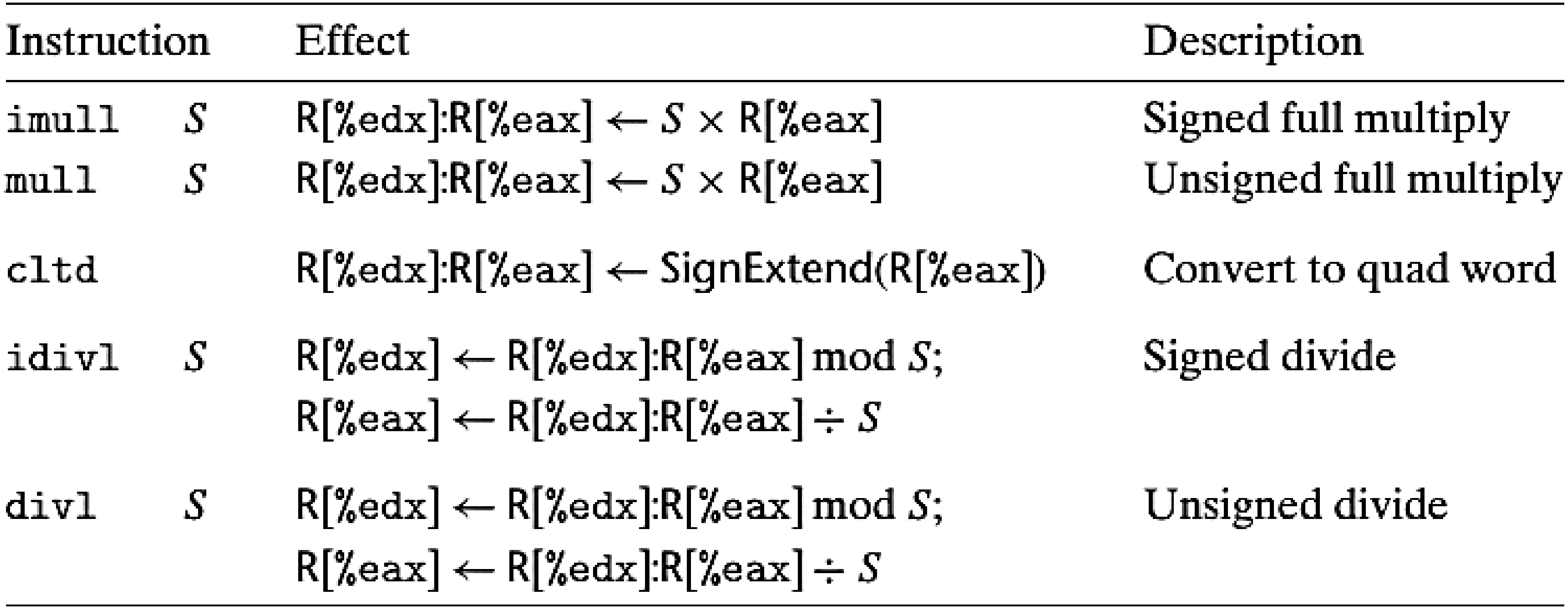
IA32 PROGRAMMING II

Dr. Arka Prokash Mazumdar

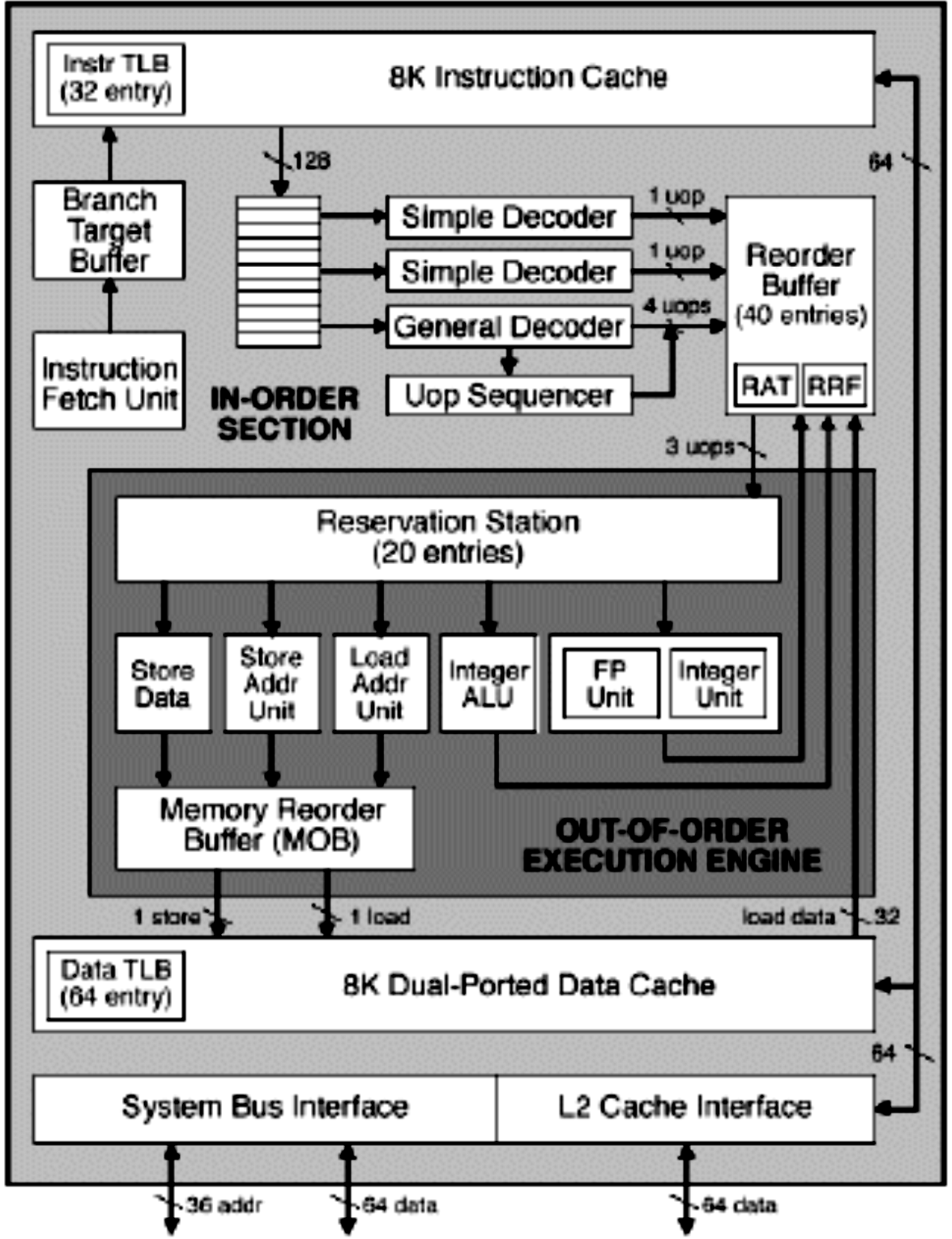
Advanced Arithmetic Operations



Instruction		Effect	Description
`imull`	*S*	R[%edx]:R[%eax] ← *S* × R[%eax]	Signed full multiply
`mull`	*S*	R[%edx]:R[%eax] ← *S* × R[%eax]	Unsigned full multiply
`cltd`		R[%edx]:R[%eax] ← SignExtend(R[%eax])	Convert to quad word
`idivl`	*S*	R[%edx] ← R[%edx]:R[%eax] mod *S*; R[%eax] ← R[%edx]:R[%eax] ÷ *S*	Signed divide
`divl`	*S*	R[%edx] ← R[%edx]:R[%eax] mod *S*; R[%eax] ← R[%edx]:R[%eax] ÷ *S*	Unsigned divide

Also see:	**sall**	**sarl**	(Arithmetic)
	shll	**shrl**	(Logical)

