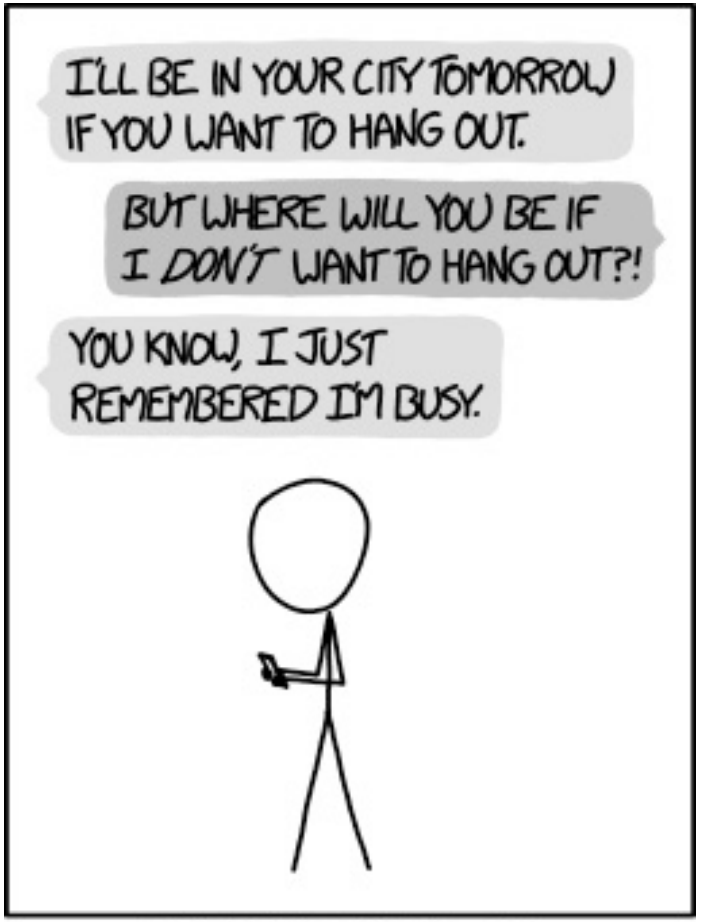
x86-64 Programming III

CSE 351 Winter 2022

Instructor:
Sam Wolfson

Teaching Assistants:
Angela Xu
Anirudh Kumar
Catherine Guevara
Dara Stotland
Harrison Bay
Ian Hsiao
Kevin Wang
Mara Kirdani-Ryan
Nick Durand
Sanjana Sridhar



http://xkcd.com/1652/

Relevant Course Information

- Lab 2 due next Friday (2/4)

- Midterm is in two weeks (take home, 2/8 – 2/11)
  - Open book; make notes and use midterm reference sheet
  - Individual, but discussion allowed via "Gilligan's Island Rule"
  - Mix of "traditional" and design/reflection questions
    - Form study groups and look at past exams!

Condition Codes (Explicit Setting: Compare)

- *Explicitly* set by **Compare** instruction
  - **`cmpq`** `src1, src2`
  - **`cmpq`** `a, b` sets flags based on `b-a`, but doesn't store

  - **CF=1** if carry out from MSB (good for *unsigned* comparison)
  - **ZF=1** if `a==b`
  - **SF=1** if `(b-a)<0` (if MSB is 1)
  - **OF=1** if *signed* overflow

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

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

Condition Codes (Explicit Setting: Test)

- *Explicitly* set by **Test** instruction
  - **`testq`** `src2, src1`
  - **`testq`** `a, b` sets flags based on `a&b`, but doesn't store
    - Useful to have one of the operands be a ***mask***
  - Can't have carry out (**`CF`**) or overflow (**`OF`**)
  - **`ZF=1`** if `a&b==0`
  - **`SF=1`** if `a&b<0` (signed)

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

Example Condition Code Setting

- Assuming that `%al` = `0x80` and `%bl` = `0x81`, which flags (CF, ZF, SF, OF) are set when we execute **`cmpb %al, %bl`**?

Using Condition Codes: Jumping

- `j*` Instructions
  - Jumps to ***target*** (an address) based on condition codes

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

Using Condition Codes: Setting

- `set*` Instructions
  - Set low-order byte of `dst` to 0 or 1 based on condition codes
  - Does not alter remaining 7 bytes

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

Reading Condition Codes

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rax	return value

- `set*` Instructions
  - Set a low-order byte to 0 or 1 based on condition codes
  - Operand is byte register (*e.g.*, `%al`) or a byte in memory
  - Do not alter remaining bytes in register
    - Typically use `movzbl` (zero-extended `mov`) to finish job

```
int gt(long x, long y)
{
  return x > y;
}
```

```
cmpq    %rsi, %rdi    #
setg    %al           #
movzbl  %al, %eax     #
ret
```

Reading Condition Codes

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rax	return value

- set* Instructions
  - Set a low-order byte to 0 or 1 based on condition codes
  - Operand is byte register (*e.g.*, %al) or a byte in memory
  - Do not alter remaining bytes in register
    - Typically use movzbl (zero-extended mov) to finish job

```
int gt(long x, long y)
{
  return x > y;
}
```

```
cmpq    %rsi, %rdi    # Compare x:y
setg    %al           # Set when >
movzbl  %al, %eax     # Zero rest of %rax
ret
```

Choosing instructions for conditionals

- All arithmetic instructions set condition flags based on result of operation (`op`)
  - Conditionals are comparisons against 0
- Come in instruction *pairs*

```
      addq 5, (p)
je:    *p+5 == 0
jne:   *p+5 != 0
jg:    *p+5 >  0
jl:    *p+5 <  0
```

```
      orq a, b
je:    b|a == 0
jne:   b|a != 0
jg:    b|a >  0
jl:    b|a <  0
```

