Instruction Set Architectures

Part II: x86, RISC, and CISC

Readings: 2.16-2.18

Which ISA runs in most cell phones and tablets?

Letter	Answer
A	ARM
B	x86
C	MIPS
D	VLIW
E	CISC

Was the full x86 instruction set we have today carefully planned out?

Letter	Answer
A	Yes
B	I wish I could unlearn everything I know about x86. I feel unclean.
C	Are you kidding? I've never seen a more poorly planned ISA!
D	*sob*
E	B, C, or D

Why did AMD and ARM (and MIPS) introduce 64-bit versions of their ISAs?

Letter	Answer
A	To make the CPU smaller.
B	Support more memory
C	To allow for more opcodes
D	B and C
E	A and B

X86 Registers…

Letter	Answer
A	Have fixed functions
B	Are generic, like in MIPS
C	Were originally (in 1978) 64 bits wide
D	Are implemented in main memory
E	None of the above.

Which of these is Amdahl's law?

Letter	Answer
A	Stot = 1/(S/x+(1-x))
B	EP = IC * CPI * CT
C	Stot = x/S+(1-x)
D	Stot = 1/(x/S + (1 – x))
E	E = MC^2

End of Quiz

Fair reading quiz questions?

Letter	Answer
A	Very fair
B	Sort of fair
C	Not very fair
D	Totally unfair

How do you like the class so far overall?

Letter	Answer
A	Very well
B	Good
C	Ok
D	Not so much
E	Not at all

How do you like using the clickers?

Letter	Answer
A	Very well
B	Good
C	Ok
D	Not so much
E	Not at all

How does your experience with clickers in this class compare with your experience with them in other classes?

Letter	Answer
A	This class is better
B	The other classes have been better
C	About the same
D	I haven’t used clickers before.

Have you been going to the discussion section on Wednesday?

Letter	Answer
A	Yes, frequently
B	Yes, once or twice
C	No
D	We have a discussion section on Wednesday?

How is 141L going for you?

Letter	Answer
A	Going well. It's fun!
B	Going ok so far…
C	Not going so well
D	Not going well at all
E	I'm not in 141L

Has this class been helpful for 141L?

Letter	Answer
A	Very much
B	Some
C	Not really
D	Not at all
E	I’m not in 141L

Start, Keep, Stop

- One the piece of paper write
- One thing I should start doing
- One thing I should keep doing
- One thing I should stop doing

Goals for this Class

- Understand how CPUs run programs
  - How do we express the computation the CPU?
  - How does the CPU execute it?
  - How does the CPU support other system components (e.g., the OS)?
  - What techniques and technologies are involved and how do they work?
- Understand why CPU performance varies
  - How does CPU design impact performance?
  - What trade-offs are involved in designing a CPU?
  - How can we meaningfully measure and compare computer performance?
- Understand why program performance varies
  - How do program characteristics affect performance?
  - How can we improve a programs performance by considering the CPU running it?
  - How do other system components impact program performance?

Goals

- Start learning to *read* x86 assembly
- Understand the design trade-offs involved in crafting an ISA
- Understand RISC and CISC
  - Motivations
  - Origins
- Learn something about other current ISAs
  - Very long instruction word (VLIW)
  - Arm and Thumb

The Stack Frame

- A function's "stack frame" holds
  - It's local variables
  - Copies of callee-saved registers (if needs to used them)
  - Copies of caller-saved registers (when it makes function calls).
- The frame pointer ($fp) points to the base of the frame stack frame.
- The frame pointer in action.
  - Adjust the stack pointer to allocate the frame
  - Save the $fp into the frame (it's callee-saved)
  - Copy from the $sp to the $fp
  - Use the $sp as needed for function calls.
  - Refer to local variables relative to $fp.
  - Clean up when you're done.

Example

```
main:
        addiu$sp,$sp,-32
        sw      $fp,24($sp)
        move    $fp,$sp
        sw      $0,8($fp)
        li      $v0,1
        sw      $v0,12($fp)
        li      $v0,2
        sw      $v0,16($fp)
        lw      $3,12($fp)
        lw      $v0,16($fp)
        addu    $v0,$3,$v0
        sw      $v0,8($fp)
        lw      $v0,8($fp)
        move    $sp,$fp
        lw      $fp,24($sp)
        addiu$sp,$sp,32
        j       $ra
```

The Stack Frame

```
main:
    addiu   $sp,$sp,-32
    sw      $fp,24($sp)
    move    $fp,$sp
    sw      $0,8($fp)
    li      $v0,1
    sw      $v0,12($fp)
    li      $v0,2
    sw      $v0,16($fp)
    lw      $3,12($fp)
    lw      $v0,16($fp)
    addu    $v0,$3,$v0
    sw      $v0,8($fp)
    lw      $v0,8($fp)
    move    $sp,$fp
    lw      $fp,24($sp)
    addiu   $sp,$sp,32
    j       $ra
```

PC->

sp->

The stack frame

...	value	fp-relative
0x1020		
0x101C		+32
0x1018	old fp	+24
0x1014		+20
0x1010	2	+16
0x100C	1	+12
0x1008	3	+8
0x1004		+4
0x1000		+0
0x0FFC		

x86 Assembly

x86 ISA Caveats

- x86 is a poorly-designed ISA
  - It breaks almost every rule of good ISA design.
  - There is nothing “regular” or predictable about its syntax.
  - We don’t have time to learn how to write x86 with any kind of thoroughness.
- It is the most widely used ISA in the world today.
  - It is the ISA you are most likely to see in the “real world”
  - So it’s useful to study.
- Intel and AMD have managed to engineer (at considerable cost) their CPUs so that this ugliness has relatively little impact on their processors’ performance (more on this later)

Some Differences Between MIPS and x86

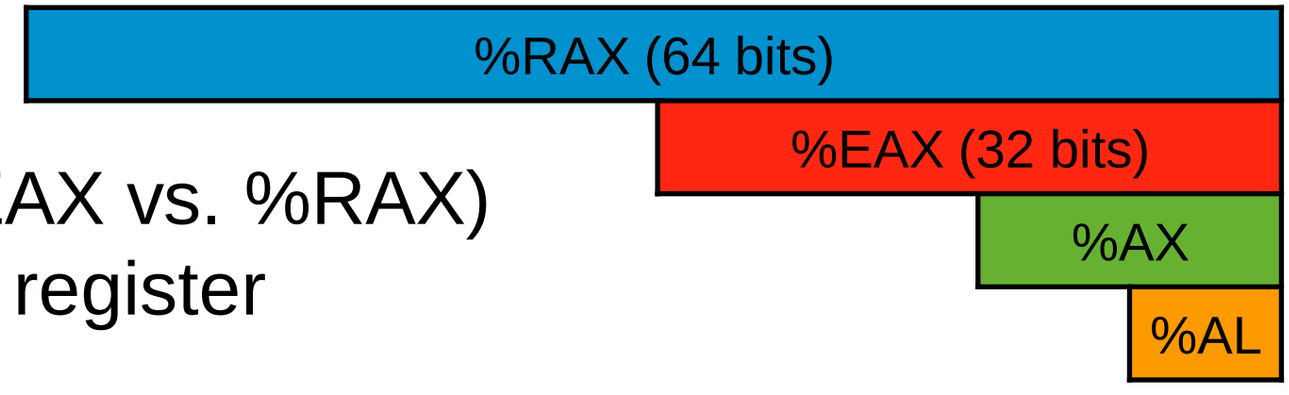
- x86 instructions can operate on memory or registers or both
- x86 is a "two address" ISA
  - Both arguments are sources.
  - One is also the destination
- x86 has (lots of) special-purpose registers
- x86 has variable-length instructions
  - Between 1 and 15 bytes

x86-64 Assembly Syntax

- There are two syntaxes for x86 assembly
- We will use the "gnu assembler (gas) syntax", aka "AT&T syntax". This is different than "Intel Syntax"
- The most confusing difference: argument order
  - AT&T/gas
    - <instruction> <src> <dst>
  - Intel
    - <instruction> <dst> <src>
- Also, different instruction names
- There are some other differences too (see http://en.wikipedia.org/wiki/X86_assembly_language#Syntax)
- If you go looking for help online, make sure it uses the AT&T syntax (or at least be aware, if it doesn't)!

