198:211
Computer Architecture

- Topics:
  - Processor Design

@2010 Badri Computer Architecture Assembly II

Where are we now?

- C-Programming
  - A "real" computer language…
- Data Representation
  - Everything goes down to bits and bytes
- Machine representation Language
  - Very limited programming model
- Digital Logic
  - Transistors → Gates → Circuits
  - Circuits → { Memory, Registers, and Components}
- What are we going to do now….

@2010 Badri Computer Architecture Assembly II

Processor Design

- Figure out how a small subset of the instruction set works in hardware
- Use Hardware blocks to describe data path
- Use control logic to ensure steps of the instruction flows smoothly

@2010 Badri Computer Architecture Assembly II

Add instruction

- Type of add depends on where operands are located
  - add reg1, reg2
    - Add contents of reg1 to reg2 and store result in reg2
  - add $constant, reg2
    - Add contents of memory immediately following the instruction to reg 2 and store result in reg2
  - add (effective-address), reg2
    - Add contents of effective address to reg2 and store result in reg2
    - Effective address could be relative; relative + offset; relative + offset, index; relative + offset, index*scaled

@2010 Badri Computer Architecture Assembly II

Instruction Example

- Addition Instruction

Generic Form

Encoded Representation

```
addl %edx, %eax    0 3 rA rB
```

- Add value in register %edx to that in register %eax
  - Store result in register %eax
- Two-byte encoding
  - First indicates instruction type
  - Second gives source and destination registers

ADD-Add : note src, dst are reversed in the intel manual

ADD—Add

Opcode	Instruction	Description
04 *ib*	ADD AL,*imm8*	Add *imm8* to AL
05 *iw*	ADD AX,*imm16*	Add *imm16* to AX
05 *id*	ADD EAX,*imm32*	Add *imm32* to EAX
80 /0 *ib*	ADD *r/m8*,*imm8*	Add *imm8* to *r/m8*
81 /0 *iw*	ADD *r/m16*,*imm16*	Add *imm16* to *r/m16*
81 /0 *id*	ADD *r/m32*,*imm32*	Add *imm32* to *r/m32*
83 /0 *ib*	ADD *r/m16*,*imm8*	Add sign-extended *imm8* to *r/m16*
83 /0 *ib*	ADD *r/m32*,*imm8*	Add sign-extended *imm8* to *r/m32*
00 /*r*	ADD *r/m8*,*r8*	Add *r8* to *r/m8*
01 /*r*	ADD *r/m16*,*r16*	Add *r16* to *r/m16*
01 /*r*	ADD *r/m32*,*r32*	Add r32 to *r/m32*
02 /*r*	ADD *r8*,*r/m8*	Add *r/m8* to *r8*
03 /*r*	ADD *r16*,*r/m16*	Add *r/m16* to *r16*
03 /*r*	ADD *r32*,*r/m32*	Add *r/m32* to *r32*

In assembly language add src, dst means src + dst → dst

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) /digit (Opcode) REG =			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--][1]		100	04	0C	14	1C	24	2C	34	3C
disp32[2]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
disp8[EAX][3]	01	000	40	48	50	58	60	68	70	78
disp8[ECX]		001	41	49	51	59	61	69	71	79
disp8[EDX]		010	42	4A	52	5A	62	6A	72	7A
disp8[EBX];		011	43	4B	53	5B	63	6B	73	7B
disp8[--][--]		100	44	4C	54	5C	64	6C	74	7C
disp8[EBP]		101	45	4D	55	5D	65	6D	75	7D
disp8[ESI]		110	46	4E	56	5E	66	6E	76	7E
disp8[EDI]		111	47	4F	57	5F	67	6F	77	7F
disp32[EAX]	10	000	80	88	90	98	A0	A8	B0	B8
disp32[ECX]		001	81	89	91	99	A1	A9	B1	B9
disp32[EDX]		010	82	8A	92	9A	A2	AA	B2	BA
disp32[EBX]		011	83	8B	93	9B	A3	AB	B3	BB
disp32[--][--]		100	84	8C	94	9C	A4	AC	B4	BC
disp32[EBP]		101	85	8D	95	9D	A5	AD	B5	BD
disp32[ESI]		110	86	8E	96	9E	A6	AE	B6	BE
disp32[EDI]		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM1/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

