CS 110
Computer Architecture
Review for Midterm I

Instructor:
Sören Schwertfeger

http://shtech.org/courses/ca/

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkley's CS61C

Midterm I

- Date: Friday, Apr. 8
- Time: 10:15 - 11:55 (normal lecture slot)
- Venue: H2 109 + H2 103
- One table per student
- Closed book:
  - You can bring **one** A4 page with notes (both sides; Chinese is OK): Write you Chinese and pingying name on the top!
  - You will be provided with the MIPS "green sheet"
  - No other material allowed!

Midterm I

- Switch cell phones **off**! (not silent mode – off!)
  - Put them in your bags.
- Bags under the table. Nothing except paper, pen, 1 drink, 1 snack on the table!
- No other electronic devices are allowed!
  - No ear plugs, music, …
- Anybody touching any electronic device will FAIL the course!
- Anybody found cheating (copy your neighbors answers, additional material, ...) will FAIL the course!

Midterm I

- Ask questions today!
- Next weeks discussion is Q&A session
  - Suggest topics for review in piazza!

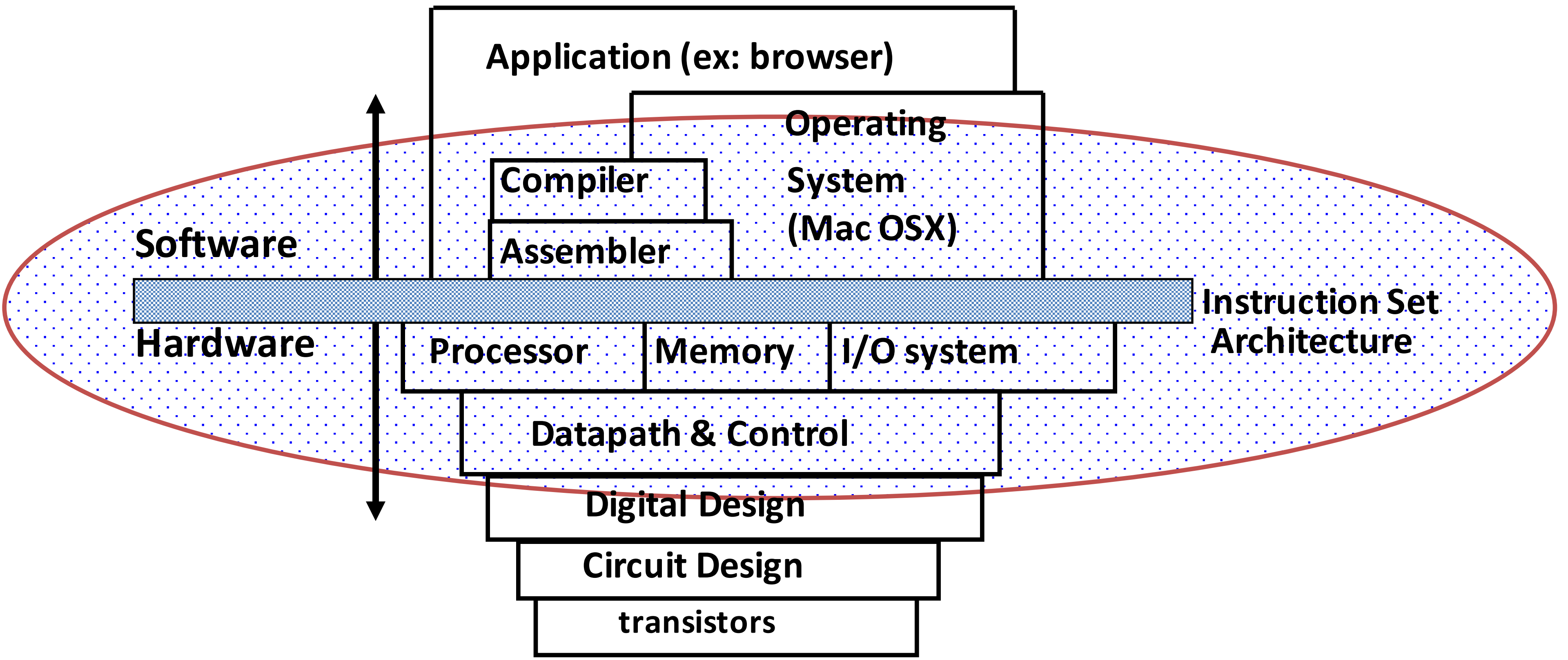
- This review session does not/ can not cover all possible topics!

- Please answer the polls – anonymous!

Lab next Monday

- No Lab on Monday (April 4)
- Do your lab-work...
- Check-off and help options:
  - Beginning of next weeks Lab
  - OH of Zhu Chen 朱晨 and Xu Qingwen 徐晴雯
  - Lab 2 and 3 (Tuesday, Thursday 3pm)

