CS33 Discussion 5

Control and Optimization

Brandon Wu

11.7.14

A Recap of Last time

- Procedure calls (the swap function)
- Jumps and branches
- Control structures: if, do while, while, for
- Structs, unions and alignment

SWITCH STATEMENTS

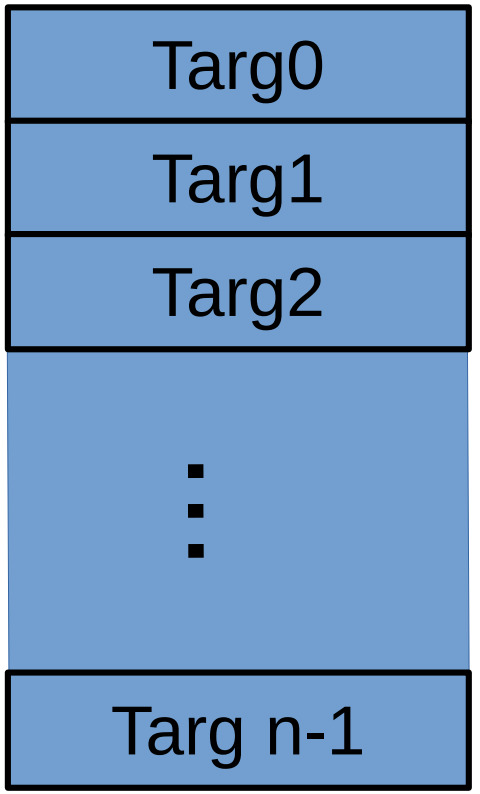
The last control structure

Review of Switch statements in C

```
switch(x) {
  case 1:
    /* Code block for x == 1 */
    break;
  case 2:
    /* Code block for x == 2*/
  case 3:
    /* Code block for x ==2 and x == 3 */
    break;
  case 5:
    /* Code block for x == 5 */
    break;
  default:
    /*default code*/
}
```

Implementing Case: Jump Table

- JT entry for each case in swich statement
  - At least!
- Contains addr of instructions

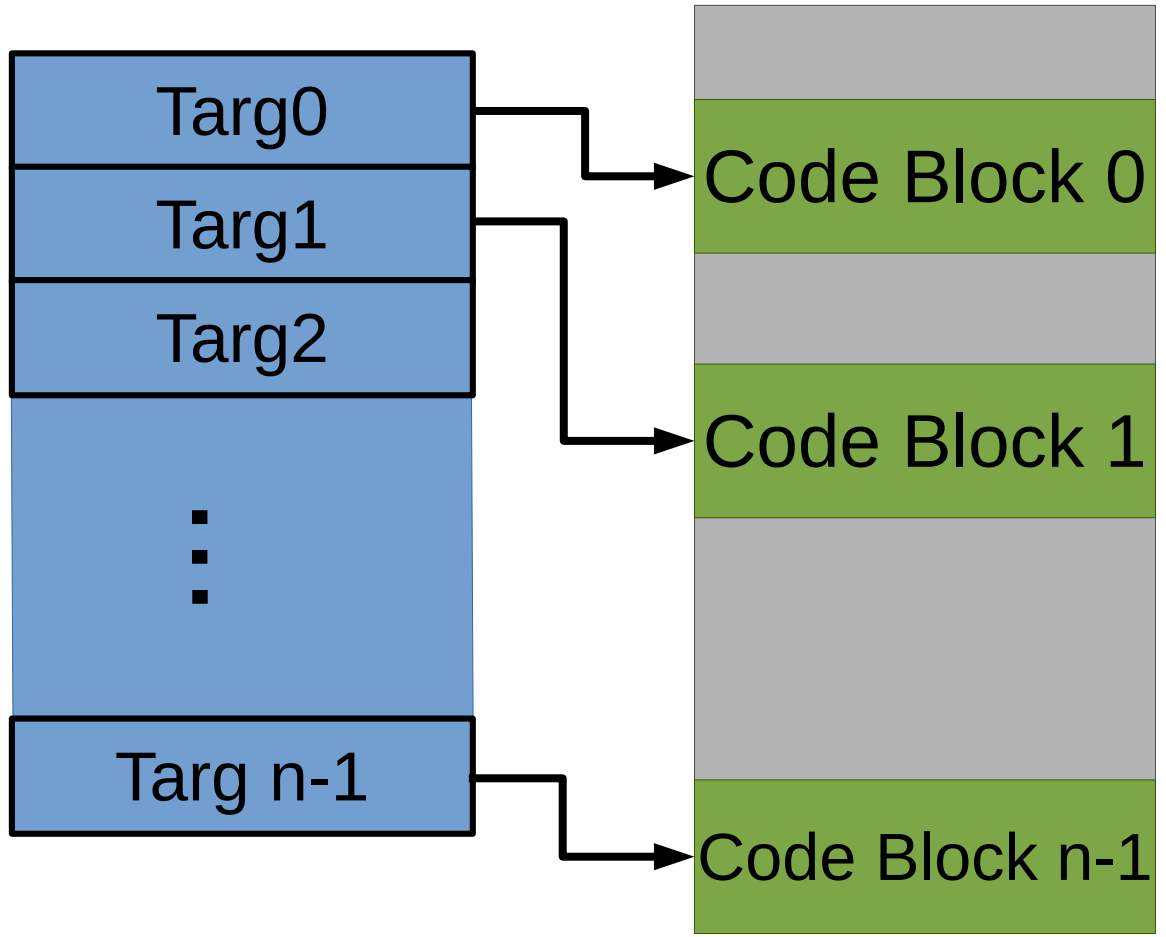


Implementing Case: Jump Table

- Use indirect jump

switch(x) ~

jmp *JTab[i]



Switch Statement example

```
switch(x) {
case 1:
   printf("One!\n");
   break;
case 2:
case 4:
   printf("Either 2 or 4!\n");
   break;
case 3:
   printf("Three!\n");
   break;
case 5:
   printf("Five!\n");
   break;
default:
   printf("Something else\n");
}
```

x is a parameter to our function

Switch Statement example

```
switch(x) {
case 1:
   printf("One!\n");
   break;
case 2:
case 4:
   printf("Either 2 or 4!\n");
   break;
case 3:
   printf("Three!\n");
   break;
case 5:
   printf("Five!\n");
   break;
default:
   printf("Something else\n");
}
```

Notice: multiple cases mapping to same code

```
0x0804844f <+0>:  push   %ebp
0x08048450 <+1>:  mov    %esp,%ebp
0x08048452 <+3>:  sub    $0x18,%esp
0x08048455 <+6>:  cmpl   $0x5,0x8(%ebp)
0x08048459 <+10>: ja     0x80484a2 <switchMe+83>
0x0804845b <+12>: mov    0x8(%ebp),%eax
0x0804845e <+15>: shl    $0x2,%eax
0x08048461 <+18>: add    $0x8048570,%eax
0x08048466 <+23>: mov    (%eax),%eax
0x08048468 <+25>: jmp    *%eax
0x0804846a <+27>: movl   $0x8048540,(%esp)
0x08048471 <+34>: call   0x80482f0 <puts@plt>
0x08048476 <+39>: jmp    0x80484ae <switchMe+95>
0x08048478 <+41>: movl   $0x8048545,(%esp)
0x0804847f <+48>:  call   0x80482f0 <puts@plt>
0x08048484 <+53>: jmp    0x80484ae <switchMe+95>
0x08048486 <+55>: movl   $0x8048554,(%esp)
0x0804848d <+62>: call   0x80482f0 <puts@plt>
0x08048492 <+67>: jmp    0x80484ae <switchMe+95>
0x08048494 <+69>: movl   $0x804855b,(%esp)
0x0804849b <+76>: call   0x80482f0 <puts@plt>
0x080484a0 <+81>: jmp    0x80484ae <switchMe+95>
0x080484a2 <+83>: movl   $0x8048561,(%esp)
0x080484a9 <+90>: call   0x80482f0 <puts@plt>
0x080484ae <+95>: leave
0x080484af <+96>:  ret
```

Too much code...

```
0x08048455 <+6>:   cmpl   $0x5,0x8(%ebp)
0x08048459 <+10>:  ja     0x80484a2 <switchMe+83>
0x0804845b <+12>:  mov    0x8(%ebp),%eax
0x0804845e <+15>:  shl    $0x2,%eax
0x08048461 <+18>:  add    $0x8048570,%eax
0x08048466 <+23>:  mov    (%eax),%eax
0x08048468 <+25>:  jmp    *%eax
0x0804846a <+27>:  movl   $0x8048540,(%esp)
0x08048471 <+34>:  call   0x80482f0 <puts@plt>
0x08048476 <+39>:  jmp    0x80484ae <switchMe+95>
0x08048478 <+41>:  movl   $0x8048545,(%esp)
0x0804847f <+48>:  call   0x80482f0 <puts@plt>
0x08048484 <+53>:  jmp    0x80484ae <switchMe+95>
0x08048486 <+55>:  movl   $0x8048554,(%esp)
0x0804848d <+62>:  call   0x80482f0 <puts@plt>
0x08048492 <+67>:  jmp    0x80484ae <switchMe+95>
0x08048494 <+69>:  movl   $0x804855b,(%esp)
0x0804849b <+76>:  call   0x80482f0 <puts@plt>
0x080484a0 <+81>:  jmp    0x80484ae <switchMe+95>
0x080484a2 <+83>:  movl   $0x8048561,(%esp)
0x080484a9 <+90>:  call   0x80482f0 <puts@plt>
```

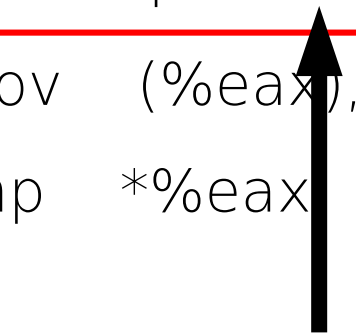
Q1: What is this?