Registers

8-bit	16-bit	32-bit	64-bit	Description	Notes
%AL	%AX	%EAX	%RAX	The accumulator register	These can be used more or less interchangeably, like the registers in MIPS.
%BL	%BX	%EBX	%RBX	The base register	
%CL	%CX	%ECX	%RCX	The counter	
%DL	%DX	%EDX	%RDX	The data register	
%SPL	%SP	%ESP	%RSP	Stack pointer	
%SBP	%BP	%EBP	%RBP	Points to the base of the stack frame	
%R*n*B	%RnW	%R*n*D	%R*n*	(n = 8...15) General purpose registers	
%SIL	%SI	%ESI	%RSI	Source index for string operations	
%DIL	%DI	%EDI	%RDI	Destination index for string operations	
	%IP	%EIP	%RIP	Instruction Pointer	
%FLAGS				Condition codes	



Different names (e.g. %AX vs. %EAX vs. %RAX) refer to different parts of the same register

Instruction Suffixes

Instruction Suffixes		
b	byte	8 bits
s	short	16 bits
w	word	16 bits
l	long	32 bits
q	quad	64 bits

Example

```
addb    $4, %al
addw    $4, %ax
addl    $4, %eax
addq    %rcx, %rax
```

Arguments/Addressing Modes

Type	Syntax	Meaning	Example
Register	%<reg>	R[%reg]	%RAX
Immediate	$nnn	constant	$42
Label	$label	label	$foobar
Displacement	*n(%reg)*	Mem[R[%reg] + n]	-42(%RAX)
Base-Offset	(%r1, %r2)	Mem[R[%r1] + %R[%r2]]	(%RAX,%AL)
Scaled Offset	(%r1, %r2, 2^n)	Mem[R[%r1] + %R[%r2] * 2^n]	(%RAX,%AL, 4)
Scaled Offset Displacement	*k*(%r1, %r2, 2^n)	Mem[R[%r1] + %R[%r2] * 2^n + k]	-4(%RAX,%AL, 2)

mov

- x86 does not have loads and stores. It has mov.

x86 Instruction	RTL	MIPS Equivalent
movb $0x05, %al	R[al] = 0x05	ori $t0, $zero, 5
movl -4(%ebp), %eax	R[eax] = mem[R[ebp] -4]	lw $t0, -4($t1)
movl %eax, -4(%ebp)	mem[R[ebp] -4] = R[eax]	sw $t0, -4($t1)
movl $LC0, (%esp)	mem[R[esp]] = $LC0	la $at, LC0 sw $at, 0($t0)

movl %R0, -4(%R1,%R2,4)	mem[R[%R1] + R[%R2] * 2^n + k] = %R0	slr $at, $t2, 2
		add $at, $at, $t1
		sw $t0, k($at)

movl %R0, %R1	R[%R1] = R[%R0]	ori $t1, $t0, $zero

Arithmetic

Instruction	RTL
subl $0x05, %eax	R[eax] = R[eax] - 0x05
subl %eax, -4(%ebp)	mem[R[ebp] -4] = mem[R[ebp] -4] - R[eax]
subl -4(%ebp), %eax	R[eax] = R[eax] - mem[R[ebp] -4]

Stack Management

Instruction	Meaning	x86 Equivalent	MIPS equivalent
pushl %eax	Push %eax onto the stack	subl $4, %esp; movl %eax, (%esp)	subi $sp, $sp, 4 sw $t0, ($sp)
popl %eax	Pop %eax off the stack	movl (%esp), %eax addl $4, %esp	lw $t0, ($sp) addi $sp, $sp, 4

enter *n*	Save stack pointer, allocate stack frame with *n* bytes for locals	push %BP
		mov %SP, %BP
		sub $n, %SP	

leave	Restore the callers stack pointer.	movl %ebp, %esp pop %ebp	

None of these are pseudo instructions. They are real instructions, just very complex.

The Stack Frame

- A function's "stack frame" holds
  - It's local variables
  - Copies of callee-saved registers (if needs to used them)
  - Copies of caller-saved registers (when it makes function calls).
- The base pointer (%ebp) points to the base of the frame stack frame.
- The base pointer in action
  - Save the old stack pointer.
  - Align the stack pointer
  - Save the old %ebp
  - Copy from the %esp to the %ebp
  - Allocate the frame by decrementing %esp
  - Refer to local variables relative to %ebp
  - Clean up when you're done.

Example

```
main:
        leal   4(%esp), %ecx
        andl   $-16, %esp
        pushl  -4(%ecx)
        pushl  %ebp
        movl   %esp, %ebp
        subl   $16, %esp
        movl   $0, -16(%ebp)
        movl   $1, -12(%ebp)
        movl   $2, -8(%ebp)
        movl   -8(%ebp), %eax
        addl   -12(%ebp), %eax
        movl   %eax, -16(%ebp)
        movl   -16(%ebp), %eax
        addl   $16, %esp
        popl   %ebp
        leal   -4(%ecx), %esp
        ret
```

Branches

- x86 uses condition codes for branches
  - Condition codes are special-purpose bits that make up the flags register
  - Arithmetic ops set the flags register
  - carry, parity, zero, sign, overflow

Instruction	Meaning
cmpl %r1 %r2	Set flags register for %r2 - %r1
jmp <location>	Jump to <location>
je <location>	Jump to <location> if the equal flag is set
jg, jge, jl, jle, jnz, ...	jump if {>, >=, <, <=, != 0,}

Function Calls

Instruction	Meaning	MIPS
call <label>	Push the return address onto the stack. Jump to the function.	Homework?

ret	Pop the return address off the stack and jump to it.	lw $at, 0($sp)
		addi $sp, $sp, 4
		jr $at

- Return address goes on the stack (rather than a register as in MIPS)
- Arguments are passed on the stack (with push)
- Return value in %eax/%rax

Example

```
int foo(int x, int y);
...
d = foo(a, b);

pushq %R9
pushq %R8
call foo
movq  %eax, d
```

x86 Assembly Resources

- These slides don’t cover everything you’ll need for the homeworks on x86 assembly
  - There’s too many ugly details to cover in class.
  - But you may still encounter this code in real life (or on the homeworks).
- You’ll need to do some looking of your own to find the missing bits
  - http://en.wikipedia.org/wiki/X86_architecture
  - http://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax
  - The text book.
- Make sure you know if the resources you find are AT&T or Intel syntax!
  - If there aren’t any “%”, it’s probably Intel, and the dst comes first, rather than last.

Which of the following is NOT correct about these two ISAs?

MIPS + x86

Selection	Statement
A	x86 provides more instructions than MIPS
B	x86 usually needs more instructions to express a program
C	An x86 instruction may access memory 3 times
D	An x86 instruction may be shorter than a MIPS instruction
E	An x86 instruction may be longer than a MIPS instruction

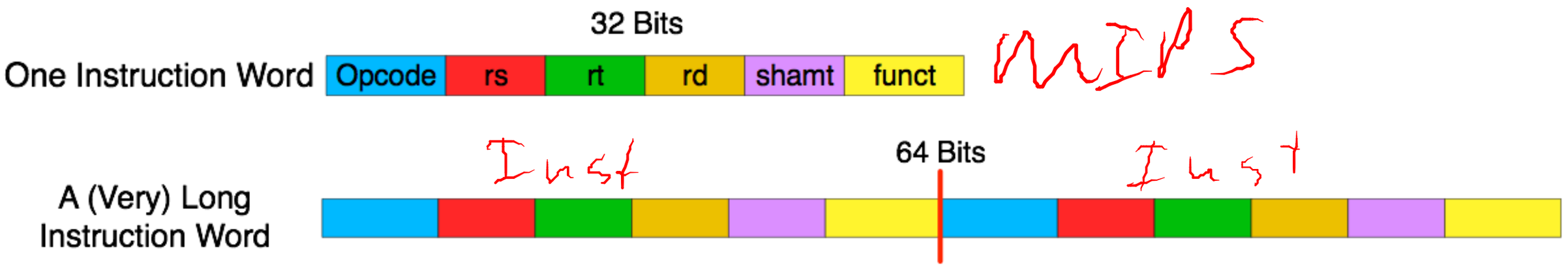
Other ISAs

Designing an ISA to Improve Performance

- The PE tells us that we can improve performance by reducing CPI. Can we get CPI to be less than 1?
  - Yes, but it means we must execute more the one instruction per cycle.
  - That means parallelism.
- How can we modify the ISA to support the execution of multiple instructions each cycle?
- Later, we'll look at modifying the processor implementation to do the same thing without changing the ISA.