Old School Machine Structures

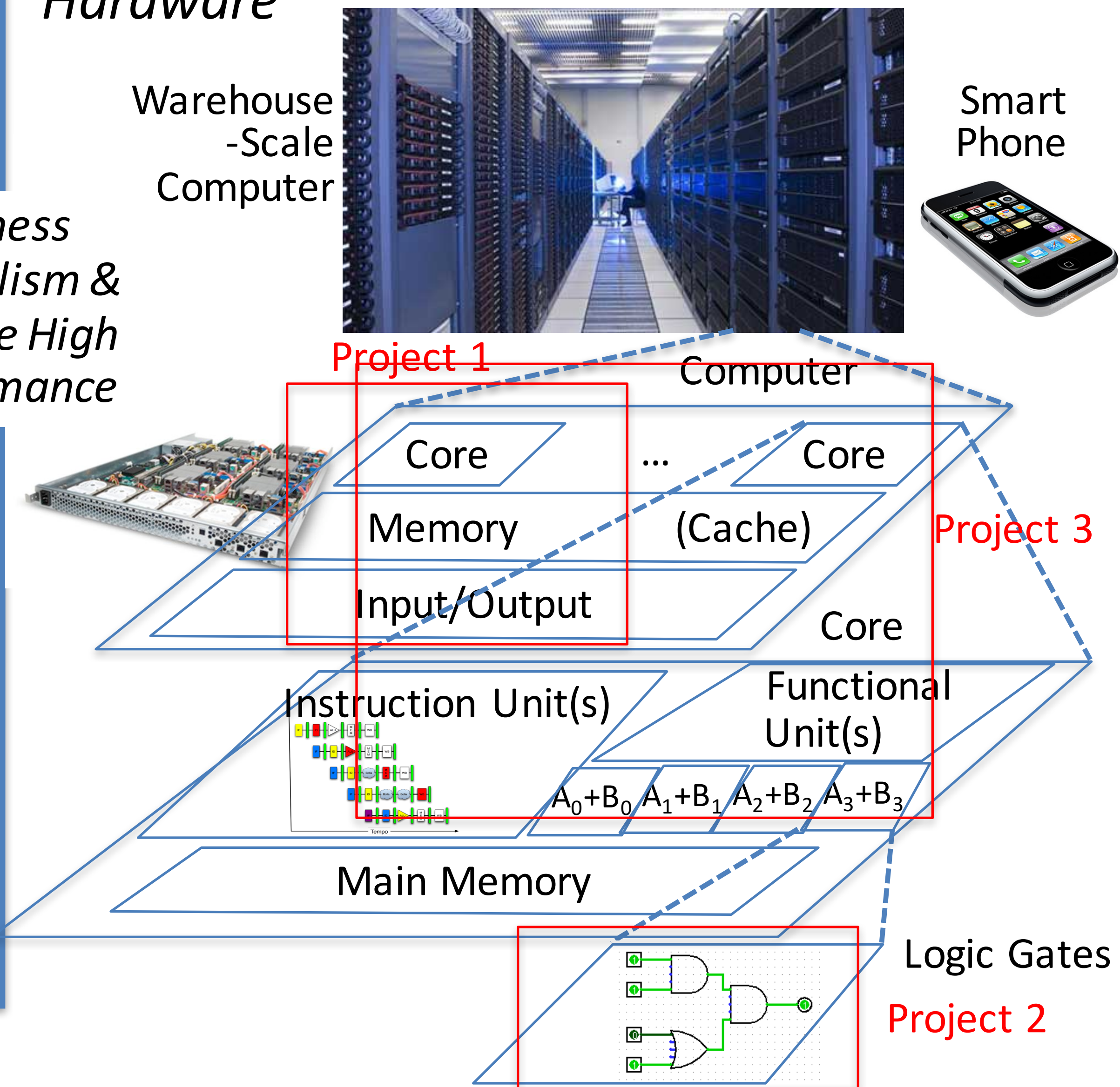


New-School Machine Structures (It's a bit more complicated!)

Software | *Hardware*

- Parallel Requests
  Assigned to computer
  e.g., Search "cats"
- Parallel Threads
  Assigned to core
  e.g., Lookup, Ads
- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions
- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words
- Hardware descriptions
  All gates functioning in
  parallel at same time

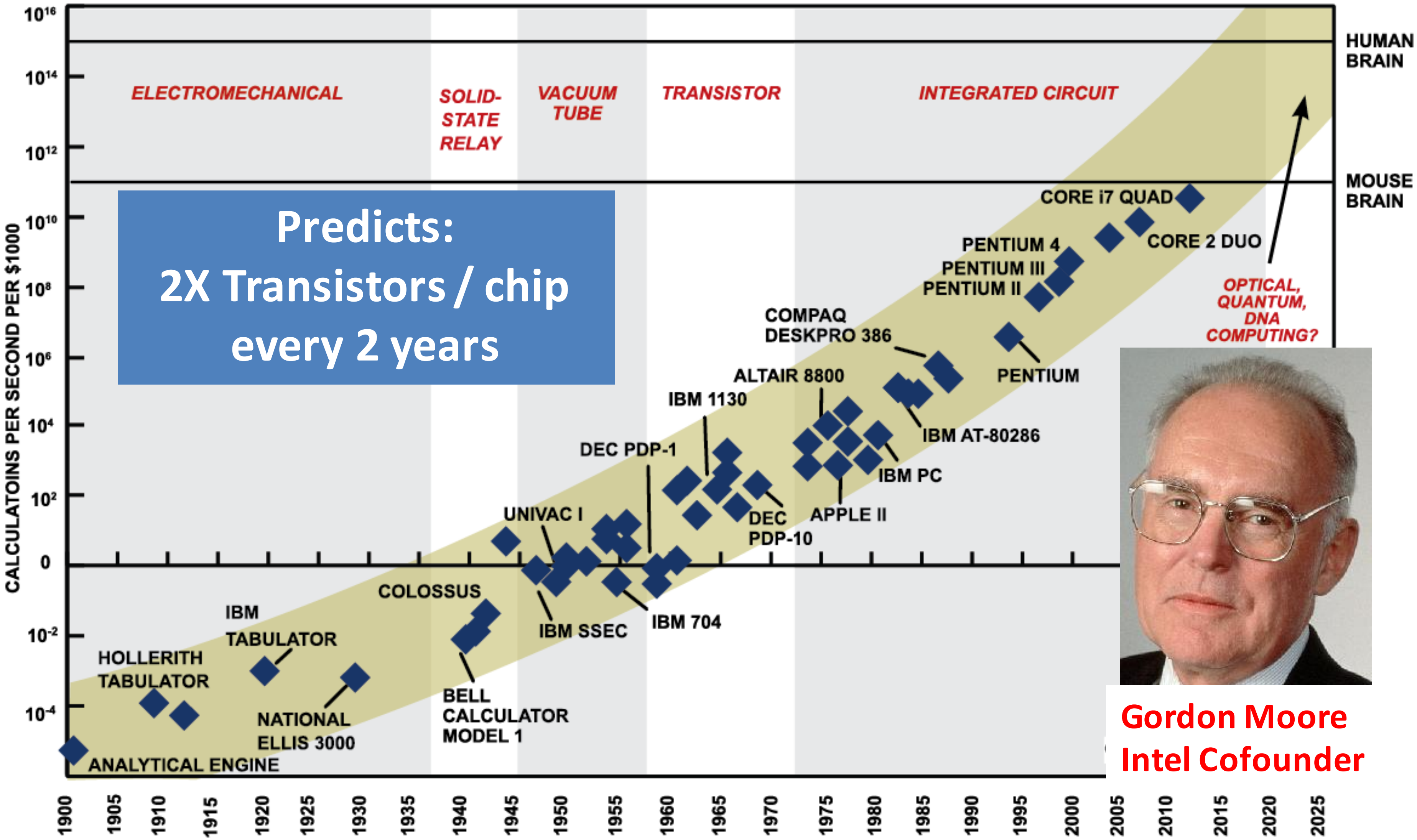
Harness Parallelism & Achieve High Performance



6 Great Ideas in Computer Architecture

1. Abstraction
   (Layers of Representation/Interpretation)
2. Moore's Law (Designing through trends)
3. Principle of Locality (Memory Hierarchy)
4. Parallelism
5. Performance Measurement & Improvement
6. Dependability via Redundancy