Instruction Example in x86

- Addition Instruction

Generic Form

Encoded Representation

```
addl %edx, %eax    0 3 C 2
```

- Opcode is 03 and src, dst are %edx, %eax
  - from previous table (Table 2.2) code is C2
- So, 03 C2 means add %edx, %e
- 03 45
- means add 8(%ebp), %eax
- 05
- add $cons, %eax

```
addl 8(%ebp), %eax    0 3 4 5
```

```
addl $4, %eax    0 5 000 4
```

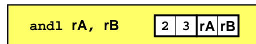
Arithmetic and Logical Operations

Instruction Code

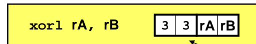
Function Code

Subtract (rA from rB)

And



Exclusive-Or



- The second byte will vary based on type of operands
- Set condition codes as side effect

Lower 4 bits can vary depending upon the type Of operand
20 means 8 byte register transfer,

Lower 4 bits can vary
30 means 8 byte register transfer,

@2010 Badri Computer Architecture Assembly II

Subtract operation- Instruction Set Reference

SUB—Subtract

Opcode	Instruction	Description
2C *ib*	SUB AL,*imm8*	Subtract *imm8* from AL
2D *iw*	SUB AX,*imm16*	Subtract *imm16* from AX
2D *id*	SUB EAX,*imm32*	Subtract *imm32* from EAX
80 /5 *ib*	SUB *r/m8*,*imm8*	Subtract *imm8* from *r/m8*
81 /5 *iw*	SUB *r/m16*,*imm16*	Subtract *imm16* from *r/m16*
81 /5 *id*	SUB *r/m32*,*imm32*	Subtract *imm32* from *r/m32*
83 /5 *ib*	SUB *r/m16*,*imm8*	Subtract sign-extended *imm8* from *r/m16*
83 /5 *ib*	SUB *r/m32*,*imm8*	Subtract sign-extended *imm8* from *r/m32*
28 /*r*	SUB *r/m8*,*r8*	Subtract *r8* from *r/m8*
29 /*r*	SUB *r/m16*,*r16*	Subtract *r16* from *r/m16*
29 /*r*	SUB *r/m32*,*r32*	Subtract *r32* from *r/m32*
2A /*r*	SUB *r8*,*r/m8*	Subtract *r/m8* from *r8*
2B /*r*	SUB *r16*,*r/m16*	Subtract *r/m16* from *r16*
2B /*r*	SUB *r32*,*r/m32*	Subtract *r/m32* from *r32*

@2010 Badri Computer Architecture Assembly II

AND—Logical AND

Opcode	Instruction	Description
24 *ib*	AND AL,*imm8*	AL AND *imm8*
25 *iw*	AND AX,*imm16*	AX AND *imm16*
25 *id*	AND EAX,*imm32*	EAX AND *imm32*
80 /4 *ib*	AND *r/m8*,*imm8*	*r/m8* AND *imm8*
81 /4 *iw*	AND *r/m16*,*imm16*	*r/m16* AND *imm16*
81 /4 *id*	AND *r/m32*,*imm32*	*r/m32* AND *imm32*
83 /4 *ib*	AND *r/m16*,*imm8*	*r/m16* AND *imm8 (sign-extended)*
83 /4 *ib*	AND *r/m32*,*imm8*	*r/m32* AND *imm8 (sign-extended)*
20 /*r*	AND *r/m8*,*r8*	*r/m8* AND *r8*
21 /*r*	AND *r/m16*,*r16*	*r/m16* AND *r16*
21 /*r*	AND *r/m32*,*r32*	*r/m32* AND *r32*
22 /*r*	AND *r8*,*r/m8*	*r8* AND *r/m8*
23 /*r*	AND *r16*,*r/m16*	*r16* AND *r/m16*
23 /*r*	AND *r32*,*r/m32*	*r32* AND *r/m32*

Description

This instruction performs a bitwise AND operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. Two memory operands cannot, however, be used in one instruction. Each bit of the instruction result is a 1 if both corresponding bits of the operands are 1; otherwise, it becomes a 0.

