

Today

- Finish up the ISA design example
 - Memory
 - Large constants
 - Functions
- Questions about project.
- x86 assembly overview.
- Immediately after class: |4|L lab hours in b240

Accessing Memory

- Load and store instructions should be the only instructions that access memory
- Loads in MIPS
 - `lw r1, offset(r2)` -> $R[rt] = \text{mem}[R[rs] + \text{imm}]$
- Stores in MIPS
 - `sw r1, offset(r2)` -> $\text{mem}[R[rs] + \text{imm}] = R[rt]$
- Does it makes sense that `rt` is an input to `sw` and an output of `lw`?

Large Constants

- Some constants are too big for the immediate field.
 - example: Create 0xDEADBEEF -- 32 bit values
- MIPS -- 16 bit immediate
 - add r1, zero, 0xDEAD
 - sll r1, r1, 16
 - ori r1, r1, 0xBEEF
- Alternative:
 - Assembly: LoadConst r1, 0xDEADBEEF
 - RTL: $R[r1] = \text{mem}[PC+4]; PC = PC + 8.$

Large Constants

- Some constants are too big for the immediate field.
 - example: Create 0xDEADBEEF -- 32 bit values
- MIPS -- 16 bit immediate
 - add r1, zero, 0xDEAD
 - sll r1, r1, 16
 - ori r1, r1, 0xBEEF
- Alternative:
 - Assembly: LoadConst r1, 0xDEADBEEF
 - RTL: $R[r1] = \text{mem}[PC+4]; PC = PC + 8.$

Is this a good idea?
Look on the next slide



Uniformity in MIPS

- 3 instruction formats: I, R, and J.
 - R-type: Register-register Arithmetic
 - I-type: immediate arithmetic; loads/stores
 - J-type: Non-conditional, non-relative branches
 - opcodes are always in the same place
 - rs and rt are always in the same place
 - The immediate is always in the same place
- Similar amounts of work per instruction
 - 1 read from instruction memory
 - ≤ 1 arithmetic operations
 - ≤ 2 register reads
 - ≤ 1 register write
 - ≤ 1 data store/load
- Fixed instruction length
- Relatively large register file: 32
- Reasonably large immediate field: 16 bits
- Wise use of opcode space
 - 6 bits of opcode
 - I-type gets another 6 bits of “function”

Uniformity in MIPS

- 3 instruction formats: I, R, and J.
 - R-type: Register-register Arithmetic
 - I-type: immediate arithmetic; loads/stores
 - J-type: Non-conditional, non-relative branches
 - opcodes are always in the same place
 - rs and rt are always in the same place
 - The immediate is always in the same place
 - Similar amounts of work per instruction
 - 1 read from instruction memory
 - ≤ 1 arithmetic operations
 - ≤ 2 register reads
 - ≤ 1 register write
 - ≤ 1 data store/load
 - Fixed instruction length
 - Relatively large register file: 32
 - Reasonably large immediate field: 16 bits
 - Wise use of opcode space
 - 6 bits of opcode
 - I-type gets another 6 bits of “function”
- $R[r] = \text{mem}[PC+4]$
breaks this uniformity
-

Supporting Function Calls

- Functions are an essential feature of modern languages

- What does a function need?

- Arguments.
- Storage for local variables.
- To return control to the caller.
- To execute regardless of who called it.
- To call other functions (that call other functions...that call other functions)

```
int Factorial(int x) {  
    if (x == 0)  
        return 1;  
    else  
        return x * Factorial(x - 1);  
}
```

- There are not *instructions* for this

- It is a contract about how the function behaves
- In particular, how it treats the resources that are shared between functions -- the registers and memory

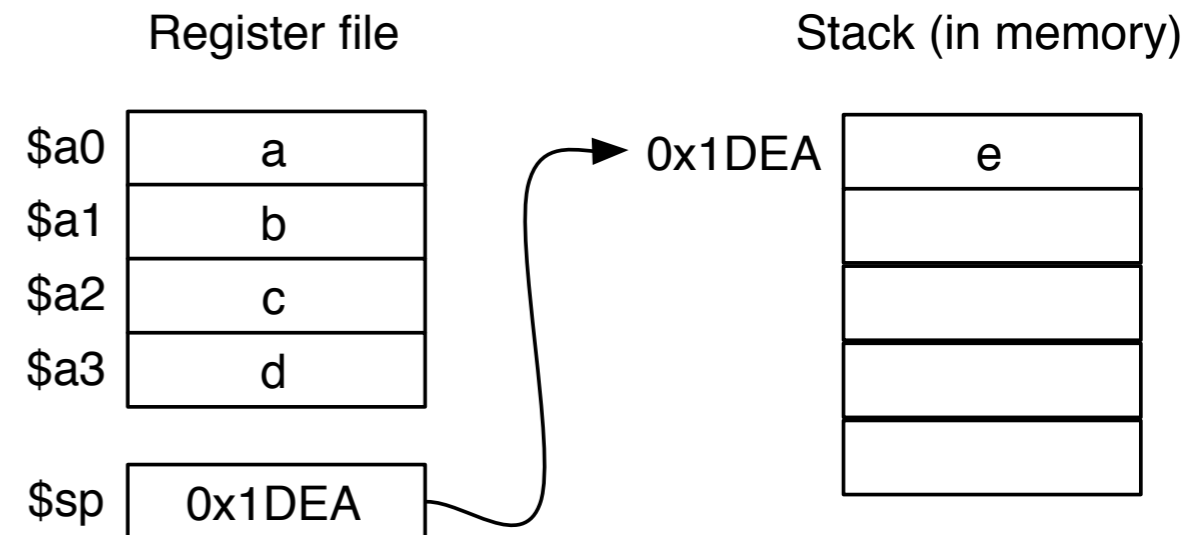
Register Discipline

- All registers are the same, but we assign them different uses.

Name	number	use	saved?
\$zero	0	zero	n/a
\$v0-\$v1	2-3	return value	no
\$a0-\$a3	4-7	arguments	no
\$t0-\$t7	8-15	temporaries	no
\$s0-\$s7	26-23	saved	yes
\$t8-\$t9	24-25	temporaries	no
\$gp	26	global ptr	yes
\$sp	29	stack ptr	yes
\$fp	30	frame ptr	yes
\$ra	31	return address	yes

Arguments

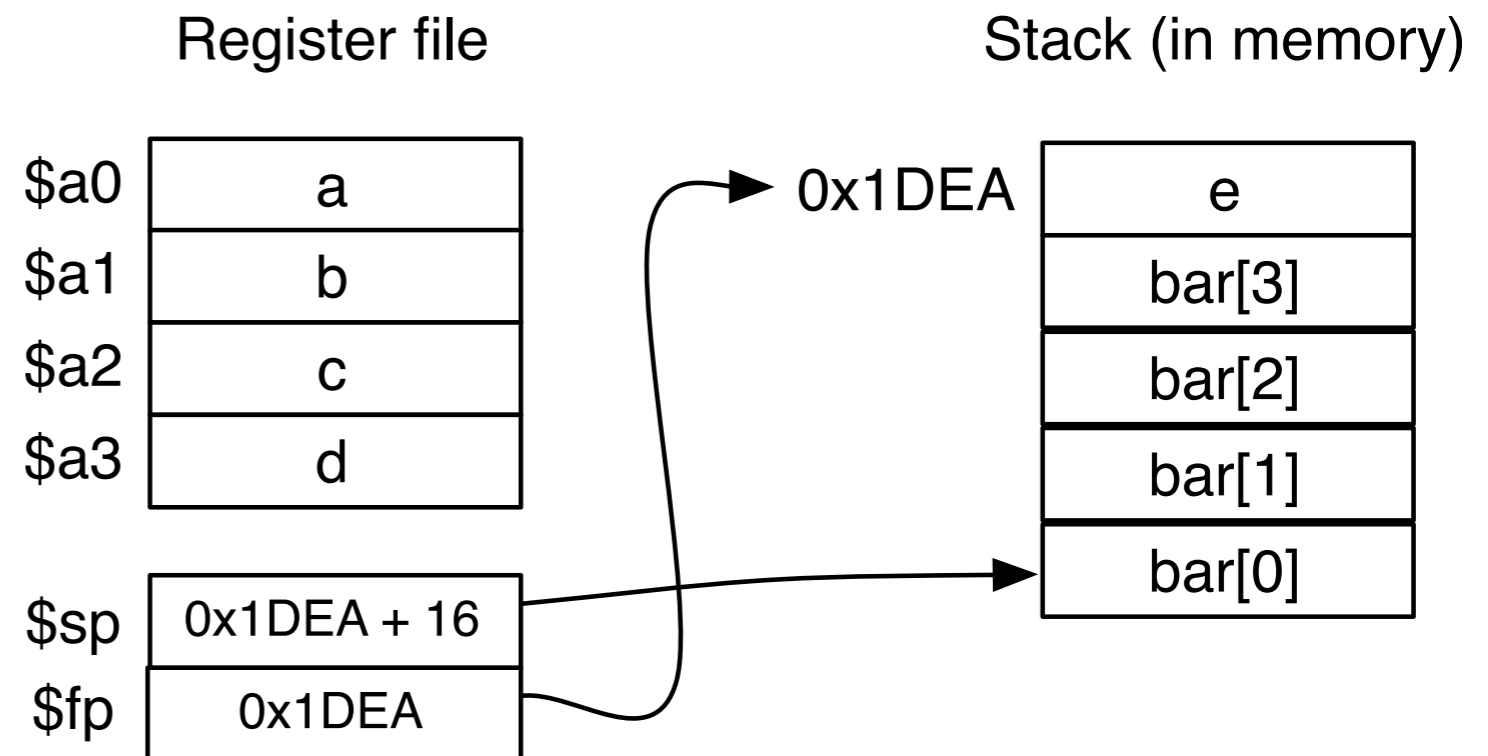
- How many arguments can function have?
 - unbounded.
 - But most functions have just a few.
- Make the common case fast
 - Put the first 4 argument in registers (\$a0-\$a3).
 - Put the rest on the “stack”



```
int Foo(int a, int b, int c, int d, int e) {  
    ...  
}
```

Storage for Local Variables

- Local variables go on the stack too.
 - `$fp` -- frame pointer (points to base of this frame)
 - `$sp` -- stack pointer



```
int Foo(int a, int b, int c, int d, int e) {  
    int bar[4];  
    ...  
}
```