#2: Moore's Law

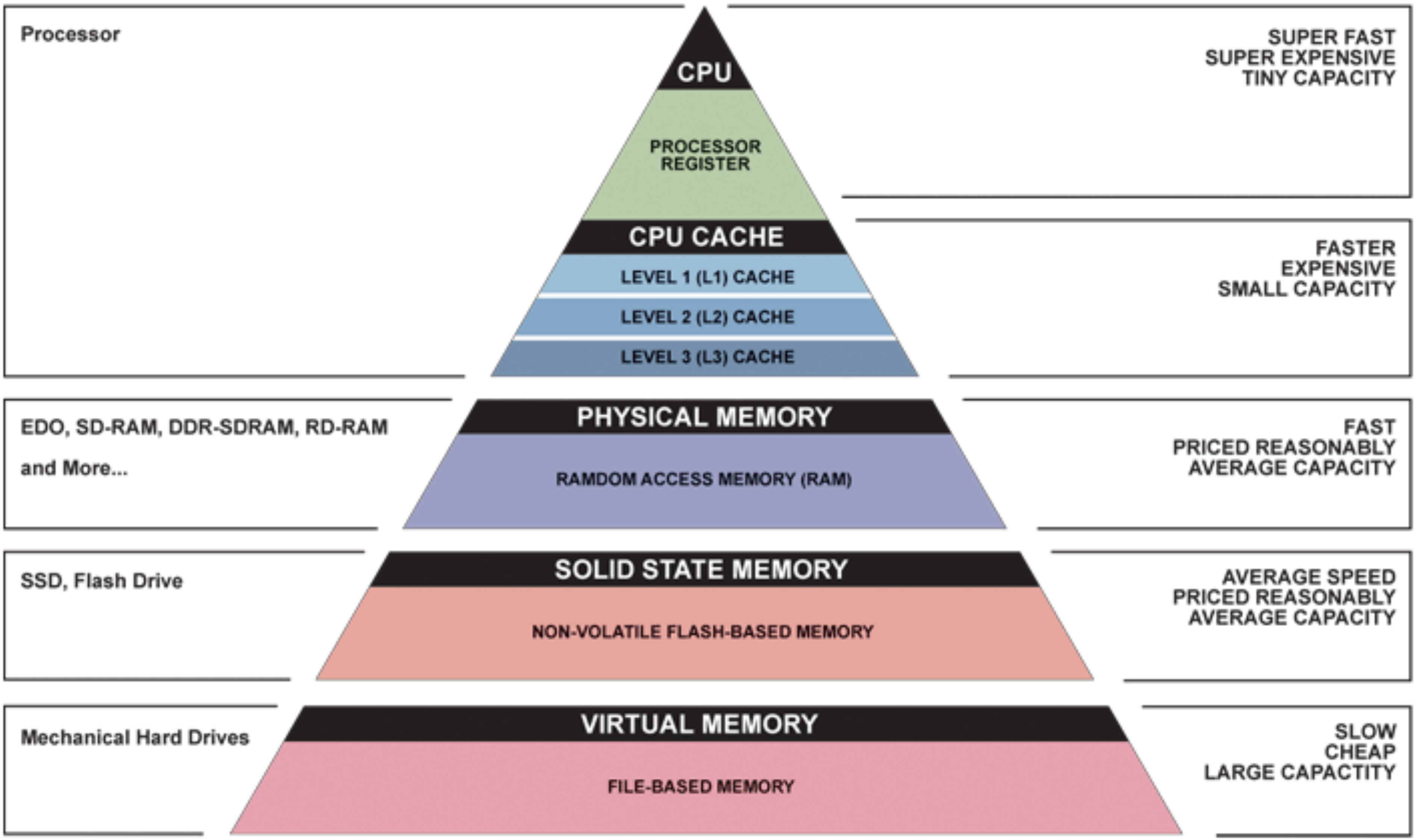


Predicts:
2X Transistors / chip every 2 years

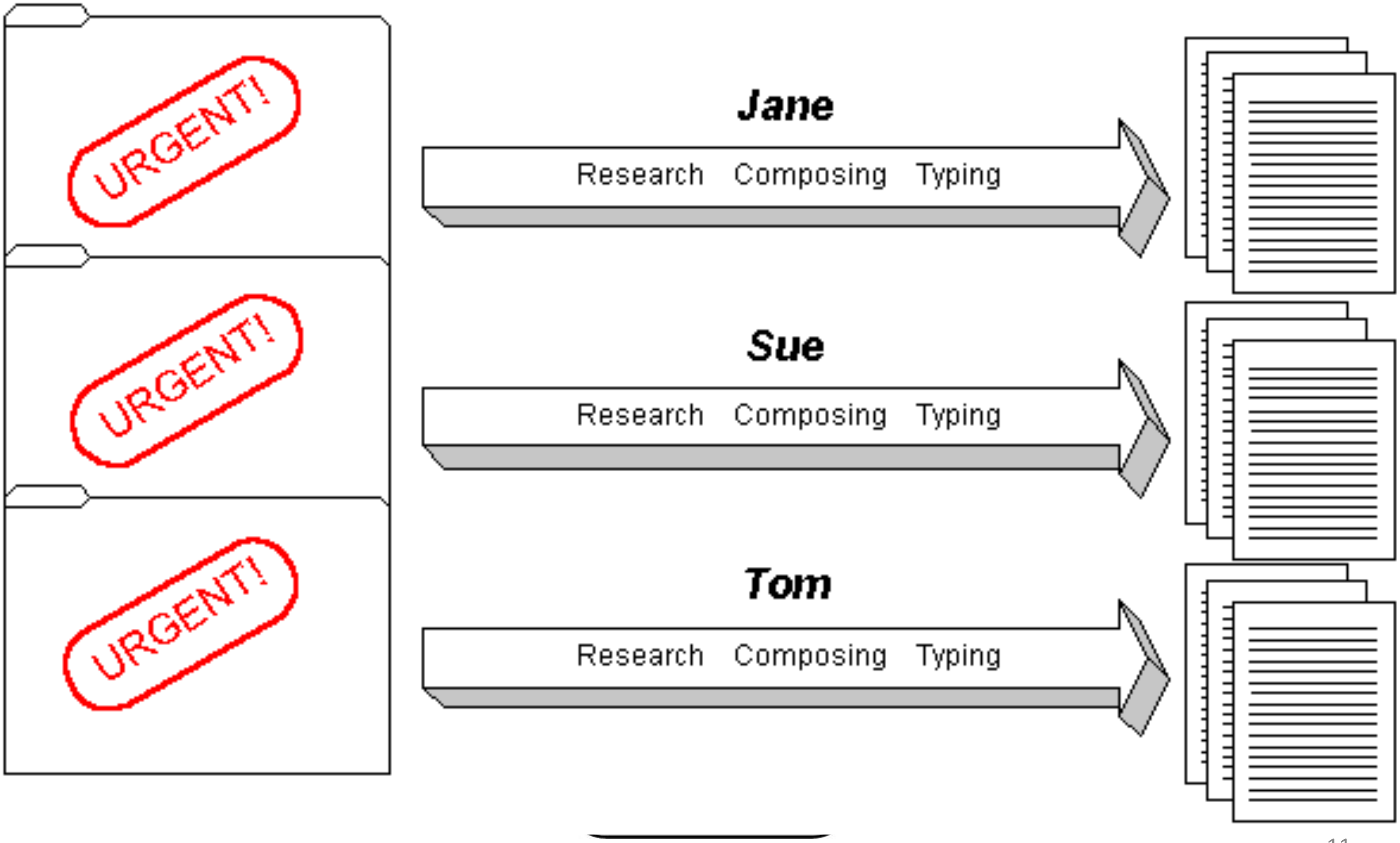
Gordon Moore
Intel Cofounder

SOURCE: RAY KURZWEIL, "THE SINGULARITY IS NEAR: WHEN HUMANS TRANSCEND BIOLOGY", P.67, *THE VIKING PRESS*, 2006. DATAPOINTS BETWEEN 2000 AND 2012 REPRESENT BCA ESTIMATES.

