معماری کامپیوتر (۱۳-۱۱-۵۱۳)
جلسه‌ی ششم

دانشگاه شهید بهشتی

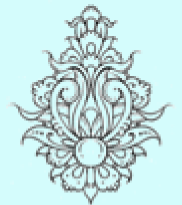
دانشکده‌ی مهندسی برق و کامپیوتر

زمستان ۱۳۹۰

احمد محمودی ازناوه

فهرست مطالب

- پردازش موازی، همگاه‌سازی و نقش سخت‌افزار
- نقش کامپایلر در کارایی
  - اشاره‌گر و آرایه
- نگاهی به دو نوع معماری
- مدارهای حسابی



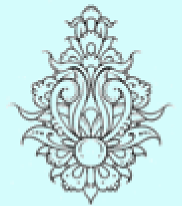
Synchronization

پردازش موازی و همگاه‌سازی

- در باره‌ی مزایای پردازش موازی، سخن‌های فروانی گفته شد، در این بخش به یکی از چالش‌های فراروی اجرای موازی فرآیندها خواهیم پرداخت.
- دو فرآیند را تصور کنید که بخشی از فضای حافظه را به صورت مشترک مورد استفاده قرار می‌دهند.
  - P1 می‌نویسد، P2 می‌خواند.
  - P1 و P2 باید هماهنگ باشند، در غیر این صورت «رقابت داده» پیش خواهد آمد.

Data race

- پاسخ به ترتیب دسترسی دو فرآیند به حافظه وابسته است.



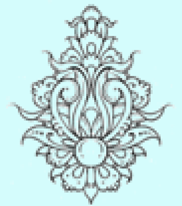
نه تنها در سیستم‌های چند پردازنده‌ای، بلکه در یک سیستم تک پردازنده‌ای با قابلیت multitasking نیز امکان رقابت داده وجود دارد.

پردازش موازی و همگاه‌سازی (ادامه...)

- سازوکارهای همگاه‌سازی، معمولا در لایه‌ی روال‌های کاربر انجام‌می‌پذیرد.
- اما این روال‌ها به پشتیانی دستورات سخت‌افزاری وابسته هستند.
  - یکی از دستوراتی که برای همگاه‌سازی استفاده می‌شوند، نوشتن و خواندن تجزیه‌ناپذیر است.
  - تجزیه‌ناپذیری، بدین معناست که بین این دو کار، عملیات دیگری نمی‌تواند انجام شود.
  - این دو، یک دستور انگاشته می‌شوند.

Atomic Operation

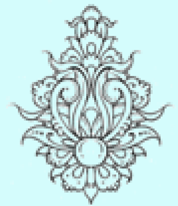
- بدون پشتیبانی سخت‌افزاری، هزینه‌ی همگاه‌سازی بسیار بالا خواهد بود و با تعداد پردازنده‌ها نیز افزایش خواهد یافت.
- چنین دستوراتی برای برنامه‌نویسان سیستم در نظر گرفته شده است.



- یکی از این دستورات جابه‌جایی تجزیه‌ناپذیر است.

Atomic Swap/exchange

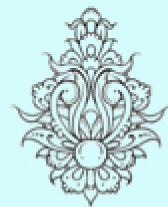
- با جابجایی تجزیه‌ناپذیر، محتوای ثبات و یک خانه‌ی حافظه به صورت تجزیه‌ناپذیر جابه‌جا خواهد شد.
- با استفاده از چنین دستوری می‌توان lock را به گونه‌ای طراحی نمود که در صورت 0 بودن به معنای آزاد بودن قفل و در غیر این صورت به معنای در دسترس نبودن آن است.



mutual exclusion (Mutex)

انحصار متقابل: الگوریتمی است که در برنامه‌نویسی همروند برای جلوگیری از استفاده از منابع مشترک.

- یک فرآیند، با استفاده از جابه‌جایی تجزیه‌ناپذیر اقدام به تغییر قفل می‌کند. در صورتی که پیش از این، فرآیند دیگری قفل را در اختیار گرفته باشد، مقدار 0 وگرنه مقدار 1 را برمی‌گرداند.
- در صورت رها بودن قفل، آن را مقداردهی کرده و مقدار 0 را باز می‌گرداند.
- بدین‌ترتیب دو فرآیند، به طور همزمان نمی‌توانند قفل را در اختیار بگیرند.
- نکته‌ای که به همگاه‌سازی کمک می‌کند، تجزیه‌ناپذیری دستور است.