PentiumPro Block Diagram



Operations

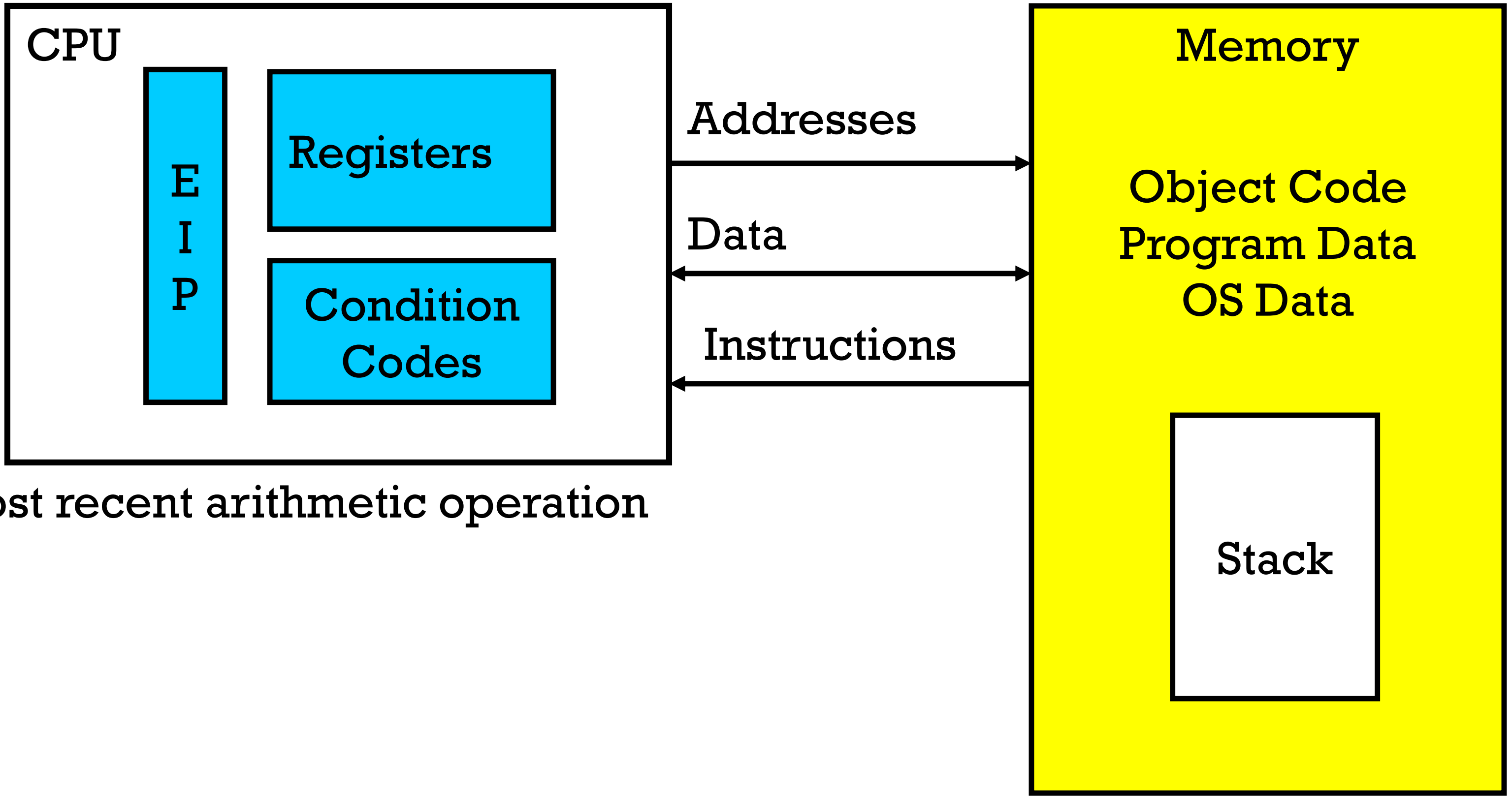
- Translates instructions dynamically into "Uops" / "μops"
  - 128 bits wide
- Holds
  - Operation 32
  - Two sources, and + 32 + 32
  - Destination + 32 = 128bits **WHY??**
- Executes Uops with **"Out of Order"** engine
  - Uop executed when
    1. Operands are available
    2. Functional unit available

Operations

  - Execution controlled by "Reservation Stations"
    - Keeps track of data **dependencies** between uops
    - Allocates resources
- Consequences
  - Indirect relationship between
    - IA32 code &
    - What actually gets executed
  - Tricky to predict / optimize performance at assembly level

Machine View

- EIP (Program Counter)
  - Address of next instruction
- Register File
  - Heavily used program data
- Condition Codes
  - Store status information about most recent arithmetic operation
  - Used for conditional branching
- Memory
  - Byte addressable array
  - Code, user data, (some) OS data
  - Includes stack used to support procedures



Instruction Example

- C Code
  - Add two signed integers
- Assembly
  - Add TWO 4-byte integers
    - "Long" words in GCC
    - Same instruction whether signed or unsigned
  - Operands:

x:	Register	%eax
y:	Memory	M[%ebp+8]
t:	Register	%eax

      - Return function value in **%eax**
- Object Code
  - 3-byte instruction
  - Stored at address 0x401046

```
int t = x+y;
```

```
addl 8(%ebp),%eax
```

Similar to expression
`x += y`

```
0x401046:     03 45 08
```

Condition Codes

- Single Bit Registers

CF	Carry Flag
SF	Sign Flag
ZF	Zero Flag
OF	Overflow Flag

Condition Codes

- Implicitly Set By Arithmetic Operations

  `addl` *Src*,*Dest*

  C analog: `t = a + b`

  - CF set if carry out from most significant bit
    - Used to detect unsigned overflow
  - ZF set if `t == 0`
  - SF set if `t < 0`
  - OF set if two's complement overflow

    `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`
- ***Not*** Set by `leal` instruction

