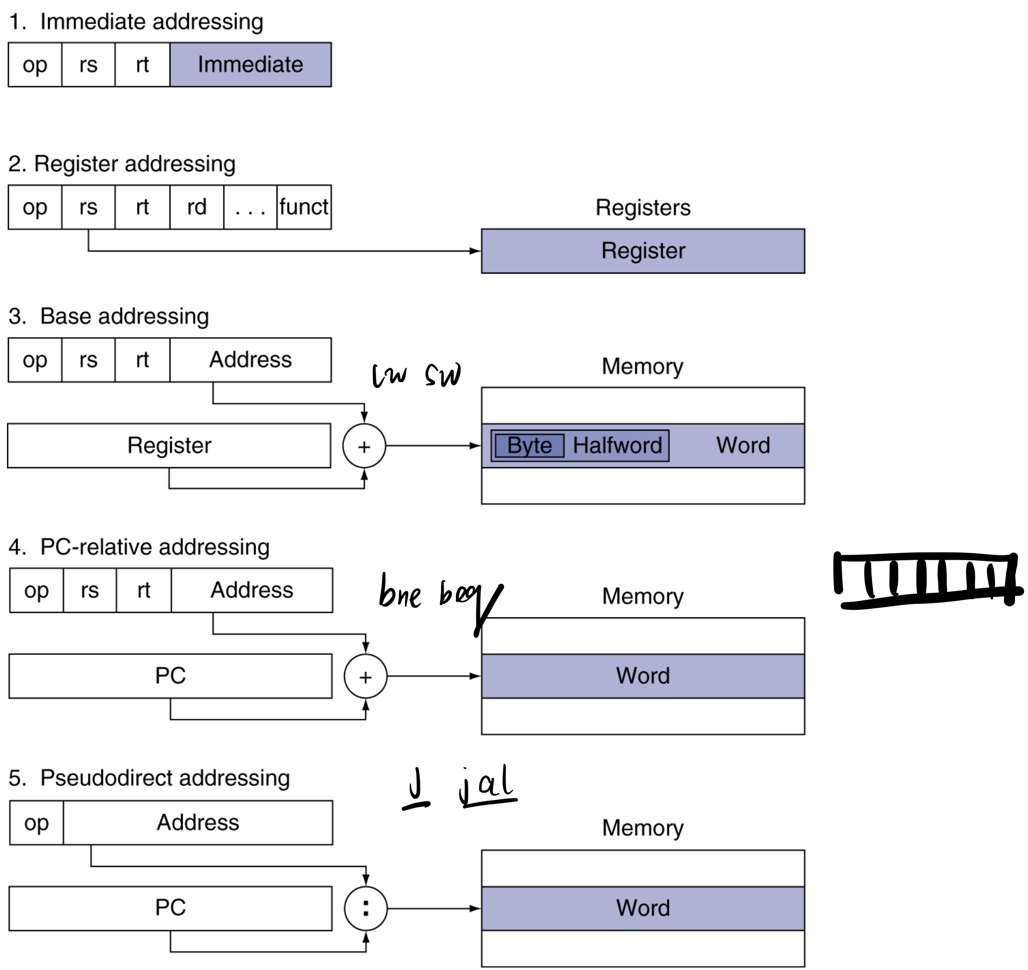
$\boxed{\text{datatype}}$

Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

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

Addressing Mode Summary



There is no "direct addressing"

```
.data
        str: .asciiz "the answer = "
.text
main:
        li $v0, 4
        la $a0, str
        syscall
        lb $t0, ($a0)

        li $v0, 10
        syscall
```

We can only use lw/sw to visit the memory

load upper immediate.

ori/lui is immediate addressing
lb is base/displacement addressing

Address	Code	Basic		
0x00400000	0x24020004	addiu $2, $0, 0x00000004	5:	li $v0, 4
0x00400004	0x3c011001	lui $1, 0x00001001	6:	la $a0, str
0x00400008	0x34240000	ori $4, $1, 0x00000000		
0x0040000c	0x0000000c	syscall	7:	syscall
0x00400010	0x80880000	lb $8, 0x00000000($4)	8:	lb $t0, ($a0)
0x00400014	0x2402000a	addiu $2, $0, 0x0000000a	10:	li $v0, 10
0x00400018	0x0000000c	syscall	11:	syscall

immediate addressing

N...	Nu...
$...	0
$at	1
$v0	2
$v1	3
$a0	4

Decoding Machine Language

- What is the assembly language of the following machine instruction?