0x08048455 <+6>: cmpl $0x5,0x8(%ebp)
0x08048459 <+10>: ja 0x80484a2 <switchMe+83>

```
0x08048455 <+6>:     cmpl   $0x5,0x8(%ebp)
0x08048459 <+10>:    ja     0x80484a2 <switchMe+83>
0x0804845b <+12>:    mov    0x8(%ebp),%eax
0x0804845e <+15>:    shl    $0x2,%eax
0x08048461 <+18>:    add    $0x8048570,%eax
0x08048466 <+23>:    mov    (%eax),%eax
0x08048468 <+25>:    jmp    *%eax
```

Do Something...

```
0x08048455 <+6>:     cmpl   $0x5,0x8(%ebp)
0x08048459 <+10>:    ja     0x80484a2 <switchMe+83>
0x0804845b <+12>:    mov    0x8(%ebp),%eax
0x0804845e <+15>:    shl    $0x2,%eax
0x08048461 <+18>:    add    $0x8048570,%eax
0x08048466 <+23>:    mov    (%eax),%eax
0x08048468 <+25>:    jmp    *%eax
```



Hmmm... this looks important

Ask GDB

```
0x08048461 <+18>: add    $0x8048570,%eax
```

(gdb) x/6w 0x8048570

```
0x8048570:  0x080484a2  0x0804846a  0x08048478  0x08048486
0x8048580:  0x08048478  0x08048494
```

- i.e. print out 6 words from memory starting at base addr 0x8048570

```
0x08048455 <+6>:   cmpl   $0x5,0x8(%ebp)
0x08048459 <+10>: ja       0x80484a2 <switchMe+83>
0x0804845b <+12>: mov    0x8(%ebp),%eax
0x0804845e <+15>: shl    $0x2,%eax
0x08048461 <+18>: add    $0x8048570,%eax
0x08048466 <+23>: mov    (%eax),%eax
0x08048468 <+25>: jmp    *%eax
0x0804846a <+27>: movl   $0x8048540,(%esp)
0x08048471 <+34>: call   0x80482f0 <puts@plt>
0x08048476 <+39>: jmp    0x80484ae <switchMe+95>
0x08048478 <+41>: movl   $0x8048545,(%esp)
0x0804847f <+48>:  call   0x80482f0 <puts@plt>
0x08048484 <+53>: jmp    0x80484ae <switchMe+95>
0x08048486 <+55>: movl   $0x8048554,(%esp)
0x0804848d <+62>: call   0x80482f0 <puts@plt>
0x08048492 <+67>: jmp    0x80484ae <switchMe+95>
0x08048494 <+69>: movl   $0x804855b,(%esp)
0x0804849b <+76>: call   0x80482f0 <puts@plt>
0x080484a0 <+81>: jmp    0x80484ae <switchMe+95>
0x080484a2 <+83>: movl   $0x8048561,(%esp)
0x080484a9 <+90>: call   0x80482f0 <puts@plt>
```

jmp *%eax

case 1

case 2, 4

case 3

case 5

default

```
0x08048455 <+6>:   cmpl   $0x5,0x8(%ebp)
0x08048459 <+10>: ja       0x80484a2 <switchMe+83>
0x0804845b <+12>: mov    0x8(%ebp),%eax
0x0804845e <+15>: shl    $0x2,%eax
0x08048461 <+18>: add    $0x8048570,%eax
0x08048466 <+23>: mov    (%eax),%eax
0x08048468 <+25>: jmp    *%eax
0x0804846a <+27>: movl   $0x8048540,(%esp)
0x08048471 <+34>: call   0x80482f0 <puts@plt>
0x08048476 <+39>: jmp    0x80484ae <switchMe+95>
0x08048478 <+41>: movl   $0x8048545,(%esp)
0x0804847f <+48>:  call   0x80482f0 <puts@plt>
0x08048484 <+53>: jmp    0x80484ae <switchMe+95>
0x08048486 <+55>: movl   $0x8048554,(%esp)
0x0804848d <+62>: call   0x80482f0 <puts@plt>
0x08048492 <+67>: jmp    0x80484ae <switchMe+95>
0x08048494 <+69>: movl   $0x804855b,(%esp)
0x0804849b <+76>: call   0x80482f0 <puts@plt>
0x080484a0 <+81>: jmp    0x80484ae <switchMe+95>
0x080484a2 <+83>: movl   $0x8048561,(%esp)
0x080484a9 <+90>: call   0x80482f0 <puts@plt>
```

jmp *%eax

```
0x08048455 <+6>:   cmpl   $0x5,0x8(%ebp)
0x08048459 <+10>:  ja     0x80484a2 <switchMe+83>
0x0804845b <+12>:  mov    0x8(%ebp),%eax
0x0804845e <+15>:  shl    $0x2,%eax
0x08048461 <+18>:  add    $0x8048570,%eax
0x08048466 <+23>:  mov    (%eax),%eax
0x08048468 <+25>:  jmp    *%eax
0x0804846a <+27>:  movl   $0x8048540,(%esp)
0x08048471 <+34>:  call   0x80482f0 <puts@plt>
0x08048476 <+39>:  jmp    0x80484ae <switchMe+95>
0x08048478 <+41>:  movl   $0x8048545,(%esp)
0x0804847f <+48>:  call   0x80482f0 <puts@plt>
0x08048484 <+53>:  jmp    0x80484ae <switchMe+95>
0x08048486 <+55>:  movl   $0x8048554,(%esp)
0x0804848d <+62>:  call   0x80482f0 <puts@plt>
0x08048492 <+67>:  jmp    0x80484ae <switchMe+95>
0x08048494 <+69>:  movl   $0x804855b,(%esp)
0x0804849b <+76>:  call   0x80482f0 <puts@plt>
0x080484a0 <+81>:  jmp    0x80484ae <switchMe+95>
0x080484a2 <+83>:  movl   $0x8048561,(%esp)
0x080484a9 <+90>:  call   0x80482f0 <puts@plt>
```

jmp *%eax

Break statements

A final note on stack frames

Q2: Why bother learning about the Stack Discipline?

Gets

```
char* gets(char* s) {
    int c = getchar();
    while (c != EOF && c != '\n' {
        c = getchar();
        *s++ = c;
    }
   *s++ = '\0';
    return s;
}
```

- Is there a problem here??? (HINT: yes)