Great Idea #3: Principle of Locality/ Memory Hierarchy



Great Idea #4: Parallelism

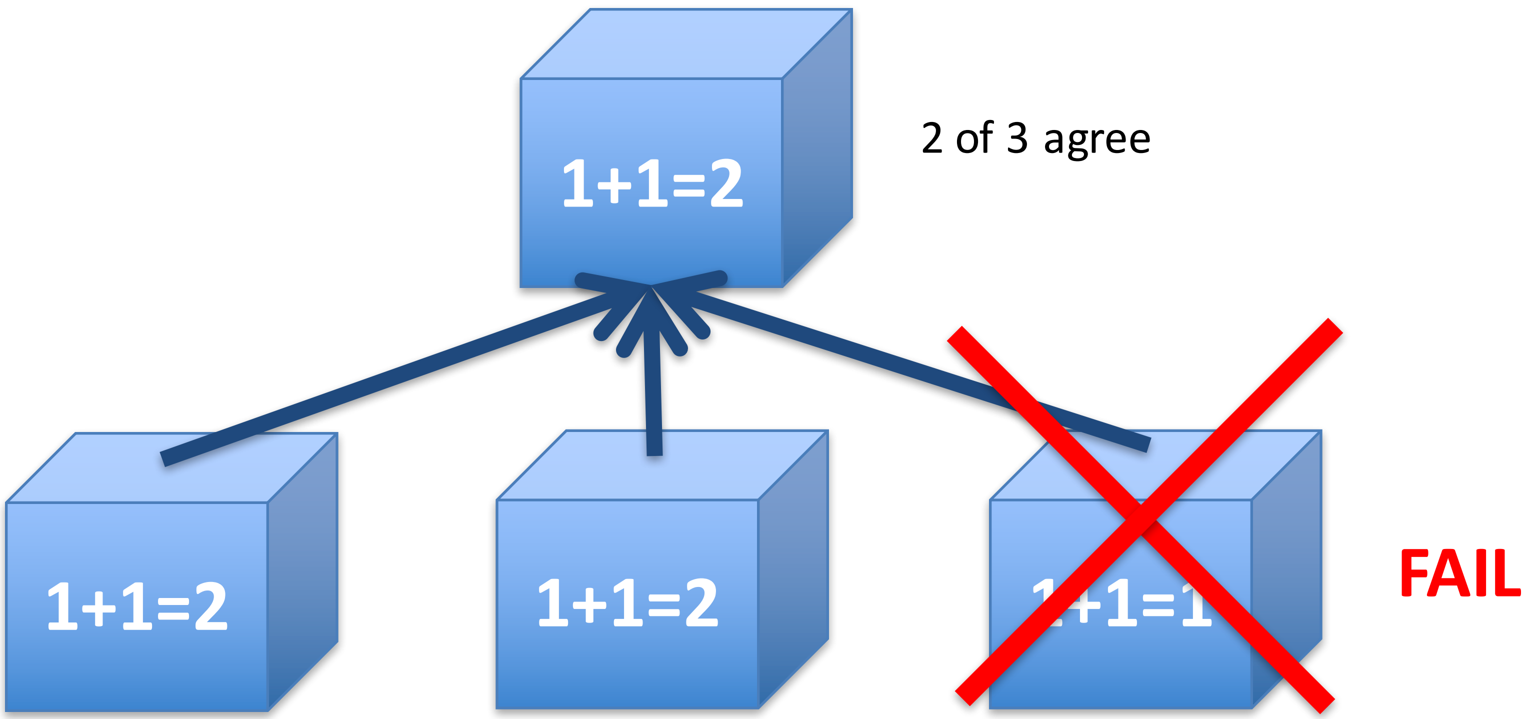


Great Idea #5: Performance Measurement and Improvement

- Tuning application to underlying hardware to exploit:
  - Locality
  - Parallelism
  - Special hardware features, like specialized instructions (e.g., matrix manipulation)
- Latency
  - How long to set the problem up
  - How much faster does it execute once it gets going
  - It is all about *time to finish*

Great Idea #6: Dependability via Redundancy

- Redundancy so that a failing piece doesn’t make the whole system fail



Increasing transistor density reduces the cost of redundancy

Key Concepts

- Inside computers, everything is a number
- But numbers usually stored with a fixed size
  - 8-bit bytes, 16-bit half words, 32-bit words, 64-bit double words, …
- Integer and floating-point operations can lead to results too big/small to store within their representations: *overflow*/*underflow*

Number Representation

Number Representation

- Value of i-th digit is $d \times Base^i$ where i starts at 0 and increases from right to left:
- $123_{10} = 1_{10} \text{ x } 10_{10}{}^2 + 2_{10} \text{ x } 10_{10}{}^1 + 3_{10} \text{ x } 10_{10}{}^0$

  $= 1\text{x}100_{10} + 2\text{x}10_{10} + 3\text{x}1_{10}$

  $= 100_{10} + 20_{10} + 3_{10}$

  $= 123_{10}$
- Binary (Base 2), Hexadecimal (Base 16), Decimal (Base 10) different ways to represent an integer
  - We use 1_{two}, 5_{ten}, 10_{hex} to be clearer (vs. 1_2, 4_8, 5_{10}, 10_{16})

Number Representation

- Hexadecimal digits: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
- $FFF_{hex} = 15_{ten} x\ 16_{ten}{}^{2} + 15_{ten} x\ 16_{ten}{}^{1} + 15_{ten} x\ 16_{ten}{}^{0}$
  $= 3840_{ten} + 240_{ten} + 15_{ten}$
  $= 4095_{ten}$
- $1111\ 1111\ 1111_{two} = FFF_{hex} = 4095_{ten}$
- May put blanks every group of binary, octal, or hexadecimal digits to make it easier to parse, like commas in decimal

Signed Integers and Two's-Complement Representation

- Signed integers in C; want ½ numbers <0, want ½ numbers >0, and want one 0
- *Two's complement* treats 0 as positive, so 32-bit word represents 2^{32} integers from -2^{31} (–2,147,483,648) to $2^{31}-1$ (2,147,483,647)
  - Note: one negative number with no positive version
  - Book lists some other options, all of which are worse
  - Every computer uses two's complement today
- *Most-significant bit* (leftmost) is the *sign bit*, since 0 means positive (including 0), 1 means negative
  - Bit 31 is most significant, bit 0 is least significant

Two's-Complement Integers

Sign Bit

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$
... ...
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 2{,}147{,}483{,}645_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -2{,}147{,}483{,}646_{ten}$
... ...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1_{ten}$

Ways to Make Two's Complement

- For N-bit word, complement to 2_{ten}^{N}
  - For 4 bit number $3_{ten}=0011_{two}$, two's complement (i.e. -3_{ten}) would be $16_{ten}-3_{ten}=13_{ten}$ or $10000_{two} - 0011_{two} = 1101_{two}$
- Here is an easier way:
  - Invert all bits and add 1
  - Computers actually do it like this, too

3_{ten}		0011_{two}
	Bitwise complement	1100_{two}
	+	1_{two}
-3_{ten}		1101_{two}

Two's-Complement Examples

- Assume for simplicity 4 bit width, -8 to +7 represented

3		0011
+2		0010
5		0101

3		0011
+ (-2)		1110
1	1	0001

-3		1101
+ (-2)		1110
-5	1	1011

Overflow when magnitude of result too big small to fit into result representation

7		0111
+1		0001
-8		1000

Overflow!

-8		1000
+ (-1)		1111
+7	1	0111

Overflow!