Operation

DEST ← DEST AND SRC;

@2010 Badri Computer Architecture Assembly II

XOR—Logical Exclusive OR

Opcode	Instruction	Description
34 *ib*	XOR AL,*imm8*	AL XOR *imm8*
35 *iw*	XOR AX,*imm16*	AX XOR *imm16*
35 *id*	XOR EAX,*imm32*	EAX XOR *imm32*
80 /6 *ib*	XOR *r/m8*,*imm8*	*r/m8* XOR *imm8*
81 /6 *iw*	XOR *r/m16*,*imm16*	*r/m16* XOR *imm16*
81 /6 *id*	XOR *r/m32*,*imm32*	*r/m32* XOR *imm32*
83 /6 *ib*	XOR *r/m16*,*imm8*	*r/m16* XOR *imm8 (sign-extended)*
83 /6 *ib*	XOR *r/m32*,*imm8*	*r/m32* XOR *imm8 (sign-extended)*
30 /*r*	XOR *r/m8*,*r8*	*r/m8* XOR *r8*
31 /*r*	XOR *r/m16*,*r16*	*r/m16* XOR *r16*
31 /*r*	XOR *r/m32*,*r32*	*r/m32* XOR *r32*
32 /*r*	XOR *r8*,*r/m8*	*r8* XOR *r/m8*
33 /*r*	XOR *r16*,*r/m16*	*r8* XOR *r/m8*
33 /*r*	XOR *r32*,*r/m32*	*r8* XOR *r/m8*

Description

This instruction performs a bitwise exclusive OR (XOR) operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result is 1 if the corresponding bits of the operands are different; each bit is 0 if the corresponding bits are the same.

@2010 Badri Computer Architecture Assembly II

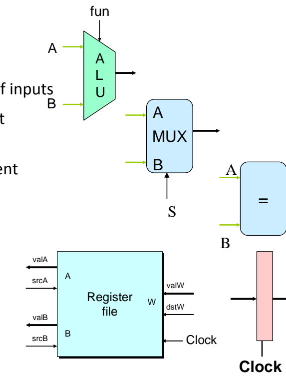
Single cycle stages

- Fetch Instruction (IF)
- Decode Instruction (ID)
- Execute Instruction (EX)
- Memory access (MEM)
- Write back (WB)
- Update Program Counter (PC)

@2010 Badri Computer Architecture Assembly II

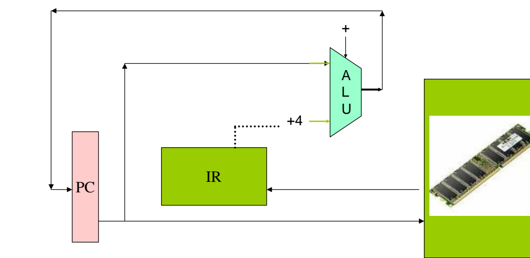
Building Blocks

- Combinational Logic
  - Compute Boolean functions of inputs
  - Continuously respond to input changes
  - Operate on data and implement control
- Storage Elements
  - Store bits
  - Addressable memories
  - Non-addressable registers
  - Loaded only as clock rises



@2010 Badri Computer Architecture Assembly II

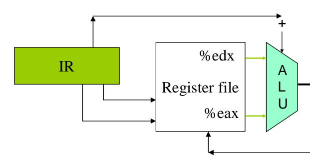
Fetch Instruction



Fetch Instruction and increment Program counter

@2010 Badri Computer Architecture Assembly II

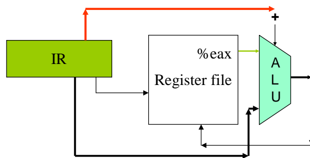
Add instruction (add %edx, %eax)



- Op code determines type of operation
  - 03 C2
- Operands are in registers
- They are fed to the adder (ALU)
- Result stored back in register
- Based On Instruction the signal to ALU can be +, -, * or /