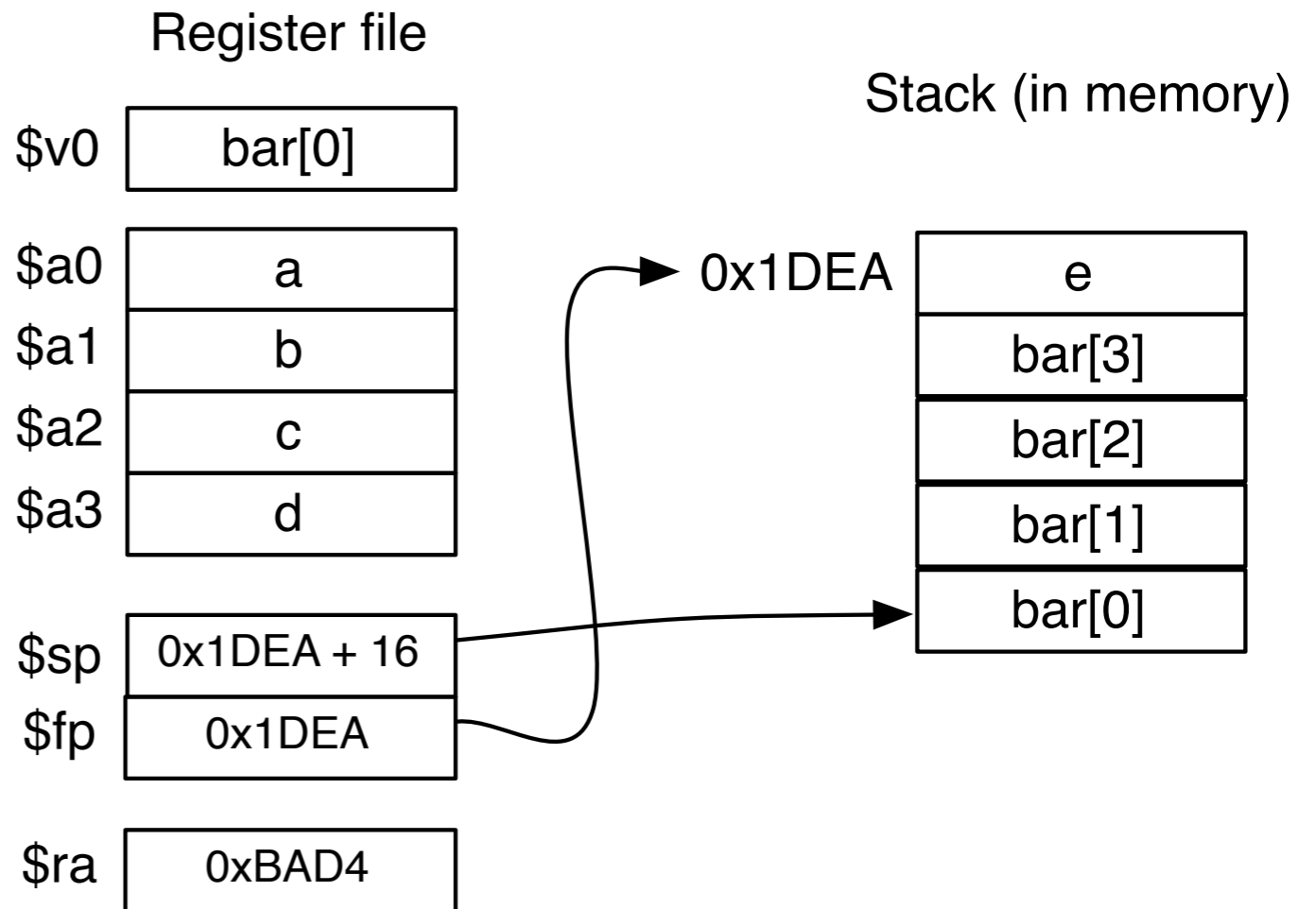
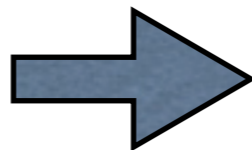
Returning Control

Caller

```
...  
move $a0, $t1  
move $a1, $s4  
move $a2, $s3  
move $a3, $s3  
sw   $t2, 0($sp)  
0xBAD0: jal   Foo
```

Callee

```
int Foo(int a, ...) {  
    int bar[4];  
    ...  
    return bar[0];  
}
```

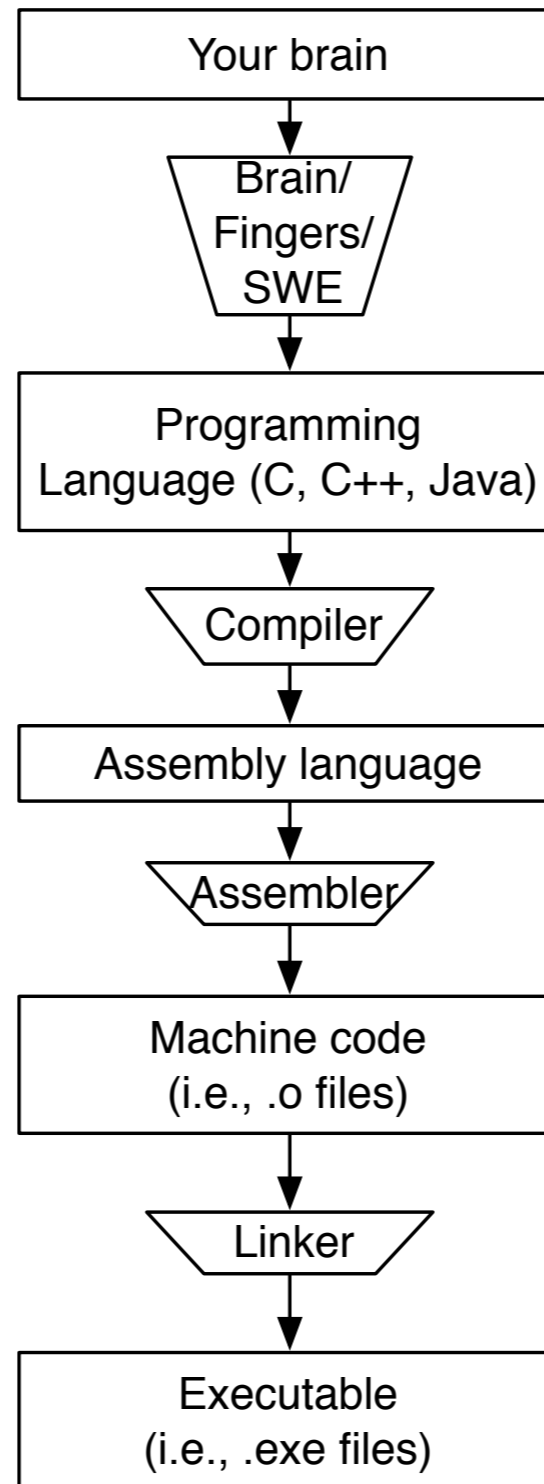


```
...  
subi $sp, $sp, 16 // Allocate bar  
...  
lw   $v0, 0($sp)  
addi $sp, $sp, 16 // deallocate bar  
jr   $ra          // return
```

Saving Registers

- Some registers are preserved across function calls
 - If a function needs a value after the call, it uses one of these
 - But it must also preserve the previous contents (so it can honor its obligation to its caller)
 - Push these registers onto the stack.
 - See figure 2.12 in the text.