Setting Codes

Explicit Setting by Compare Instruction

`cmpl` ***Src2*, *Src1***

- like computing ***Src1–Src2*** without setting destination

- CF set if carry out from most significant bit
  - Used for unsigned comparisons
- ZF set if `a == b`
- SF set if `(a-b) < 0`
- OF set if two's complement overflow

```
(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)
```

Setting Codes

Explicit Setting by Test instruction

`testl` *Src2*,*Src1*

- Sets condition codes based on value of: ***Src1 & Src2***
  - Useful to have one of the operands be a mask
- **`testl b,a`**
  - like computing `a&b` without setting destination
- ZF set when `a&b == 0`
- SF set when `a&b < 0`

Accessing Condition Codes: SETX

SetX	Condition	Description
`sete`	`ZF`	Equal / Zero
`setne`	`~ZF`	Not Equal / Not Zero
`sets`	`SF`	Negative
`setns`	`~SF`	Nonnegative
`setg`	`~(SF^OF)&~ZF`	Greater (Signed)
`setge`	`~(SF^OF)`	Greater or Equal (Signed)
`setl`	`(SF^OF)`	Less (Signed)
`setle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`seta`	`~CF&~ZF`	Above (unsigned)
`setb`	`CF`	Below (unsigned)

Accessing Condition Codes: JX

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~(SF^OF)&~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF)|ZF	Less or Equal (Signed)
ja	~CF&~ZF	Above (unsigned)
jb	CF	Below (unsigned)

Condition Code Example

```
int max(int x, int y)
{
  if (x > y)
    return x;
  else
    return y;
}
```

```
_max:
        pushl %ebp              }  Set
        movl %esp,%ebp          }  Up

        movl 8(%ebp),%edx       }
        movl 12(%ebp),%eax      }
        cmpl %eax,%edx          }   Body
        jle L9                  }
        movl %edx,%eax          }
L9:

        movl %ebp,%esp          }
        popl %ebp               }  Finish
        ret                     }
```

Condition Code Example

```
int max(int x, int y)
{
  if (x > y)
    goto Flag;

  return y;

Flag:
  return x;
}
```

```
_max:
       pushl %ebp
       movl %esp,%ebp

       movl 8(%ebp),%edx
       movl 12(%ebp),%eax
       cmpl %eax,%edx
       jle L9
       movl %edx,%eax
L9:

       movl %ebp,%esp
       popl %ebp
       ret
```

Set Up

Body

Finish

“Do-While” Example

C Code

```
int fact_do
  (int x)
{
  int result = 1;
  do {
    result *= x;
    x = x-1;
  } while (x > 1);
  return result;
}
```

Goto Version

```
int fact_goto(int x)
{
  int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
  return result;
}
```

“Do-While” Example

Goto Version

```
int fact_goto
  (int x)
{
  int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
  return result;
}
```

- Registers

```
%edx   x
%eax   result
```

Assembly

```
_fact_goto:
  pushl %ebp            # Setup
  movl %esp,%ebp        # Setup
  movl $1,%eax          # eax = 1
  movl 8(%ebp),%edx     # edx = x

L11:
  imull %edx,%eax       # result *= x
  decl %edx             # x--
  cmpl $1,%edx          # Compare x : 1
  jg L11                # if > goto loop

  movl %ebp,%esp        # Finish
  popl %ebp             # Finish
  ret                   # Finish
```