@2010 Badri Computer Architecture Assembly II

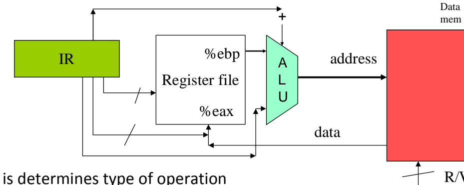
Add instruction (add $4, %eax)



- Op code is determines type of operation
- 03 05
- One Operand is a register, the other is part of the instruction
- %eax, immediate value
- They are fed to the adder (ALU)
- Result stored back in register

@2010 Badri Computer Architecture Assembly II

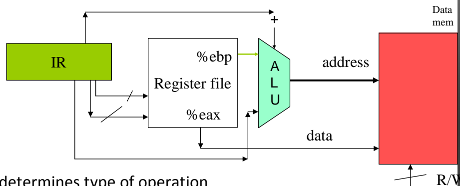
mov instruction (mov 8(%ebp), %eax)



- Op code is determines type of operation
- 8b 45
- Load word from memory onto register %eax
- Address is 8 + (%ebp)

@2010 Badri Computer Architecture Assembly II

mov instruction (mov %eax,8(%ebp))



- Op code determines type of operation
- 89 45
- Store word from register %eax onto memory
- Address is 8 + (%ebp)

@2010 Badri Computer Architecture Assembly II

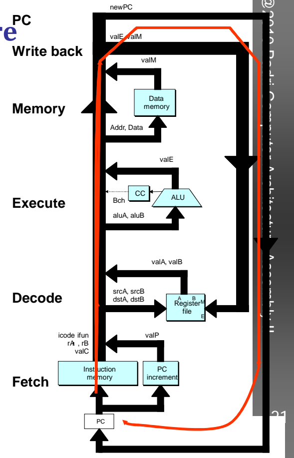
MOV—Move

Opcode	Instruction	Description
88 /r	MOV r/m8,r8	Move r8 to r/m8
89 /r	MOV r/m16,r16	Move r16 to r/m16
89 /r	MOV r/m32,r32	Move r32 to r/m32
8A /r	MOV r8,r/m8	Move r/m8 to r8
8B /r	MOV r16,r/m16	Move r/m16 to r16
8B /r	MOV r32,r/m32	Move r/m32 to r32
8C /r	MOV r/m16,Sreg**	Move segment register to r/m16
8E /r	MOV Sreg,r/m16**	Move r/m16 to segment register
A0	MOV AL,moffs8*	Move byte at (seg:offset) to AL
A1	MOV AX,moffs16*	Move word at (seg:offset) to AX
A1	MOV EAX,moffs32*	Move doubleword at (seg:offset) to EAX
A2	MOV moffs8*,AL	Move AL to (seg:offset)
A3	MOV moffs16*,AX	Move AX to (seg:offset)
A3	MOV moffs32*,EAX	Move EAX to (seg:offset)
B0+ rb	MOV r8,imm8	Move imm8 to r8
B8+ rw	MOV r16,imm16	Move imm16 to r16
B8+ rd	MOV r32,imm32	Move imm32 to r32
C6 /0	MOV r/m8,imm8	Move imm8 to r/m8
C7 /0	MOV r/m16,imm16	Move imm16 to r/m16
C7 /0	MOV r/m32,imm32	Move imm32 to r/m32

@2010 Badri Computer Architecture Assembly II

SEQ Hardware Structure

- State
  - Program counter register (PC)
  - Condition code register (CC)
  - Register File
  - Memories
    - Access same memory space
    - Data: for reading/writing program data
    - Instruction: for reading instructions
- Instruction Flow
  - Read instruction at address specified by PC
  - Process through stages
  - Update program counter