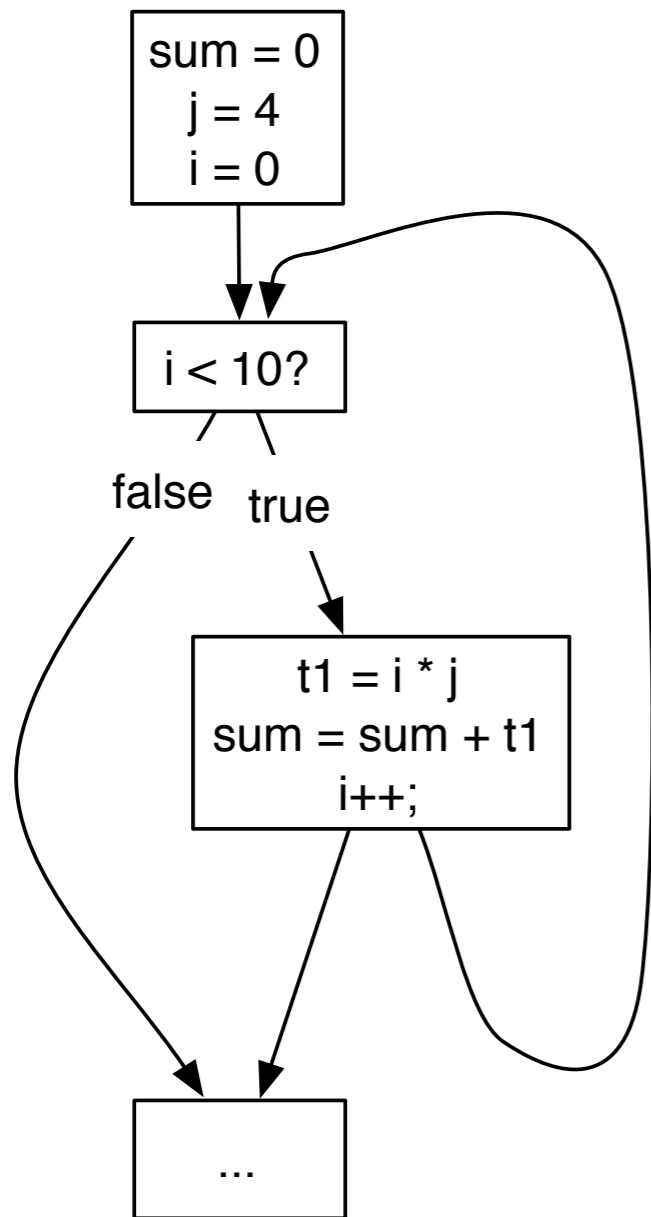
From Brain to Bits



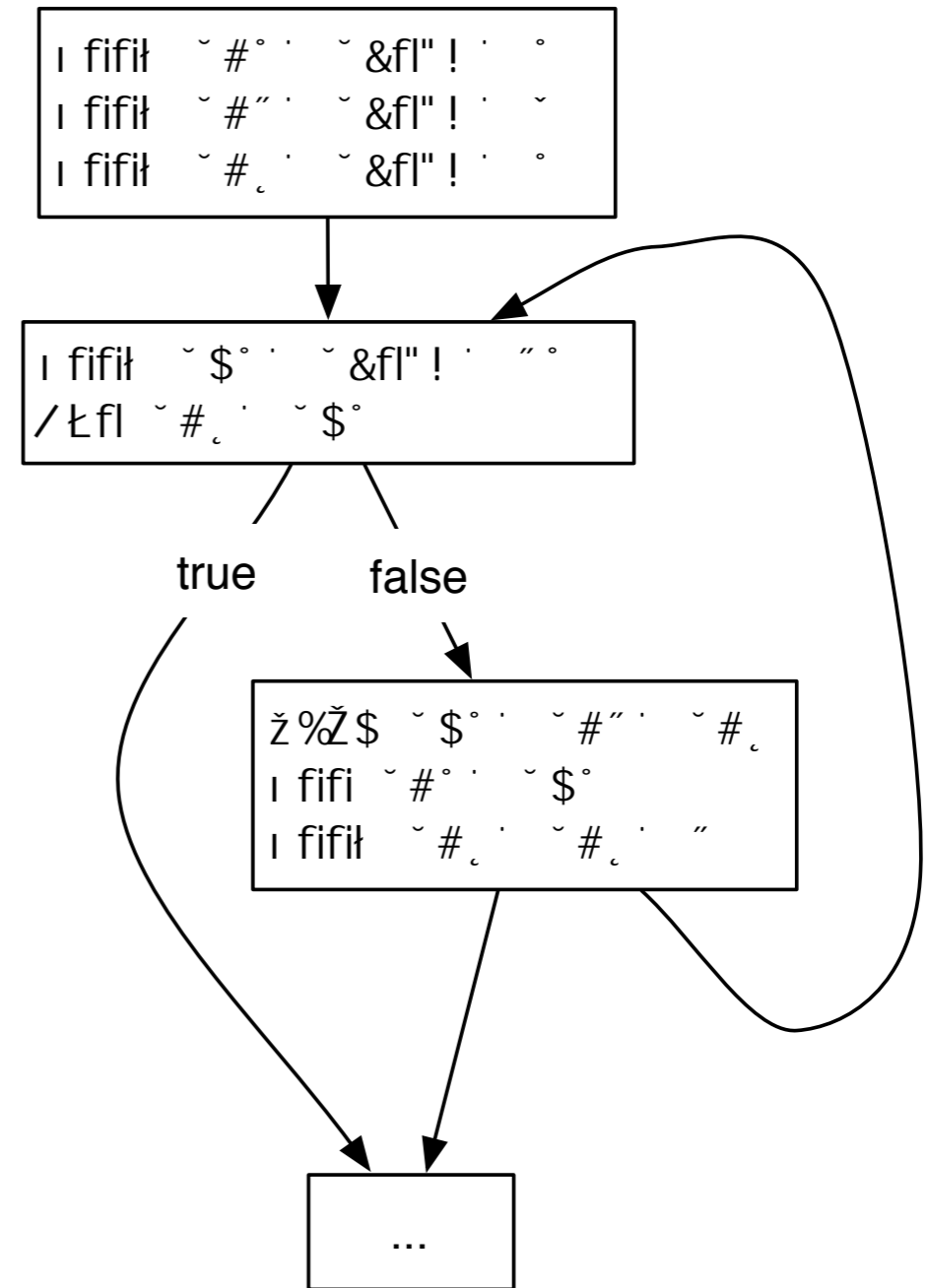
C Code

```
int i;  
int sum = 0;  
int j = 4;  
for(i = 0; i < 10; i++) {  
    sum = i * j + sum;  
}
```


In the Compiler

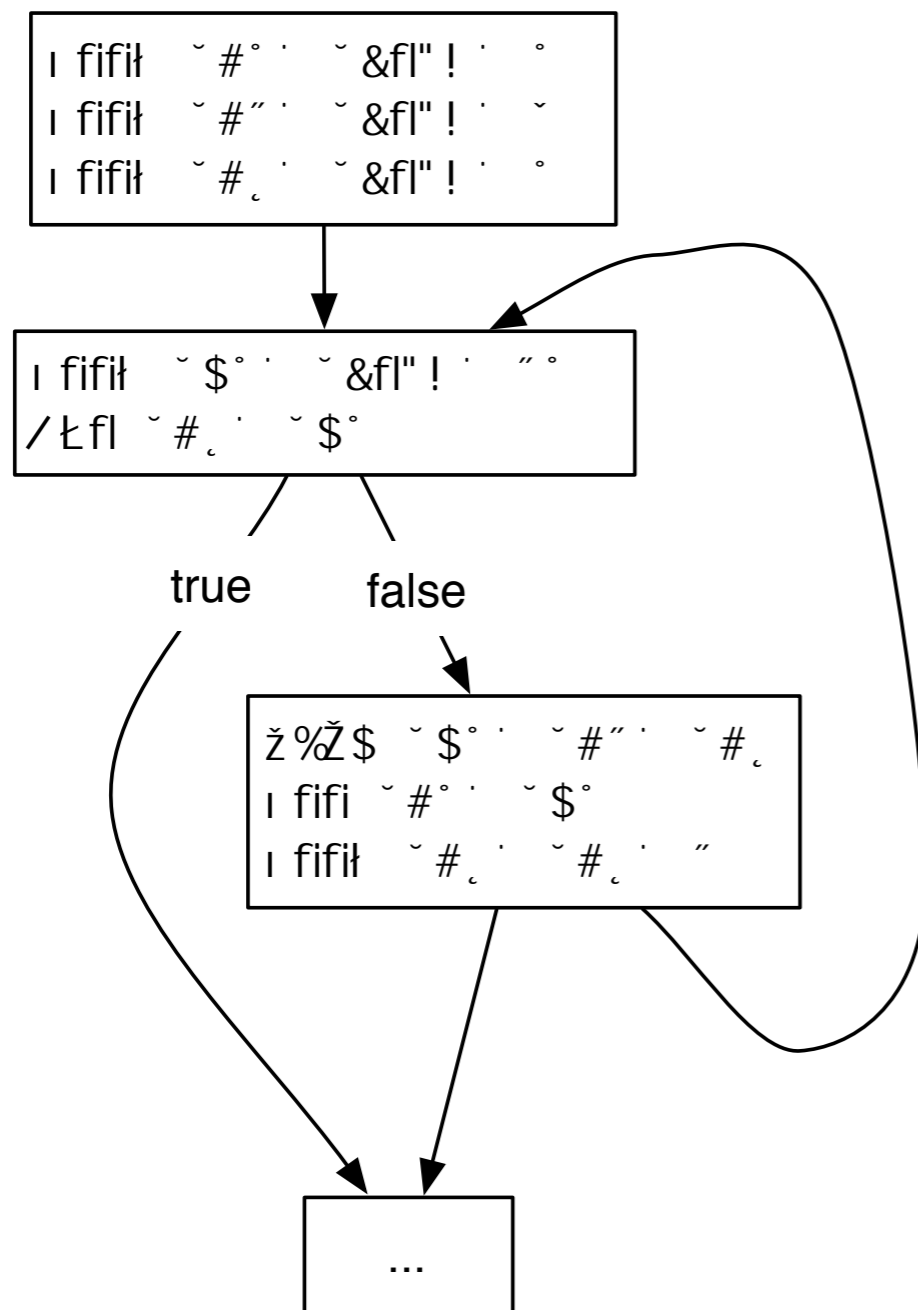


Control flow graph
w/high-level
instructions



Control flow graph
w/real instructions

Out of the Compiler



```

addi $s0, $zero, 0
addi $s1, $zero, 4
addi $s2, $zero, 0

```

```

top:
addi $t0, $zero, 10
bge $s2, $t0, after

```

```

body:
mult $t0, $s1, $s2
add $s0, $t0
addi $s2, $s2, 1
br top

```

```

after:
...

```

Assembly language

Labels in the Assembler

```
addi $s0, $zero, 0
addi $s1, $zero, 4
addi $s2, $zero, 0
```

```
top:
addi $t0, $zero, 10
bge $s2, $t0, after
```

```
mult $t0, $s1, $s2
add $s0, $t0
addi $s2, $s2, 1
br top
```

```
after:
```

```
...
```

'after' is defined at 0x20
used at 0x10

The value of the immediate for the branch
is $0x20 - 0x10 = 0x10$

'top' is defined at 0x0C
used at 0x1C

The value of the immediate for the branch
is $0x0C - 0x1C = 0xFFFF0$ (i.e., $-0x10$)

Labels in the Assembler

```
0x00 addi $s0, $zero, 0
0x04 addi $s1, $zero, 4
0x08 addi $s2, $zero, 0
```

top:

```
0x0C addi $t0, $zero, 10
0x10 bge $s2, $t0, after
```

```
mult $t0, $s1, $s2
```

```
0x14 add $s0, $t0
0x18 addi $s2, $s2, 1
0x1C br top
```

```
0x20 after:
...
```

'after' is defined at 0x20
used at 0x10

The value of the immediate for the branch
is $0x20 - 0x10 = 0x10$

'top' is defined at 0x0C
used at 0x1C

The value of the immediate for the branch
is $0x0C - 0x1C = 0xFFFF0$ (i.e., $-0x10$)

Assembly Language

- “Text section”
 - Hold assembly language instructions
 - In practice, there can be many of these.
- “Data section”
 - Contain definitions for static data.
 - It can contain labels as well.
- The addresses in the data section have no relation to the addresses in the data section.
- Pseudo instructions
 - Convenient shorthand for longer instruction sequences.

.data and pseudo instructions

```
void foo() {  
    static int a = 0;  
    a++;  
    ...  
}
```

```
.data  
foo_a:  
    .word 0  
  
.text  
foo:  
    lda    $t0, foo_a  
    ld     $s0, 0($t0)  
    addi   $s0, $s0, 1  
    st     $s0, 0($t0)  
after:  
    addi   $s2, $s2, 1  
    ...  
    bne    $s2, after
```

.data and pseudo instructions

```
void foo() {  
    static int a = 0;  
    a++;  
    ...  
}
```

```
lda $t0, foo_a
```

becomes these instructions (this is not assembly language!)

```
ori $t0, $zero, ((foo_a & 0xffff0000) >> 16)
```

```
sll $t0, $t0, 16
```

```
ori $t0, $t0, (foo_a & 0xffff)
```

```
.data  
foo_a:  
    .word 0
```

```
.text  
foo:  
    lda    $t0, foo_a  
    ld     $s0, 0($t0)  
    addi   $s0, $s0, 1  
    st     $s0, 0($t0)  
after:  
    addi   $s2, $s2, 1  
    ...  
    bne    $s2, after
```

.data and pseudo instructions

```
void foo() {  
    static int a = 0;  
    a++;  
    ...  
}
```

```
lda $t0, foo_a
```

becomes these instructions (this is not assembly language!)

```
ori $t0, $zero, ((foo_a & 0xffff0000) >> 16)
```

```
sll $t0, $t0, 16
```

```
ori $t0, $t0, (foo_a & 0xffff)
```

```
.data  
foo_a:  
    .word 0
```

```
.text  
foo:  
    lda    $t0, foo_a  
    ld     $s0, 0($t0)  
    addi   $s0, $s0, 1  
    st     $s0, 0($t0)  
after:  
    addi   $s2, $s2, 1  
    ...  
    bne    $s2, after
```

The assembler computes and inserts these values.

.data and pseudo instructions

```
void foo() {  
    static int a = 0;  
    a++;  
    ...  
}
```

```
lda $t0, foo_a
```

becomes these instructions (this is not assembly language!)

```
ori $t0, $zero, ((foo_a & 0xffff0000) >> 16)
```

```
sll $t0, $t0, 16
```

```
ori $t0, $t0, (foo_a & 0xffff)
```

```
.data  
foo_a:  
    .word 0
```

```
.text  
foo:  
    lda $t0, foo_a  
    ld $s0, 0($t0)  
    addi $s0, $s0, 1  
    st $s0, 0($t0)  
after:  
    addi $s2, $s2, 1  
    ...  
    bne $s2, after
```

If foo is address 0x0,
where is after?