Carry into MSB = Carry Out MSB

Carry into MSB ≠ Carry Out MSB

Carry in = carry from less significant bits

Carry out = carry to more significant bits

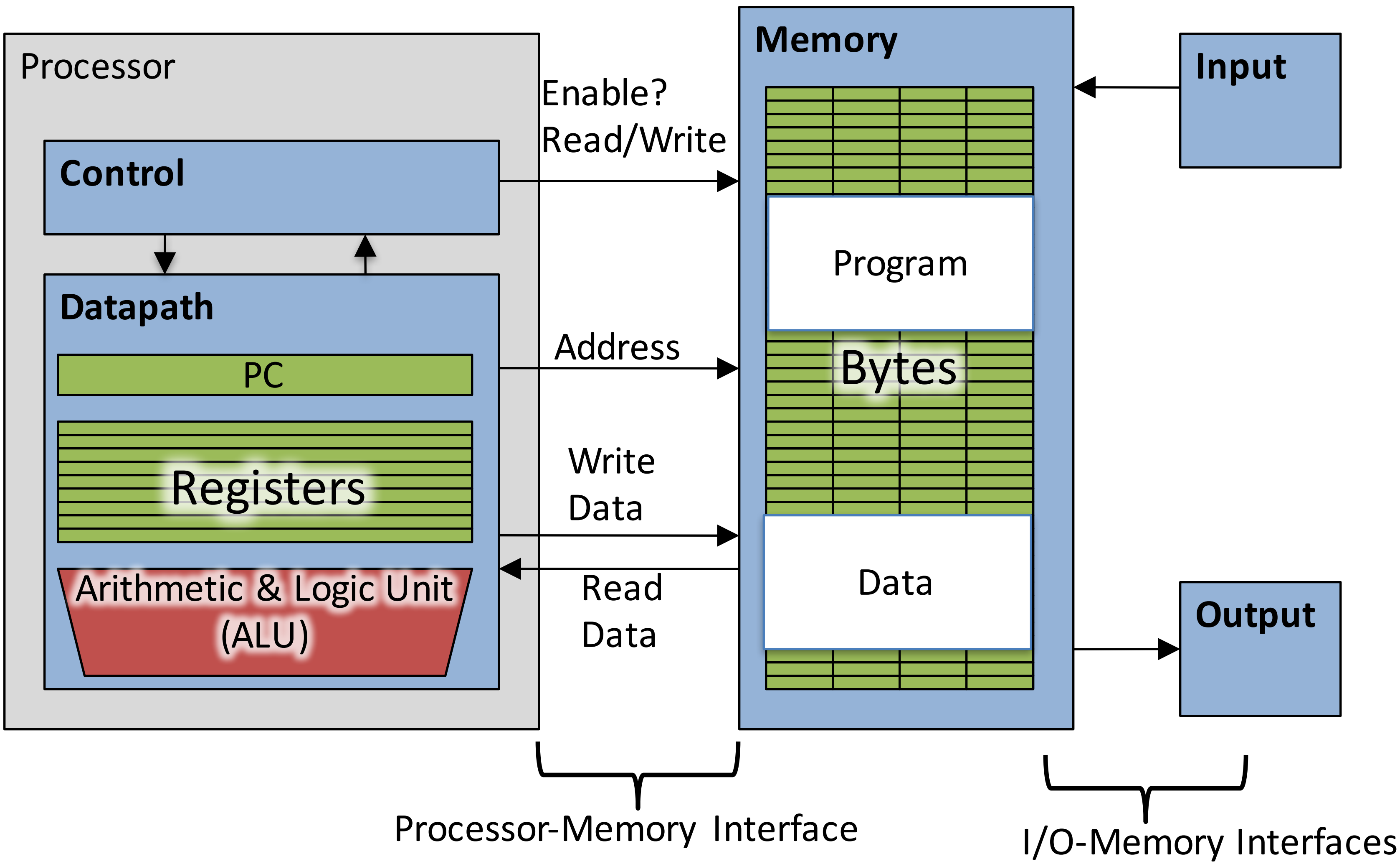
Suppose we had a 5-bit word. What integers can be represented in two's complement?

- ☐ -32 to +31
- ☐ 0 to +31
- ☐ -16 to +15
- ☐ -15 to +16

Suppose we had a 5-bit word. What integers can be represented in two's complement?

- ☐ -32 to +31
- ☐ 0 to +31
- ☐ -16 to +15
- ☐ -15 to +16

Components of a Computer



C Programming

Quiz: Pointers

```
void foo(int *x, int *y)
{  int t;
   if ( *x > *y ) { t = *y; *y = *x; *x = t; }
}
int a=3, b=2, c=1;
foo(&a, &b);
foo(&b, &c);
foo(&a, &b);
printf("a=%d b=%d c=%d\n", a, b, c);
```

Result is:

A: **a=3 b=2 c=1**

B: **a=1 b=2 c=3**

C: **a=1 b=3 c=2**

D: **a=3 b=3 c=3**

E: **a=1 b=1 c=1**

Arrays and Pointers

```
int
foo(int array[],
    unsigned int size)
{
   …
   printf("%d\n", sizeof(array));
}

int
main(void)
{
   int a[10], b[5];
   … foo(a, 10)… foo(b, 5)  …
   printf("%d\n", sizeof(a));
}
```

What does this print? **8**

… because **`array`** is really a pointer (and a pointer is architecture dependent, but likely to be 8 on modern machines!)

What does this print? **40**

Quiz:

```
int x[] = { 2, 4, 6, 8, 10 };
int *p = x;
int **pp = &p;
(*pp)++;
(*(*pp))++;
printf("%d\n", *p);
```

Result is:

A: 2

B: 3

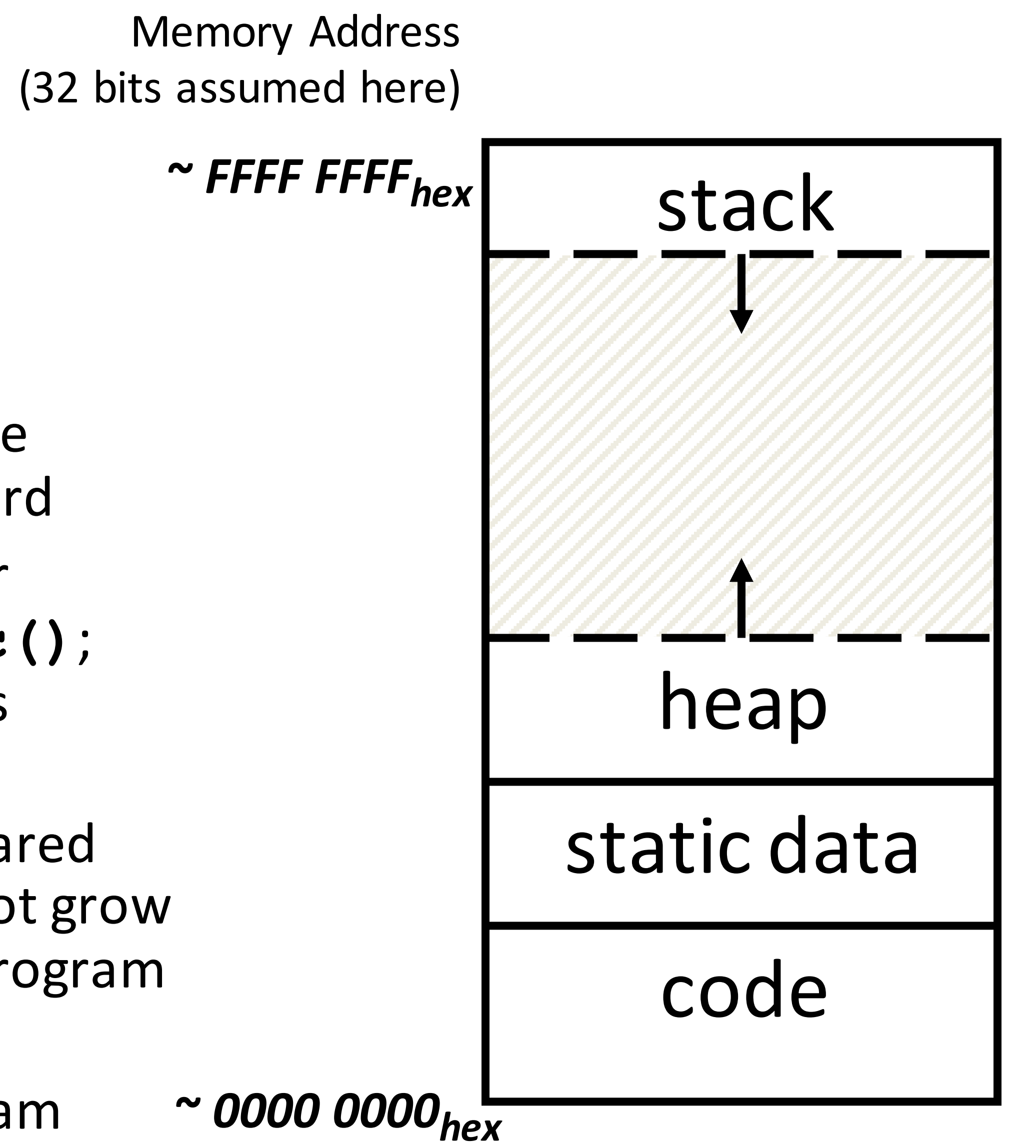
C: 4

D: 5

E: None of the above

C Memory Management

- Program's *address space* contains 4 regions:
  - stack: local variables inside functions, grows downward
  - heap: space requested for dynamic data via **`malloc()`**; resizes dynamically, grows upward
  - static data: variables declared outside functions, does not grow or shrink. Loaded when program starts, can be modified.
  - code: loaded when program starts, does not change

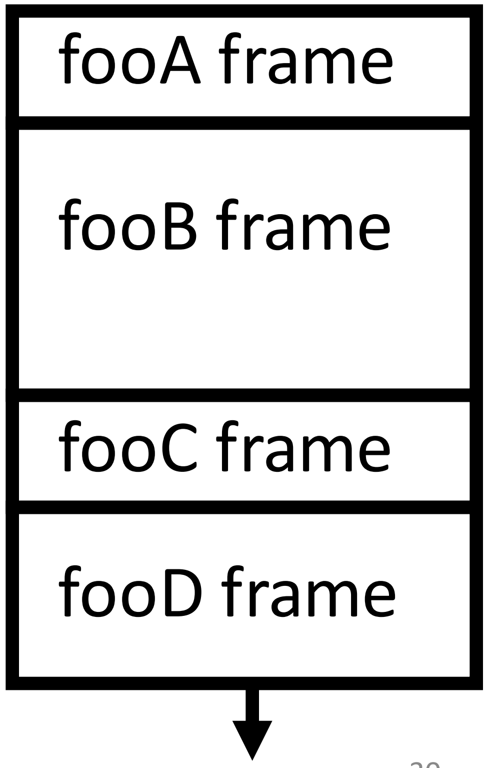


The Stack

- Every time a function is called, a new frame is allocated on the stack
- Stack frame includes:
  - Return address (who called me?)
  - Arguments
  - Space for local variables
- Stack frames contiguous blocks of memory; stack pointer indicates start of stack frame
- When function ends, stack frame is tossed off the stack; frees memory for future stack frames
- We’ll cover details later for MIPS processor

```
fooA() { fooB(); }
fooB() { fooC(); }
fooC() { fooD(); }
```

Stack Pointer →



Question!

```
int x = 2;
int result;

int foo(int n)
{    int y;
     if (n <= 0) { printf("End case!\n"); return 0; }
     else
     {  y = n + foo(n-x);
        return y;
     }
}
result = foo(10);
```

Right after the **`printf`** executes but before the **`return 0`**, how many copies of **`x`** and **`y`** are there allocated in memory?

A: #x = 1, #y = 1

B: #x = 1, #y = 5

C: #x = 5, #y = 1

D: #x = 1, #y = 6

E: #x = 6, #y = 6

Faulty Heap Management

- What is wrong with this code?
- Memory leak!

```
int foo() {
   int *value = malloc(sizeof(int));
   *value = 42;
   return *value;
}
```

Using Memory You Don't Own

- What is wrong with this code?

```
int* init_array(int *ptr, int new_size) {
   ptr = realloc(ptr, new_size*sizeof(int));
   memset(ptr, 0, new_size*sizeof(int));
   return ptr;
}

int* fill_fibonacci(int *fib, int size) {
   int i;
   init_array(fib, size);
   /* fib[0] = 0; */ fib[1] = 1;
   for (i=2; i<size; i++)
    fib[i] = fib[i-1] + fib[i-2];
   return fib;
}
```

Using Memory You Don't Own

- Improper matched usage of mem handles

```
int* init_array(int *ptr, int new_size) {
   ptr = realloc(ptr, new_size*sizeof(int));
   memset(ptr, 0, new_size*sizeof(int));
   return ptr;
}
```

Remember: `realloc` may move entire block

```
int* fill_fibonacci(int *fib, int size) {
   int i;
   /* oops, forgot: fib = */ init_array(fib, size);
   /* fib[0] = 0; */ fib[1] = 1;
   for (i=2; i<size; i++)
    fib[i] = fib[i-1] + fib[i-2];
   return fib;
}
```

What if array is moved to new location?

And In Conclusion, …

- Pointers are an abstraction of machine memory addresses
- Pointer variables are held in memory, and pointer values are just numbers that can be manipulated by software
- In C, close relationship between array names and pointers
- Pointers know the type of the object they point to (except void *)
- Pointers are powerful but potentially dangerous

And In Conclusion, …

- C has three main memory segments in which to allocate data:
  - Static Data: Variables outside functions
  - Stack: Variables local to function
  - Heap: Objects explicitly malloc-ed/free-d.
- Heap data is biggest source of bugs in C code

MIPS

Addition and Subtraction of Integers Example 1

- How to do the following C statement?

  a = b + c + d - e;   a = ((b + c) + d) - e;

  b → $s1; c → $s2; d → $s3; e → $s4; a → $s0

- Break into multiple instructions

```
add $t0, $s1, $s2 # temp = b + c
add $t0, $t0, $s3 # temp = temp + d
sub $s0, $t0, $s4 # a = temp - e
```

- A single line of C may break up into several lines of MIPS.
- Notice the use of temporary registers – don't want to modify the variable registers `$s`
- Everything after the hash mark on each line is ignored (comments)

Overflow handling in MIPS

- Some languages detect overflow (Ada), some don't (most C implementations)
- MIPS solution is 2 kinds of arithmetic instructions:
  - These cause overflow to be detected
    - add (add)
    - add immediate (addi)
    - subtract (sub)
  - These do not cause overflow detection
    - add unsigned (addu)
    - add immediate unsigned (addiu)
    - subtract unsigned (subu)
- Compiler selects appropriate arithmetic
  - MIPS C compilers produce addu, addiu, subu

Question:

We want to translate `*x = *y +1` into MIPS
(x, y int pointers stored in: `$s0 $s1`)