چالش‌های دستورات تجزیه‌ناپذیر

- اجرای چنین دستوری، مستلزم خواندن، بررسی مقدار و در صورت نیاز نوشتن در خانه‌ی حافظه طی انجام یک دستور بی‌وقفه است.
- به جای این کار، می‌توان از دو دستور متوالی بهره جست، به گونه‌ای دو دستور بر روی هم تجزیه‌ناپذیر باشند.

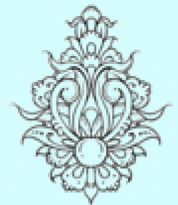
```
ll rt, offset(rs)
```

load link

```
sc rt, offset(rs)
```

store conditional

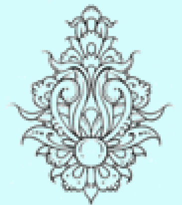
در صورتی که خانه‌ای که توسط ll خوانده شده است، تغییر نکند اجرای sc موفقیت‌آمیز انگاشته شده؛ خانه‌ی حافظه را با مقدار rt مقداردهی کرده و در ثبات مزبور مقدار 1 را قرار می‌دهد
در صورت شکست، مقدار 0 در rt قرار خواهد گرفت.



جابجایی تجزیه‌ناپذیر

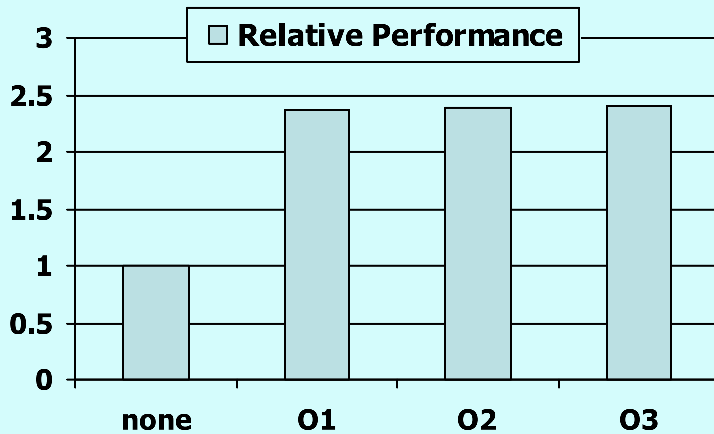
```
try: add $t0,$zero,$s4 ;copy exchange value
     ll  $t1,0($s1)    ;load linked
     sc  $t0,0($s1)    ;store conditional
     beq $t0,$zero,try ;branch store fails
     add $s4,$zero,$t1 ;put load value in $s4
```

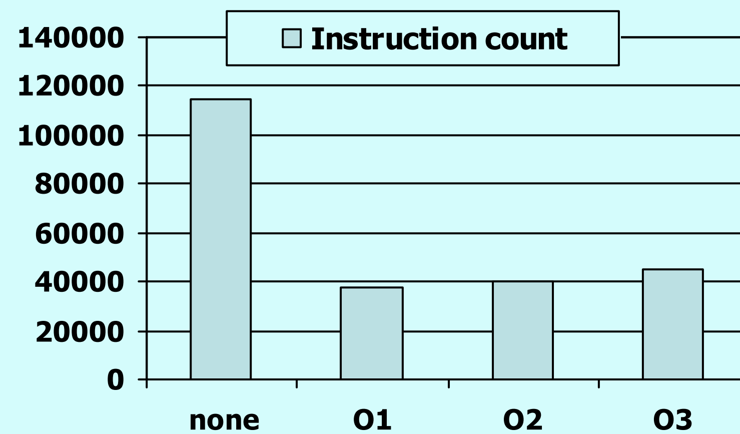
- از این دو دستور، برای پیاده‌سازی تجزیه‌ناپذیر دستورات زیر می‌توان بهره جست:
  - مقایسه و جابه‌جایی
  - واکشی و افزایش