The assembler computes and inserts these values.

.data and pseudo instructions

```
void foo() {  
    static int a = 0;  
    a++;  
    ...  
}
```

```
lda $t0, foo_a
```

becomes these instructions (this is not assembly language!)

```
ori $t0, $zero, ((foo_a & 0xffff0000) >> 16)  
sll $t0, $t0, 16  
ori $t0, $t0, (foo_a & 0xffff)
```

```
.data  
foo_a:  
    .word 0
```

```
.text  
foo:
```

```
0x00 lda $t0, foo_a  
0x0C ld $s0, 0($t0)  
0x10 addi $s0, $s0, 1  
0x14 st $s0, 0($t0)
```

```
after:
```

```
0x18 addi $s2, $s2, 1  
...  
bne $s2, after
```

If foo is address 0x0,
where is after?

The assembler computes and inserts these values.

ISA Alternatives

- MIPS is a 3-address, RISC ISA
 - add rs, rt, rd -- 3 operands
 - RISC -- reduced instruction set. Relatively small number of operation. Very regular encoding. RISC is the “right” way to build ISAs.
- 2-address
 - add r1, r2 --> $r1 = r1 + r2$
 - + few operands, so more bits for each.
 - - lots of extra copy instructions
- 1-address
 - Accumulator architectures
 - add r1 -> $acc = acc + r1$

Stack-based ISA

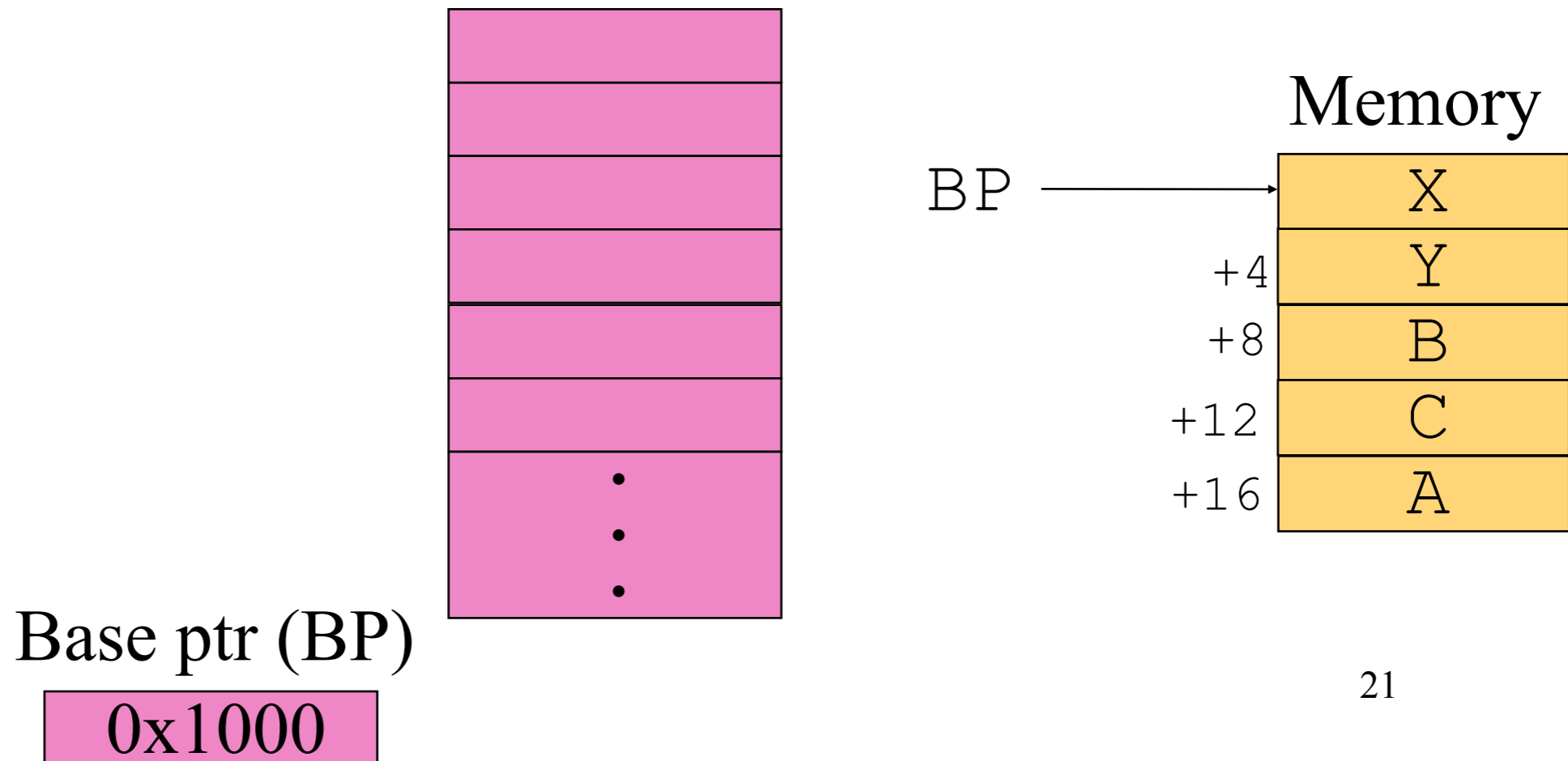
- A push-down stack holds arguments
- Some instruction manipulate the stack
 - push, pop, swap, etc.
- Most instructions operate on the contents of the stack
 - Zero-operand instructions
 - add --> t1 = pop; t2 = pop; push t1 + t2;
- Elegant in theory.
- Clumsy in hardware.
 - How big is the stack?
- Java byte code is a stack-based ISA
- So is the x86 floating point ISA

compute $A = X * Y - B * C$

- Stack-based ISA

- Processor state: PC, "operand stack", "Base ptr"
- Push -- Put something from memory onto the stack
- Pop -- take something off the top of the stack
- +, -, *, ... -- Replace top two values with the result

PC
→



compute $A = X * Y - B * C$

- Stack-based ISA

- Processor state: PC, "operand stack", "Base ptr"
- Push -- Put something from memory onto the stack
- Pop -- take something off the top of the stack
- +, -, *, ... -- Replace top two values with the result

PC



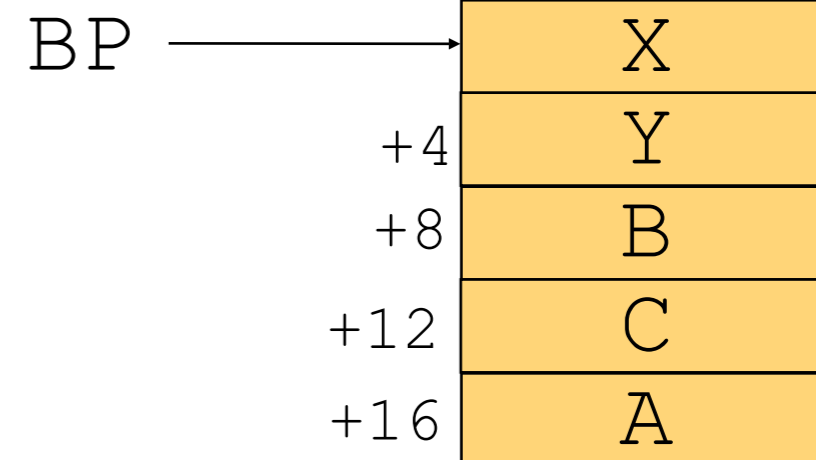
```
Push 12 (BP)
Push 8 (BP)
Mult
Push 0 (BP)
Push 4 (BP)
Mult
Sub
Store 16 (BP)
Pop
```



Base ptr (BP)