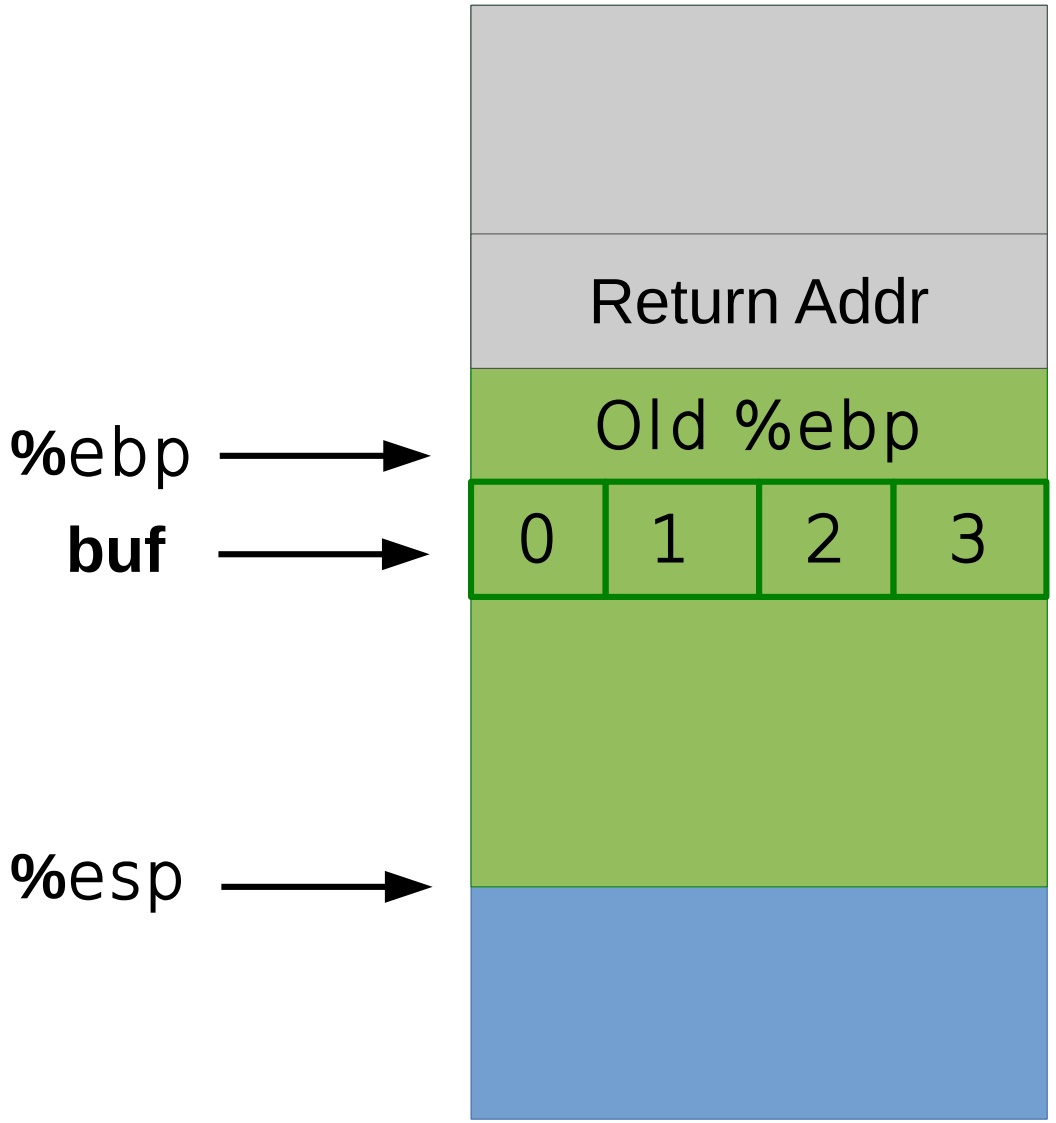
A call to gets

```
void echo {
    char buf [4];
    gets(buf); // prompt user for input
    puts(buf); // echos the user input to stdout
}
```

- What are the consequences to a call to echo when the user is irresponsible?

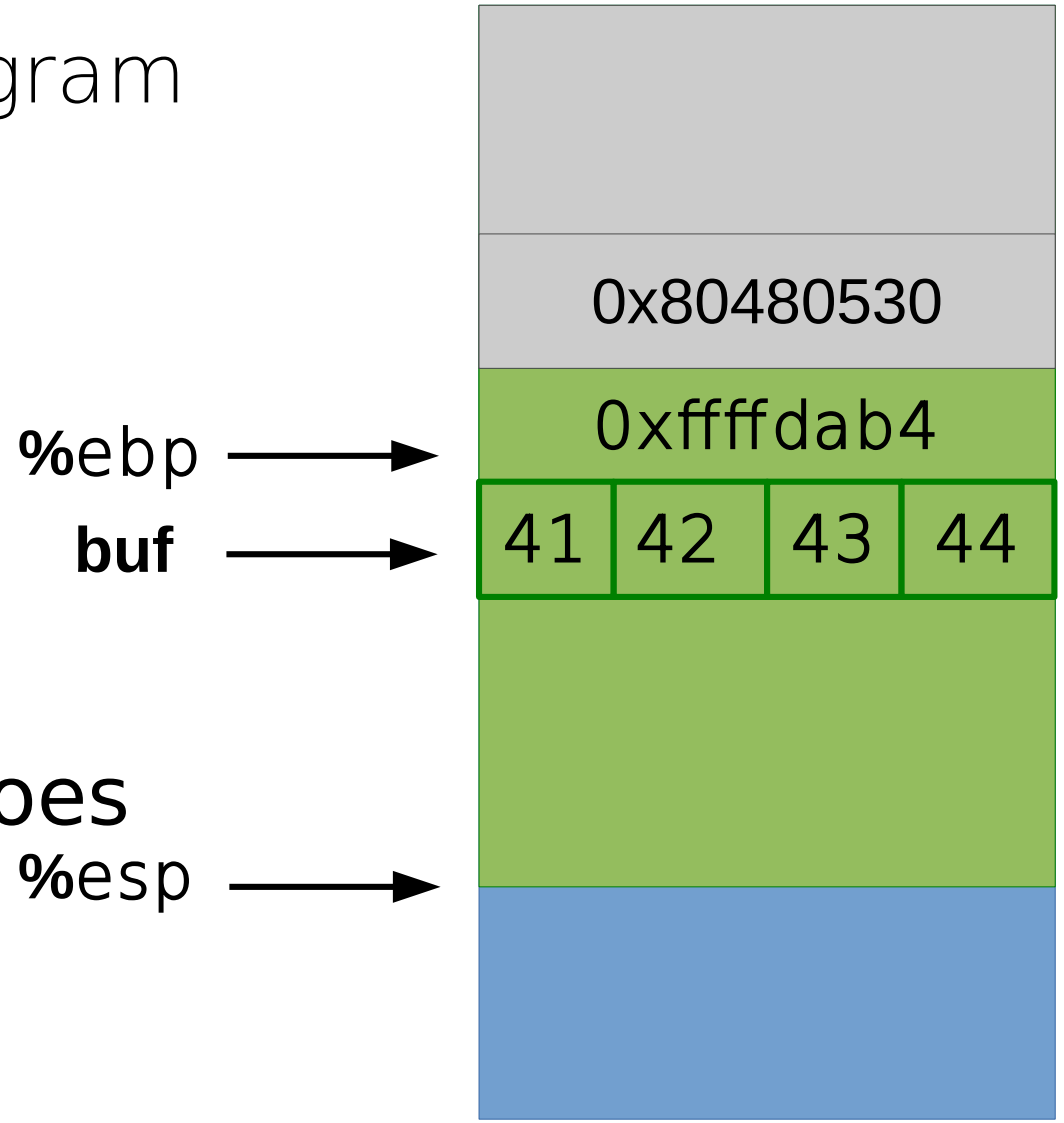
The echo Stack Frame

- See any problems???



Case 1: A Well Behaved User :-)

```
bash ~>  ./myProgram
ABCD
p $ebp = 0xffffda90
x $ebp = 0xffffdab4
p $esp = 0xffffda80
```



- What if our user does **not** behave?