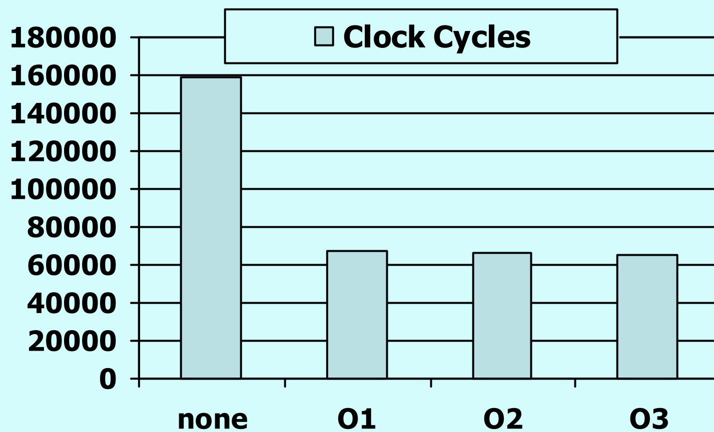


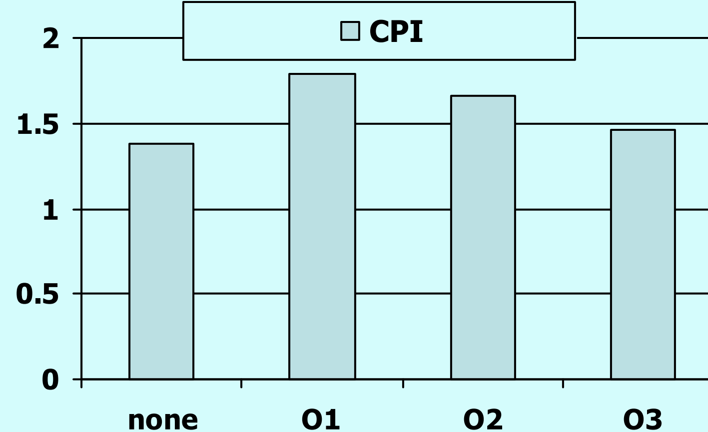
بهینه‌سازی کامپایلر

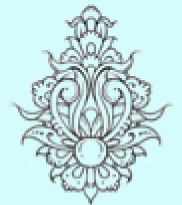
Compiled with gcc for Pentium 4 under Linux