0x1000

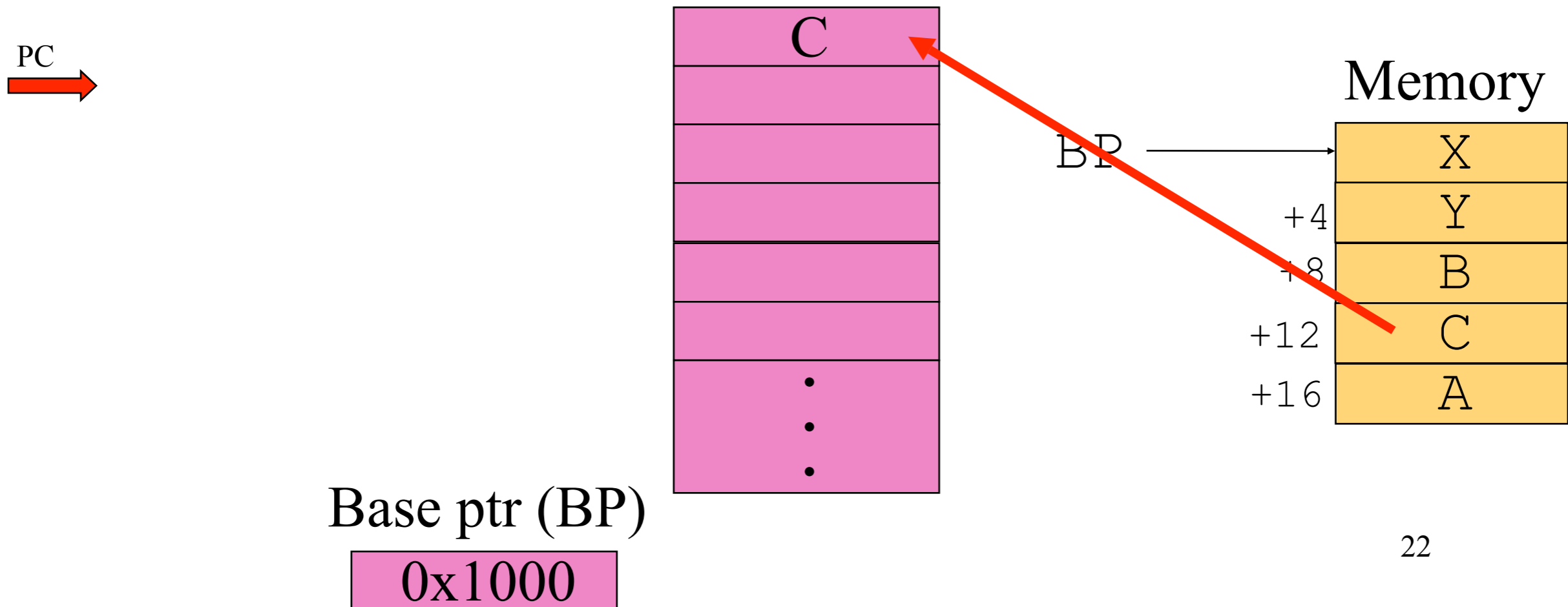
Memory



compute $A = X * Y - B * C$

- Stack-based ISA

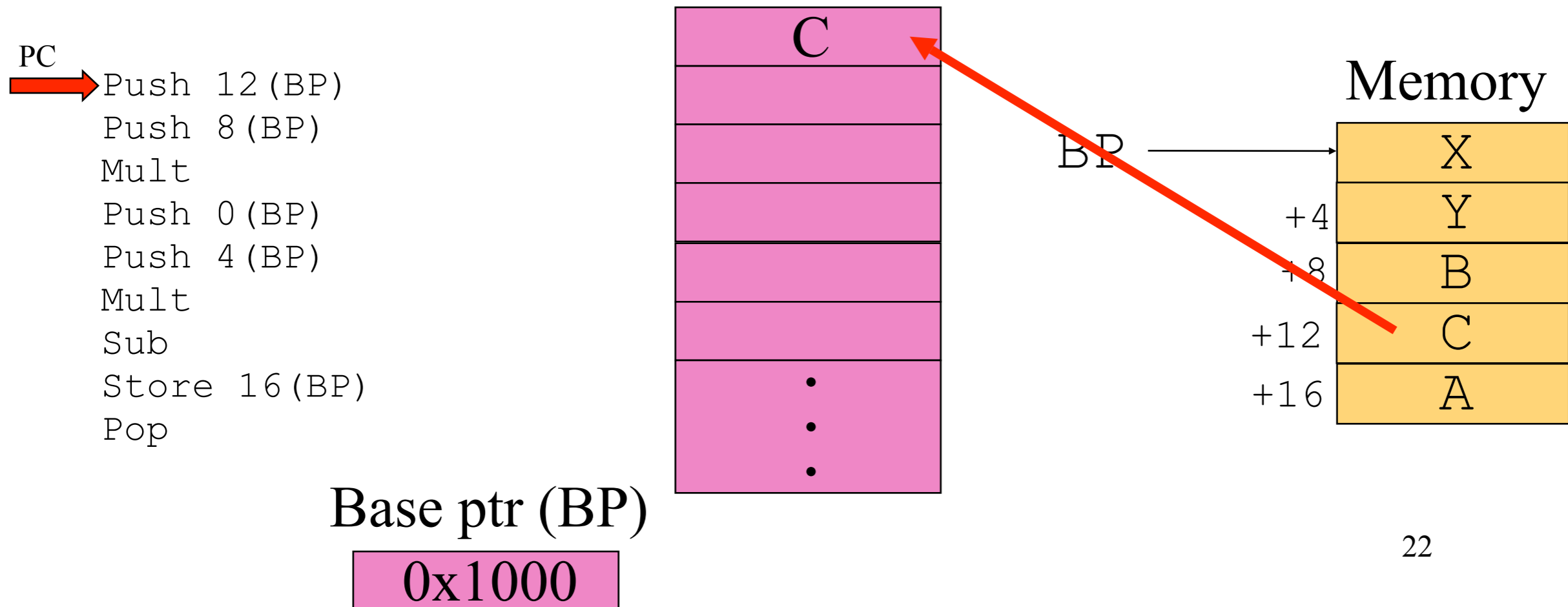
- Processor state: PC, "operand stack", "Base ptr"
- Push -- Put something from memory onto the stack
- Pop -- take something off the top of the stack
- +, -, *, ... -- Replace top two values with the result



compute $A = X * Y - B * C$

- Stack-based ISA

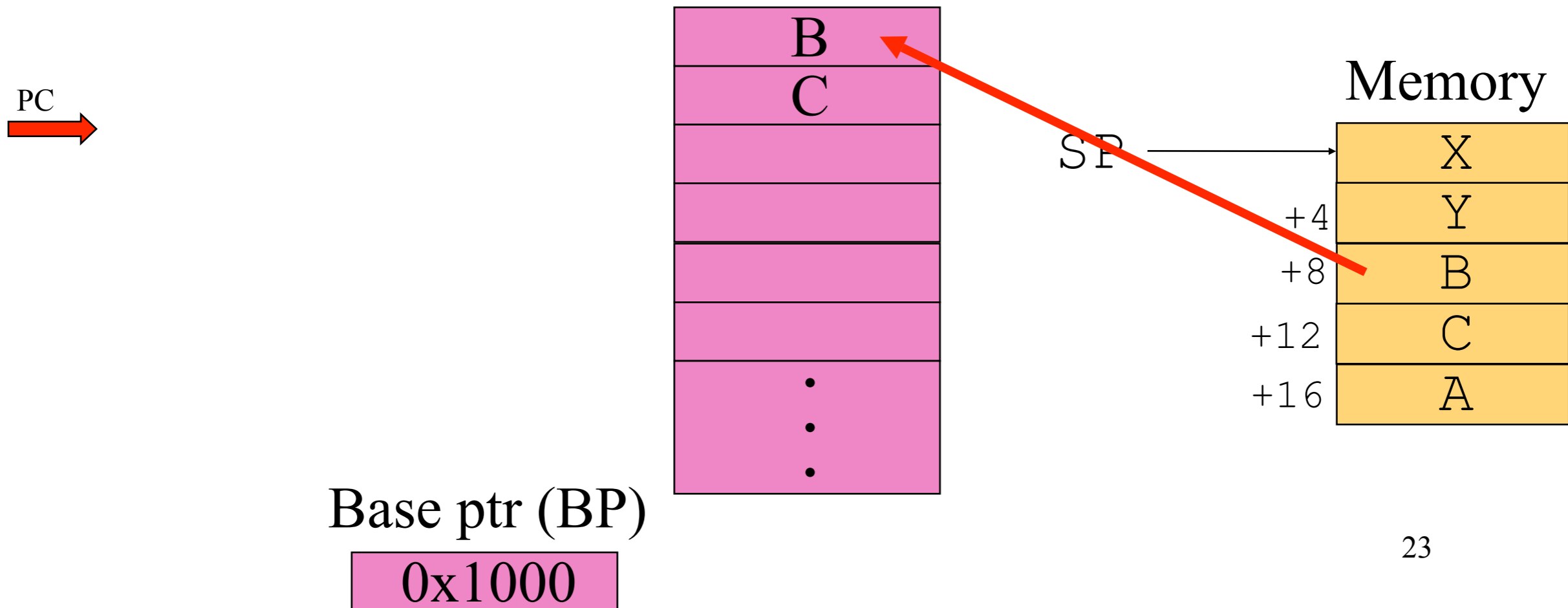
- Processor state: PC, "operand stack", "Base ptr"
- Push -- Put something from memory onto the stack
- Pop -- take something off the top of the stack
- +, -, *, ... -- Replace top two values with the result



compute $A = X * Y - B * C$

- Stack-based ISA

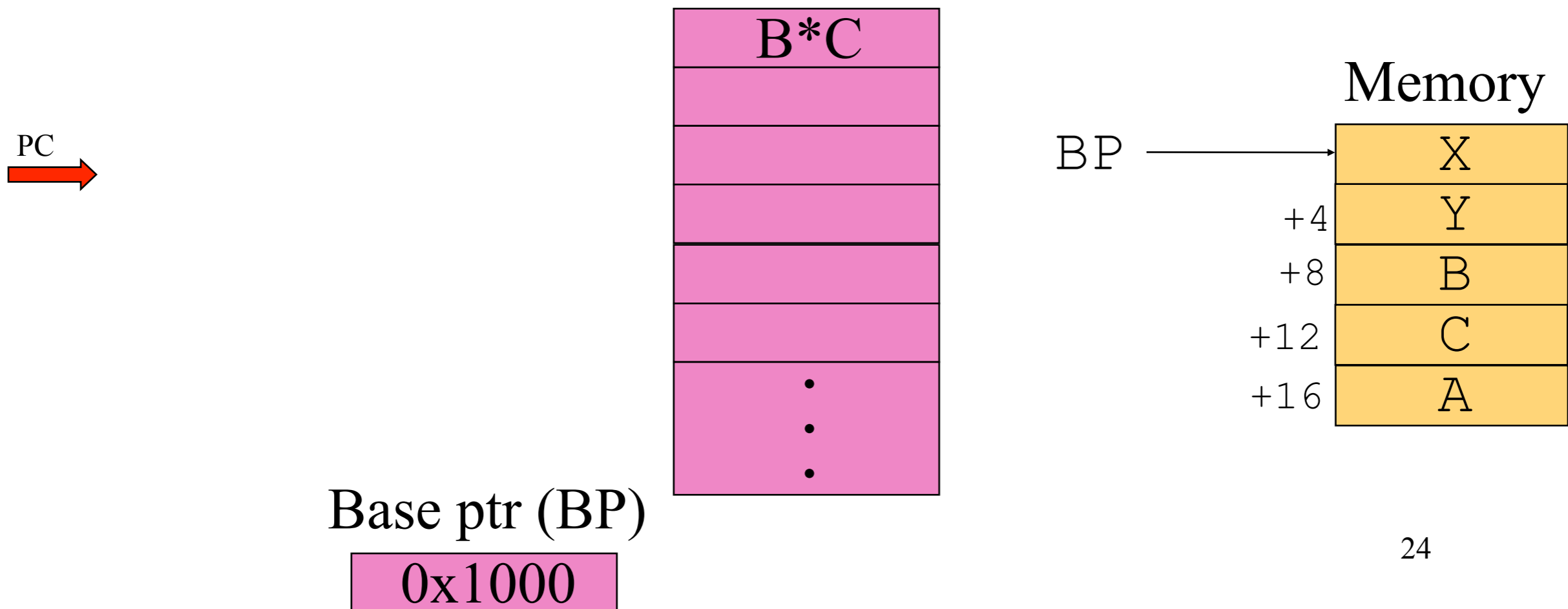
- Processor state: PC, "operand stack", "Base ptr"
- Push -- Put something from memory onto the stack
- Pop -- take something off the top of the stack
- +, -, *, ... -- Replace top two values with the result