Case 2: A Stupid User <:-}

bash ~> ./myProgram

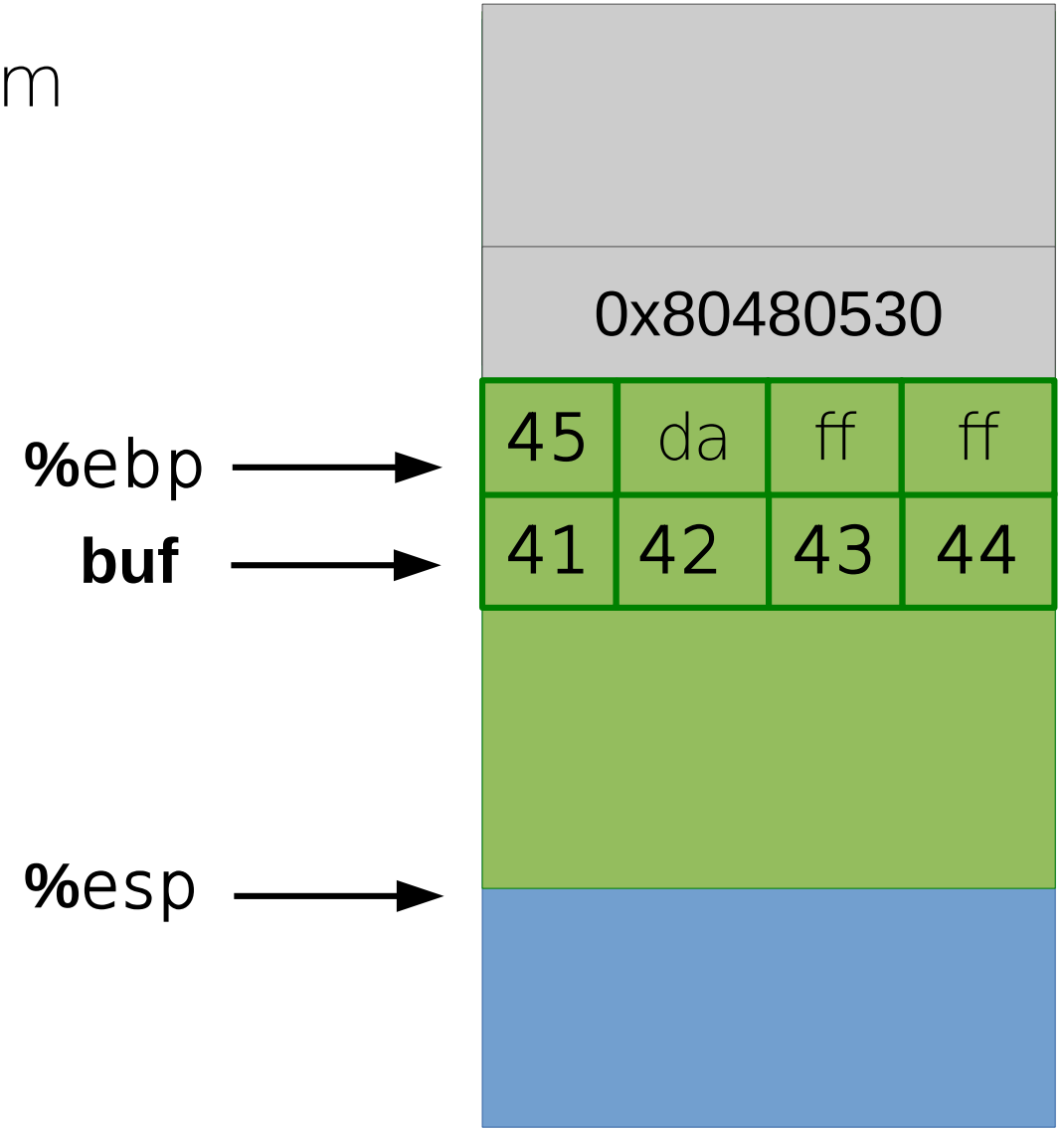
ABCDE

p $ebp = 0xffffda90

x $ebp = **0xffffda45**

p $esp = 0xffffda80

- What happened?
- What are the consequences?



Seg Fault! :'(

- Old ebp no longer valid (maybe)
- On return, we set ebp = old ebp
- Bad things happen when we access garbage addresses
  - (%ebp) , 0x8(%ebp), -0x4(%ebp) , etc

SOOO... IS THIS AS BAD AS IT GETS?

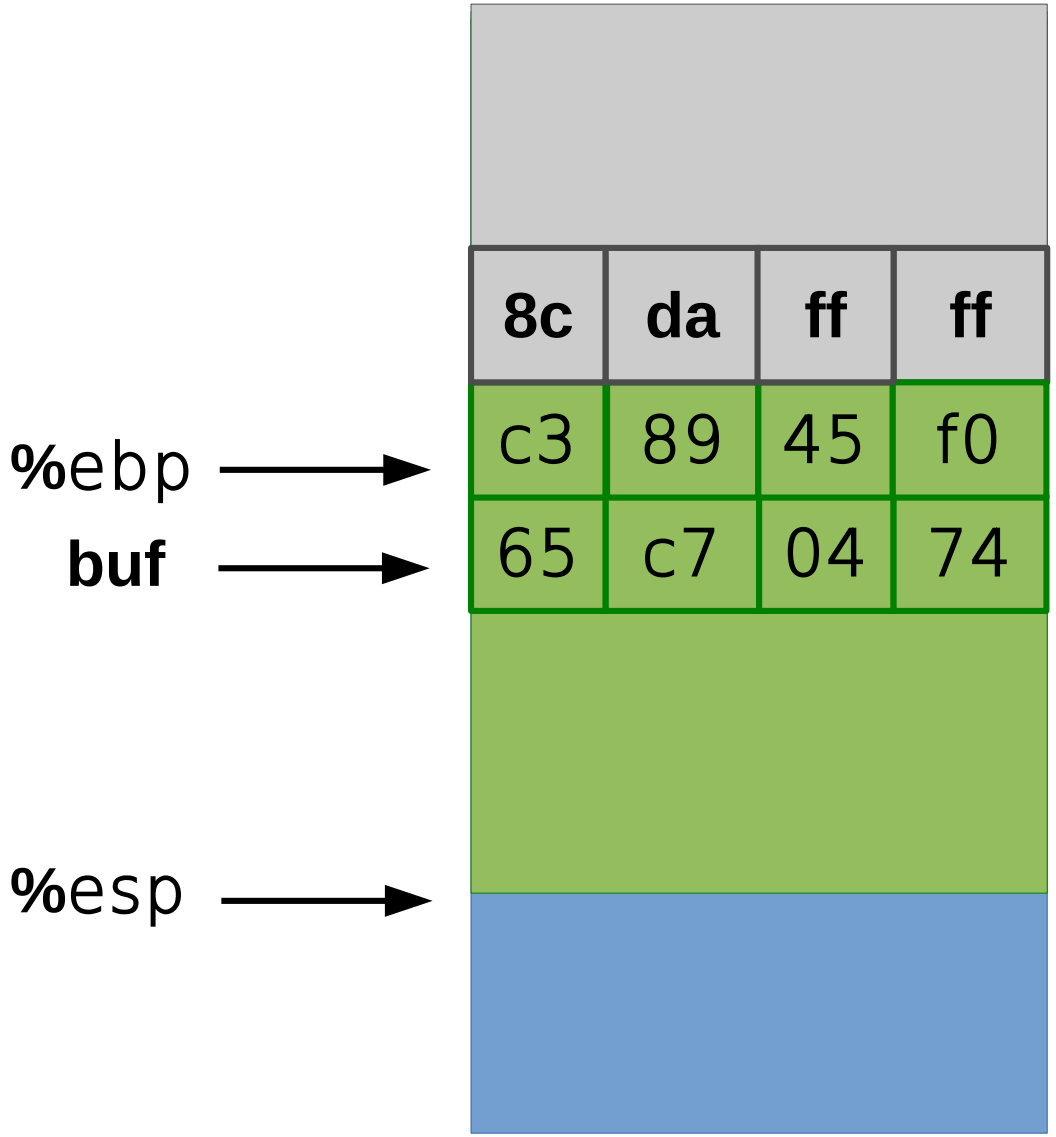
Case 3: An Evil User >:-)

p $ebp = 0xffffda90

x $ebp = 0xf04589c3

p $esp = 0xffffda80

- **What just happened?**



Stack Smashing

- Evil User can set arbitrary Return Addr
- Can even point **somewhere else in stack**
  - e.g even &buf
  - Can plant evil code as part as input string

Buffer Overflow Attack

Fixed-sized Stack Buffer +

Known frame Offset +

Naive gets Function

= BAD News

- The “Internet Worm”, the AOL/MSN Messaging War (1999)

Buffer Overflow Prevention

- Stack Randomization
  - Defeated by “Nop sled”
- Stack corruption detection
  - e.g. using “Canary”
- Require Executable Permissions
- Don't use gets!!!

CHAPTER 5: CODE OPTIMIZATION

But first, some notes about micro-architecture...

But first, lets talk about laundry...

Wait...what?

- Time to complete:
  - 1 Load in washer: 45 min
  - 1 Load in dryer: 60min
  - Folding 1 Load: 20min
- Bob needs to complete 2 loads of laundry before his parents come over in 3 hours 30 min
- Can it be done?

Naive Solution

Time	1	2	3	4	5	6
WASH	L1			L2		
DRY		L1			L2	
FOLD			L1			L2

TOTAL TIME TO COMPLETE = ?

Naive Solution

Time	1	2	3	4	5	6
WASH	L1			L2		
DRY		L1			L2	
FOLD			L1			L2

TOTAL TIME TO COMPLETE = 2*(45 + 60 + 20) = 250
= **4 hrs, 10 min**

Can we do better?

Bob's Laundry Pipeline

Time	1	2	3	4
WASH	L1	L2		
DRY		L1	L2	
FOLD			L1	L2

TOTAL TIME TO COMPLETE = ?

Bob's Laundry Pipeline

Time	1	2	3	4
WASH	L1	L2		
DRY		L1	L2	
FOLD			L1	L2

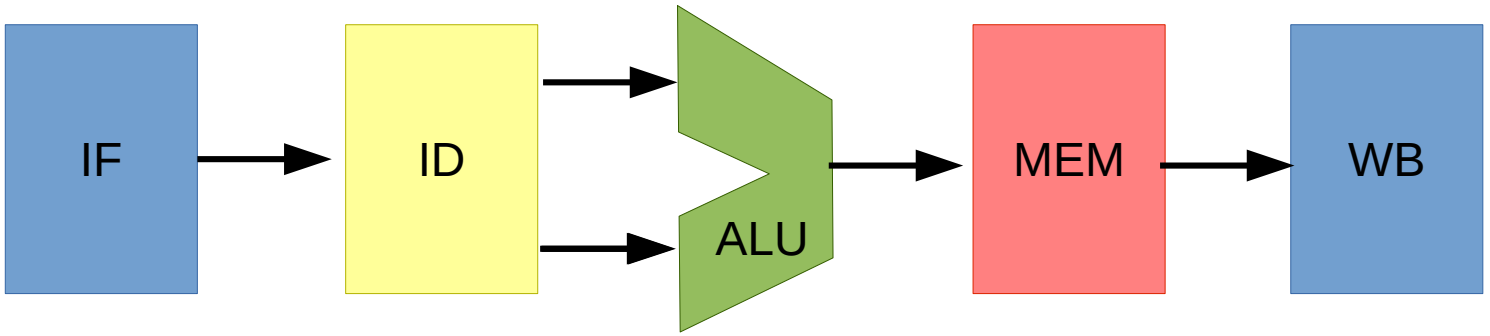
TOTAL TIME TO COMPLETE = 45 + 60 + 60 + 20 = 180
= **3 hrs, 5 min**

Definition

- **Latency**: the time delay between the input and output

The processor model

- Instruction fetch, Decode stage, Execution, Memory access, Write back



- How to translate assembly instructions?