		(op) s, d
je	"Equal"	d (op) s == 0
jne	"Not equal"	d (op) s != 0
js	"Sign" (negative)	d (op) s < 0
jns	(non-negative)	d (op) s >= 0
jg	"Greater"	d (op) s > 0
jge	"Greater or equal"	d (op) s >= 0
jl	"Less"	d (op) s < 0
jle	"Less or equal"	d (op) s <= 0
ja	"Above" (unsigned >)	d (op) s > 0U
jb	"Below" (unsigned <)	d (op) s < 0U

Choosing instructions for conditionals

- Reminder: `cmp` is like `sub`, `test` is like `and`
  - Result is not stored anywhere

		cmp a,b	**test a,b**
je	"Equal"	b == a	b&a == 0
jne	"Not equal"	b != a	b&a != 0
js	"Sign" (negative)	b−a < 0	b&a < 0
jns	(non-negative)	b−a >=0	b&a >= 0
jg	"Greater"	b > a	b&a > 0
jge	"Greater or equal"	b >= a	b&a >= 0
jl	"Less"	b < a	b&a < 0
jle	"Less or equal"	b <= a	b&a <= 0
ja	"Above" (unsigned >)	$b >_U a$	b&a > 0U
jb	"Below" (unsigned <)	$b <_U a$	b&a < 0U

```
    cmpq 5, (p)
je:   *p == 5
jne:  *p != 5
jg:   *p >  5
jl:   *p <  5
```

```
    testq a, a
je:    a == 0
jne:   a != 0
jg:    a >  0
jl:    a <  0
```

```
    testb a, 0x1
je:    a_LSB == 0
jne:   a_LSB == 1
```

Choosing instructions for conditionals

		cmp a,b	test a,b
je	"Equal"	b == a	b&a == 0
jne	"Not equal"	b != a	b&a != 0
js	"Sign" (negative)	b-a < 0	b&a < 0
jns	(non-negative)	b-a >=0	b&a >= 0
jg	"Greater"	b > a	b&a > 0
jge	"Greater or equal"	b >= a	b&a >= 0
jl	"Less"	b < a	b&a < 0
jle	"Less or equal"	b <= a	b&a <= 0
ja	"Above" (unsigned >)	$b >_U a$	b&a > 0U
jb	"Below" (unsigned <)	$b <_U a$	b&a < 0U

Register	Use(s)
%rdi	argument x
%rsi	argument y
%rax	return value

```
if (x < 3) {
  return 1;
}
return 2;
```

```
  cmpq $3, %rdi
  jge T2
T1: # x < 3:
  movq $1, %rax
  ret
T2: # !(x < 3):
  movq $2, %rax
  ret
```

Practice Question 1

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rax	return value

A.
```
cmpq  %rsi, %rdi
jle   .L4
```

B.
```
cmpq  %rsi, %rdi
jg    .L4
```

C.
```
testq %rsi, %rdi
jle   .L4
```

D.
```
testq %rsi, %rdi
jg    .L4
```

E. We're lost…

```
long absdiff(long x, long y)
{
  long result;
  if (x > y)
    result = x-y;
  else
    result = y-x;
  return result;
}
```

```
absdiff:
    ____________________
    ____________________
                          # x > y:
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret
.L4:                      # x <= y:
    movq    %rsi, %rax
    subq    %rdi, %rax
    ret
```

Reading Review

- Terminology:
  - Label, jump target
  - Program counter
  - Jump table, indirect jump

- Questions from the Reading?

Labels

```
swap:
    movq   (%rdi), %rax
    movq   (%rsi), %rdx
    movq   %rdx, (%rdi)
    movq   %rax, (%rsi)
    ret
```

```
max:
    movq %rdi, %rax
    cmpq %rsi, %rdi
    jg   done
    movq %rsi, %rax
done:
    ret
```

- A jump changes the program counter (`%rip`)
  - `%rip` tells the CPU the *address* of the next instruction to execute
- **Labels** give us a way to refer to a specific instruction in our assembly/machine code
  - Associated with the *next* instruction found in the assembly code (ignores whitespace)
  - Each ***use*** of the label will eventually be replaced with something that indicates the final address of the instruction that it is associated with

x86 Control Flow

- Condition codes
- Conditional and unconditional branches
- **Loops**
- Switches

Expressing with Goto Code

```
long absdiff(long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff_j(long x, long y)
{
    long result;
    int ntest = (x <= y);
    if (ntest) goto Else;
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```

- C allows `goto` as means of transferring control
  - Closer to assembly programming style
  - Don't do this!! Bad!!!

Compiling Loops

C/Java code:

```
while ( sum != 0 ) {
    <loop body>
}
```

Assembly code:

```
loopTop:    testq %rax, %rax
            je    loopDone
            <loop body code>
            jmp   loopTop
loopDone:
```

- Other loops compiled similarly
  - Will show variations and complications in coming slides, but may skip a few examples in the interest of time
- Most important to consider:
  - When should conditionals be evaluated? (*while* vs. *do-while*)
  - How much jumping is involved?

Compiling Loops

While Loop:

C:
```
while ( sum != 0 ) {
    <loop body>
}
```

x86-64:
```
loopTop:    testq %rax, %rax
            je    loopDone
            <loop body code>
            jmp   loopTop
loopDone:
```

Do-while Loop:

C:
```
do {
    <loop body>
} while ( sum != 0 )
```

x86-64:
```
loopTop:
            <loop body code>
            testq  %rax, %rax
            jne    loopTop
loopDone:
```

While Loop (ver. 2):

C:
```
while ( sum != 0 ) {
    <loop body>
}
```

x86-64:
```
            testq %rax, %rax
            je    loopDone
loopTop:
            <loop body code>
            testq %rax, %rax
            jne   loopTop
loopDone:
```

For-Loop → While-Loop

For-Loop:

```
for (Init; Test; Update) {
    Body
}
```

While-Loop Version:

```
Init;
while (Test) {
    Body
    Update;
}
```

Caveat: C and Java have `break` and `continue`

- Conversion works fine for break
  - Jump to same label as loop exit condition
- But not continue: would skip doing *Update*, which it should do with for-loops
  - Introduce new label at *Update*

Practice Question 2

- The following is assembly code for a for-loop; identify the corresponding parts (Init, Test, Update)
  - i → %eax, x → %rdi, y → %esi

```
Line
  1          movl    $0, %eax
  2  .L2:    cmpl    %esi, %eax
  3          jge     .L4
  4          movslq  %eax, %rdx
  5          leaq    (%rdi,%rdx,4), %rcx
  6          movl    (%rcx), %edx
  7          addl    $1, %edx
  8          movl    %edx, (%rcx)
  9          addl    $1, %eax
 10          jmp     .L2
 11  .L4:
```

Summary

- Control flow in x86 determined by Condition Codes
  - Showed **C**arry, **Z**ero, **S**ign, and **O**verflow, though others exist
  - Set flags with arithmetic instructions (implicit) or Compare and Test (explicit)
  - Set instructions read out flag values
  - Jump instructions use flag values to determine next instruction to execute
  - Most control flow constructs (*e.g.*, if-else, for-loop, while-loop) can be implemented in assembly using combinations of conditional and unconditional jumps

How did we get here?

- Loops: do the same thing over and over again
- Distinction between *creating* and *running* programs
  - Values inform which is "better" and which is "worse"
- Modern hardware, modern "house", historical relics
- Decisions were not an accident!
  - Priorities may have been dated, inconsistent, prejudiced

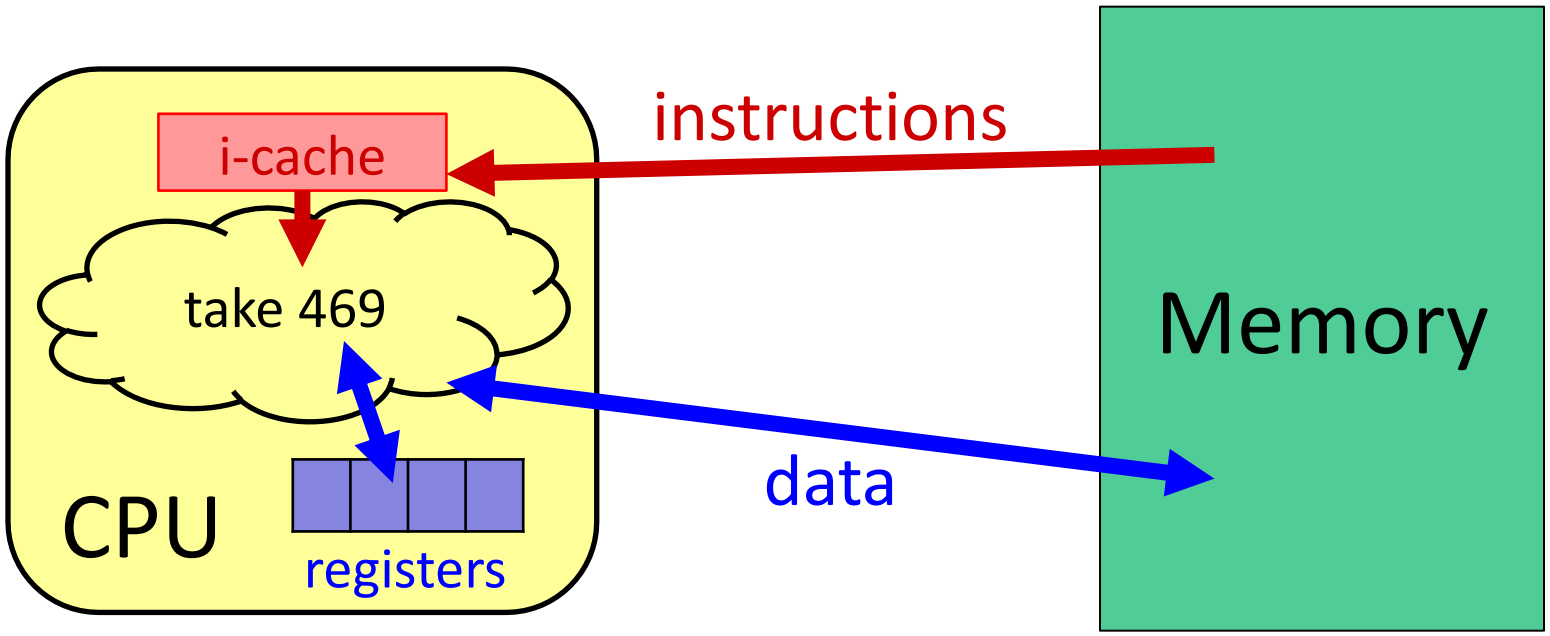
- *Who* acted as the first computers?
- **What were the prevailing narratives that informed computing during its modern* incarnation?**

* starting in the early 20th century

Prevailing Narratives in Computer Science

- **"Boring, repetitive work" should be automated or augmented for efficiency and profit**
- "Boring, repetitive work" is "robot work"
- Augmentation is highly valued and exclusive

Hardware: 351 View (version 1)



- More CPU details:
  - Instructions are held temporarily in the instruction cache
  - Other data are held temporarily in registers
- Instruction fetching is hardware-controlled
- Data movement is programmer-controlled (assembly)

(Modern) Hardware: Historic View

- **Computer**: one who computes

The women of Bletchley Park, Credit: BBC

- Mostly white cis-women
- "Boring, repetitive work", doing math quickly

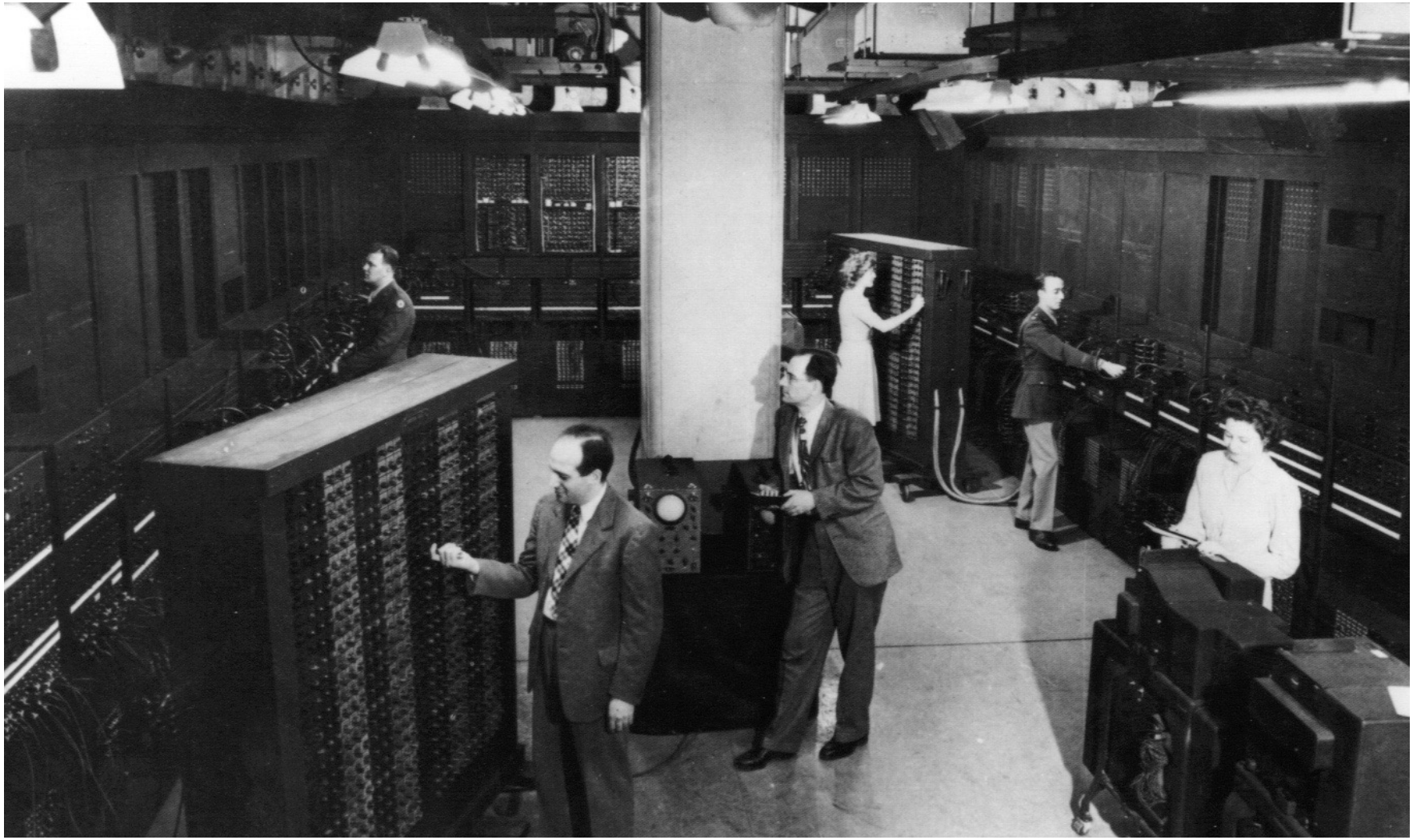
Computing in the US

- **Computer**: one who computes
- Observatory calculations @ Harvard (1870s)

Human Computers at NACA, Credit: NASA

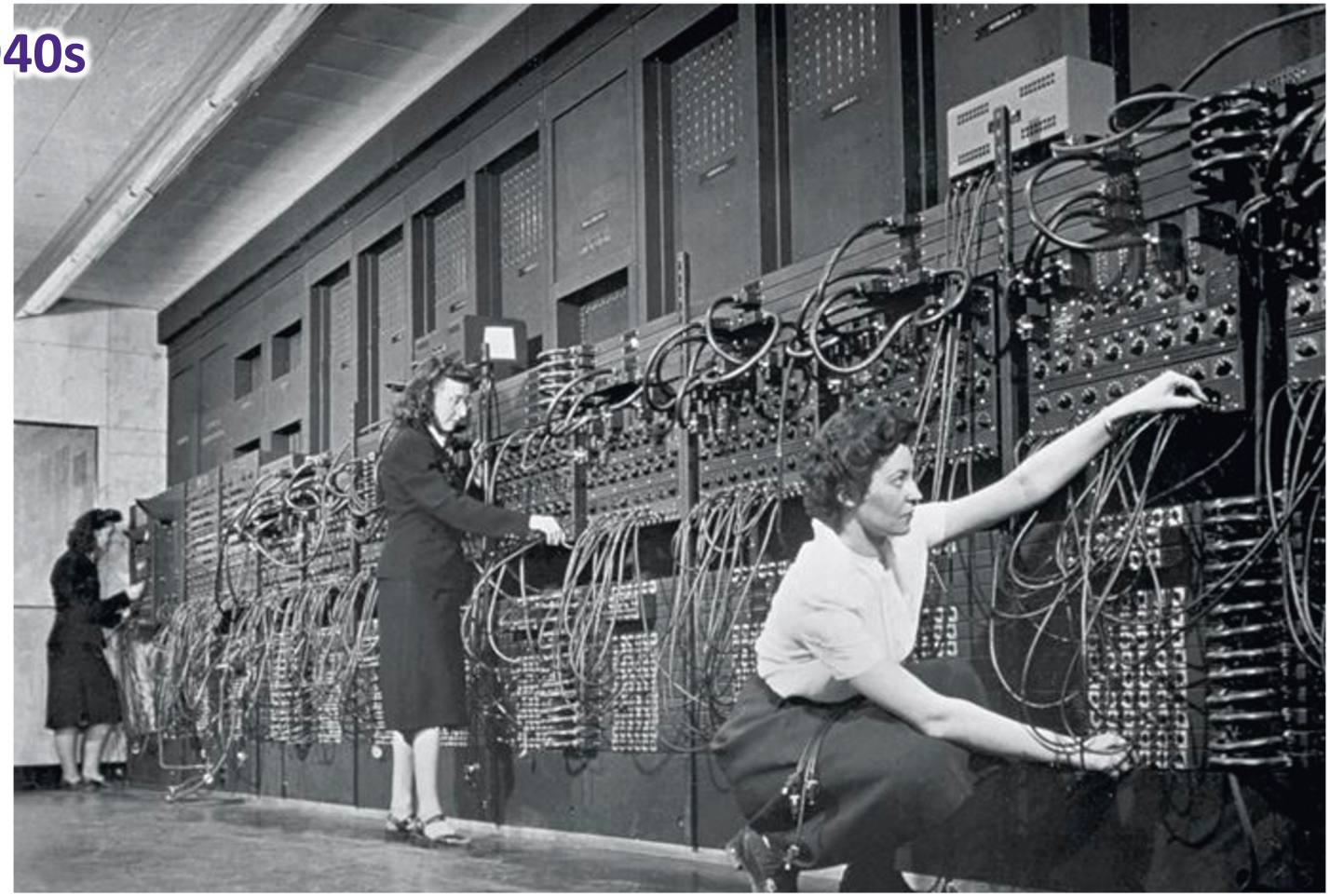
Human Computers at JPL, Credit: JPL

The ENIAC: Augmenting/Automating



Programming, historically

1940s



Jean Jennings (left), Marlyn Wescoff (center), and Ruth Lichterman program ENIAC at the University of Pennsylvania, circa 1946.
Photo: Corbis
http://fortune.com/2014/09/18/walter-isaacson-the-women-of-eniac/

The Computer Girls

BY LOIS MANDEL

A trainee gets $8,000 a year ...a girl "senior systems analyst" gets $20,000—and up! Maybe it's time to investigate....

Ann Richardson, IBM systems engineer, designs a bridge via computer. Above (left) she checks her facts with fellow systems engineer, Marvin V. Fuchs. Right, she feeds facts into the computer. Below, Ann demonstrates on a viewing screen how her facts designed the bridge, and makes changes with a "light pen."

Twenty years ago, a girl could be a secretary, a school teacher . . . maybe a librarian, a social worker or a nurse. If she was really ambitious, she could go into the professions and compete with men . . . usually working harder and longer to earn less pay for the same job.

Now have come the big, dazzling computers—and a whole new kind of work for women: programming. Telling the miracle machines what to do and how to do it. Anything from predicting the weather to sending out billing notices from the local department store.

And if it doesn't sound like woman's work—well, it just is.

("I had this idea I'd be standing at a big machine and pressing buttons all day long," says a girl who programs for a Los Angeles bank. I couldn't have been further off the track. I figure out how the computer can solve a problem, and then instruct the machine to do it."