10

0x00af8020

- Hex to bin: 0000 0000 1010 1111 1000 0000 0010 0000

op	rs	rt	rd	shamt	funct
000000	00101	01111	10000	00000	100000

- Get the instruction: add $s0,$a1,$t7

32.

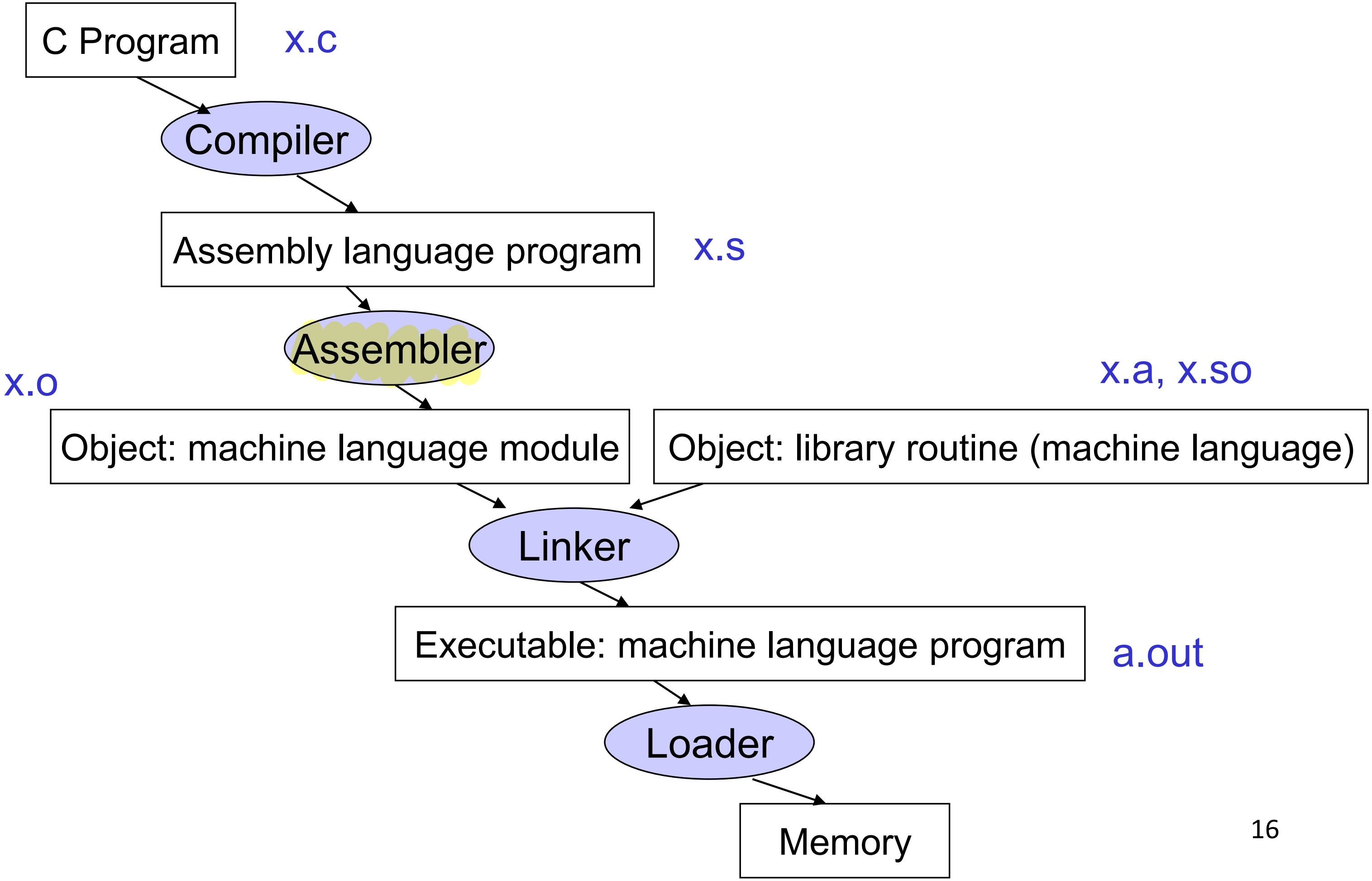
Decoding Machine Language

op(31:26)								
28–26 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	set less than imm. unsigned	andi	ori	xori	load upper immediate
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	load half	lwl	load word	load byte unsigned	load half unsigned	lwr	
5(101)	store byte	store half	swl	store word			swr	
6(110)	load linked word	lwc1						
7(111)	store cond. word	swc1						