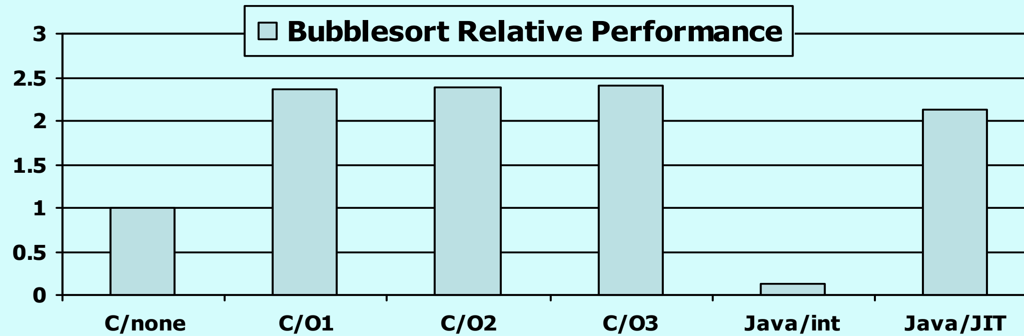


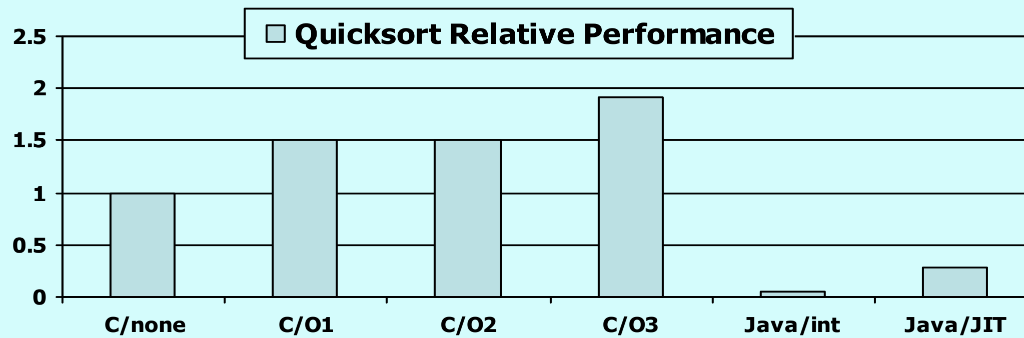


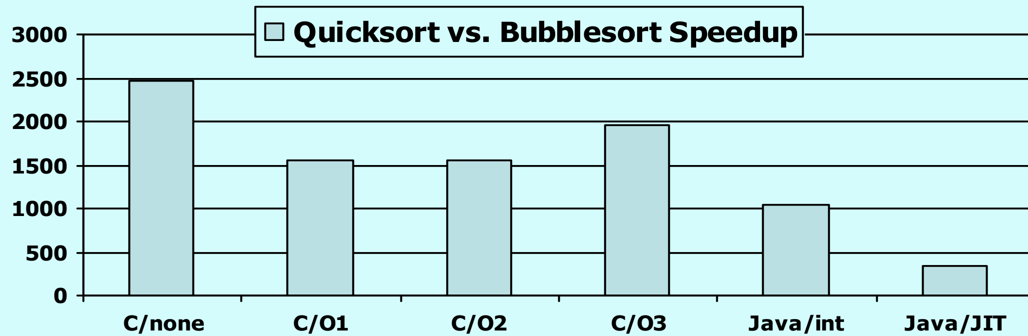


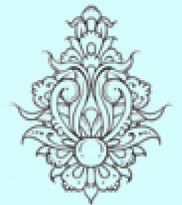


نقش زبان برنامه‌نویسی در کارایی







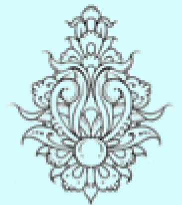


آرایه در برابر اشاره‌گر

array در $a0، size در $a1، i در قرار داده می‌شوند.

```
clear1(int
array[], int size)
{
  int i;
  for (i = 0; i <
size; i += 1)
    array[i] = 0;
}
```

```
       move $t0,$zero    # i = 0
loop1: sll $t1,$t0,2     # $t1 = i * 4
       add $t2,$a0,$t1   # $t2 =
                         #   &array[i]
       sw $zero, 0($t2)  # array[i] = 0
       addi $t0,$t0,1    # i = i + 1
       slt $t3,$t0,$a1   # $t3 =
                         #   (i < size)
       bne $t3,$zero,loop1 # if (…)
                           # goto loop1
```

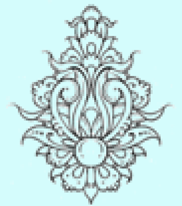


آرایه در برابر اشاره‌گر (ادامه...)

array در $a0، size در $a1، p در $t0 قرار داده می‌شوند،

```
clear2(int *array, int
size) {
  int *p;
  for (p = &array[0];
p < &array[size];
       p = p + 1)
     *p = 0;
}
```

```
       move   $t0,$a0     # p = & array[0]#
loop2: sw $zero,0($t0) # Memory[p] = 0
       addi $t0,$t0,4  # p = p + 4
       sll $t1,$a1,2   # $t1 = size * 4
       add $t2,$a0,$t1 # $t2 = &array[size]
       slt $t3,$t0,$t2 # $t3 = (p<&array[size])
       bne $t3,$zero,loop2 # if (…) goto loop2
```



آرایه در برابر اشاره‌گر (ادامه...)

<table>
<tr>
<td>

```
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

</td>
<td>

```
clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p < &array[size];
       p = p + 1)
     *p = 0;
}
```

</td>
</tr>
<tr>
<td>

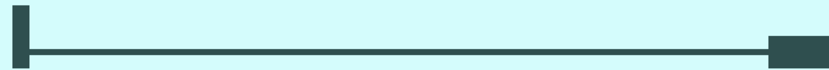
```
       move $t0,$zero   # i = 0
loop1: sll $t1,$t0,2    # $t1 = i * 4
       add $t2,$a0,$t1  # $t2 =
                        #   &array[i]
       sw $zero, 0($t2) # array[i] = 0
       addi $t0,$t0,1   # i = i + 1
       slt $t3,$t0,$a1  # $t3 =
                        #   (i < size)
       bne $t3,$zero,loop1 # if (…)
                           # goto loop1
```

</td>
<td>