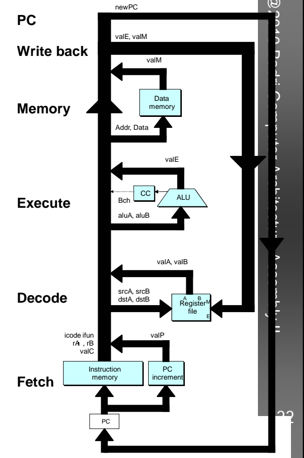


@2010 Badri Computer Architecture Assembly II

21

SEQ Stages

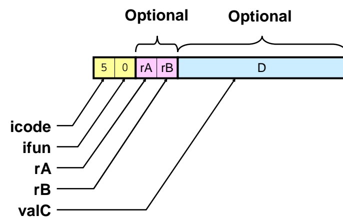
- Fetch
  - Read instruction from instruction memory
- Decode
  - Read program registers
- Execute
  - Compute value or address
- Memory
  - Read or write data
- Write Back
  - Write program registers
- PC
  - Update program counter



@2010 Badri Computer Architecture Assembly II

22

Instruction Decoding

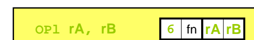


- Instruction Format
  - Instruction byte icode:ifun
  - Optional register byte rA:rB
  - Optional constant word valC

@2010 Badri Computer Architecture Assembly II

23

Executing Arith./Logical Operation



- Fetch
  - Read 2 bytes
- Decode
  - Read operand registers
- Execute
  - Perform operation
  - Set condition codes
- Memory
  - Do nothing
- Write back
  - Update register
- PC Update
  - Increment PC by 2

@2010 Badri Computer Architecture Assembly II

24

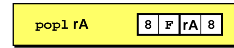
Stage Computation: Arith/Log. Ops

	OPl rA, rB	
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	Read instruction byte Read register byte Compute next PC
Decode	valA ← R[rA] valB ← R[rB]	Read operand A Read operand B
Execute	valE ← valB OP valA Set CC	Perform ALU operation Set condition code register
Memory		
Write back	R[rB] ← valE	Write back result
PC update	PC ← valP	Update PC

- Formulate instruction execution as sequence of simple steps
- Use same general form for all instructions

@2010 Badri Computer Architecture Assembly II

Executing `popl`



- Fetch
  - Read 2 bytes
- Decode
  - Read stack pointer
- Execute
  - Increment stack pointer by 4
  - Read from old stack pointer
- Write back
  - Update stack pointer
  - Write result to register
- PC Update
  - Increment PC by 2

@2010 Badri Computer Architecture Assembly II

POP Instruction

POP—Pop a Value from the Stack

Opcode	Instruction	Description
8F /0	POP *m16*	Pop top of stack into *m16*; increment stack pointer
8F /0	POP *m32*	Pop top of stack into *m32*; increment stack pointer
58+ *rw*	POP *r16*	Pop top of stack into *r16*; increment stack pointer
58+ *rd*	POP *r32*	Pop top of stack into *r32*; increment stack pointer
1F	POP DS	Pop top of stack into DS; increment stack pointer
07	POP ES	Pop top of stack into ES; increment stack pointer
17	POP SS	Pop top of stack into SS; increment stack pointer
0F A1	POP FS	Pop top of stack into FS; increment stack pointer
0F A9	POP GS	Pop top of stack into GS; increment stack pointer

@2010 Badri Computer Architecture Assembly II

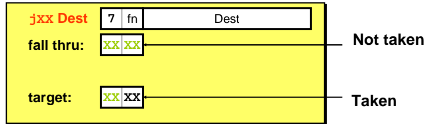
Stage Computation: `popl`

	popl rA	
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	Read instruction byte Read register byte Compute next PC
Decode	valA ← %esp valB ← %esp	Read stack pointer Read stack pointer
Execute	valE ← valB + 4	Increment stack pointer
Memory	valM ← M_4[valA]	Read from stack
Write back	R[%esp] ← valE R[rA] ← valM	Update stack pointer Write back result
PC update	PC ← valP	Update PC

- Use ALU to increment stack pointer
- Must update two registers
  - Popped value
  - New stack pointer