Decoding Machine Language

op(31:26)=000000 (R-format), funct(5:0)								
2–0 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump register	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set l.t.	set l.t. unsigned				
6(110)								
7(111)								

Starting a C Program



Role of Assembler

- Convert pseudo-instructions into actual hardware instructions – pseudo-instrs make it easier to program in assembly – examples: “move”, “blt”, 32-bit immediate operands, etc.

- Convert assembly instrs into machine instrs – a separate object file (x.o) is created for each C file (x.c) – compute the actual values for instruction labels – maintain info on external references and debugging information

Role of Linker

- Stitches different object files into a single executable
    - patch internal and external references
    - determine addresses of data and instruction labels
    - organize code and data modules in memory
- Some libraries (DLLs) are dynamically linked – the executable points to dummy routines – these dummy routines call the dynamic linker-loader so they can update the executable to jump to the correct routine

Role of Linker

Object file 1:

Object file header			
	Name	Procedure A	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	Instruction	
	0	lw $a0, 0($gp)	
	4	jal 0	
	...	...	
Data segment	0	(X)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	lw	X
	4	jal	B
Symbol table	Label	Address	
	X	–	
	B	–	

Role of Linker

Object file 2:

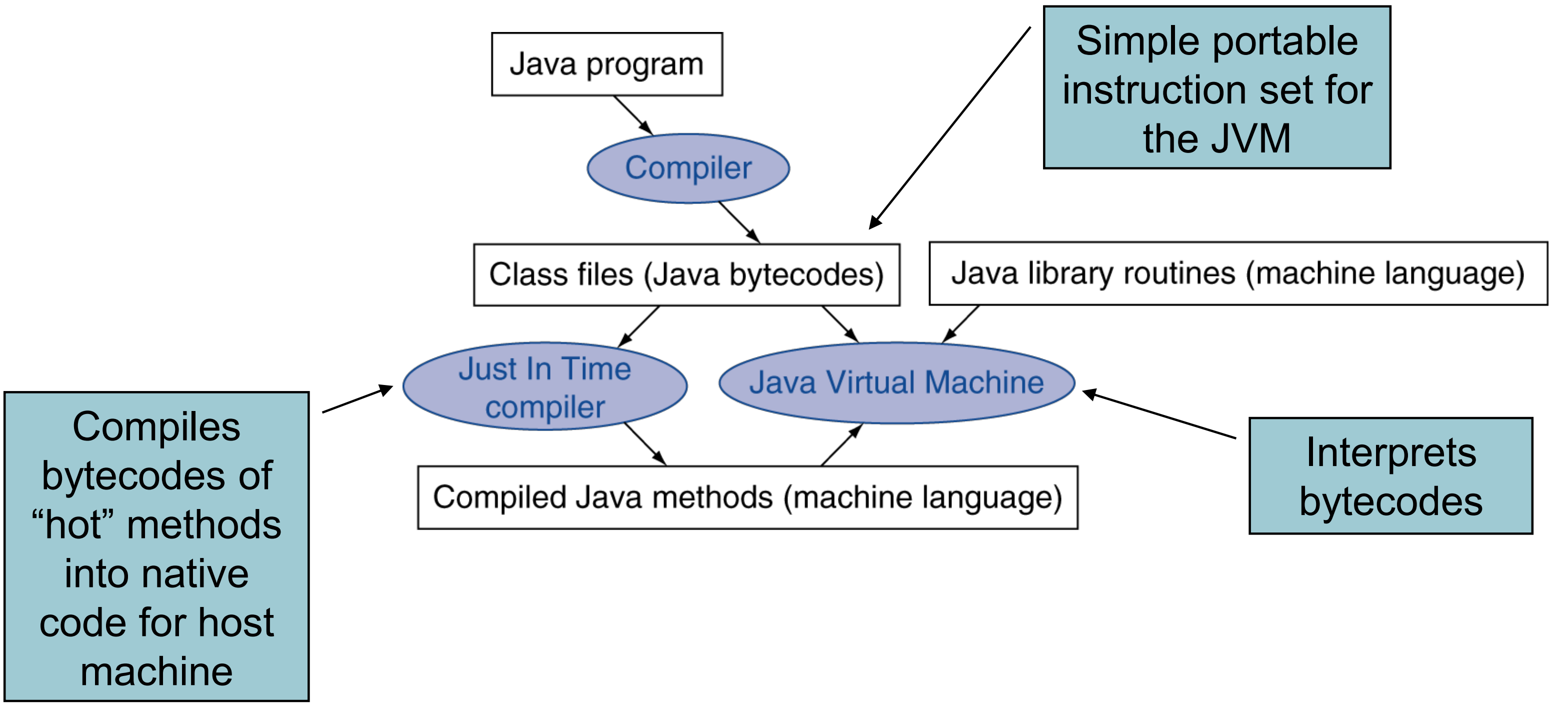
Object file header			
	Name	Procedure B	
	Text size	200_{hex}	
	Data size	30_{hex}	
Text segment	Address	Instruction	
	0	`sw $a1, 0($gp)`	
	4	`jal 0`	
	…	…	
Data segment	0	`(Y)`	
	…	…	
Relocation information	Address	Instruction type	Dependency
	0	`sw`	`Y`
	4	`jal`	`A`
Symbol table	Label	Address	
	`Y`	–	
	`A`	–	