compute $A = X * Y - B * C$

- Stack-based ISA

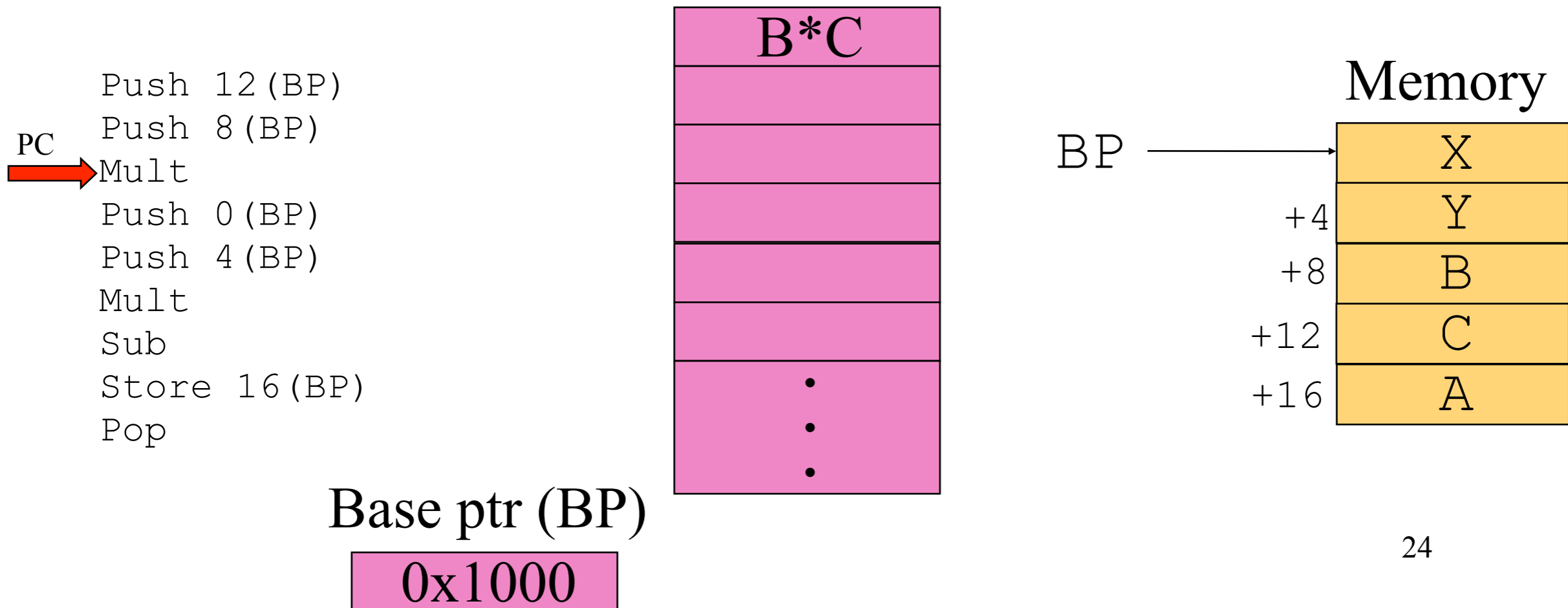
- Processor state: PC, "operand stack", "Base ptr"
- Push -- Put something from memory onto the stack
- Pop -- take something off the top of the stack
- +, -, *, ... -- Replace top two values with the result



compute $A = X * Y - B * C$

- Stack-based ISA

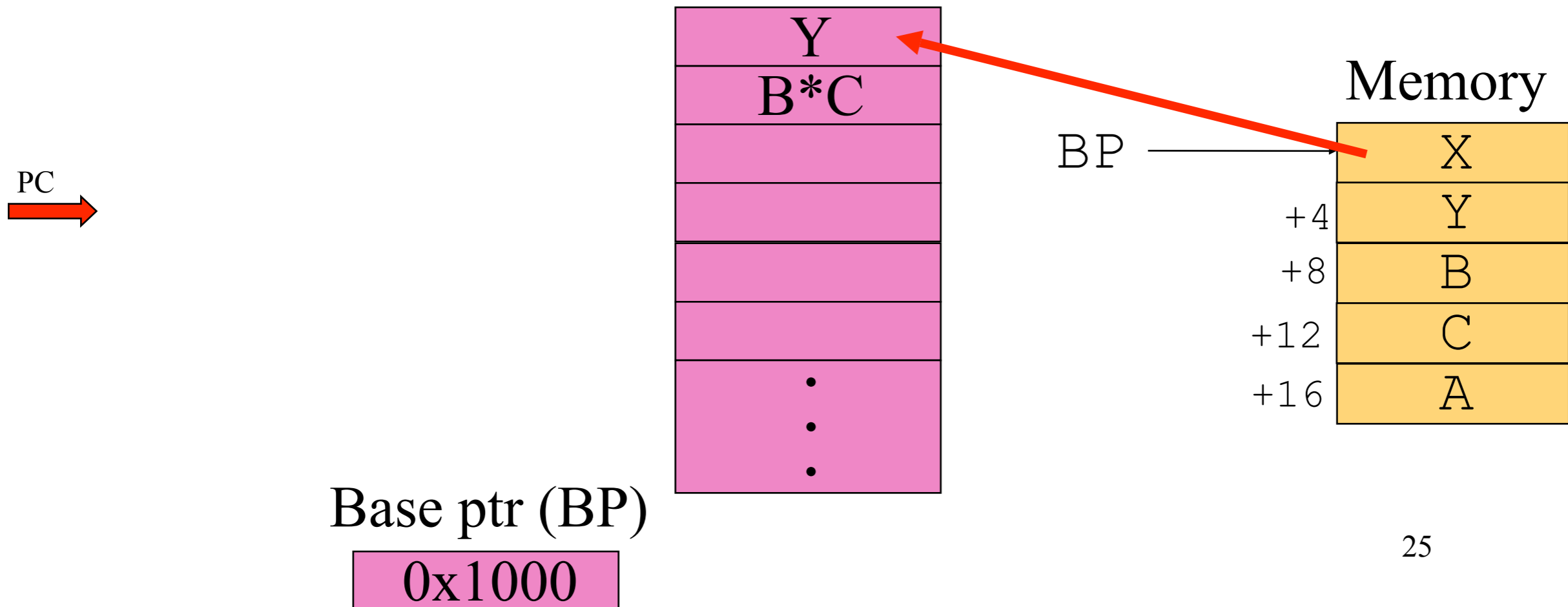
- Processor state: PC, "operand stack", "Base ptr"
- Push -- Put something from memory onto the stack
- Pop -- take something off the top of the stack
- +, -, *, ... -- Replace top two values with the result



compute $A = X * Y - B * C$

- Stack-based ISA

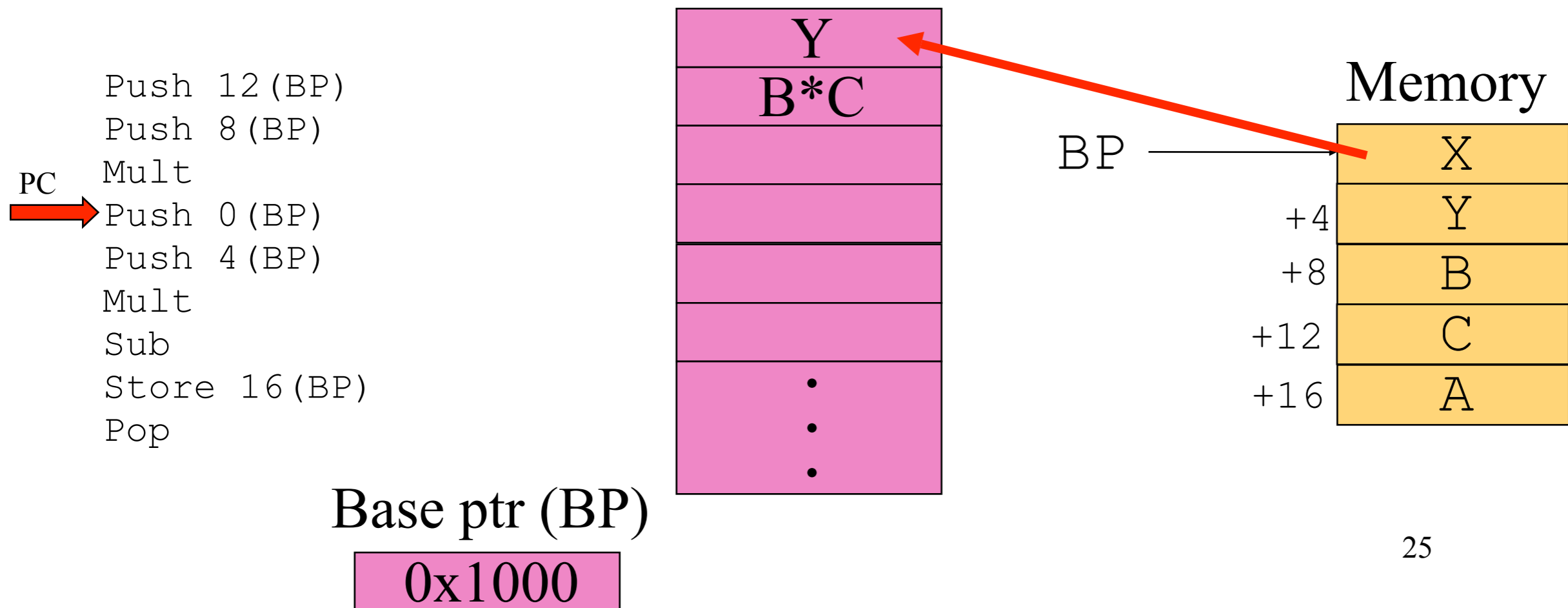
- Processor state: PC, "operand stack", "Base ptr"
- Push -- Put something from memory onto the stack
- Pop -- take something off the top of the stack
- +, -, *, ... -- Replace top two values with the result



compute $A = X * Y - B * C$

- Stack-based ISA

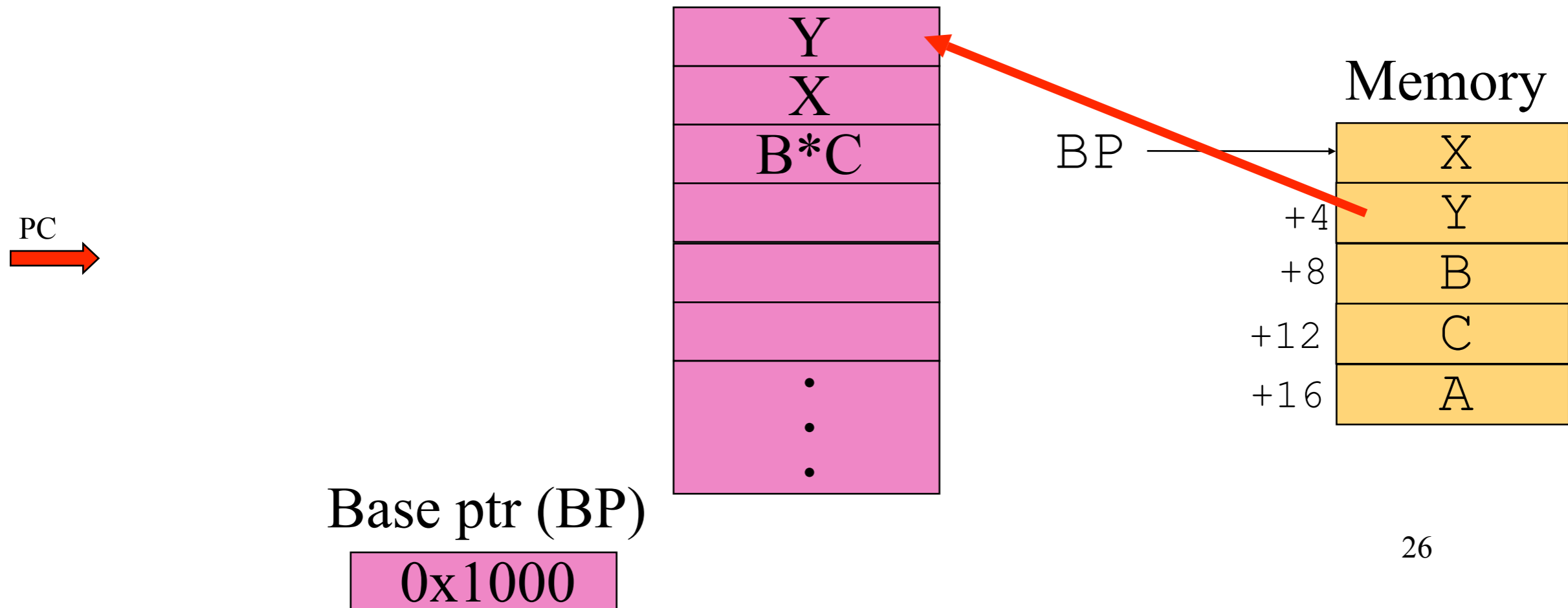
- Processor state: PC, "operand stack", "Base ptr"
- Push -- Put something from memory onto the stack
- Pop -- take something off the top of the stack
- +, -, *, ... -- Replace top two values with the result