@2010 Badri Computer Architecture Assembly II

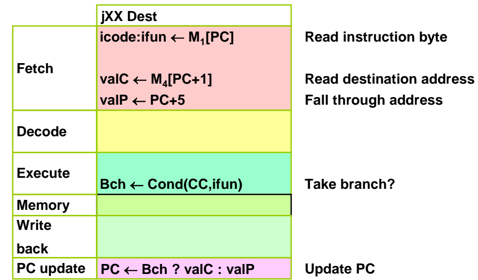
Executing Jumps



- Fetch
  - Read 5 bytes
  - Increment PC by 5
- Decode
  - Do nothing
- Execute
  - Determine whether to take branch based on jump condition and condition codes
- Memory
  - Do nothing
- Write back
  - Do nothing
- PC Update
  - Set PC to Dest if branch taken or to incremented PC if not branch

@2010 Badri Computer Architecture Assembly II

Stage Computation: Jumps



	jXX Dest	
Fetch	icode:ifun ← M_1[PC] valC ← M_4[PC+1] valP ← PC+5	Read instruction byte Read destination address Fall through address
Decode		
Execute	Bch ← Cond(CC,ifun)	Take branch?
Memory		
Write back		
PC update	PC ← Bch ? valC : valP	Update PC

- Compute both addresses
- Choose based on setting of condition codes and branch condition

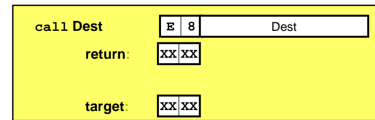
@2010 Badri Computer Architecture Assembly II

Jcc—Jump if Condition Is Met

Opcode	Instruction	Description
77 cb	JA rel8	Jump short if above (CF=0 and ZF=0)
73 cb	JAE rel8	Jump short if above or equal (CF=0)
72 cb	JB rel8	Jump short if below (CF=1)
76 cb	JBE rel8	Jump short if below or equal (CF=1 or ZF=1)
72 cb	JC rel8	Jump short if carry (CF=1)
E3 cb	JCXZ rel8	Jump short if CX register is 0
E3 cb	JECXZ rel8	Jump short if ECX register is 0
74 cb	JE rel8	Jump short if equal (ZF=1)
7F cb	JG rel8	Jump short if greater (ZF=0 and SF=OF)
7D cb	JGE rel8	Jump short if greater or equal (SF=OF)
7C cb	JL rel8	Jump short if less (SF<>OF)
7E cb	JLE rel8	Jump short if less or equal (ZF=1 or SF<>OF)
76 cb	JNA rel8	Jump short if not above (CF=1 or ZF=1)
72 cb	JNAE rel8	Jump short if not above or equal (CF=1)
73 cb	JNB rel8	Jump short if not below (CF=0)
77 cb	JNBE rel8	Jump short if not below or equal (CF=0 and ZF=0)
73 cb	JNC rel8	Jump short if not carry (CF=0)
75 cb	JNE rel8	Jump short if not equal (ZF=0)
7E cb	JNG rel8	Jump short if not greater (ZF=1 or SF<>OF)
7C cb	JNGE rel8	Jump short if not greater or equal (SF<>OF)
7D cb	JNL rel8	Jump short if not less (SF=OF)
7F cb	JNLE rel8	Jump short if not less or equal (ZF=0 and SF=OF)
71 cb	JNO rel8	Jump short if not overflow (OF=0)
7B cb	JNP rel8	Jump short if not parity (PF=0)
79 cb	JNS rel8	Jump short if not sign (SF=0)
75 cb	JNZ rel8	Jump short if not zero (ZF=0)
70 cb	JO rel8	Jump short if overflow (OF=1)
7A cb	JP rel8	Jump short if parity (PF=1)
7A cb	JPE rel8	Jump short if parity even (PF=1)
7B cb	JPO rel8	Jump short if parity odd (PF=0)
78 cb	JS rel8	Jump short if sign (SF=1)
74 cb	JZ rel8	Jump short if zero (ZF = 1)
0F 87 cw/cd	JA rel16/32	Jump near if above (CF=0 and ZF=0)
0F 83 cw/cd	JAE rel16/32	Jump near if above or equal (CF=0)
0F 82 cw/cd	JB rel16/32	Jump near if below (CF=1)
0F 86 cw/cd	JBE rel16/32	Jump near if below or equal (CF=1 or ZF=1)
0F 82 cw/cd	JC rel16/32	Jump near if carry (CF=1)