Role of Linker

Executable file:

Executable file header		
	Text size	300_{hex}
	Data size	50_{hex}
Text segment	Address	Instruction
	$0040\ 0000_{hex}$	`lw $a0,` 8000_{hex}`($gp)`
	$0040\ 0004_{hex}$	`jal` $40\ 0100_{hex}$
	...	...
	$0040\ 0100_{hex}$	`sw $a1,` 8020_{hex}`($gp)`
	$0040\ 0104_{hex}$	`jal` $40\ 0000_{hex}$
	...	...
Data segment	Address	
	$1000\ 0000_{hex}$	(X)
	...	...
	$1000\ 0020_{hex}$	(Y)
	...	...

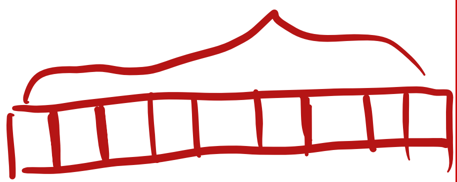
Starting Java Applications



Full Example – Sort in C (pg. 133)

```
void sort (int v[], int n)
{
    int i, j;
    for (i=0; i<n; i+=1) {
        for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) {
            swap (v,j);
        }
    }
}
```

insertion



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

- Allocate registers to program variables
- Produce code for the program body
- Preserve registers across procedure invocations

The swap Procedure

- Register allocation: \$a0 and \$a1 for the two arguments, \$t0 for the temp variable – no need for saves and restores as we're not using \$s0-\$s7 and this is a leaf procedure (won't need to re-use \$a0 and \$a1)

```
swap:   sll    $t1, $a1, 2
        add    $t1, $a0, $t1
        lw     $t0, 0($t1)
        lw     $t2, 4($t1)
        sw     $t2, 0($t1)
        sw     $t0, 4($t1)
        jr     $ra
```

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

The sort Procedure

- Register allocation: arguments v and n use \$a0 and \$a1, i and j use \$s0 and \$s1; must save \$a0 and \$a1 before calling the leaf procedure

- The outer for loop looks like this: (note the use of pseudo-instrs)

```
            move   $s0, $zero         # initialize the loop
loopbody1:  bge    $s0, $a1, exit1    # will eventually use slt and beq
            … body of inner loop …
            addi   $s0, $s0, 1
            j      loopbody1
exit1:
```

```
for (i=0; i<n; i+=1) {
  for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) {
        swap (v,j);
  }
}
```

The sort Procedure

- The inner for loop looks like this:

```
            addi    $s1, $s0, -1          # initialize the loop
loopbody2:  blt     $s1, $zero, exit2     # will eventually use slt and beq
            sll     $t1,  $s1, 2
            add     $t2, $a0, $t1
            lw      $t3, 0($t2)
            lw      $t4, 4($t2)
            bgt     $t3, $t4, exit2
            … body of inner loop …
            addi    $s1, $s1, -1
            j       loopbody2
exit2:
```

```
for (i=0; i<n; i+=1) {
  for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) {
       swap (v,j);
  }
}
```