"It's just like planning a dinner," explains Dr. Grace Hopper, now a staff scientist in systems programming for Univac. (She helped develop the first electronic digital computer, the Eniac, in 1946.) "You have to plan ahead and schedule everything so it's ready when you need it. Programming requires patience and the ability to handle detail. Women are 'naturals' at computer programming."

What she's talking about is *aptitude*—the one most important quality a girl needs to become a programmer. She also needs a keen, logical mind. And if that zeroes out the old Billie Burke-Gracie Allen image of femininity, it's about time, because this is the age of the Computer Girls. There are twenty thousand of them in the United *(cont. on page 54)*

"People like Grace Hopper were very consciously mobilizing gender stereotypes to get women in."

Janet Abbate, *Recoding Gender*

Prevailing Narratives in Computer Science

- **"Boring, repetitive work" should be automated or augmented for efficiency and profit**
  - Consistently eliminating jobs of marginalized folks
- "Boring, repetitive work" is "robot work"
- Augmentation is highly valued and exclusive

Prevailing Narratives in Computer Science

- "Boring, repetitive work" should be automated or augmented for efficiency and profit
  - Consistently eliminating jobs of marginalized folks
- **"Boring, repetitive work" is "robot work"**
- Augmentation is highly valued and exclusive

Historic Robots

- *Robot:* (Czech) compulsory service
  - Slav *robota*: servitude, hardship
- Robots: tool to replace "unskilled" work, servants

You'll Own "Slaves" by 1965

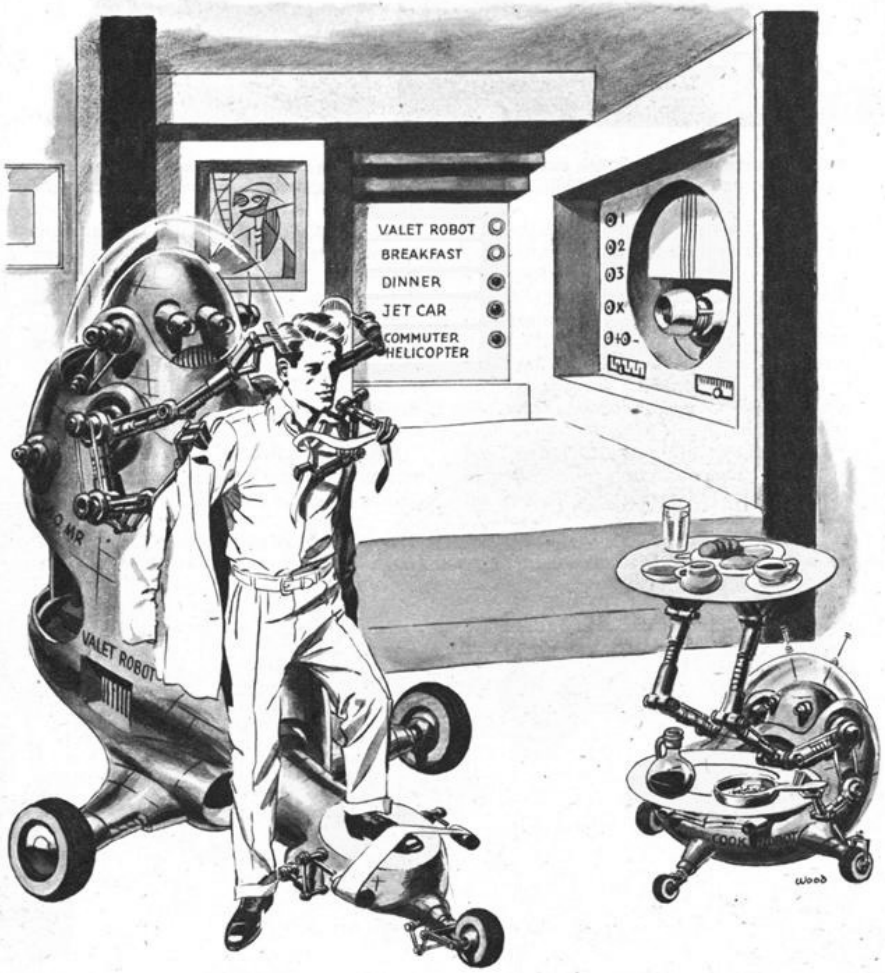
The robots are coming! When they do, you'll command a host of push-button servants.

By O. O. Binder

IN 1863, Abe Lincoln freed the slaves. But by 1965, slavery will be back! We'll all have personal slaves again, only this time we won't fight a Civil War over them. Slavery will be here to stay.

Don't be alarmed. We mean robot "slaves." Let's take a peek into the future to see what the Robot Age will bring. It is a morning of 1965. . .

Robots will dress you, comb your hair and serve meals in a jiffy.