compute $A = X * Y - B * C$

- Stack-based ISA

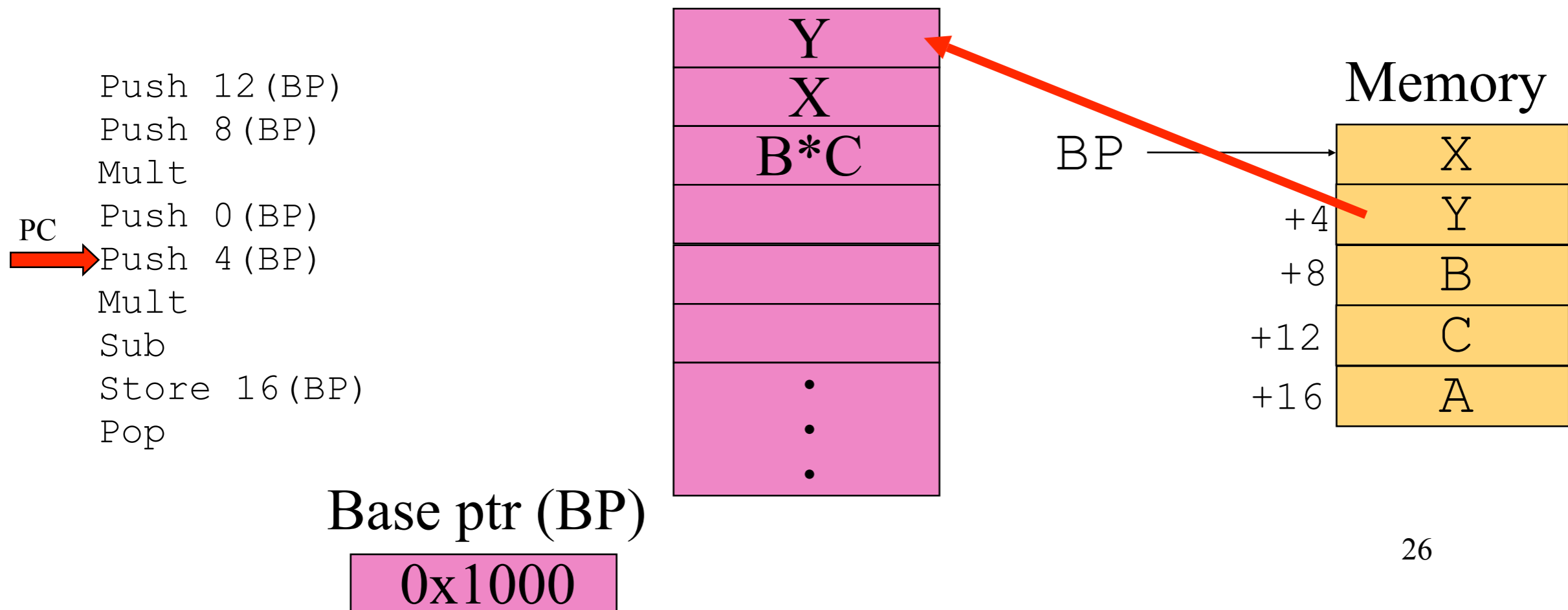
- Processor state: PC, "operand stack", "Base ptr"
- Push -- Put something from memory onto the stack
- Pop -- take something off the top of the stack
- +, -, *, ... -- Replace top two values with the result



compute $A = X * Y - B * C$

- Stack-based ISA

- Processor state: PC, "operand stack", "Base ptr"
- Push -- Put something from memory onto the stack
- Pop -- take something off the top of the stack
- +, -, *, ... -- Replace top two values with the result



compute $A = X * Y - B * C$

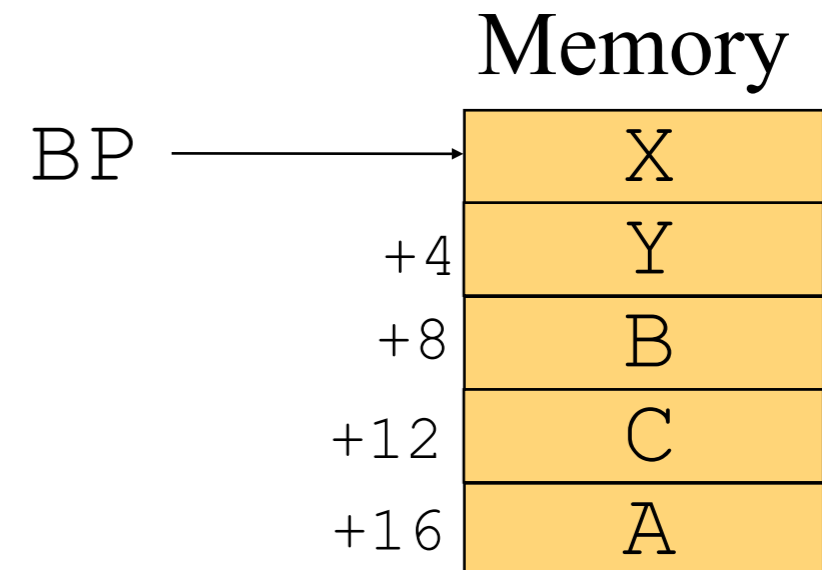
- Stack-based ISA

- Processor state: PC, "operand stack", "Base ptr"
- Push -- Put something from memory onto the stack
- Pop -- take something off the top of the stack
- +, -, *, ... -- Replace top two values with the result
- Store -- Store the top of the stack

PC
→

Base ptr (BP)

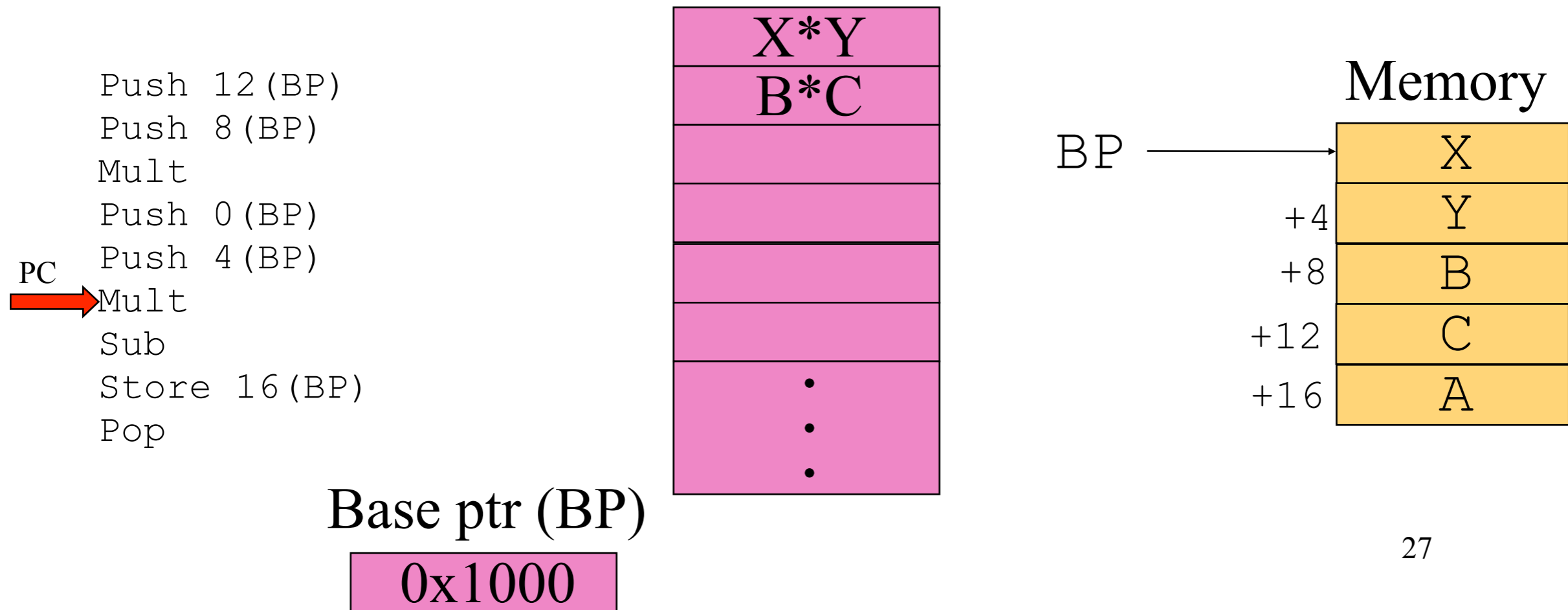
0x1000



compute $A = X * Y - B * C$

• Stack-based ISA

- Processor state: PC, "operand stack", "Base ptr"
- Push -- Put something from memory onto the stack
- Pop -- take something off the top of the stack
- +, -, *, ... -- Replace top two values with the result
- Store -- Store the top of the stack



compute $A = X * Y - B * C$

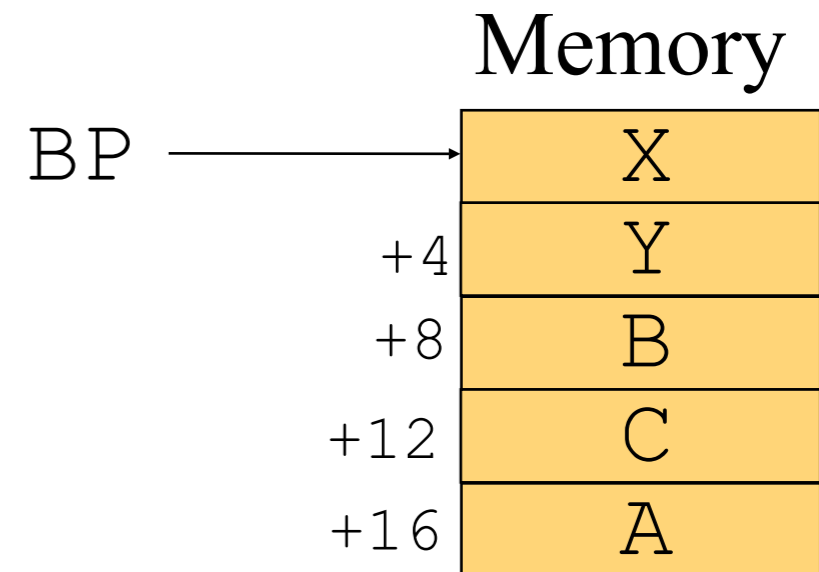
- Stack-based ISA

- Processor state: PC, "operand stack", "Base ptr"
- Push -- Put something from memory onto the stack
- Pop -- take something off the top of the stack
- +, -, *, ... -- Replace top two values with the result