Very Long Instruction Word (VLIW)

- Put two (or more) instructions in one!



- Each sub-instruction is just like a normal instruction.
- The instructions execute *at the same time.*
- The processor can treat them as a single unit.
- Typical VLIW widths are 2-4 instructions, but some machine have been much higher

VLIW Example

- VLIW-MIPS
  - Two MIPS instruction/VLIW instruction word
  - Not a real VLIW ISA.

MIPS Code

```
ori $s2, $zero, 6
ori $s3, $zero, 4
add $s2, $s2, $s3
sub $s4, $s2, $s3
```

```
Results:
$s2 = 10
$s4 = 6
```

Since the add and sub execute *sequentially,* the sub sees the new value for $s2

VLIW-MIPS Code

```
<ori $s2, $zero,6;  ori $s3, $zero, 4>
<add $s2, $s2, $s3; sub $s4, $s2, $s3>
```

```
Results:
$s2 = 10
$s4 = 2
```

Since the add and sub execute *at the same time* they both see the original value of `$s2`

VLIW Challenges

- VLIW has been around for a long time, but it’s not seen mainstream success.
- The main challenging is finding instructions to fill the VLIW slots.
- This is tortuous by by hand, and difficult for the compiler.

VLIW-MIPS Code

```
<ori $s2, $zero,6;  ori $s3, $zero, 4>
<add $s2, $s2, $s3; nop                >
<sub $s4, $s2, $s3; nop                >

Results:
$s2 = 10
$s4 = 6
```

Now, the add and sub execute sequentially, but we’ve wasted space and resources executing `nops`.

VLIW’s History

- VLIW has been around for a long time
  - It’s the simplest way to get CPI < 1.
  - The ISA specifies the parallelism, the hardware can be very simple
  - When hardware was expensive, this seemed like a good idea.
- However, the compiler problem (previous slide) is extremely hard.
  - There end up being lots of noops in the long instruction words.
  - Especially for “branchy” code (word processors, compilers, games, etc.)
- As a result, they have either
  - 1. met with limited commercial success as general purpose machines (many companies) or,
  - 2. Become very complicated in new and interesting ways (for instance, by providing special registers and instructions to eliminate branches), or
  - 3. Both 1 and 2 -- See the Itanium from intel.

Consider a 2-wide VLIW processor whose cycle time is 0.75x that our baseline MIPS processors. For your code, the compiler ends up including one nop in ½ of the VLIW instruction words it generates. What's the overall speedup of the VLIW processor vs. the baseline MIPS? Assume the number of non-nops doesn't change.

Selection	VLIW CPI	Total Speedup
A	1.5	1.333
B	1.5	0.666
C	0.75	1.77
D	0.666	2.002
E	0.75	1.5

VLIW’s Success Stories

- VLIW’s main success is in digital signal processing
  - DSP applications mostly comprise very regular loops
    - Constant loop bounds,
    - Simple data access patterns
    - Non-data-dependent computation
  - Since these kinds of loops make up almost all (i.e., x is almost 1.0) of the applications, Amdahl’s Laws says writing the code by hand is worthwhile.
  - These applications are cost and power sensitive
    - VLIW processors are simple
    - Simple means small, cheap, and efficient.
- I would not be surprised if there’s a VLIW processor in your cell phone.

The ARM ISA

- The ARM ISA is in most of today’s cool mobile gadgets
- It got started at about the same time as MIPS
  - ARM Holdings. Inc. owns the ISA and licenses it to other companies.
  - It does not actually build chips.
- There are ARM chips available from many vendors
  - The vendors compete or other features (e.g., integrated graphics co-processors)
  - Drives down cost.
- There’s an ARM version of your text book.

MIPS vs. ARM

- MIPS and ARM are both modern, relatively clean ISAs
- ARM has
  - Fixed-length instruction words (mostly. More in moment)
  - General-purpose registers (although only 16 of them)
  - A similar set of instructions.
- But there are some differences...

MIPS vs. ARM: Addressing Modes

- MIPS has 3 “addressing modes”
  - Register -- $s1
  - Displacement -- 4($s1)
  - Immediate -- 4
- ARM has several more

ARM Instruction	Meaning	
LDR r0,[r1,#8]	R[r0] = Mem[R[r1] + 8]	Displacement (like mips)
LDR r0,[r1,#8]!	R[r1] = R[r1] + 8 R[r0] = Mem[R[r1]];	Pre-increment Displacement
LDR r0,[r1],#8	R[r0] = Mem[R[r1]]; R[r1] = R[r1] + 8	Post-increment Displacement

MIPS vs. ARM: Shifts

- ARM likes to perform shift operations
- The second src operand of most instructions can be shifted before use
- MIPS is less shift-happy.

ARM Instruction	Meaning
Add r1,r2,r3, LSL #4	R[r1] = R[r2] + (R[r3] << 4)
Add r1,r2,r3, LSL r4	R[r1] = R[r2] + (R[r3] << R[r4])

MIPS vs. ARM: Branches

- ARM uses condition codes and predication for branches
  - Condition codes: negative, zero, carry, overflow
  - Instruction set them
- Instruction can be made conditional on one of the condition codes
  - The the corresponding condition code is set, the instruction will execute.
  - Otherwise, the instruction will be a nop.
  - An instruction suffix specifies the condition code
- This eliminates many branches.
  - We'll see later on in this class that branches can slow down execution.

C Code

```
if (x == y)
  p = q + r
```

```
x is $s0
y is $s1
p is $s2
q is $s3
r is $s4
```

MIPS Assembly

```
  bne $s0, $s1, foo
  add $s2, $s3, $s4
foo:
```

```
x is r0
y is r1
p is r2
q is r3
r is r4
```