“Switch” Example

- Series of conditionals
  - Good if few cases
  - Slow if many
- Jump Table
  - Lookup branch target
  - Avoids conditionals
  - Possible when cases are small integer constants
- GCC
  - Picks one based on case structure

```
long switch_eg
   (long x, long y, long z)
{
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

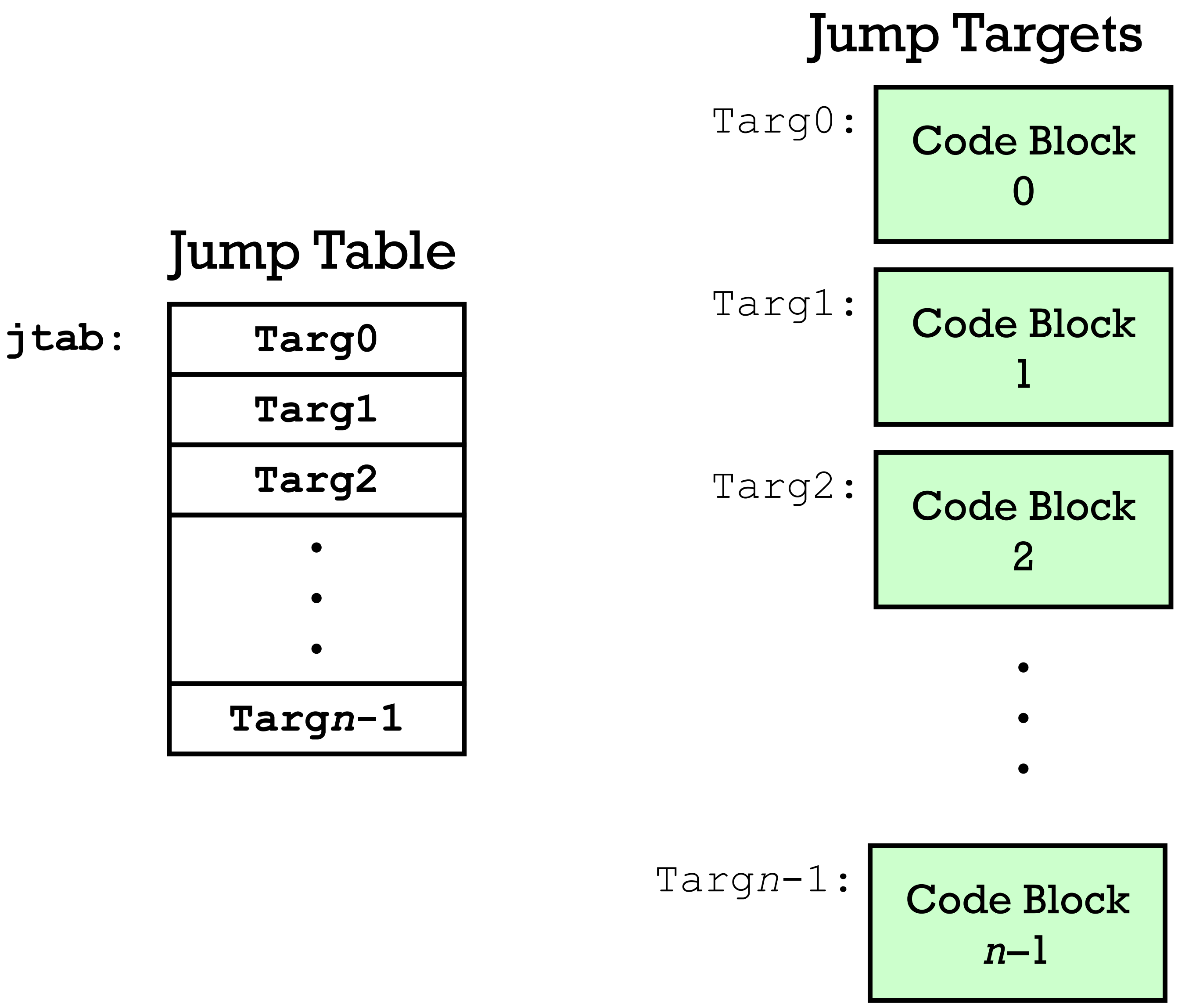
Jump Table

Switch Form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

Approx. Translation

```
target = JTab[x];
goto *target;
```



Jump Table

jtab:

Targ0
Targ1
Targ2
• • •
Targn-1

Jump Targets

Targ0: Code Block 0

Targ1: Code Block 1

Targ2: Code Block 2

• • •

Targn-1: Code Block n–1

Jump Table

```
switch_eg:
      pushl %ebp   # Setup
      movl  %esp, %ebp    # Setup
      pushl %ebx   # Setup
      movl  $1, %ebx      # w = 1
      movl  8(%ebp), %edx       # edx = x
      movl  16(%ebp), %ecx      # ecx = z
      cmpl  $6, %edx      # x:6
      ja    .L61   # if > goto default
      jmp   *.L62(,%edx,4)      # goto JTab[x]
```

```
long switch_eg
   (long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Jump Table

Table Contents

```
.section .rodata
   .align 4
.L57:
  .long .L51 #Op = 0
  .long .L52 #Op = 1
  .long .L53 #Op = 2
  .long .L54 #Op = 3
  .long .L55 #Op = 4
  .long .L56 #Op = 5
```

Enumerated Values

ADD	0
MULT	1
MINUS	2
DIV	3
MOD	4
BAD	5

```
.L51:
    movl $43,%eax # ’+’
    jmp .L49
.L52:
    movl $42,%eax # ’*’
    jmp .L49
.L53:
    movl $45,%eax # ’-’
    jmp .L49
.L54:
    movl $47,%eax # ’/’
    jmp .L49
.L55:
    movl $37,%eax # ’%’
    jmp .L49
.L56:
    movl $63,%eax # ’?’
    # Fall Through to .L49
```