A:
```
addi $s0,$s1,1
```

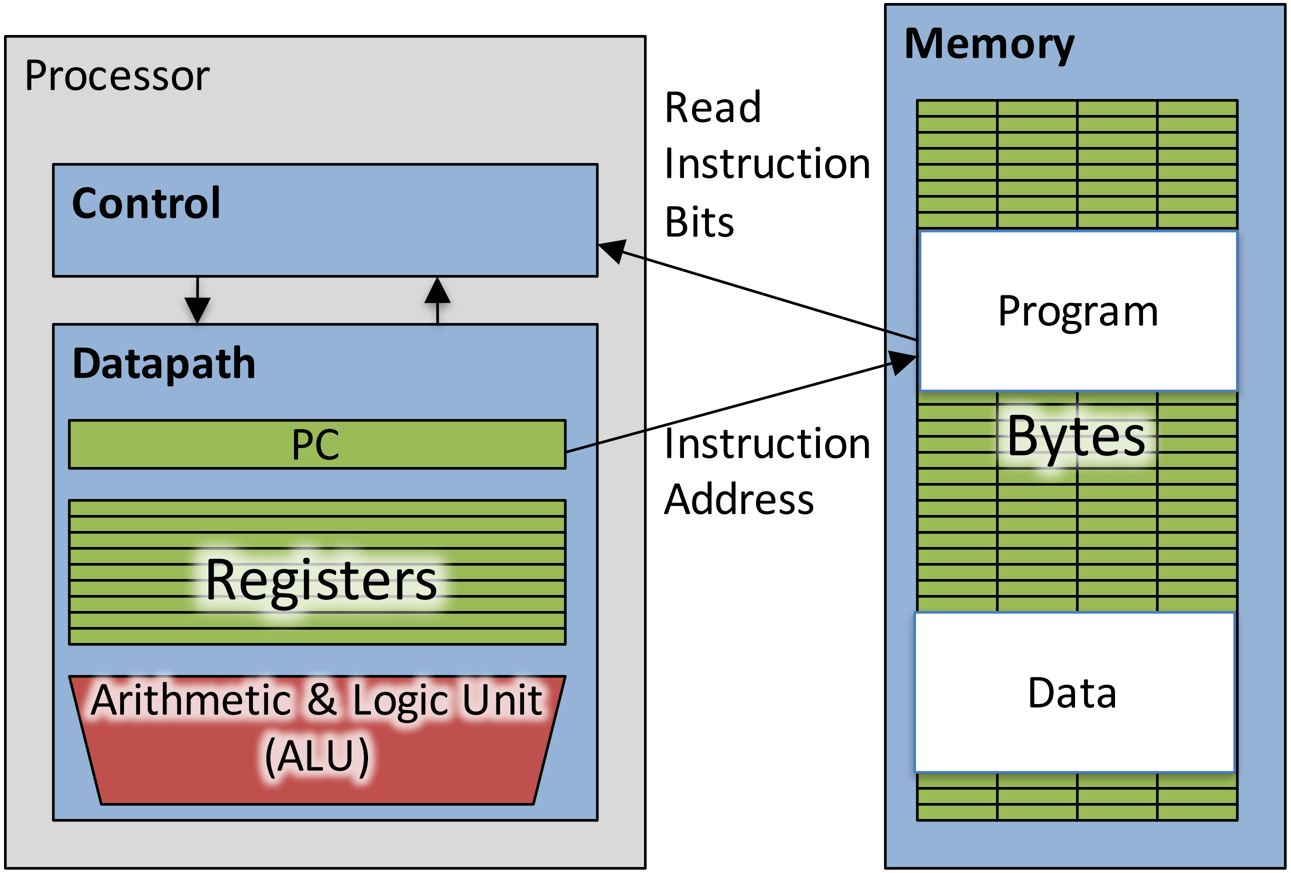
B:
```
lw   $s0,1($s1)
sw   $s1,0($s0)
```

C:
```
lw   $t0,0($s1)
addi $t0,$t0,1
sw   $t0,0($s0)
```

D:
```
sw   $t0,0($s1)
addi $t0,$t0,1
lw   $t0,0($s0)
```

E:
```
lw   $s0,1($t0)
sw   $s1,0($t0)
```

Executing a Program



- The PC (program counter) is internal register inside processor holding byte address of next instruction to be executed.
- Instruction is fetched from memory, then control unit executes instruction using datapath and memory system, and updates program counter (default is add +4 bytes to PC, to move to next sequential instruction)

Question!

```
        addi $s0,$zero,0
Start:  slt  $t0,$s0,$s1
        beq  $t0,$zero,Exit
        sll  $t1,$s0,2
        addu $t1,$t1,$s5
        lw   $t1,0($t1)
        add  $s4,$s4,$t1
        addi $s0,$s0,1
        j Start
Exit:
```

What is the code above?

A: while loop

B: do … while loop

C: for loop

D: A or C

E: Not a loop

MIPS Function Call Conventions

- Registers faster than memory, so use them
- `$a0–$a3`: four *argument* registers to pass parameters ($4 - $7)
- `$v0,$v1`: two *value* registers to return values ($2,$3)
- `$ra`: one *return address* register to return to the point of origin ($31)

Instruction Support for Functions (1/4)

C

```
... sum(a,b);... /* a,b:$s0,$s1 */
 }
 int sum(int x, int y) {
   return x+y;
 }
```

MIPS

```
address (shown in decimal)
 1000
 1004
 1008
 1012
 1016
 …
 2000
 2004
```

In MIPS, all instructions are 4 bytes, and stored in memory just like data. So here we show the addresses of where the programs are stored.

Instruction Support for Functions (2/4)

C

```
... sum(a,b);... /* a,b:$s0,$s1 */
 }
 int sum(int x, int y) {
   return x+y;
 }
```

MIPS

```
address (shown in decimal)
 1000 add  $a0,$s0,$zero    # x = a
 1004 add  $a1,$s1,$zero    # y = b
 1008 addi $ra,$zero,1016   # $ra=1016
 1012 j    sum              # jump to sum
 1016 …                     # next instruction
 …
 2000 sum: add $v0,$a0,$a1
 2004 jr   $ra    # new instr. "jump register"
```

Instruction Support for Functions (3/4)

C

```
... sum(a,b);... /* a,b:$s0,$s1 */
 }
 int sum(int x, int y) {
   return x+y;
 }
```

MIPS

- Question: Why use `jr` here? Why not use `j`?
- Answer: `sum` might be called by many places, so we can't return to a fixed place. The calling proc to `sum` must be able to say "return here" somehow.

```
2000 sum: add $v0,$a0,$a1
2004 jr   $ra # new instr. "jump register"
```

Instruction Support for Functions (4/4)

- Single instruction to jump and save return address: jump and link (`jal`)
- Before:

```
1008 addi $ra,$zero,1016  # $ra=1016
1012 j sum                # goto sum
```

- After:

```
1008 jal sum  # $ra=1012,goto sum
```

- Why have a `jal`?
  - Make the common case fast: function calls very common.
  - Don't have to know where code is in memory with `jal`!

Question

- Which statement is FALSE?