Instruction Execution Example 1

- addl %eax, %ebx
  - IF: IR = Fetch[eip] eip = next eip
  - ID: Decode[IR]: A <= Reg[eax], B <= Reg[ebx]
  - EX: Sum <= A + B
  - MEM: do nothing
  - WB: Reg[ebx] = Sum

*Example from Peng Wei's slides

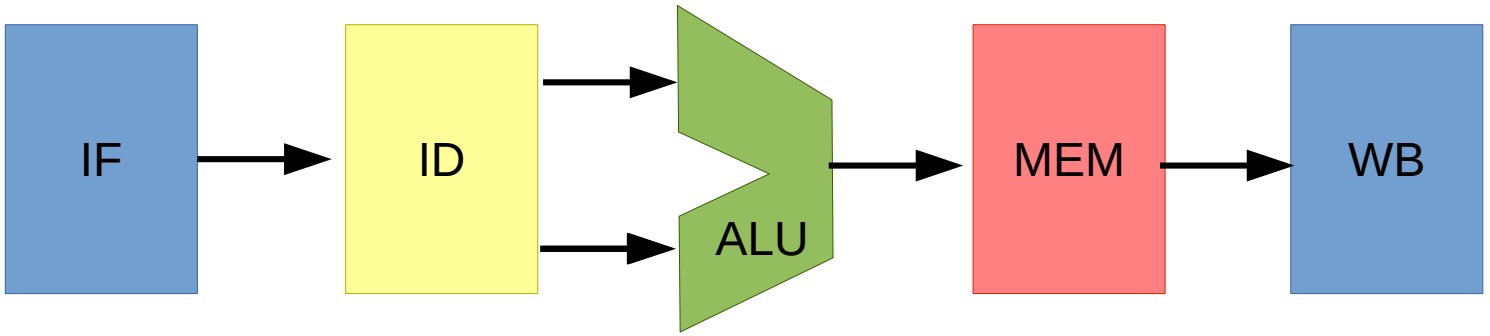
Instruction Execution Example 2

- movl 12(%eax), %ebx
  - IF: IR = Fetch[eip], eip = next eip
  - ID: Decode[IR], A <= Reg[eax], B <= 12
  - EX: Addr <= A + B
  - MEM: Rd = Mem[Addr]
  - WB: Reg[ebx] = Rd

*Example from Peng Wei's slides

Single Cycle Processor

- One instruction executes per cycle

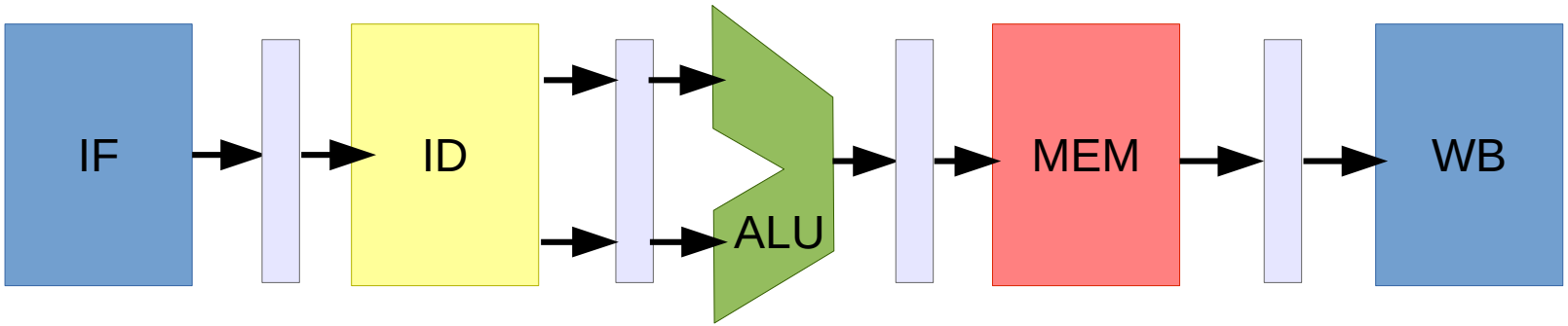


Latency: 200ps 150ps 400ps 350ps 200ps

Minimum Cycle time = Σ latency = 1300 ps
Maxium clk frequency = 1/T = 770 MHz

Pipelined Processor

- Latch after each stage

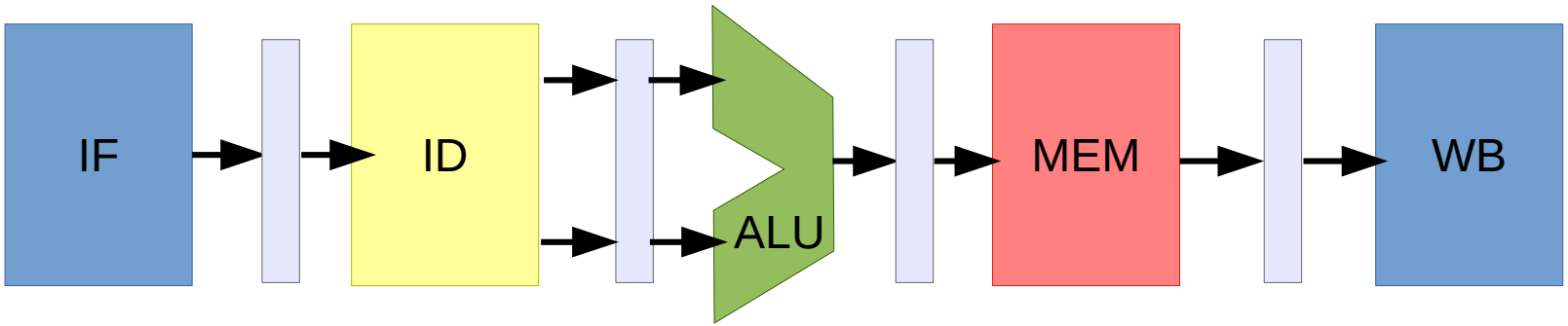


Latency: 200ps 150ps 400ps 350ps 200ps

Minimum Cycle time = ?
Maxium clk frequency = ?

Pipelined Processor

- Latch after each stage



Latency: 200ps 150ps 400ps 350ps 200ps

Minimum Cycle time = max(latency) = 400 ps
Maxium clk frequency = 1/T = 2.5 GHz

Pipeline Hazard

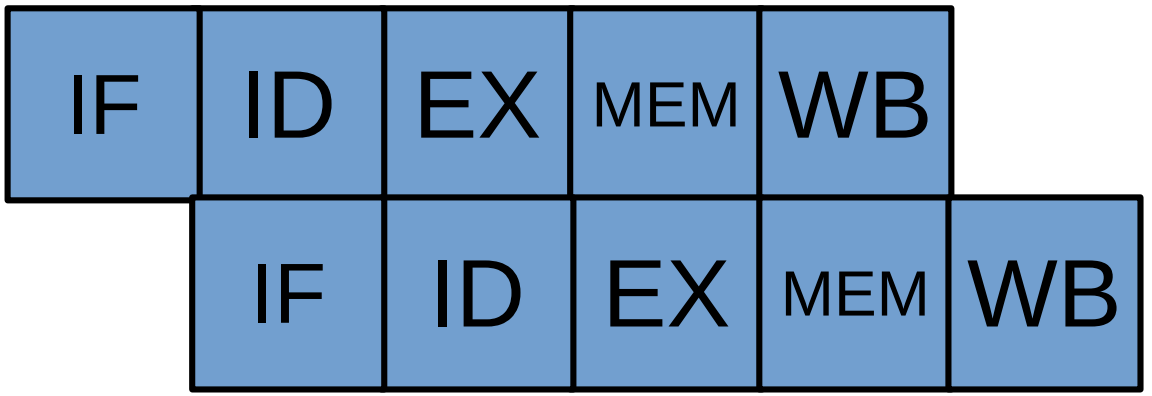
```
movl 4(%ecx), %edx
addl %edx, %eax
```