You are gently awakened by soft chimes from your robot clock, which also turns up the heat, switches on radio news and signals your robot valet, whom you've affectionately named "Jingles." He turns on your shower, dries you with a blast of warm air, and runs an electric shaver over your stubble. Jingles helps you dress, tying your necktie perfectly and parting your hair within a millimeter of where you like it.

Down in the kitchen, Steela, the robot cook, opens a door in her own alloy body and withdraws eggs, toast and coffee from her built-in stove. Then she dumps the dishes back in and you hear her internal dishwasher bubbling as you leave for the garage.

In your robot car you simply set a dial for your destination and relax. Your automatic auto does the rest—following a radar beam downtown, passing other cars, slowing down in speed zones, gently applying radar brakes when necessary, even gassing up when your tank is empty. You give a friendly wave to robot traffic cops who break up all traffic jams with electronic speed and perception. Suddenly you hear gun shots. A thief is emptying his gun at a robot cop, who just keeps coming, bullets bouncing from his steel chest. The panicky thug races away in his car but the robot cop shifts himself into eighth gear and overtakes the bandit's car on foot.

If you work at an office, your robot secretary takes dictation on voice tapes and types internally at the same time, handing you your letter as soon as you say "yours truly." If you go golfing, the secretary answers the phone, records any messages, and also delivers any prerecorded message of yours.

At home, your robot reciter reads books to you from your microfilm library. His eye can see microscopic prints. Or you play chess with a robot companion, matching your wits against an electronic brain.

In 1956 research scientists already devised robot game players who always won against human opponents. Of course the 1965 robots can be adjusted as you wish by buttons for high, average or low skill.

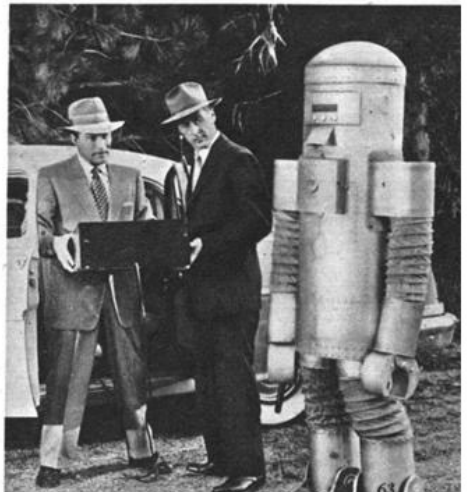
When a heavy snow falls you don't have to shovel the walk. Neither does your robot caretaker. He merely sprays cheap atomic heat around the grounds, melting the snow as fast as it falls. Yours is a robot home, too, turning all day on a foundation turntable to enjoy the utmost benefits of the sun.