Saves and Restores

- Since we repeatedly call “swap” with \$a0 and \$a1, we begin “sort” by copying its arguments into \$s2 and \$s3 – must update the rest of the code in “sort” to use \$s2 and \$s3 instead of \$a0 and \$a1

- Must save \$ra at the start of “sort” because it will get over-written when we call “swap”

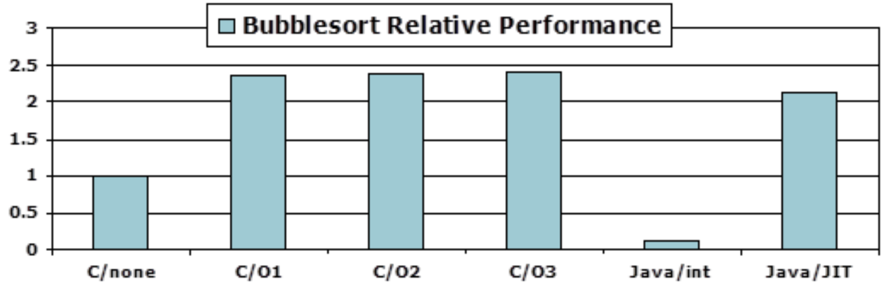
- Must also save \$s0-\$s3 so we don’t overwrite something that belongs to the procedure that called “sort”

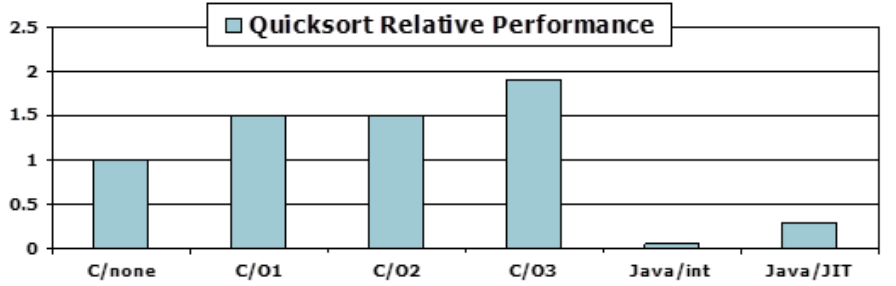
Saves and Restores

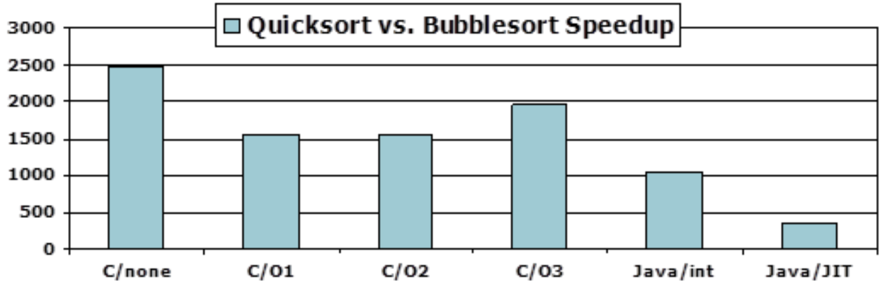
```
sort:   addi   $sp, $sp, -20
        sw     $ra, 16($sp)
        sw     $s3, 12($sp)
        sw     $s2, 8($sp)
        sw     $s1, 4($sp)
        sw     $s0, 0($sp)
        move   $s2, $a0
        move   $s3, $a1
          …
        move   $a0, $s2        # the inner loop body starts here
        move   $a1, $s1
        jal    swap
          …
exit1:  lw     $s0, 0($sp)
          …
        addi   $sp, $sp, 20
        jr     $ra
```

9 lines of C code → 35 lines of assembly

Effect of Language and Algorithm







Lessons Learnt

- Instruction count and CPI are not good performance indicators in isolation
- Compiler optimizations are sensitive to the algorithm
- Java/JIT compiled code is significantly faster than JVM interpreted
  - Comparable to optimized C in some cases
- Nothing can fix a dumb algorithm!