*Example from Peng Wei's slides

Pipeline Hazard

movl 4(%ecx), %edx

addl %edx, %eax



Pipeline Hazard

movl 4(%ecx), %edx

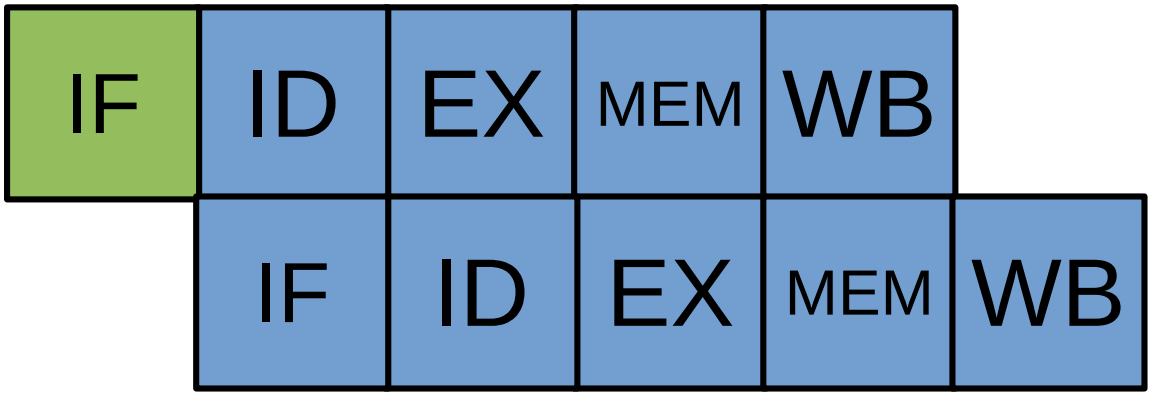
addl %edx, %eax

Suppose:

%edx = 10

%eax = 5

4(%ecx) = 95



- What should be the result?

Pipeline Hazard

movl 4(%ecx), %edx

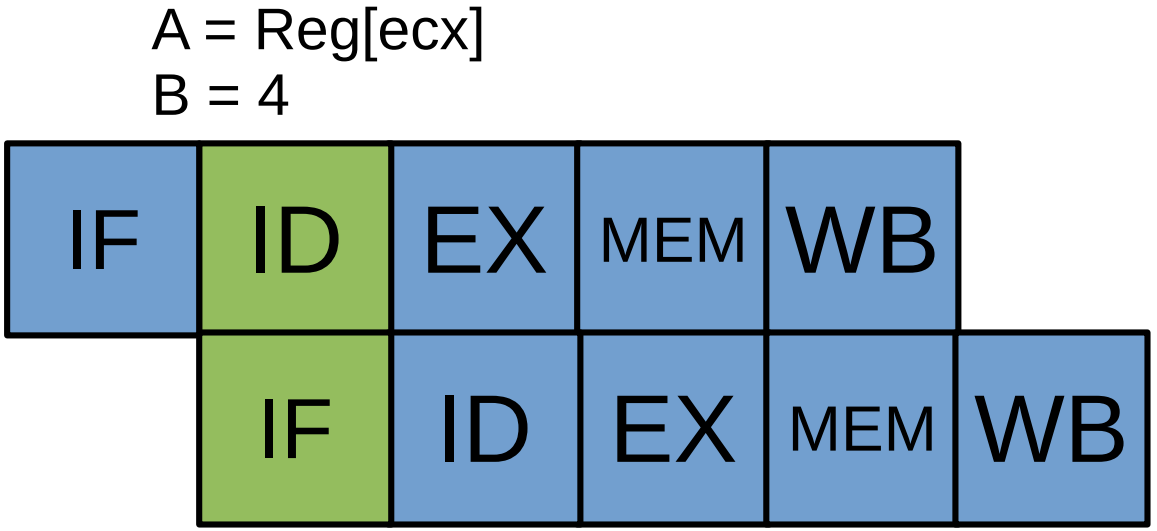
addl %edx, %eax

Suppose:

%edx = 10

%eax = 5

4(%ecx) = 95



Pipeline Hazard

movl 4(%ecx), %edx

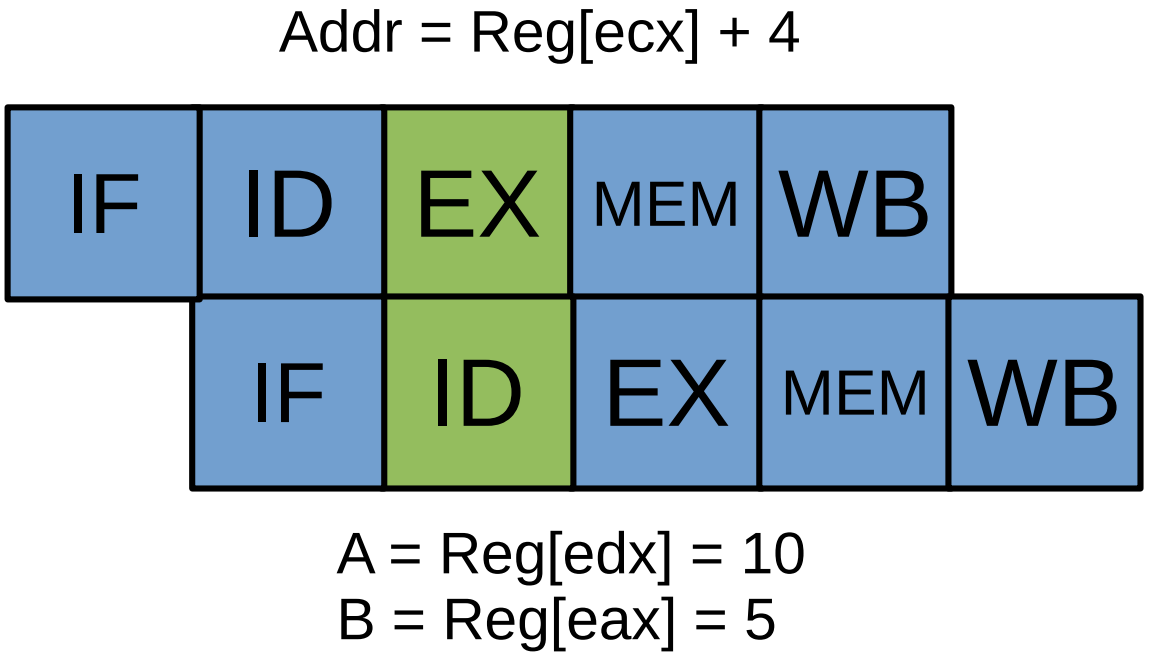
addl %edx, %eax

Suppose:

%edx = 10

%eax = 5

4(%ecx) = 95



Pipeline Hazard

movl 4(%ecx), %edx

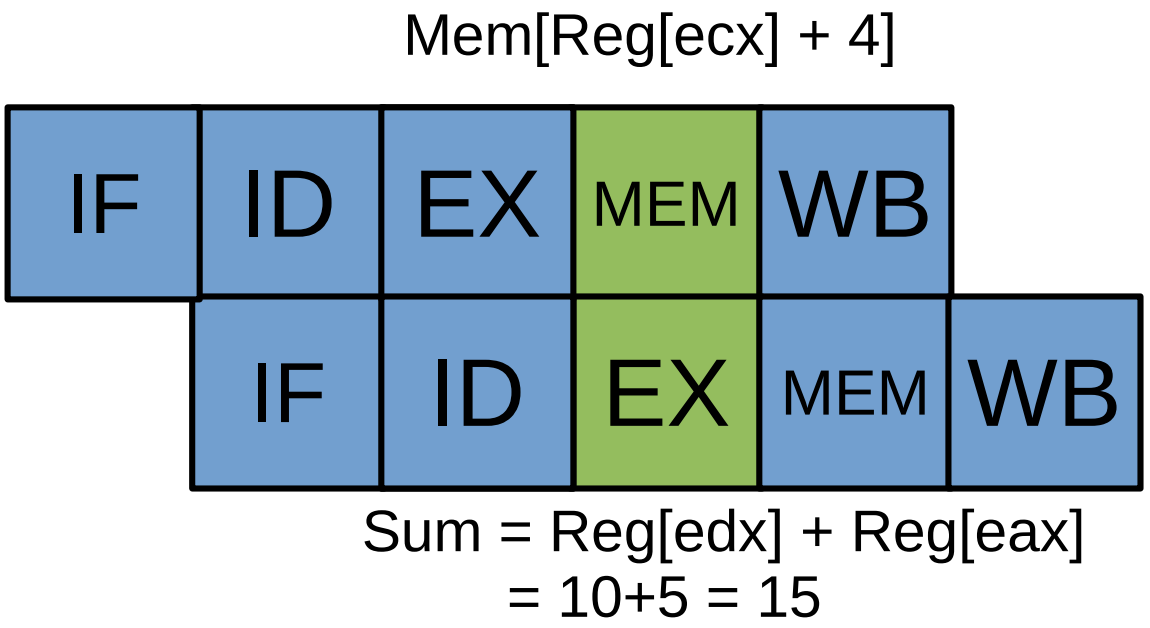
addl %edx, %eax

Suppose:

%edx = 10

%eax = 5

4(%ecx) = 95



Pipeline Hazard

movl 4(%ecx), %edx

addl %edx, %eax

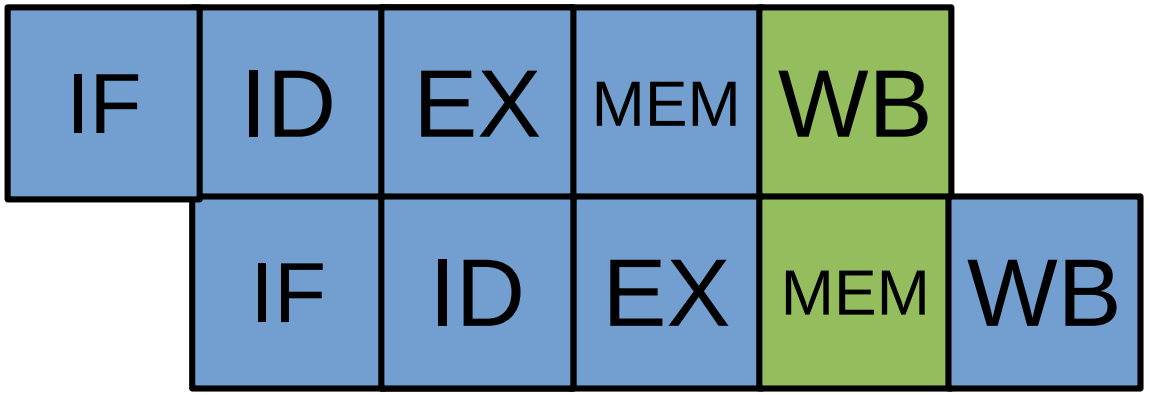
Reg[edx] = Mem[Reg[ecx] + 4]
= 95

Suppose:

%edx = 10

%eax = 5

4(%ecx) = 95



Do nothing

Pipeline Hazard

movl 4(%ecx), %edx

addl %edx, %eax

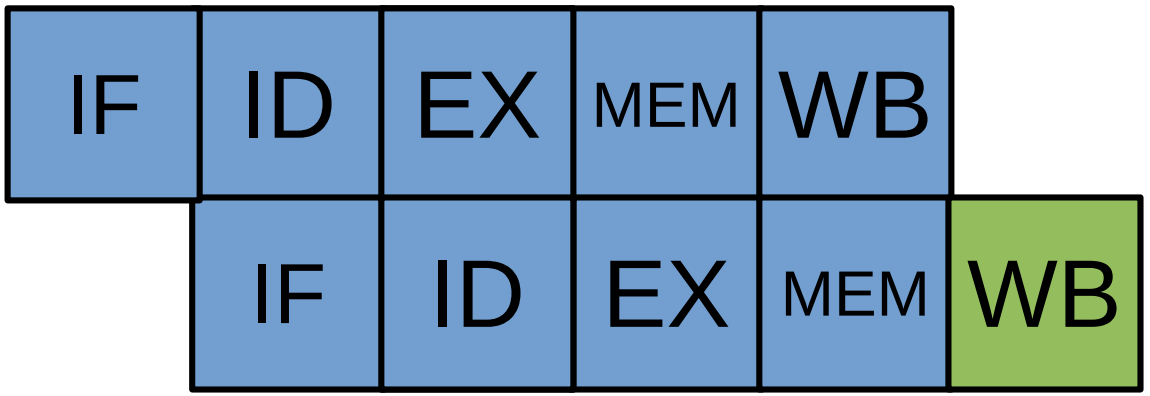
Suppose:

%edx = 10

%eax = 5

4(%ecx) = 95

Reg[edx] = 95



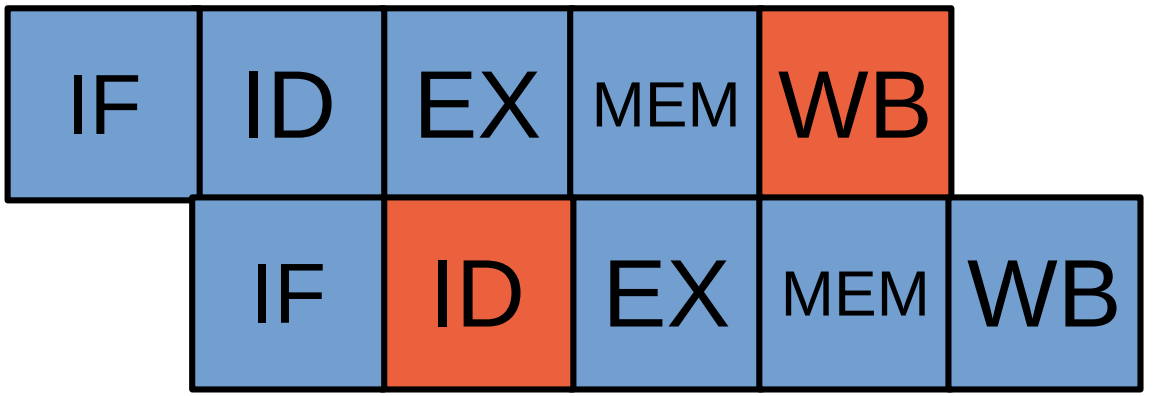
Reg[eax] = Reg[edx]+Reg[eax]
= 15

Pipeline Hazard: Read after write

movl 4(%ecx), %**edx**

addl %**edx**, %eax

- We need a value that will not be ready for 2 clk cycles



LOOP UNROLLING

Simple Loop

```
for (i=0; i<100; i++) {

   A[i] += 7;

}

Loop:

   mov (%eax), %ebx

   add $7, %ebx

   mov %ebx, (%eax)

   add $4, %eax

   cmp $400, %eax

   jne Loop
```

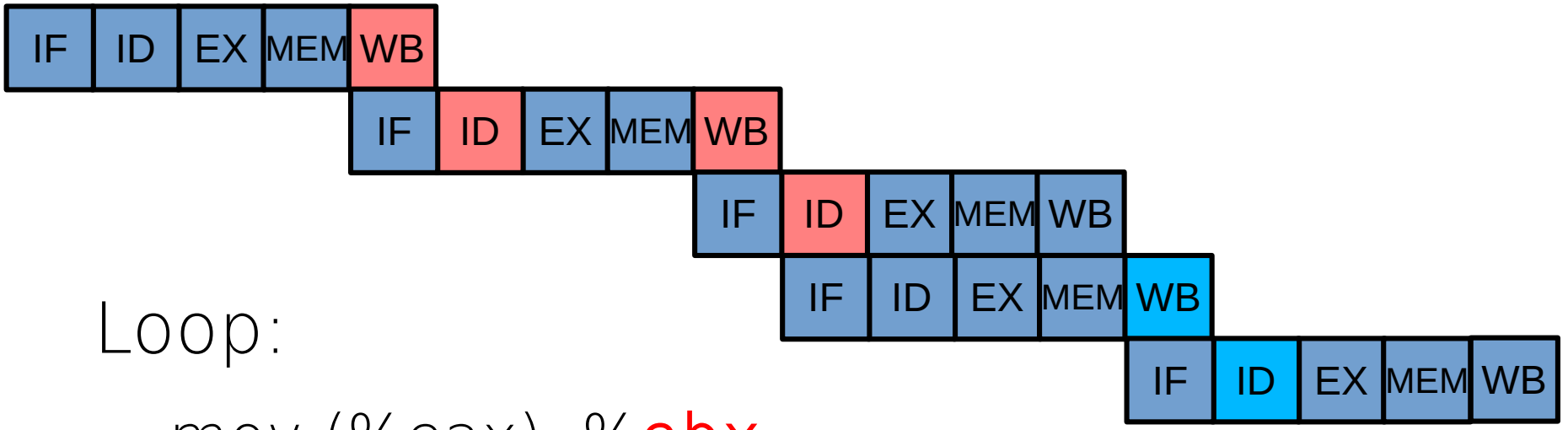
*Example from Peng Wei's slides

Simple Loop

```
for (i=0; i<100; i++) {

   A[i] += 7;

}

Loop:

   mov (%eax), %ebx
   add $7, %ebx
   mov %ebx, (%eax)
   add $4, %eax
   cmp $400, %eax
   jne Loop
```

Read after **Write** dependencies

Simple Loop, not so simple...