```
       move $t0,$a0     # p = & array[0]
       sll $t1,$a1,2    # $t1 = size * 4
       add $t2,$a0,$t1 # $t2 =
                        #   &array[size]
loop2: sw $zero,0($t0) # Memory[p] = 0
       addi $t0,$t0,4   # p = p + 4
       slt $t3,$t0,$t2 # $t3 =
                        #(p<&array[size])
       bne $t3,$zero,loop2 # if (…)
                           # goto loop2
```

</td>
</tr>
</table>

مجموعه دستورات
ARM و x86



ARM Partnership Model

ARM پردازنده نمی سازد

ARM Powered Products

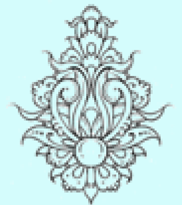
T H E A R C H I T E C T U R E F O R T H E D I G I T A L W O R L D

ARM

- پردازنده‌های ARM دو مدل دستور دارند:
  - مجموعه دستورات سی‌ودو بیتی
  - مجموعه دستورات شانزده بیتی (thumb)
- برخی هسته‌های ARM توانایی اجرای سخت‌افزاری java byte code را دارند.

Jazelle DBX (Direct Bytecode eXecution)

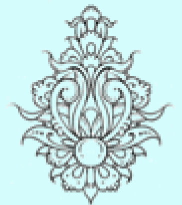
- پردازنده‌های ARM هفت اسلوب کاری دارند.



ثبات‌ها در ARM

User	FIQ	IRQ	SVC	Undef	Abort
r0	User mode r0-r7, r15, and cpsr	User mode r0-r12, r15, and cpsr	User mode r0-r12, r15, and cpsr	User mode r0-r12, r15, and cpsr	User mode r0-r12, r15, and cpsr
r1					
r2					
r3					
r4					
r5					
r6					
r7					
r8	r8				
r9	r9				
r10	r10				
r11	r11				
r12	r12				
r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)
r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)
r15 (pc)					
cpsr					
	spsr	spsr	spsr	spsr	spsr

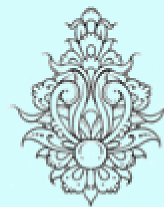
اسلوب system، از ثبات‌های کاربر استفاده می‌کند.



شباهت‌های MIPS و ARM

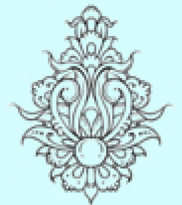
- ARM متداول‌ترین پردازنده برای سیستم‌های درون‌کار می‌باشد.

	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 × 32-bit	31 × 32-bit
Input/output	Memory mapped	Memory mapped



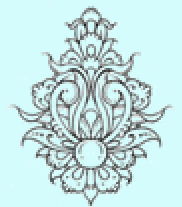
مقایسه و پرش شرطی در ARM

- در ARM از پرچم‌های وضعیت برای دستورات پرش استفاده می‌شود:
  - Negative, zero, carry, overflow
  - این پرچم‌ها در ثبات PSW ذخیره می‌شوند.
  - بعد از دستورات ریاضی و منطقی، مقدار پرچم‌ها می‌تواند تغییر کند.
- دستورات مقایسه، بدون نگهداری نتیجه مقدار پرچم‌ها را تغییر می‌دهند.



مقایسه و پرش شرطی در ARM (ادامه...)

- تمامی دستورات در ARM قابلیت اجرای مشروط را دارند. چهار بیت پرارزش دستور شرط را معین می‌کند.
- بدین‌ترتیب برای شرطی که روی یک دستور اعمال می‌شود، نیازی به دستورات شرطی نیست.
- بسته به شرط دستور به گونه‌ای خاص و یا به صورت nop اجر می‌شود.



قالب دستورها