ARM Assembly

```
CMP r0,r1
ADDEQ r2,r3,r4
```

ISA Alternatives

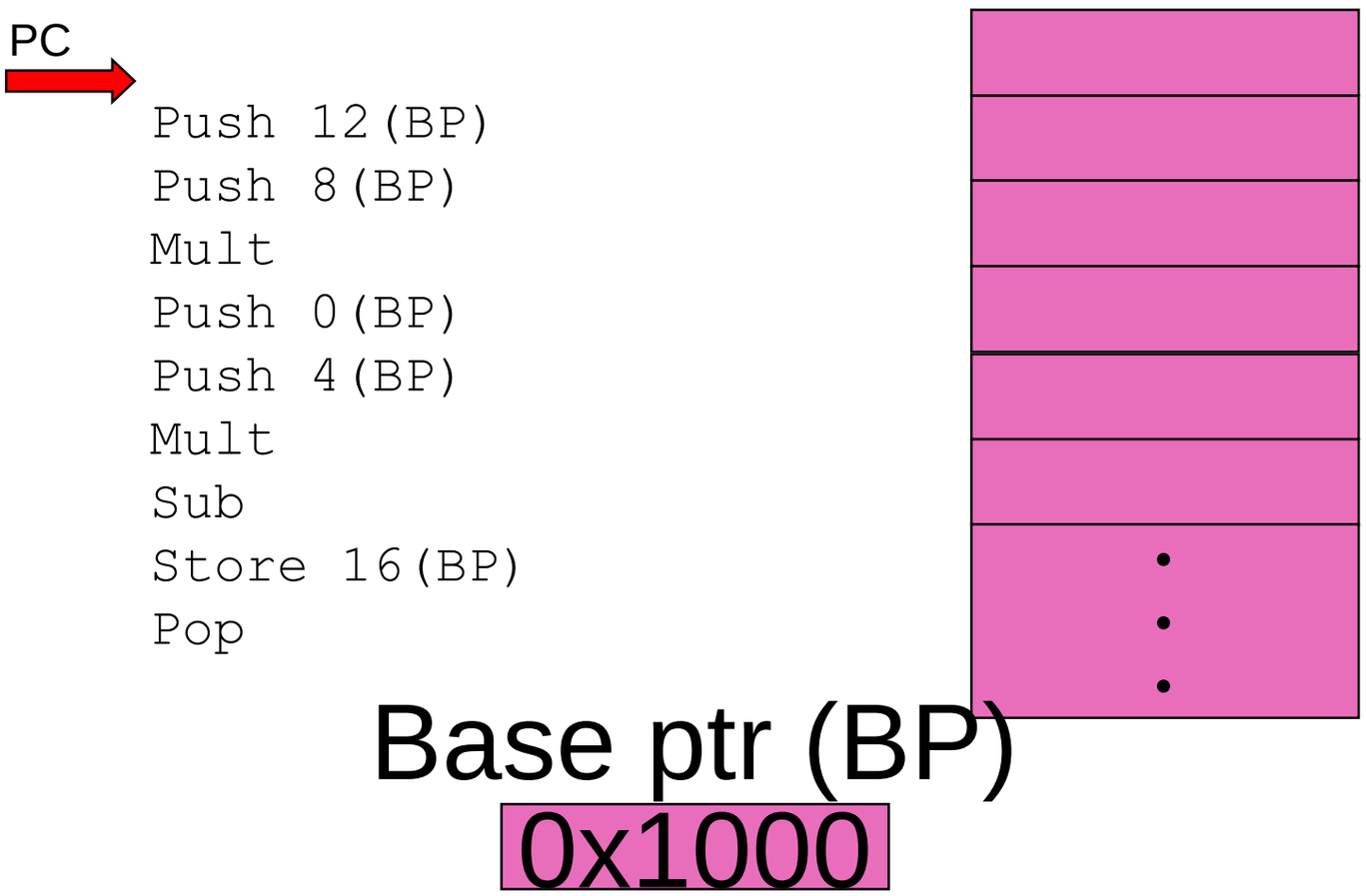
- 2-address code
  - `add r1, r2` means R[r1] = R[r1] + R[r2]
  - + few operands, so more bits for each.
  - - lots of extra copy instructions
- 1-address -- Accumulator architectures
  - An "accumulator" is a source and destination for all operations
  - `add r1` means acc = acc + R[r1]
  - `setacc r1` mean acc = R[r1]
  - `getacc r1` mean R[r1] = acc
- "0-address" code -- Stack-based architectures

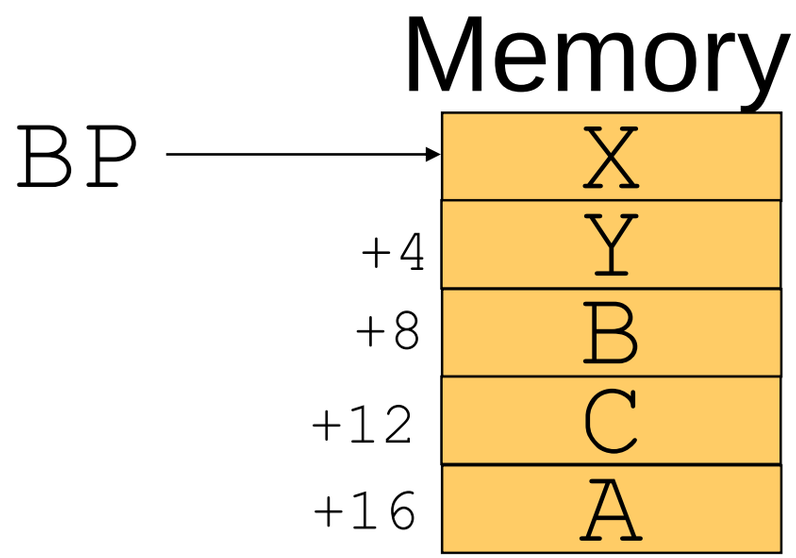
Stack-based ISA

- No register file. Instead, a stack holds values
- Some instruction manipulate the stack
  - push -- add something to the stack
  - pop -- remove the top item.
  - swap -- swaps the top two items
- Most instructions operate on the contents of the stack
  - Zero-operand instructions
  - ‘add’ is equivalent to t1 = pop; t2 = pop; push t1 + t2;
- Elegant in theory
- Clumsy in hardware.
  - How big is the stack?
- Java and Python “byte code” are stack-based ISAs
  - Infinite stack, but it runs in a VM

Stack Example: A = X * Y - B * C

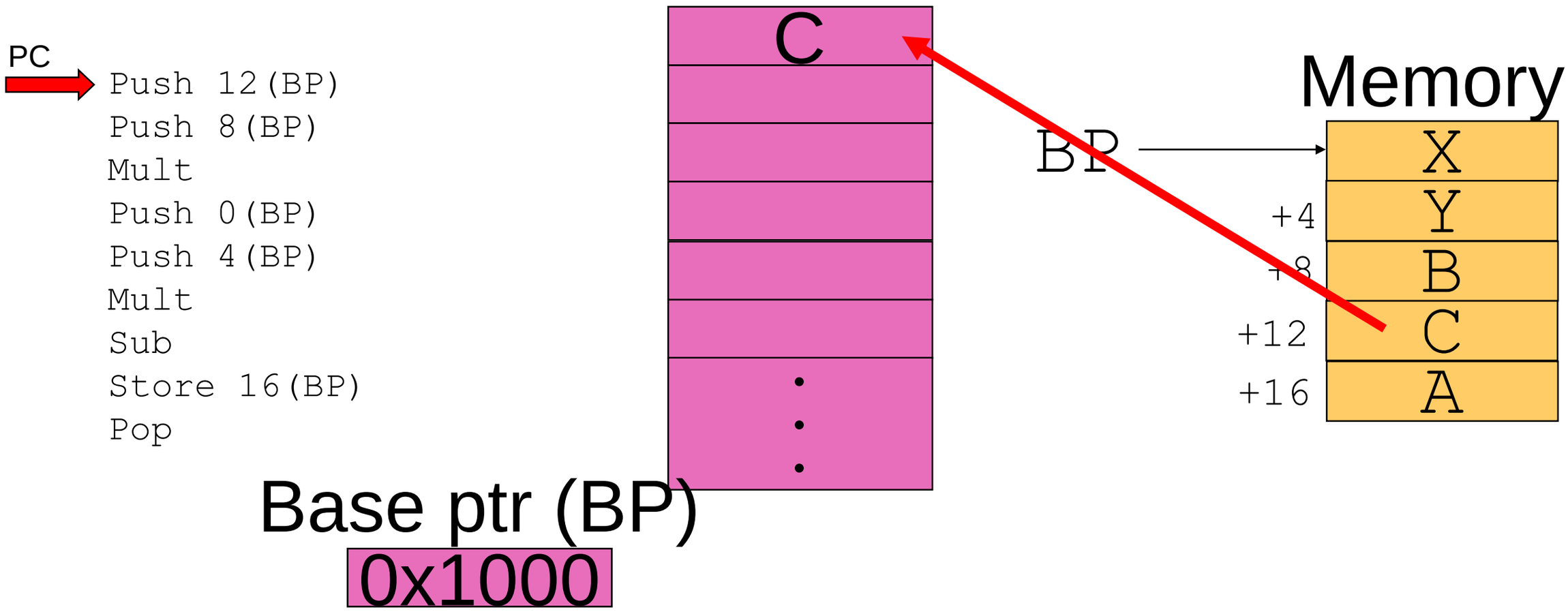
- Stack-based ISA
- Processor state: PC, “operand stack”, “Base ptr”
- Push -- Put something from memory onto the stack
- Pop -- take something off the top of the stack
- +, -, *,… -- Replace top two values with the result
- Store -- Store the top of the stack