Other ISAs

- ARM
- x86

ARM Market share

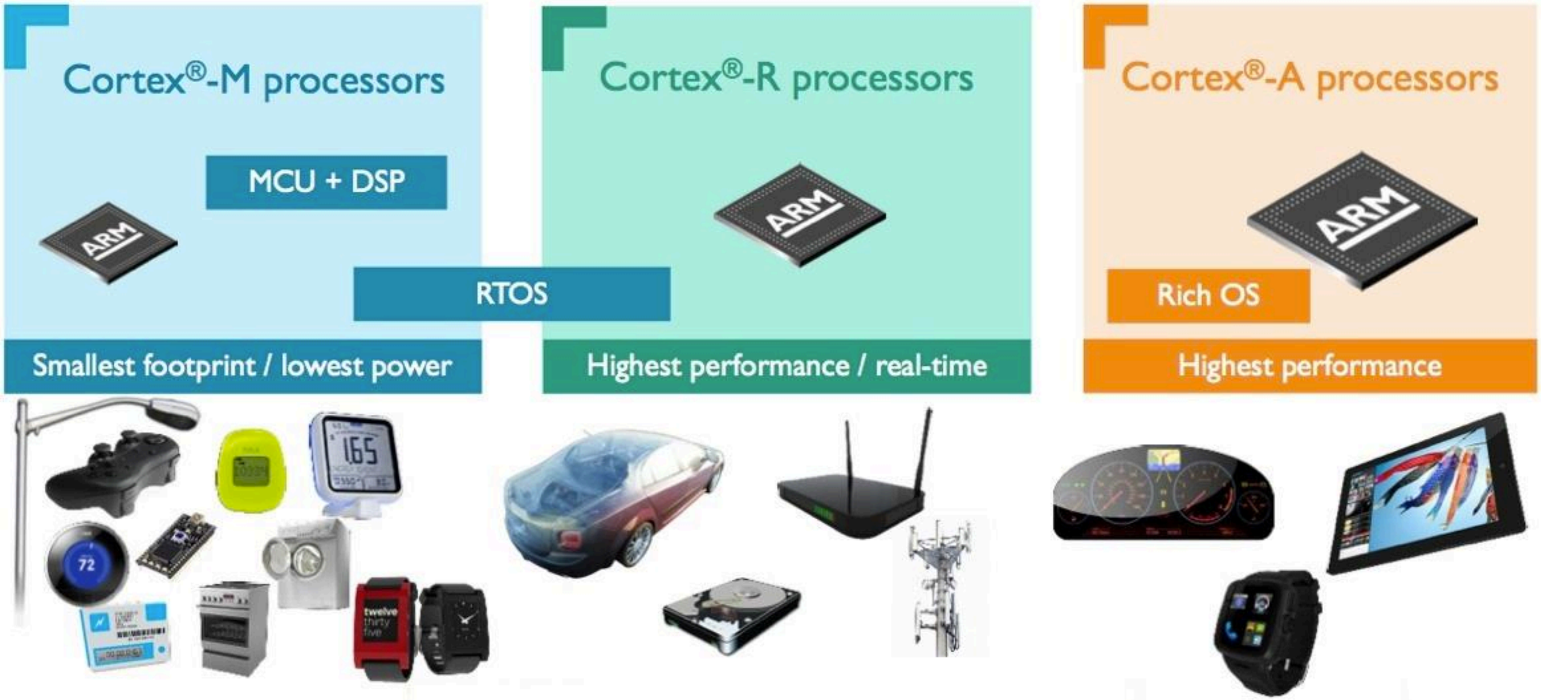
Markets for ARM in 2012

	Devices Shipped (Million of Units)	2012 Devices	Chips/ Device	TAM 2012 Chips	2012 ARM	2012 Share
Mobile	Smart Phone	730	3-5	2,500	2,200	90%
Mobile	Feature Phone	460	2-3	1,200	1,100	95%
Mobile	Low End Voice	730	1-2	730	700	95%
Mobile	Portable Media Players	130	1-3	250	220	90%
Mobile	Mobile Computing* (apps only)	400	1	400	160	40%
Home	Digital Camera	150	1-2	230	180	80%
Home	Digital TV & Set-top-box	420	1-2	640	290	45%
Enterprise	Desktop PCs & Servers (apps)	200	1	200	-	0%
Enterprise	Networking	1,200	1-2	1,300	420	35%
Enterprise	Printers	120	1	120	85	70%
Enterprise	Hard Disk & Solid State Drives	700	1	700	620	90%
Embedded	Automotive	2,600	1	2,600	210	8%
Embedded	Smart Card	6,000	1	6,000	710	13%
Embedded	Microcontrollers	8,700	1	8,700	1,500	18%
Embedded	Others **	2,000	1	2,000	300	15%
	Total	**25,500**		**27,000**	**8,700**	**32%**