Register-register

ARM

31–28	27–20	19–16	15–12	11–4	3–0
Opx^4	Op^8	$Rs1^4$	Rd^4	Opx^8	$Rs2^4$

MIPS

31–26	25–21	20–16	15–11	10–6	5–0
Op^6	$Rs1^5$	$Rs2^5$	Rd^5	$Const^5$	Opx^6

Data transfer

ARM

31–28	27–20	19–16	15–12	11–0
Opx^4	Op^8	$Rs1^4$	Rd^4	$Const^{12}$

MIPS

31–26	25–21	20–16	15–0
Op^6	$Rs1^5$	Rd^5	$Const^{16}$

Branch

ARM

31–28	27–24	23–0
Opx^4	Op^4	$Const^{24}$

MIPS

31–26	25–21	20–16	15–0
Op^6	$Rs1^5$	$Opx^5/Rs2^5$	$Const^{16}$

Jump/Call

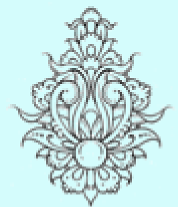
ARM

31–28	27–24	23–0
Opx^4	Op^4	$Const^{24}$

MIPS

31–26	25–0
Op^6	$Const^{26}$

■ Opcode □ Register □ Constant

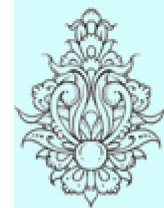


ARM

- ARM دارای دستوری است که می‌تواند گروهی از ثبات‌ها را ذخیره کند.

همچنین، محتوی ثبات دوم در دستورهای حسابی و منطقی قابلیت شیفت دادن را دارد.

Name	Definition	ARM v.4	MIPS
Load immediate	Rd = Imm	mov	addi, $0,
Not	Rd = ~(Rs1)	mvn	nor, $0,
Move	Rd = Rs1	mov	or, $0,
Rotate right	Rd = Rs i >> i $Rd_{0...i-1} = Rs_{31-i...31}$	ror	
And not	Rd = Rs1 & ~(Rs2)	bic	
Reverse subtract	Rd = Rs2 - Rs1	rsb, rsc	
Support for multiword integer add	CarryOut, Rd = Rd + Rs1 + OldCarryOut	adcs	—
Support for multiword integer sub	CarryOut, Rd = Rd – Rs1 + OldCarryOut	sbcs	—

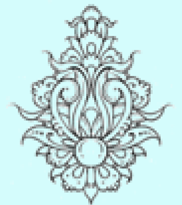


مجموعه دستورات خانواده‌ی X86

- روند تکامل با حفظ سازگاری

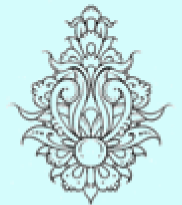
– 8080	۱۹۷۴	۸بیتی
– 8086	۱۹۷۸	۱۶بیتی
– 8087	۱۹۸۰	کمک‌پردازنده‌ی ممیز شناور
– 80286	۱۹۸۲	آدرس ۲۴بیتی همراه با MMU
– 80386	۱۹۸۵	۳۲ بیتی، اضافه شدن مودهای آدرس‌دهی جدید
– i486	۱۹۸۹	دارای خط لوله، حافظه‌ی نهان
– Pentium	۱۹۹۳	superscaler
– Pentium Pro	۱۹۹۵	ریزمعماری جدید
– Pentium III	۱۹۹۹	
– Pentium 4	۲۰۰۱	

Technical elegance ≠ market success



ثبات‌های خانواده‌ی x86

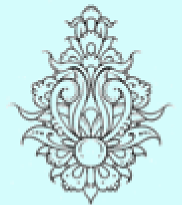
Name	31 … 0	Use
EAX		GPR 0
ECX		GPR 1
EDX		GPR 2
EBX		GPR 3
ESP		GPR 4
EBP		GPR 5
ESI		GPR 6
EDI		GPR 7
CS		Code segment pointer
SS		Stack segment pointer (top of stack)
DS		Data segment pointer 0
ES		Data segment pointer 1
FS		Data segment pointer 2
GS		Data segment pointer 3
EIP		Instruction pointer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



آدرس‌دهی در x86

- هر دستور دو عملوند دارد:

Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate



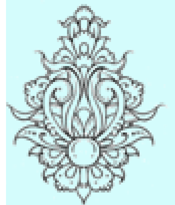
حالت‌های آدرس‌دهی حافظه:

- Address in register
- Address = R_{base} + displacement