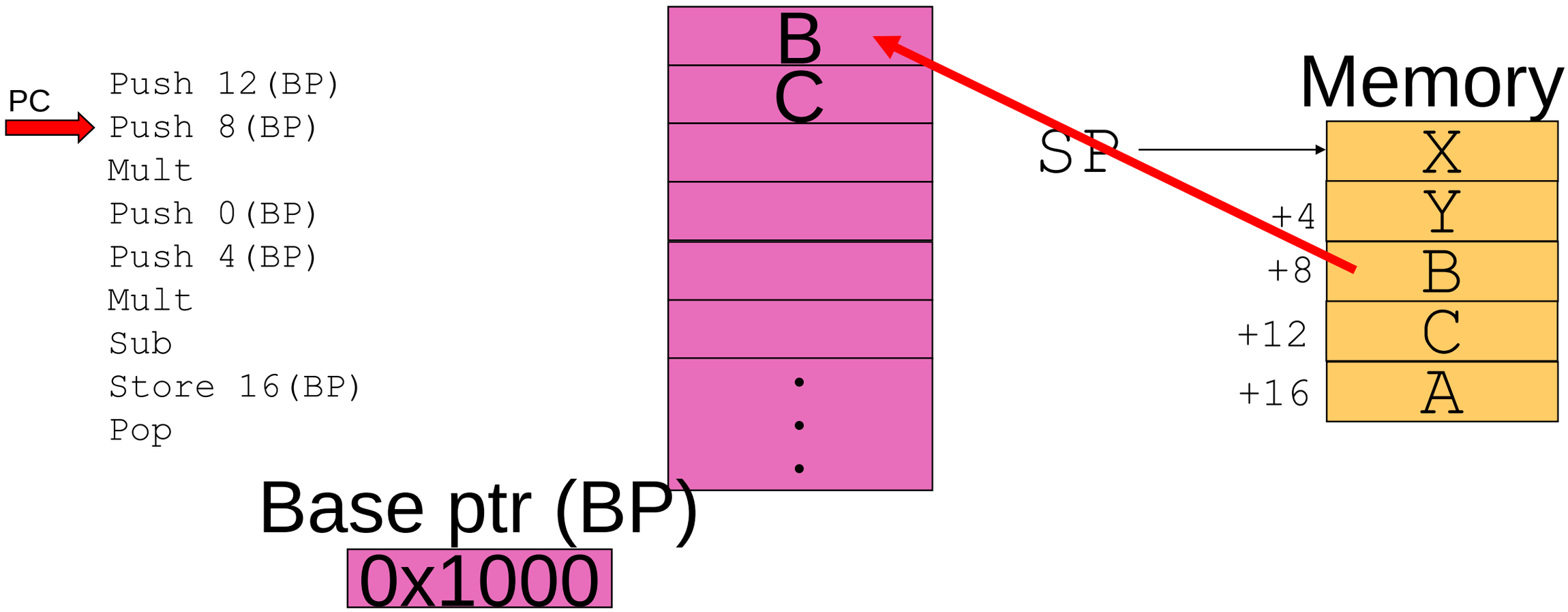
Stack Example: A = X * Y - B * C

- 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



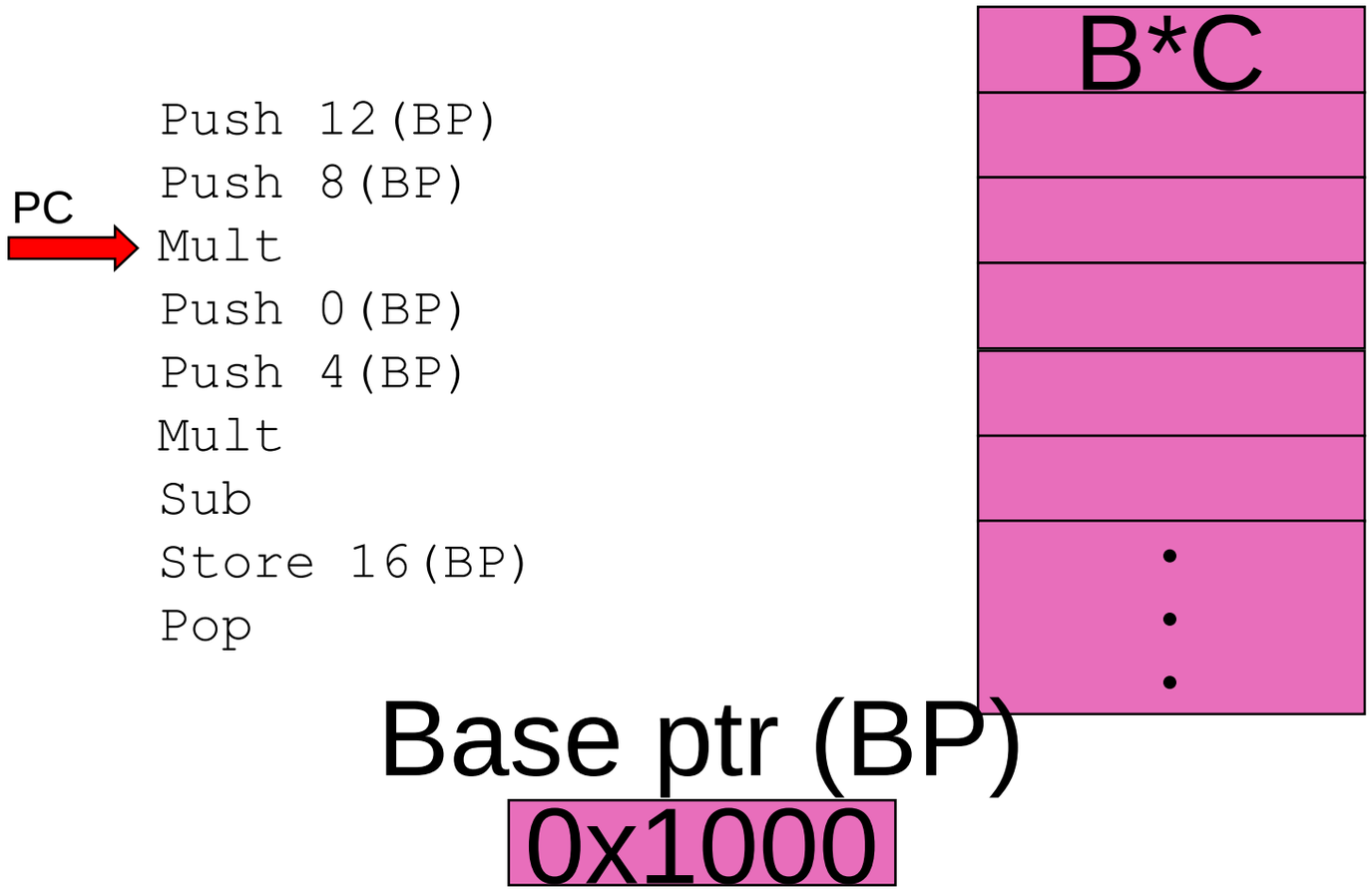
Stack Example: A = X * Y - B * C

- 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



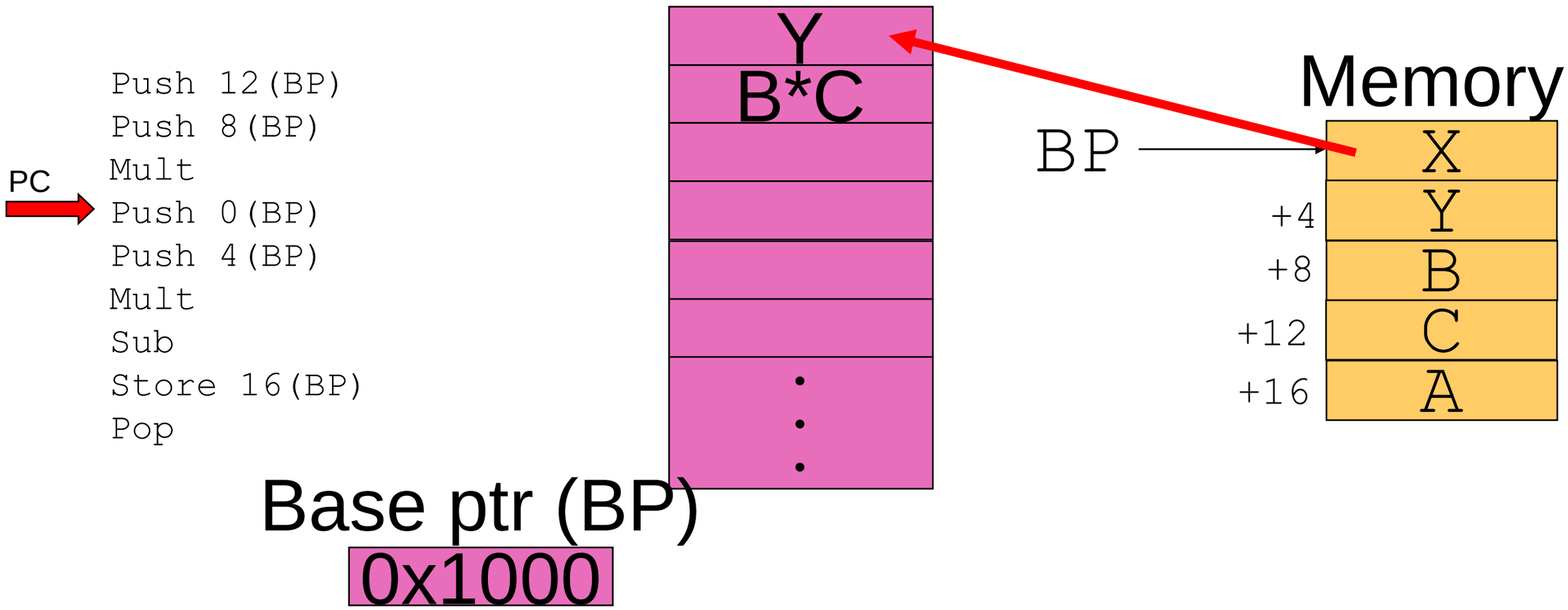
Stack Example: A = X * Y - B * C