At bedtime, you snap on the robot guard who detects any burglars electronically. It's a cheaper version of the robot alarm system in 1956, guarding precious documents like the original Constitution, in the National Archives Building.

During the night, no mice or rats can escape the super-sensitive ears and infra-red eyes of your roving robot cat. Back in 1956 scientists experimented with the first robot animals, such as the robot mole that could follow light beams, the robot moth dancing around flames and robot mice finding their way out of mazes.

Fanciful, this picture of the near future? A foretaste of such robot wonders

Metal star of Zombies Of The Stratosphere heeds his masters in science-fiction movie.



These racist origins persist

Twitter Engineering
@TwitterEng

We're starting with a set of words we want to move away from using in favor of more inclusive language, such as:

Avoid non-inclusive language	⇒	Prefer inclusive versions
Whitelist	⇒	Allowlist
Blacklist	⇒	Denylist
Master/slave	⇒	Leader/follower, primary/replica, primary/standby
Grandfathered	⇒	Legacy status
Gendered pronouns (e.g. guys)	⇒	Folks, people, you all, y'all
Gendered pronouns (e.g. he/him/his)	⇒	They, them, their
Man hours	⇒	Person hours, engineer hours
Sanity check	⇒	Quick check, confidence check, coherence check
Dummy value	⇒	Placeholder value, sample value

12:52 PM · Jul 2, 2020 · Twitter Web App

*This isn't to praise Twitter; this was the first list I found

Prevailing Narratives in Computer Science

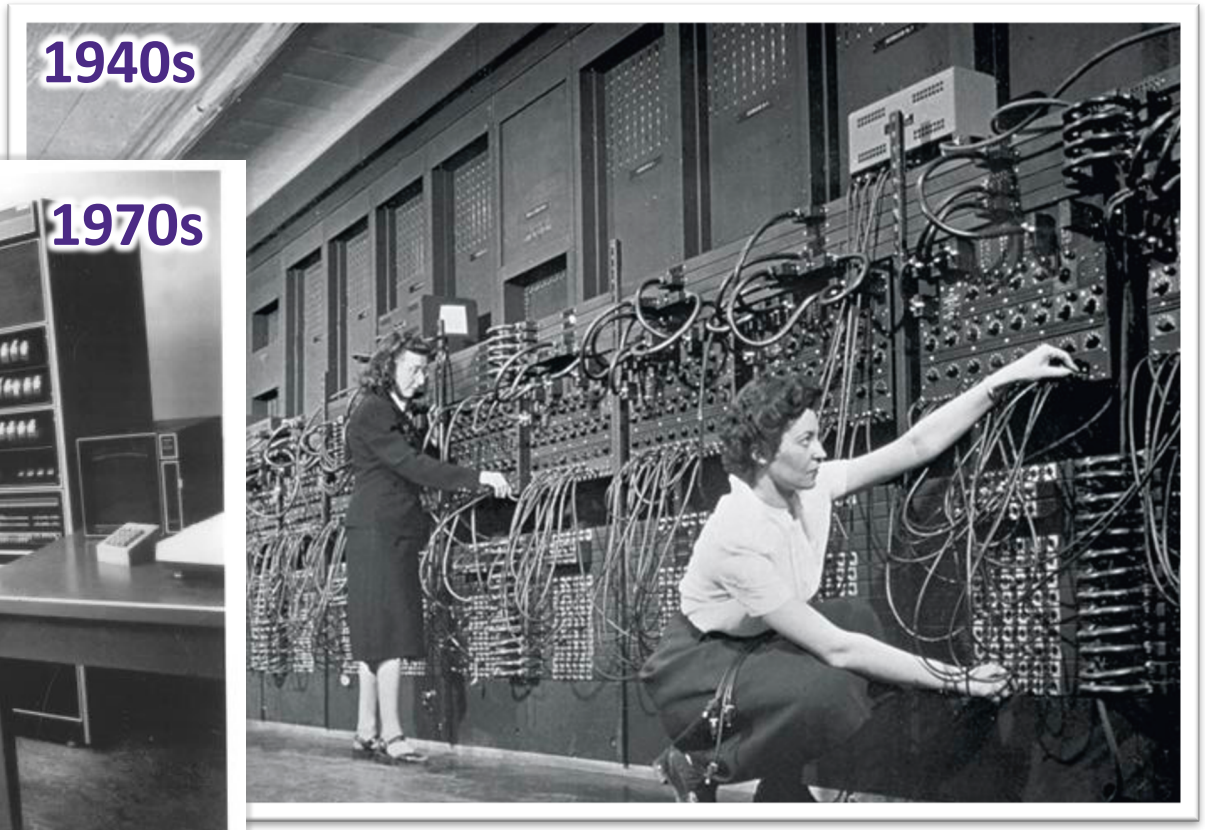
- "Boring, repetitive work" should be automated or augmented for efficiency and profit
  - Consistently eliminating jobs of marginalized folks
- **"Boring, repetitive work" is "robot work"**
  - Performed by those deemed *less than human*
  - Robot work should be done by robots (non-human)
    - *"Robot work": anything unvalued by those with systemic power*
  - If the task can't be automated, use people (less-human)
    - Frequently, this ends up being marginalized people, who later have their jobs automated
- Augmentation is highly valued and exclusive