- Address = $R_{base} + 2^{scale} \times R_{index}$ (scale = 0, 1, 2, or 3)
- Address = $R_{base} + 2^{scale} \times R_{index}$ + displacement

Mode	Description	Register restrictions	MIPS equivalent
Register indirect	Address is in a register.	Not ESP or EBP	`lw $s0,0($s1)`
Based mode with 8- or 32-bit displacement	Address is contents of base register plus displacement.	Not ESP	`lw $s0,100($s1) # <= 16-bit` `# displacement`

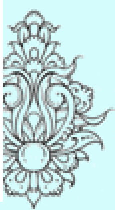
Base plus scaled index	The address is
	Base + (2^{Scale} x Index)
	where Scale has the value 0, 1, 2, or 3.
		Base: any GPR
		Index: not ESP
			`mul $t0,$s2,4`
			`add $t0,$t0,$s1`
			`lw $s0,0($t0)`

Base plus scaled index with 8- or 32-bit displacement	The address is
	Base + (2^{Scale} x Index) + displacement
	where Scale has the value 0, 1, 2, or 3.
		Base: any GPR
		Index: not ESP
			`mul $t0,$s2,4`
			`add $t0,$t0,$s1`
			`lw $s0,100($t0) # ð16-bit`
			`# displacement`



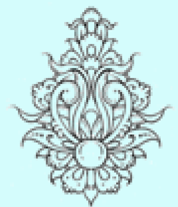
نمونه‌ای از دستورات x86

Instruction	Function
je name	if equal(condition code) {EIP=name}; EIP-128 <= name < EIP+128
jmp name	EIP=name
call name	SP=SP-4; M[SP]=EIP+5; EIP=name;
movw EBX,[EDI+45]	EBX=M[EDI+45]
push ESI	SP=SP-4; M[SP]=ESI
pop EDI	EDI=M[SP]; SP=SP+4
add EAX,#6765	EAX= EAX+6765
test EDX,#42	Set condition code (flags) with EDX and 42
movsl	M[EDI]=M[ESI]; EDI=EDI+4; ESI=ESI+4



نمونه‌ای از دستورات x86 (ادامه...)

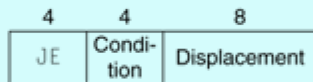
Instruction	Meaning
Control	**Conditional and unconditional branches**
`jnz, jz`	Jump if condition to EIP + 8-bit offset; `JNE` (for `JNZ`), `JE` (for `JZ`) are alternative names
`jmp`	Unconditional jump—8-bit or 16-bit offset
`call`	Subroutine call—16-bit offset; return address pushed onto stack
`ret`	Pops return address from stack and jumps to it
`loop`	Loop branch—decrement ECX; jump to EIP + 8-bit displacement if ECX $\neq$ 0
Data transfer	**Move data between registers or between register and memory**
`move`	Move between two registers or between register and memory
`push, pop`	Push source operand on stack; pop operand from stack top to a register
`les`	Load ES and one of the GPRs from memory
Arithmetic, logical	**Arithmetic and logical operations using the data registers and memory**
`add, sub`	Add source to destination; subtract source from destination; register-memory format