- 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



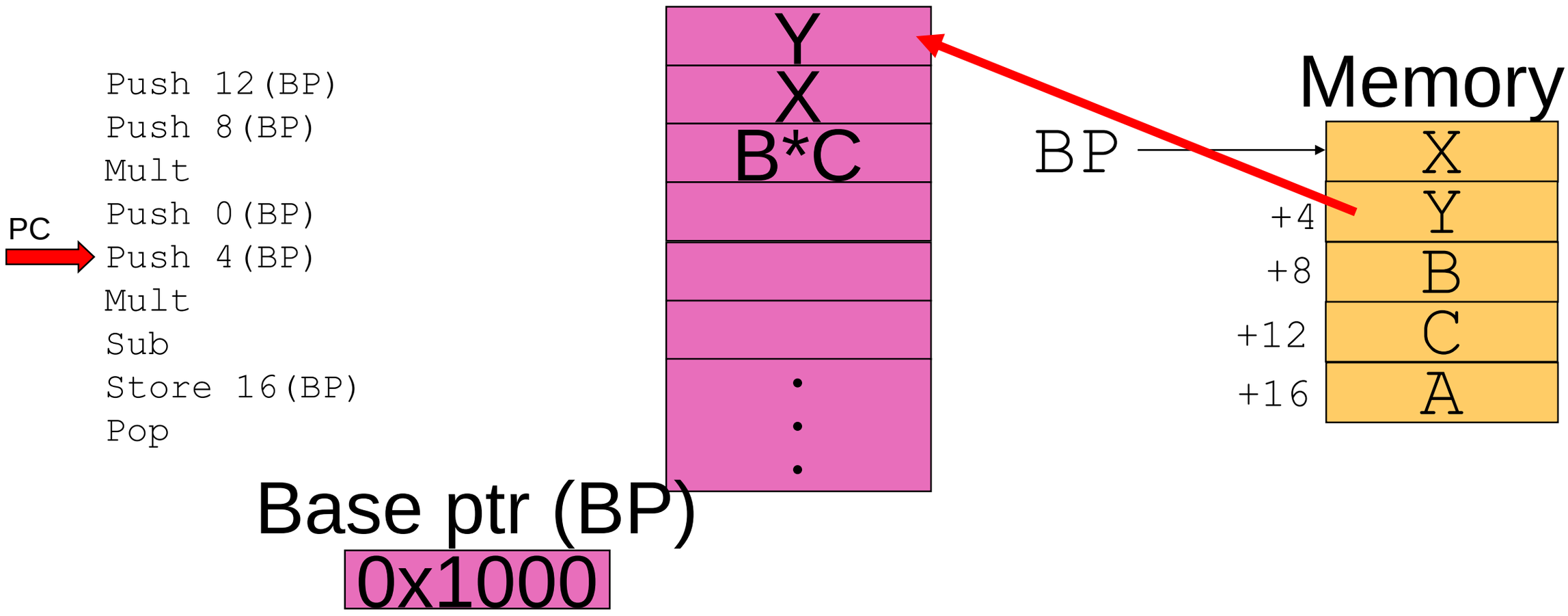
Stack Example: A = X * Y - B * C

- 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



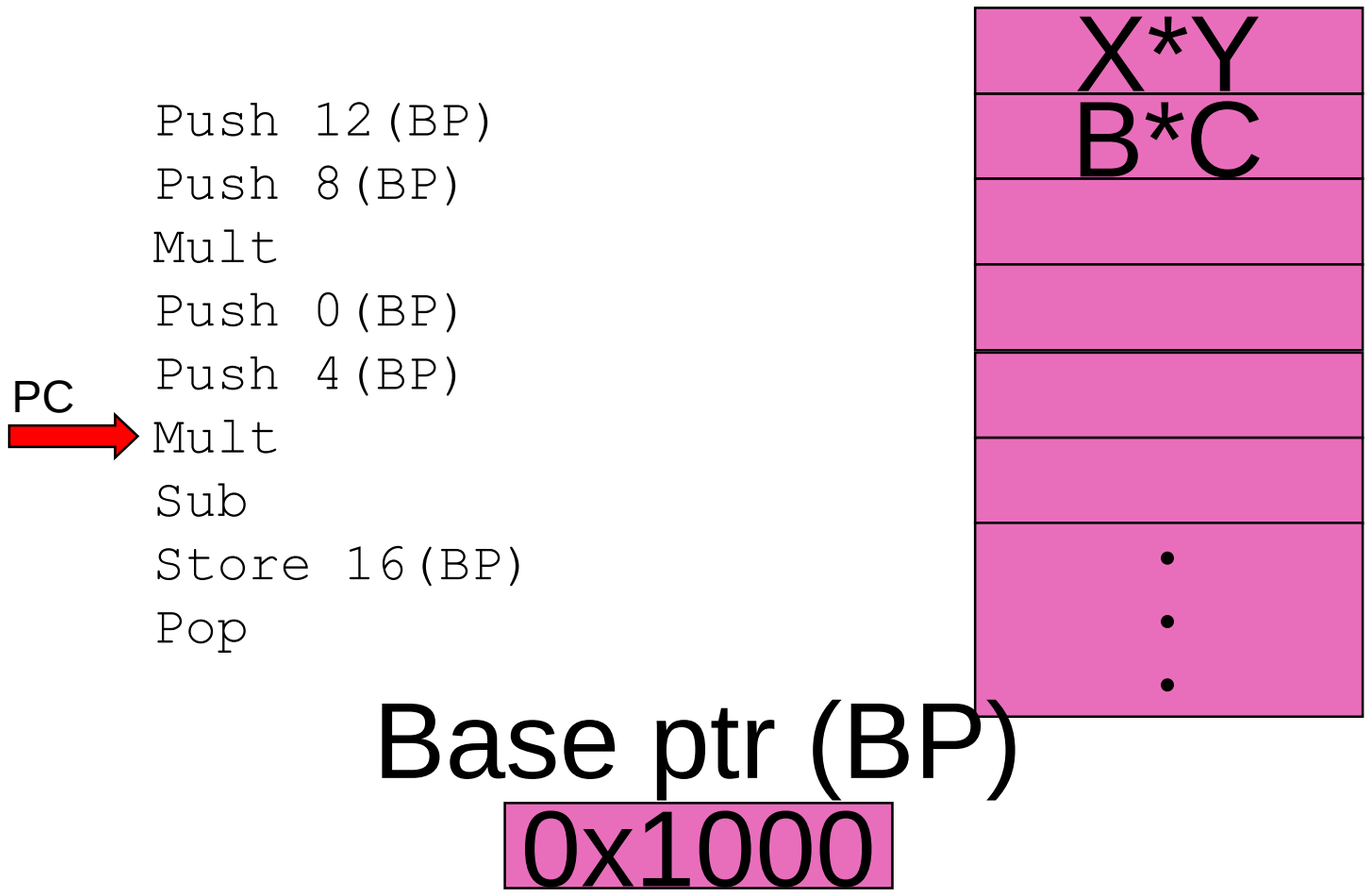
Stack Example: A = X * Y - B * C

- 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