PC →

Base ptr (BP)

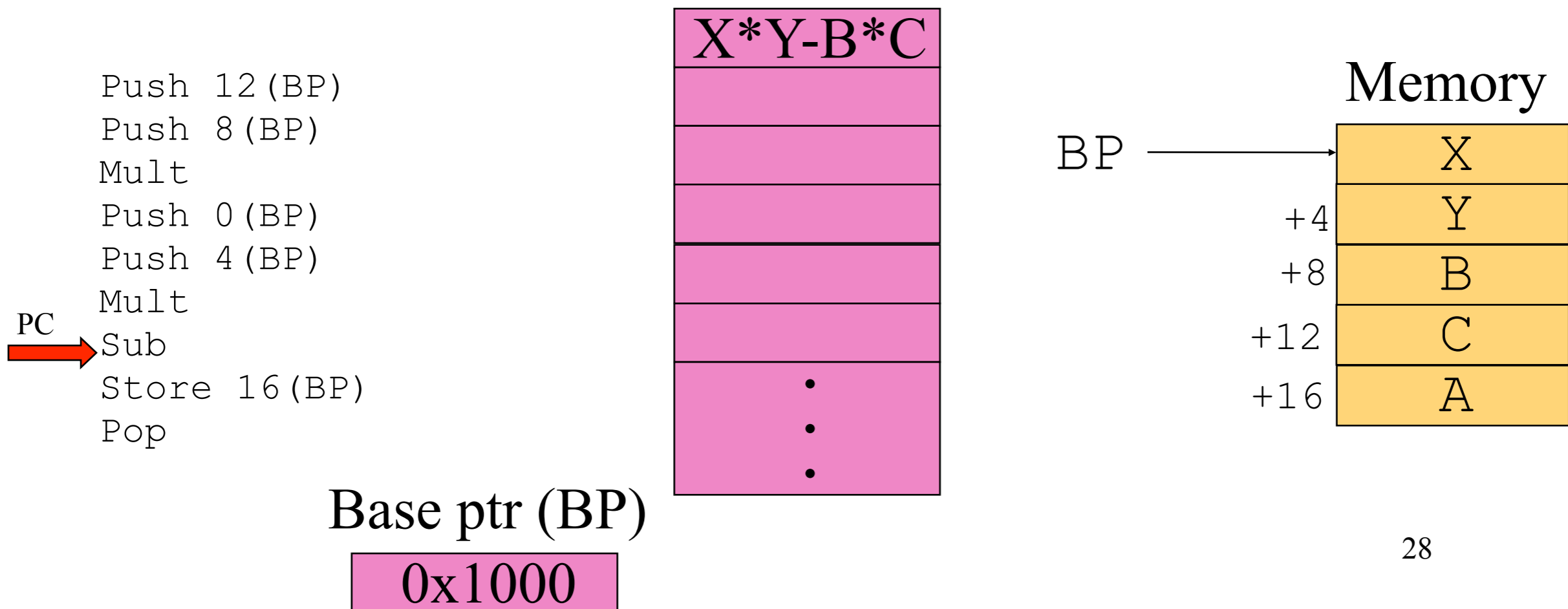
0x1000



compute $A = X * Y - B * C$

- Stack-based ISA

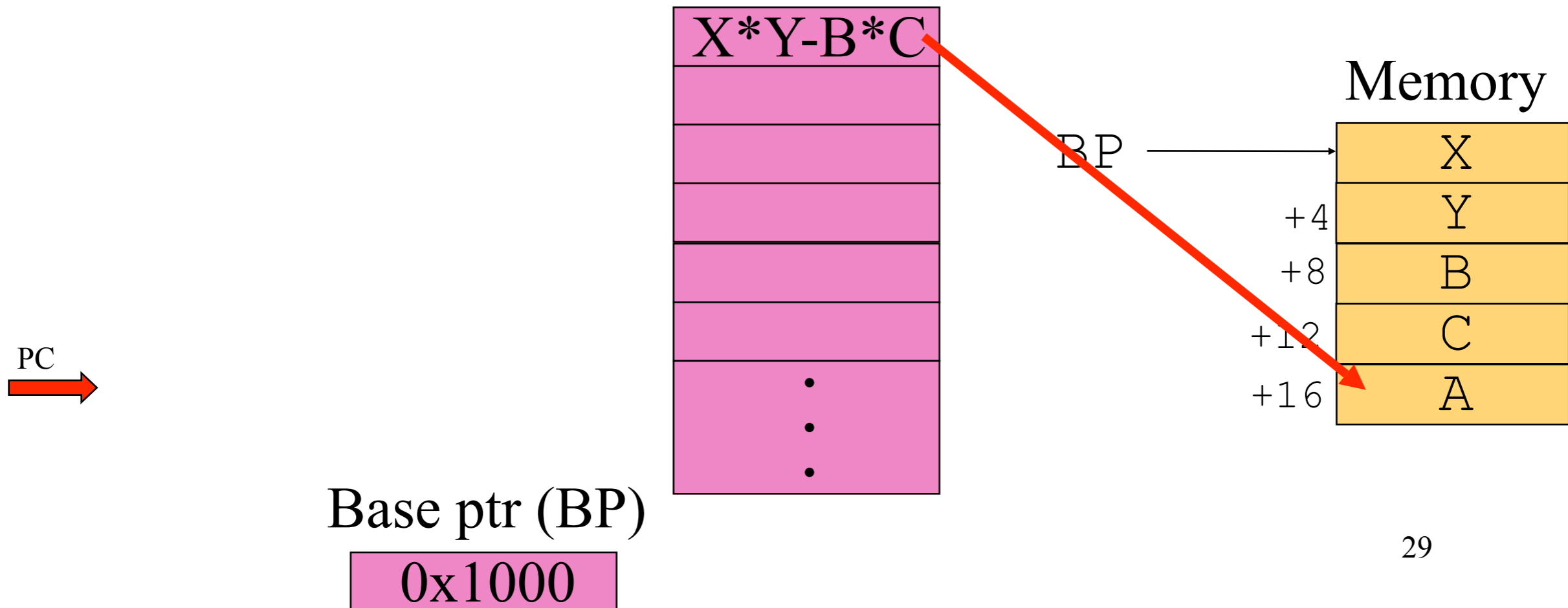
- Processor state: PC, "operand stack", "Base ptr"
- Push -- Put something from memory onto the stack
- Pop -- take something off the top of the stack
- +, -, *, ... -- Replace top two values with the result



compute $A = X * Y - B * C$

- Stack-based ISA

- Processor state: PC, "operand stack", "Base ptr"
- Push -- Put something from memory onto the stack
- Pop -- take something off the top of the stack
- +, -, *, ... -- Replace top two values with the result
- Store -- Store the top of the stack



Time-based Addressing

- Named registers are sooo last century.
- Instead, store the register file as a shift register
- Refer to temporaries by how many instructions ago they were created
- Assume there's a special stack pointer, *sp*.

`sum = a + b + c;`

```
read  sp
ld    0 (v0)
ld    4 (v1)
ld    8 (v2)
add   v0, v1
add   v0, v3
```

Time-based Addressing

- Named registers are sooo last century.
- Instead, store the register file as a shift register
- Refer to temporaries by how many instructions ago they were created
- Assume there's a special stack pointer, *sp*.

`sum = a + b + c;`

```
read  sp
ld    0 (v0)
ld    4 (v1)
ld    8 (v2)
add   v0, v1
add   v0, v3
```

register	value
v3	-
v2	-
v1	-
v0	-

Time-based Addressing

- Named registers are sooo last century.
- Instead, store the register file as a shift register
- Refer to temporaries by how many instructions ago they were created
- Assume there's a special stack pointer, *sp*.

`sum = a + b + c;`

```
read  sp
ld    0 (v0)
ld    4 (v1)
ld    8 (v2)
add   v0, v1
add   v0, v3
```



register	value
v3	-
v2	-
v1	-
v0	sp

Time-based Addressing

- Named registers are sooo last century.
- Instead, store the register file as a shift register
- Refer to temporaries by how many instructions ago they were created
- Assume there's a special stack pointer, *sp*.

`sum = a + b + c;`

```
read  sp          ←
ld    0 (v0)      ←
ld    4 (v1)
ld    8 (v2)
add   v0, v1
add   v0, v3
```

register	value
v3	-
v2	-
v1	sp
v0	a

Time-based Addressing

- Named registers are sooo last century.
- Instead, store the register file as a shift register
- Refer to temporaries by how many instructions ago they were created
- Assume there's a special stack pointer, *sp*.

`sum = a + b + c;`

```
read  sp           ←
ld    0 (v0)       ←
ld    4 (v1)       ←
ld    8 (v2)
add   v0, v1
add   v0, v3
```

register	value
v3	-
v2	sp
v1	a
v0	b

Time-based Addressing

- Named registers are sooo last century.
- Instead, store the register file as a shift register
- Refer to temporaries by how many instructions ago they were created
- Assume there's a special stack pointer, *sp*.

`sum = a + b + c;`

```
read  sp           ←
ld    0 (v0)       ←
ld    4 (v1)       ←
ld    8 (v2)       ←
add   v0, v1
add   v0, v3
```

register	value
v3	sp
v2	a
v1	b
v0	c

Time-based Addressing

- Named registers are sooo last century.
- Instead, store the register file as a shift register
- Refer to temporaries by how many instructions ago they were created
- Assume there's a special stack pointer, *sp*.

`sum = a + b + c;`

```
read  sp           ←
ld    0 (v0)       ←
ld    4 (v1)       ←
ld    8 (v2)       ←
add   v0, v1       ←
add   v0, v3
```

register	value
v3	a
v2	b
v1	c
v0	b+c

Time-based Addressing

- Named registers are sooo last century.
- Instead, store the register file as a shift register
- Refer to temporaries by how many instructions ago they were created
- Assume there's a special stack pointer, *sp*.

`sum = a + b + c;`

```
read  sp          ←
ld    0 (v0)      ←
ld    4 (v1)      ←
ld    8 (v2)      ←
add   v0, v1      ←
add   v0, v3      ←
```

register	value
v3	b
v2	c
v1	b+c
v0	a+b+c