`cmp`	Compare source and destination; register-memory format
`shl, shr, rcr`	Shift left; shift logical right; rotate right with carry condition code as fill
`cbw`	Convert byte in eight rightmost bits of EAX to 16-bit word in right of EAX
`test`	Logical AND of source and destination sets condition codes
`inc, dec`	Increment destination, decrement destination
`or, xor`	Logical OR; exclusive OR; register-memory format
String	**Move between string operands; length given by a repeat prefix**
`movs`	Copies from string source to destination by incrementing ESI and EDI; may be repeated
`lods`	Loads a byte, word, or doubleword of a string into the EAX register



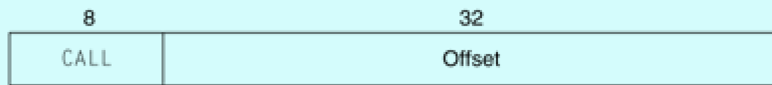
قالب دستورها

- طول دستورها متغیر است.
- سخت‌افزار دستورات را به دستورهای ساده‌تری (ریز دستور) ترجمه می‌کند.

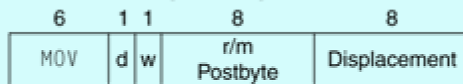
a. JE EIP + displacement



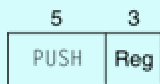
b. CALL



c. MOV EBX, [EDI + 45]

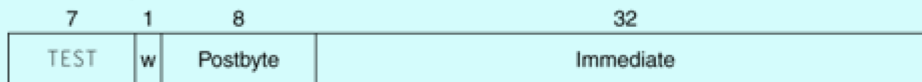


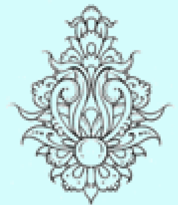
d. PUSH ESI



e. ADD EAX, #6765

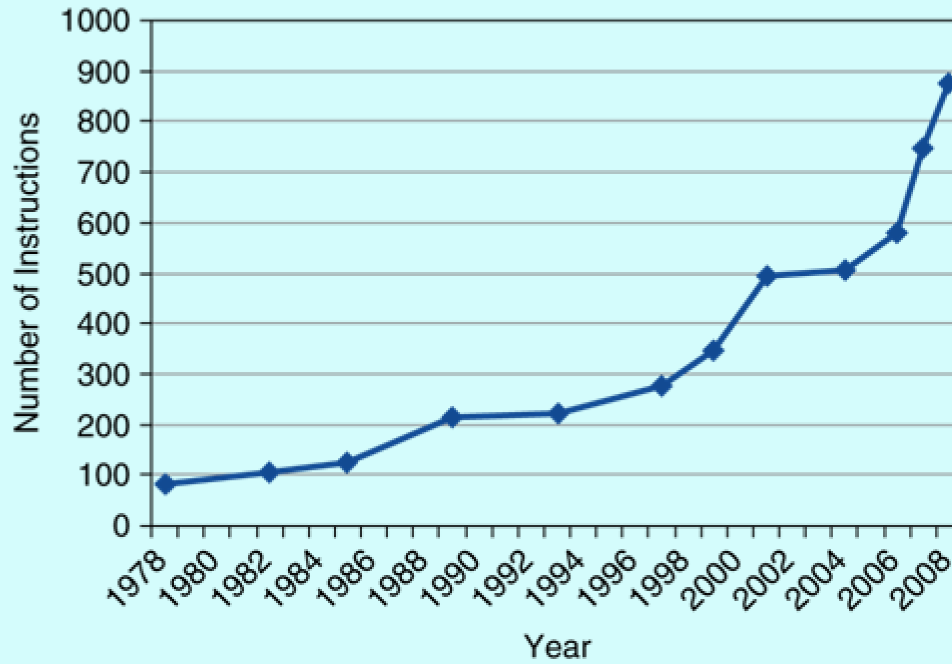
f. TEST EDX, #42

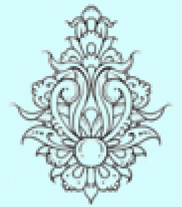




سفسطه!

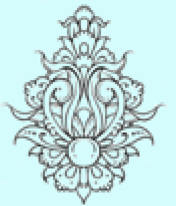
- آیا دستورات پیچیده به معنای کارایی بالاتر است؟
- آیا نوشتن برنامه به زبان اسمبلی، کارایی را افزایش می‌دهد؟



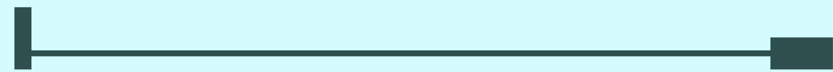


نتیجه‌گیری

- نظم منجر به سادگی بیشتر می‌شود.
- کوچک‌تر یعنی سریع‌تر
- سرعت داده به موارد پر استفاده
- طراحی خوب یعنی مصالحه‌ی خوب



مدارهای حساب •••



Numerical Precision is the very soul of science

نمایی از واحد حساب

Incr PC
Next addr
Next PC
jta
(PC)
PC
Instr cache
inst
rs
rt
rd
31
0 1 2
Reg file
(rs)
(rt)
SE
32
16
imm
op
fn
0 1
ALUOvfl
Ovfl
ALU
Func
ALU out
Data addr
Data in
Data cache
Data out
0 1 2
Register input
Br&Jump
RegDst
RegWrite
ALUSrc
ALUFunc
DataRead
DataWrite
RegInSrc

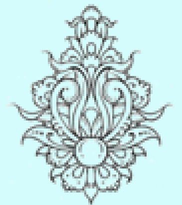
نوشتن نتیجه در ثبات

واکشی دستورات

دسترسی به محتوای ثبات

عملیات ALU

دسترسی به حافظه

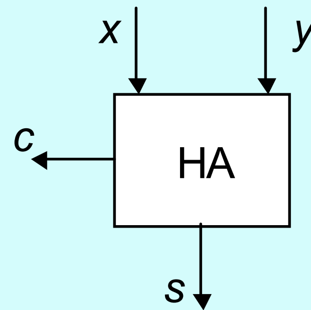


B.Parhami

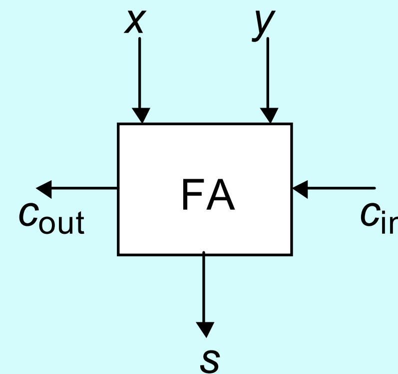
یادآوری

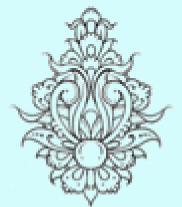
نیم‌جمع‌کننده و تمام‌جمع‌کننده

Inputs		Outputs	
x	y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



Inputs			Outputs	
x	y	c_{in}	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1





B.Parhami

شیوه‌های گوناگون ساخت تمام‌جمع‌کننده