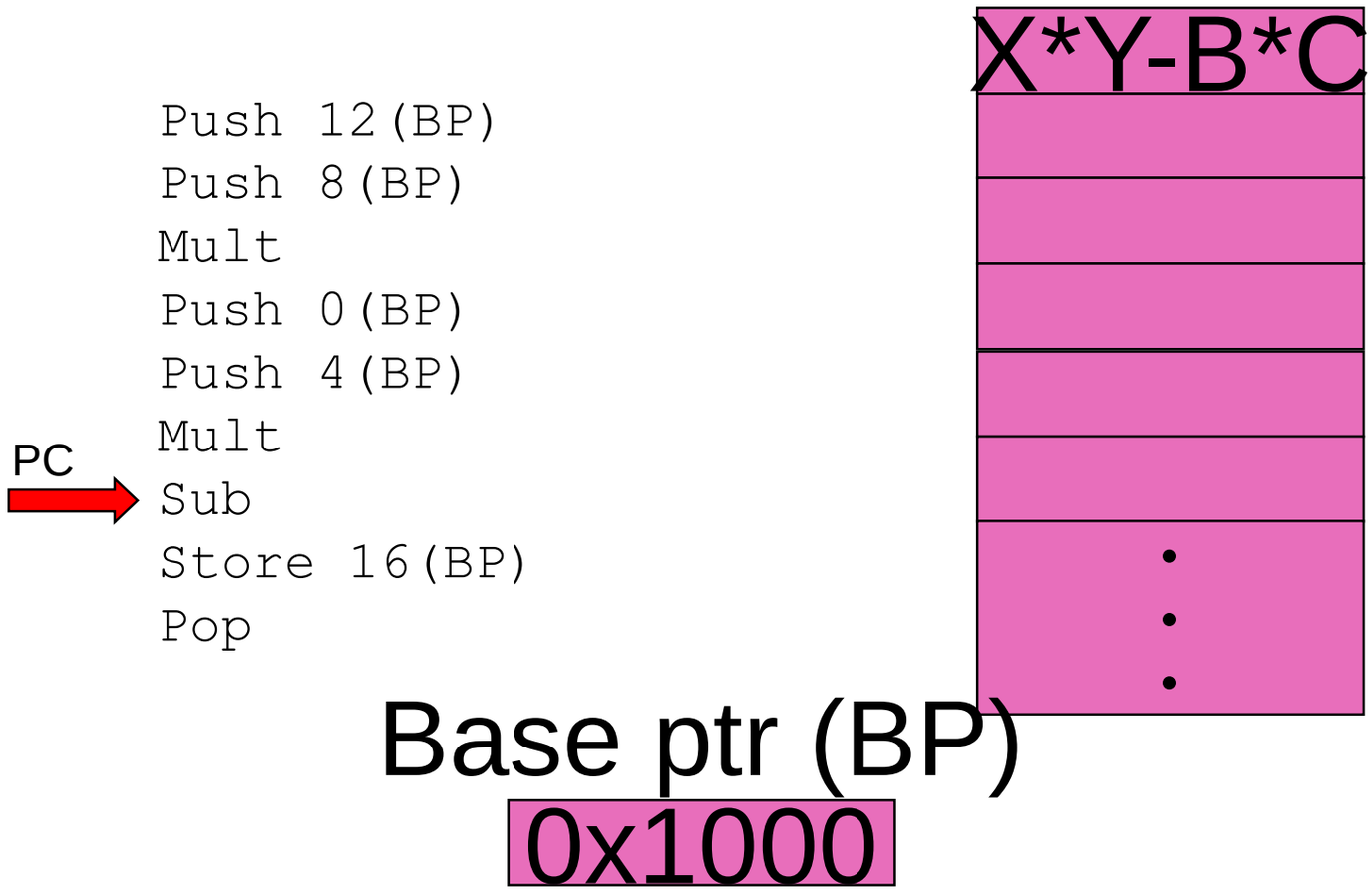
Stack Example: A = X * Y - B * C

- 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



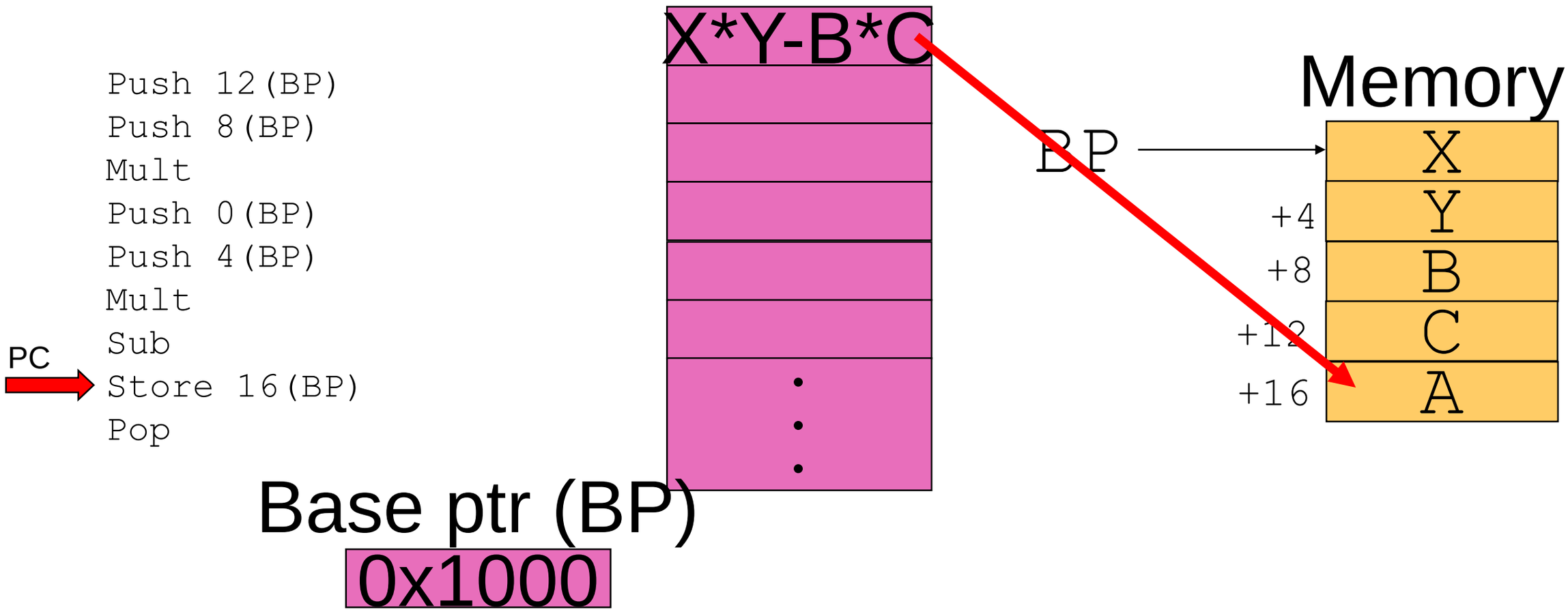
Stack Example: A = X * Y - B * C

- 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
- Store -- Store the top of the stack



RISC vs CISC

In the Beginning...