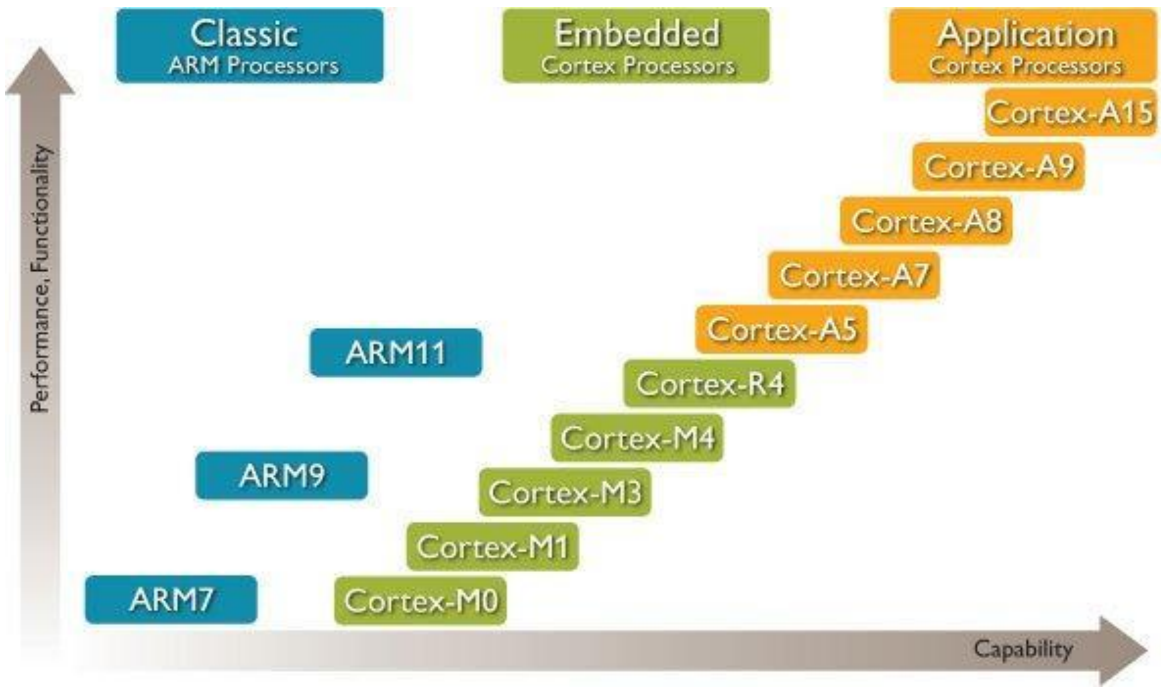
Year	Market Share
2007	17%
2008	20%
2009	22%
2010	25%
2011	29%
2012	32%

Source:
Gartner, IDC, SIA, and ARM estimates

ARM Applications



ARM CPU Series



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
Registers	15 $\times$ 32-bit	31 $\times$ 32-bit
Input/output	Memory mapped	Memory mapped

ARM v8 Instructions

- In moving to 64-bit, ARM did a complete overhaul
- ARM v8 resembles MIPS
  - Changes from v7:
    - No conditional execution field
    - Immediate field is 12-bit constant
    - Dropped load/store multiple
    - PC is no longer a GPR
    - GPR set expanded to 32
    - Addressing modes work for all word sizes
    - Divide instruction
    - Branch if equal/branch if not equal instructions

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

Basic x86 Registers

Name	31 … 0	Use
EAX		GPR 0
ECX		GPR 1
EDX		GPR 2
EBX		GPR 3
ESP		GPR 4
EBP		GPR 5
ESI		GPR 6
EDI		GPR 7
CS		Code segment pointer
SS		Stack segment pointer (top of stack)
DS		Data segment pointer 0
ES		Data segment pointer 1
FS		Data segment pointer 2
GS		Data segment pointer 3
EIP		Instruction pointer (PC)
EFLAGS		Condition codes

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