Loop:

mov (%eax), %ebx

add $7, %ebx

mov %ebx, (%eax)

add $4, %eax

cmp $400, %eax

jne Loop

Bad for CPE

Unrolled Loop

```
for (i=0; i<100; i+= 2) {

   A[i] += 7;
   A[i+1] += 7
}
```

- Not including special handling for corner cases

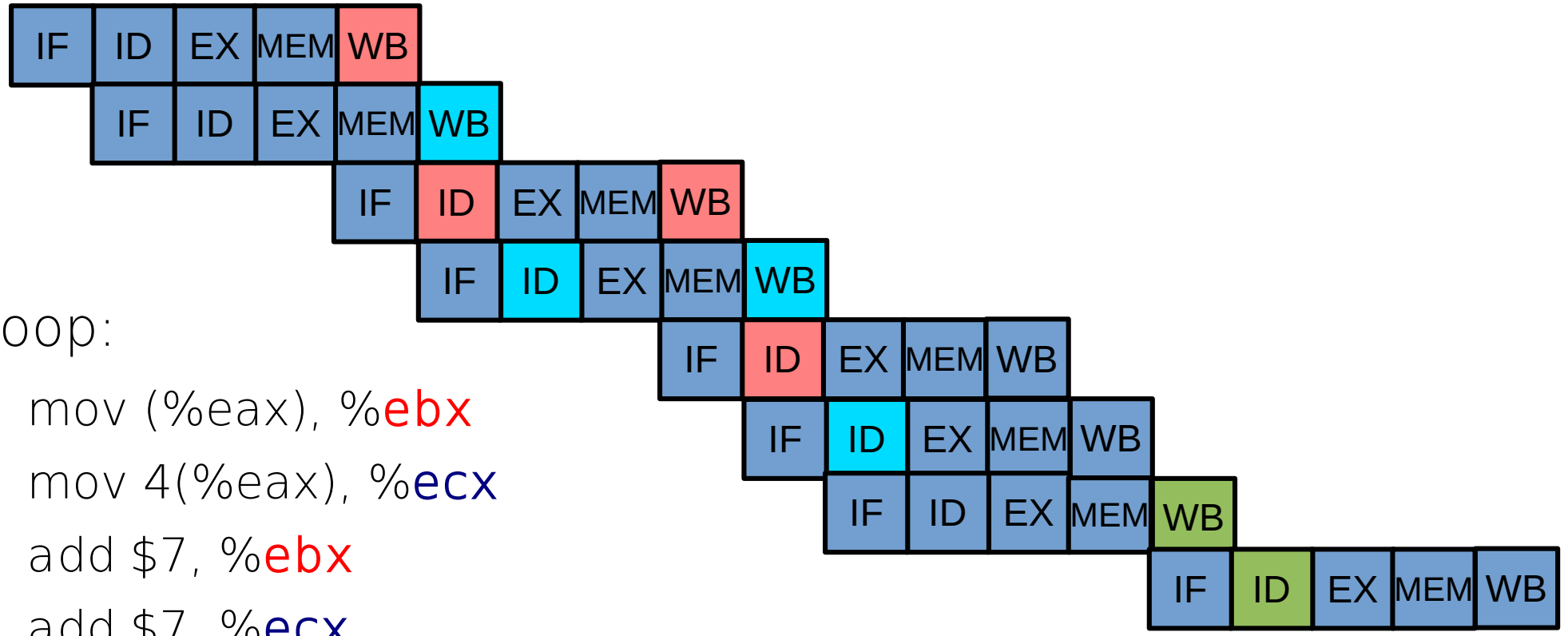
Unrolled Loop

```
Loop:
    mov (%eax), %ebx
    mov 4(%eax), %ecx
    add $7, %ebx
    add $7, %ecx
    mov %ebx, (%eax)
    mov %ecx, 4(%eax)
    add $8, %eax
    cmp $400, %eax
    jne Loop
```

Now what about **Read** after **Write** dependencies?

Unrolled Loop

```
Loop:
    mov (%eax), %ebx
    mov 4(%eax), %ecx
    add $7, %ebx
    add $7, %ecx
    mov %ebx, (%eax)
    mov %ecx, 4(%eax)
    add $8, %eax
    cmp $400, %eax
    jne Loop
```




```
Loop:
    mov (%eax), %ebx
    mov 4(%eax), %ecx
    add $7, %ebx
    add $7, %ecx
    mov %ebx, (%eax)
    mov %ecx, 4(%eax)
    add $8, %eax
    cmp $400, %eax
    jne Loop
```

A little better...

Why We Should Unroll

- Loop control structure costs CPU cycles
  - Overhead cost may be greater than that of loop body
  - If looping a known amount, can save cycles
- Penalty for close read after write dependencies
- Conditional branches are another story...

Code Motion

- If result is the same, don't do it more than once

```
for (i=0; i<N; i++) {
    for (j = 0; j<M;j++) {
        a[M*i+j] = b[j]
    }
}
```

Code Motion

- If result is the same, don't do it more than once

```
for (i=0; i<N; i++) {
    mi = M*i;
    for (j = 0; j<M;j++) {
        a[mi+j] = b[j]
    }
}
```

- strlen( ) is **O(n)**

Strength Reductions

- Modify an expensive operation to a cheaper one
- Machine dependent

$$36*x = 32*x + 4*x$$
$$= x << 5 + x << 2$$

Reducing Procedure Calls

- What is overhead associated with procedure call?

```
int max (int a, int b) {
    return (a > b) : a ? b;
}
```

Reducing Procedure Calls

```
00000000 <max>:
   0: 55                    push   %ebp
   1: 89 e5                 mov    %esp,%ebp
   3: 8b 45 08              mov    0x8(%ebp),%eax
   6: 39 45 0c              cmp    %eax,0xc(%ebp)
   9: 0f 4d 45 0c           cmovge 0xc(%ebp),%eax
   d: 5d                    pop    %ebp
   e: c3                    ret
```

Reducing Procedure Calls

```
00000000 <max>:
   0: 55                    push   %ebp
   1: 89 e5                 mov    %esp,%ebp
   3: 8b 45 08              mov    0x8(%ebp),%eax
   6: 39 45 0c              cmp    %eax,0xc(%ebp)
   9: 0f 4d 45 0c           cmovge 0xc(%ebp),%eax
   d: 5d                    pop    %ebp
   e: c3                    ret
```

Reducing Procedure Calls

- For small functions, majority of code is stack setup—what a waste!
- Would be better if we copied n pasted text of function body on every function call

Inline functions

- Compiler is very good at this
- Two ways to indicate inline as programmer:

```
#define MAX_INT((a,b)) ((a>b)?(a):(b))
inline
int max(int a, int b) { return a>b?a:b;}
```

Reduce Memory References

```
int* max_elm(int* arr, int size) {
   int i;
   int* max = arr;
   for (i=1; i<size; i++)
   if (*(arr+i) > *max)
      max = arr+i;
   return max;
}
```

Reduce Memory References

```
int* max_elm(int* arr, int size) {
    int i;
    int* max = arr;
    for (i=1; i<size; i++)
    if (*(arr+i) > *max)
        max = arr+i;
    return max;
}
```

How many Mem References do we do?

```
15:   8b 45 f8              mov    -0x8(%ebp),%eax
18:   8d 14 85 00 00 00 00 lea    0x0(,%eax,4),%edx
1f: 8b 45 08              mov    0x8(%ebp),%eax
22:   01 d0               add    %edx,%eax
24:   8b 10               mov    (%eax),%edx
26:   8b 45 fc            mov    -0x4(%ebp),%eax
29:   8b 00               mov    (%eax),%eax
2b:   39 c2               cmp    %eax,%edx
2d:   7e 12               jle    41 <max_elm+0x41>
2f: 8b 45 f8              mov    -0x8(%ebp),%eax
32:   8d 14 85 00 00 00 00 lea    0x0(,%eax,4),%edx
39:   8b 45 08            mov    0x8(%ebp),%eax
3c:01 d0                  add    %edx,%eax
3e:89 45 fc               mov    %eax,-0x4(%ebp)
41:   83 45 f8 01         addl   $0x1,-0x8(%ebp)
45:   8b 45 f8            mov    -0x8(%ebp),%eax
48:   3b 45 0c            cmp    0xc(%ebp),%eax
4b:   7c c8               jl     15 <max_elm+0x15>
```

```
15:   8b 45 f8            mov    -0x8(%ebp),%eax
18:   8d 14 85 00 00 00 00 lea    0x0(,%eax,4),%edx
1f: 8b 45 08              mov    0x8(%ebp),%eax
22:   01 d0               add    %edx,%eax
24:   8b 10               mov    (%eax),%edx
26:   8b 45 fc            mov    -0x4(%ebp),%eax
29:   8b 00               mov    (%eax),%eax
2b:   39 c2               cmp    %eax,%edx
2d:   7e 12               jle    41 <max_elm+0x41>
2f: 8b 45 f8              mov    -0x8(%ebp),%eax
32:   8d 14 85 00 00 00 00 lea    0x0(,%eax,4),%edx
39:   8b 45 08            mov    0x8(%ebp),%eax
3c:01 d0                  add    %edx,%eax
3e:89 45 fc               mov    %eax,-0x4(%ebp)
41:   83 45 f8 01         addl   $0x1,-0x8(%ebp)
45:   8b 45 f8            mov    -0x8(%ebp),%eax
48:   3b 45 0c            cmp    0xc(%ebp),%eax
4b:   7c c8               jl     15 <max_elm+0x15>
```

*max

Reduce Memory References

```
int* max_elm(int* arr, int size) {
  int i, max, curr;
  int* max_ptr;
  for (i=1; i<size; i++) {
    curr = *(arr+i);
    if (curr > max) {
      max_ptr = arr+i;
      max = curr;
    }
  return max_ptr;
}
```

Q: How many Mem References do we save?