Prevailing Narratives in Computer Science

- "Boring, repetitive work" should be automated or augmented for efficiency and profit
  - Consistently eliminating jobs of marginalized folks
- "Boring, repetitive work" is "robot work"
  - Performed by those deemed *less than human*
  - Robot work should be done by robots (non-human)
    - *"Robot work": anything unvalued by the powerful*
  - If the task can't be automated, use people (less-human)
    - Frequently, this ends up being marginalized people, who later have their jobs automated
- **Augmentation is highly valued and exclusive**

Programming, historically

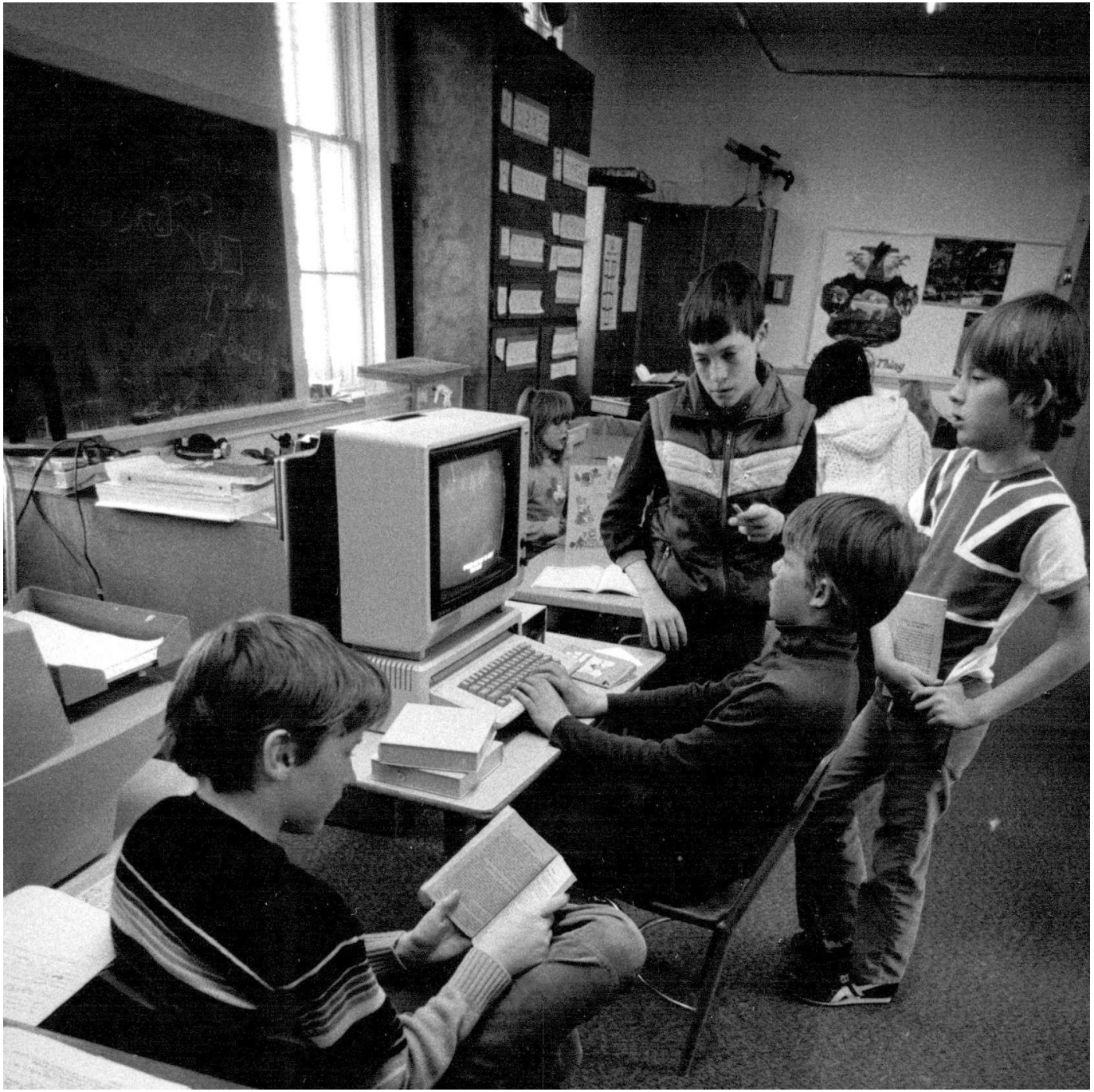
https://s-media-cache-ak0.pinimg.com/564x/91/37/23/91372375e2e6517f8af128aab655e3b4.jpg



Jean Jennings (left), Marlyn Wescoff (center), and Ruth Lichterman program ENIAC at the University of Pennsylvania, circa 1946.
Photo: Corbis
http://fortune.com/2014/09/18/walter-isaacson-the-women-of-eniac/

Modern Robots: Personal Computers


DARTMOUTH



Prevailing Narratives in Computer Science

- "Boring, repetitive work" should be automated or augmented for efficiency and profit
  - Consistently eliminating jobs of marginalized folks
- "Boring, repetitive work" is "robot work"
  - Performed by those deemed *less than human*
  - Robot work should be done by robots (non-human)
    - *"Robot work": anything unvalued by the powerful*
  - If the task can't be automated, use people (less-human)
    - Frequently, this ends up being marginalized people, who later have their jobs automated
- **Augmentation is highly valued and exclusive**
  - "Boring, repetitive work" is more available

Automation – it's complicated

- I don't mean to vilify automation indiscriminately
  - Adaptive cruise control, autopilot, medical devices
- **However**, we need to consider the values that inform whether specific tasks should be automated
  - *Why* should this task be automated?
    - 737 MAX – Boeing wanted to save money
  - *Who* does this automation seek to benefit?
    - Self driving cars replacing rideshare and taxi drivers
    - Benefit Uber and Lyft executives and shareholders
- Who do **you** want to benefit from your work?
  - Bill Gates?