@2010 Badri Computer Architecture Assembly II

Executing `call`



- Fetch
  - Read 5 bytes
  - Increment PC by 5
- Decode
  - Read stack pointer
- Execute
  - Decrement stack pointer by 4
- Memory
  - Write incremented PC to new value of stack pointer
- Write back
  - Update stack pointer
- PC Update
  - Set PC to Dest

@2010 Badri Computer Architecture Assembly II

Stage Computation: `call`

	`call` Dest	
Fetch	icode:ifun ← M_1[PC] valC ← M_4[PC+1] valP ← PC+5	Read instruction byte Read destination address Compute return point
Decode	valB ← R[`%esp`]	Read stack pointer
Execute	valE ← valB + –4	Decrement stack pointer
Memory	M_4[valE] ← valP	Write return value on stack
Write back	R[`%esp`] ← valE	Update stack pointer
PC update	PC ← valC	Set PC to destination

- Use ALU to decrement stack pointer
- Store incremented PC

@2010 Badri Computer Architecture Assembly II

Call Opcodes in X86

CALL—Call Procedure

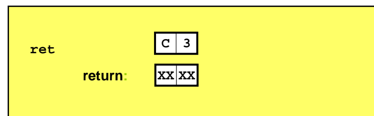
Opcode	Instruction	Description
E8 *cw*	CALL *rel16*	Call near, relative, displacement relative to next instruction
E8 *cd*	CALL *rel32*	Call near, relative, displacement relative to next instruction
FF /2	CALL *r/m16*	Call near, absolute indirect, address given in *r/m16*
FF /2	CALL *r/m32*	Call near, absolute indirect, address given in *r/m32*
9A *cd*	CALL *ptr16:16*	Call far, absolute, address given in operand
9A *cp*	CALL *ptr16:32*	Call far, absolute, address given in operand
FF /3	CALL *m16:16*	Call far, absolute indirect, address given in *m16:16*
FF /3	CALL *m16:32*	Call far, absolute indirect, address given in *m16:32*

Description

This instruction saves procedure linking information on the stack and branches to the procedure (called procedure) specified with the destination (target) operand. The target operand specifies the address of the first instruction in the called procedure. This operand can be an immediate value, a general-purpose register, or a memory location.

@2010 Badri Computer Architecture Assembly II

Executing `ret`



- Fetch
  - Read 1 byte
- Decode
  - Read stack pointer
- Execute
  - Increment stack pointer by 4
- Memory
  - Read return address from old stack pointer
- Write back
  - Update stack pointer
- PC Update
  - Set PC to return address

@2010 Badri Computer Architecture Assembly II

Stage Computation: `ret`

	`ret`	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
Decode	valA ← R[`%esp`] valB ← R[`%esp`]	Read operand stack pointer Read operand stack pointer
Execute	valE ← valB + 4	Increment stack pointer
Memory	valM ← M_4[valA]	Read return address
Write back	R[`%esp`] ← valE	Update stack pointer
PC update	PC ← valM	Set PC to return address

- Use ALU to increment stack pointer
- Read return address from memory

@2010 Badri Computer Architecture Assembly II

RET—Return from Procedure

Opcode	Instruction	Description
C3	RET	Near return to calling procedure
CB	RET	Far return to calling procedure
C2 *iw*	RET *imm16*	Near return to calling procedure and pop *imm16* bytes from stack
CA *iw*	RET *imm16*	Far return to calling procedure and pop *imm16* bytes from stack

Description

This instruction transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.

The optional source operand specifies the number of stack bytes to be released after the return address is popped; the default is none. This operand can be used to release parameters from the stack that were passed to the called procedure and are no longer needed. It must be used when the CALL instruction used to switch to a new procedure uses a call gate with a non-zero word count to access the new procedure. Here, the source operand for the RET instruction must specify the same number of bytes as is specified in the word count field of the call gate.

@2010 Badri Computer Architecture Assembly II