- 1964 -- The first ISA appears on the IBM System 360
- In the "good" old days
  - Initially, the focus was on usability by humans.
  - Lots of "user-friendly" instructions (remember the x86 addressing modes).
  - Memory was expensive, so code-density mattered.
  - Many processors were *microcoded* -- each instruction actually triggered the execution of a builtin function in the CPU. Simple hardware to execute complex instructions (but CPIs are very, very high)
- ...so...
  - Many, many different instructions, lots of bells and whistles
  - Variable-length instruction encoding to save space.
- ... their success had some downsides...
  - ISAs evolved organically.
  - They got messier, and more complex.

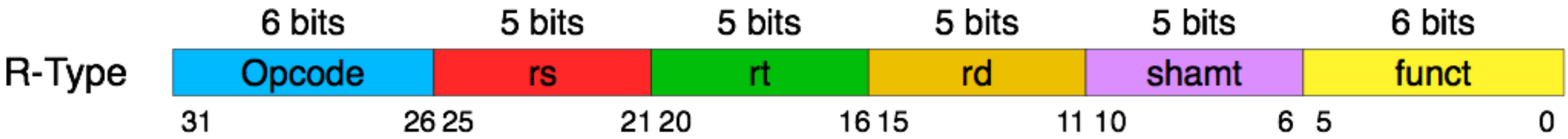
Things Changed

- In the modern era
  - Compilers write code, not humans.
  - Memory is cheap. Code density is unimportant.
  - Low CPI should be possible, but only for simple instructions
  - We learned a lot about how to design ISAs, how to let them evolve gracefully, etc.
- So, architects started with with a clean slate...

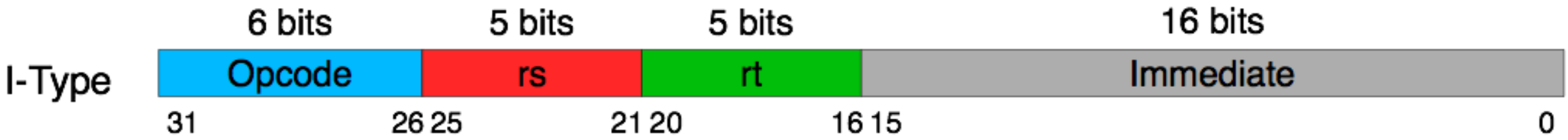
Reduced Instruction Set Computing (RISC)

- Simple, regular ISAs, mean simple CPUs, and simple CPUs can go fast.
  - Fast clocks.
  - Low CPI.
  - Simple ISAs will also mean more instruction (increasing IC), but the benefits should outweigh this.
- Compiler-friendly, not user-friendly.
  - Simple, regular ISAs, will be easy for compilers to use
  - A few, simple, flexible, fast operations that compiler can combine easily.
  - Separate memory access and data manipulation
    - Instructions access memory *or* manipulate register values. Not both.
    - "Load-store architectures" (like MIPS)

Instruction Formats



Arithmetic: Register[rd] = Register[rs] + Register[rt]
Register indirect jumps: PC = PC + Register[rs]



Arithmetic: Register[rd] = Register[rs] + Imm
Branches: If Register[rs] == Register[rt], goto PC + Immediate
Memory: Memory[Register[rs] + Immediate] = Register[rt]
Register[rt] = Memory[Register[rs] + Immediate]

Direct jumps: PC = Address
Syscalls, break, etc.

RISC Characteristics of MIPS

- All instructions have
  - <= 1 arithmetic op
  - <= 1 memory access
  - <= 2 register reads
  - <= 1 register write
  - <= 1 branch
  - It needs a small, fixed amount of hardware.
- Instructions operate on memory *or* registers *not both*
  - “Load/Store Architecture”
- Decoding is easy
  - Uniform opcode location
  - Uniform register location
  - Always 4 bytes -> the location of the next PC is to know.
- Uniform execution algorithm
  - Fetch
  - Decode
  - Execute
  - Memory
  - Write Back
- Compiling is easy
  - No complex instructions to reason about
  - No special registers
- The HW is simple
  - A skilled undergrad can build one in 10 weeks.
  - 33 instructions can run complex programs.

CISC: x86

- x86 is the prime example of CISC (there were many others long ago)
  - Many, many instruction formats. Variable length.
  - Many complex rules about which register can be used when, and which addressing modes are valid where.
  - Very complex instructions
  - Combined memory/arithmetic.
  - Special-purpose registers.
  - Many, many instructions.
- Implementing x86 correctly is almost intractable

Mostly RISC: ARM

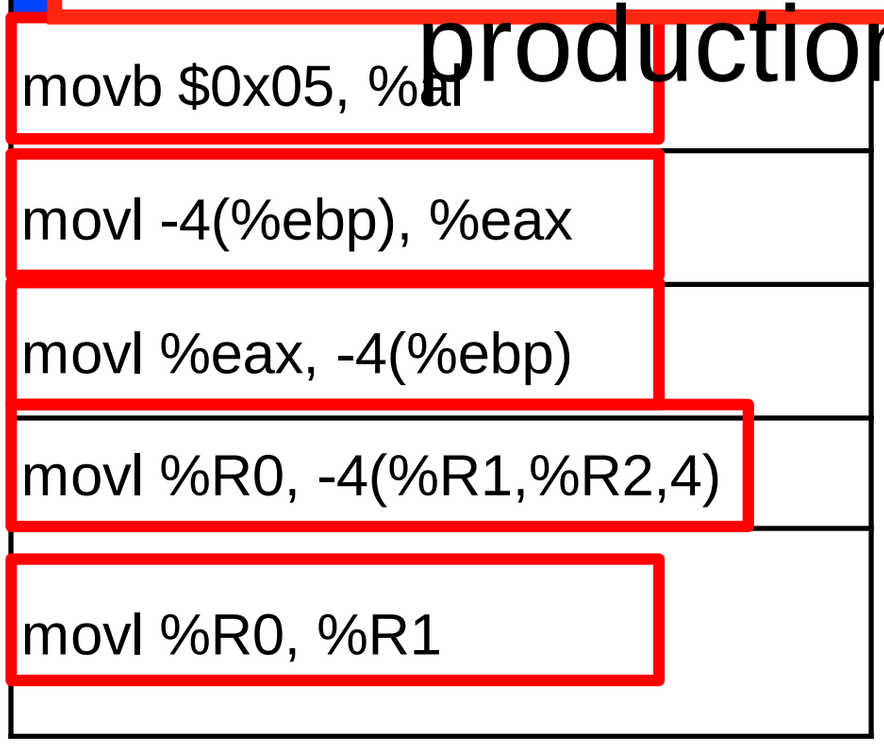
- ARM is somewhere in between
  - Four instruction formats. Fixed length.
  - General purpose registers (except the condition codes)
  - Moderately complex instructions, but they are still "regular" -- all instructions look more or less the same.
- ARM targeted embedded systems
  - Code density is important
  - Performance (and clock speed) is less critical
  - Both of these argue for more complex instructions.
  - But they can still be regular, easy to decode, and crafted to minimize hardware complexity
- Implementing an ARM processor is also tractable for 141L, but it would be harder than MIPS

RISCing the CISC

- Everyone believes that RISC ISAs are better for building fast processors.
- So, how do Intel and AMD build fast x86 processors?
  - Despite using a CISC ISA, these processors are actually RISC [illegible]
  - [illegible]-ops

The preceding was a dramatization. MIPS instructions were used for clarity and because I had some laying around.

No x86 instruction were harmed in the production of this slide.



VLIWing the CISC

- We can also get rid of x86 in software.
- Transmeta did this.
  - They built a processor that was completely hidden behind a "soft" implementation of the x86 instruction set.
  - Their system would translate x86 instruction into an internal VLIW instruction set and execute that instead.
    - Originally, their aim was high performance.
    - That turned out to be hard, so they focused low power instead.
- Transmeta eventually lost to Intel
  - Once Intel decided it cared about power (in part because Transmeta made the case for low-power x86 processors), it started producing very efficient CPUs.

The End