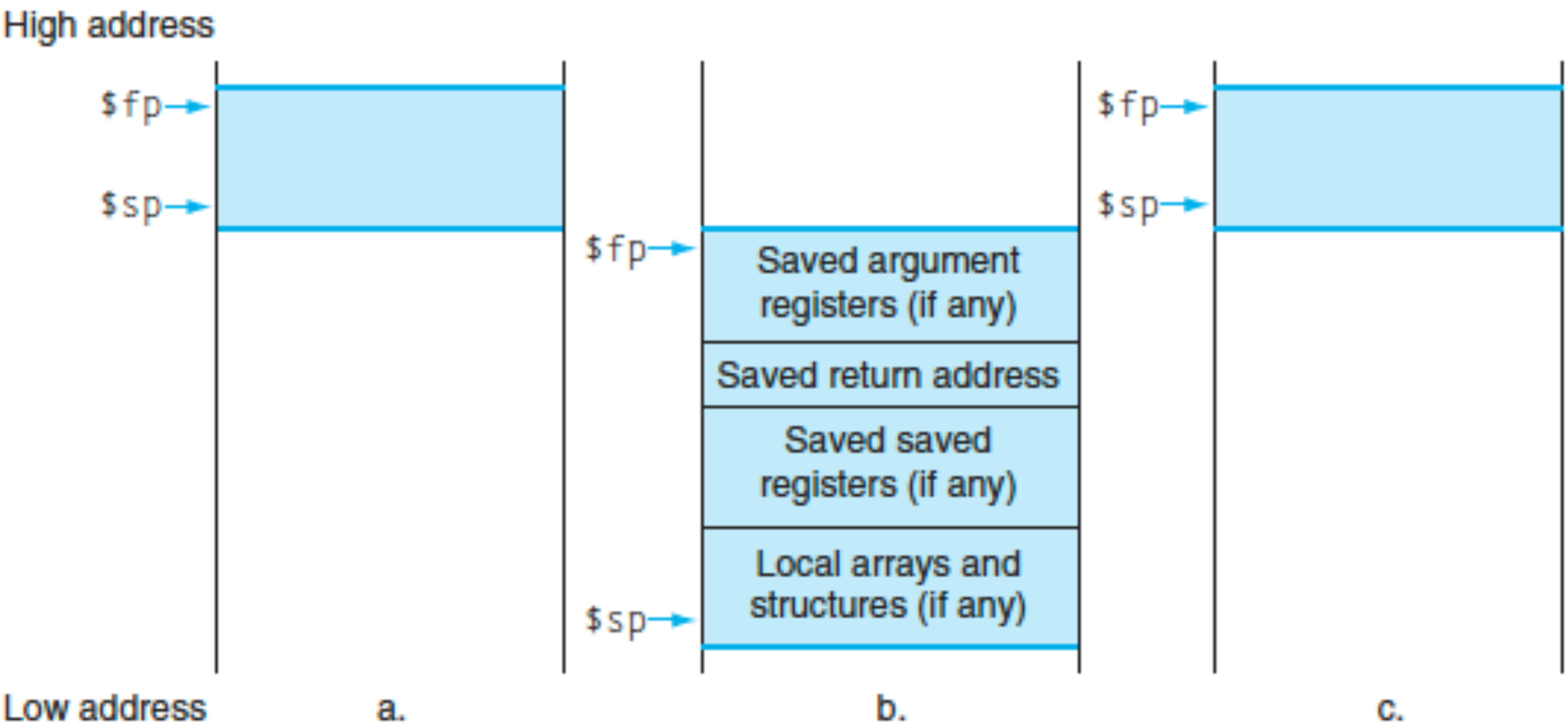
  A: MIPS uses `jal` to invoke a function and jr to return from a function

  B: `jal` saves PC+1 in `$ra`

  C: The callee can use temporary registers (`$ti`) without saving and restoring them

  D: The caller can rely on save registers (`$si`) without fear of callee changing them

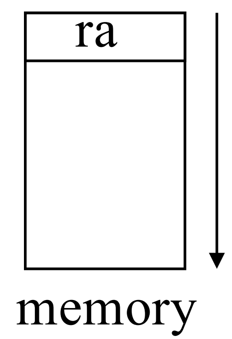
Stack Before, During, After Call



Basic Structure of a Function

Prologue

```
entry_label:
addi $sp,$sp, -framesize
sw $ra, framesize-4($sp)   # save $ra
save other regs if need be
```



Body · · · **(call other functions…)**

Epilogue

```
restore other regs if need be
lw $ra, framesize-4($sp)   # restore $ra
addi $sp,$sp, framesize
jr $ra
```

Instruction Formats

- I-format: used for instructions with immediates, **`lw`** and **`sw`** (since offset counts as an immediate), and branches (**`beq`** and **`bne`**)
  - (but not the shift instructions; later)
- J-format: used for **`j`** and **`jal`**
- R-format: used for all other instructions
- It will soon become clear why the instructions have been partitioned in this way

R-Format Instructions (1/5)

- Define "fields" of the following number of bits each: 6 + 5 + 5 + 5 + 5 + 6 = 32

6	5	5	5	5	6

- For simplicity, each field has a name:

opcode	rs	rt	rd	shamt	funct

- Important: On these slides and in book, each field is viewed as a 5- or 6-bit unsigned integer, not as part of a 32-bit integer
  - Consequence: 5-bit fields can represent any number 0-31, while 6-bit fields can represent any number 0-63

I-Format Instructions (2/4)

- Define "fields" of the following number of bits each: 6 + 5 + 5 + 16 = 32 bits

6	5	5	16

  - Again, each field has a name:

opcode	**rs**	**rt**	**immediate**

  - Key Concept: Only one field is inconsistent with R-format. Most importantly, **opcode** is still in same location.

I-Format Example (2/2)

- MIPS Instruction:

```
addi   $21,$22,-50
```

Decimal/field representation:

8	22	21	-50

Binary/field representation:

001000	10110	10101	1111111111001110

hexadecimal representation: **22D5 FFCE**$_{hex}$

Branch Example (1/2)

- MIPS Code:

```
Loop: beq   $9,$0,End
      addu  $8,$8,$10
      addiu $9,$9,-1
      j     Loop
End:
```

Start counting from instruction AFTER the branch (1, 2, 3)

- I-Format fields:

`opcode` = 4 (look up on Green Sheet)

`rs` = 9 (first operand)

`rt` = 0 (second operand)

`immediate` = 3

Branch Example (2/2)

- MIPS Code:

```
Loop: beq   $9,$0,End
      addu  $8,$8,$10
      addiu $9,$9,-1
      j     Loop
End:
```

Field representation (decimal):

31 … 0

4	9	0	3

Field representation (binary):

31 … 0

000100	01001	00000	0000000000000011

J-Format Instructions (2/4)

- Define two "fields" of these bit widths:

31	0
6	26

- As usual, each field has a name:

31	0
`opcode`	`target address`

- **Key Concepts:**
  - Keep `opcode` field identical to R-Format and I-Format for consistency
  - Collapse all other fields to make room for large target address

Summary

- I-Format: instructions with immediates, `lw`/`sw` (offset is immediate), and `beq`/`bne`
  - But not the shift instructions
  - Branches use PC-relative addressing

I:

opcode	rs	rt	immediate

- J-Format: `j` and `jal` (but not `jr`)
  - Jumps use absolute addressing

J:

opcode	target address

- R-Format: all other instructions

R:

opcode	rs	rt	rd	shamt	funct

Assembler Pseudo-Instructions

- Certain C statements are implemented unintuitively in MIPS
  - e.g. assignment (`a=b`) via add $zero
- MIPS has a set of "pseudo-instructions" to make programming easier
  - More intuitive to read, but get translated into actual instructions later
- Example:

```
move dst,src
```

translated into

```
addi dst,src,0
```

Multiply and Divide

- Example pseudo-instruction:

```
mul $rd,$rs,$rt
```

  - Consists of mult which stores the output in special hi and lo registers, and a move from these registers to $rd

```
mult $rs,$rt
mflo $rd
```

- **`mult`** and **`div`** have nothing important in the **`rd`** field since the destination registers are **`hi`** and **`lo`**
- **`mfhi`** and **`mflo`** have nothing important in the **`rs`** and **`rt`** fields since the source is determined by the instruction (see COD)

Question

Which of the following place the address of LOOP in $v0?

1)
```
la $t1, LOOP
lw $v0, 0($t1)
```

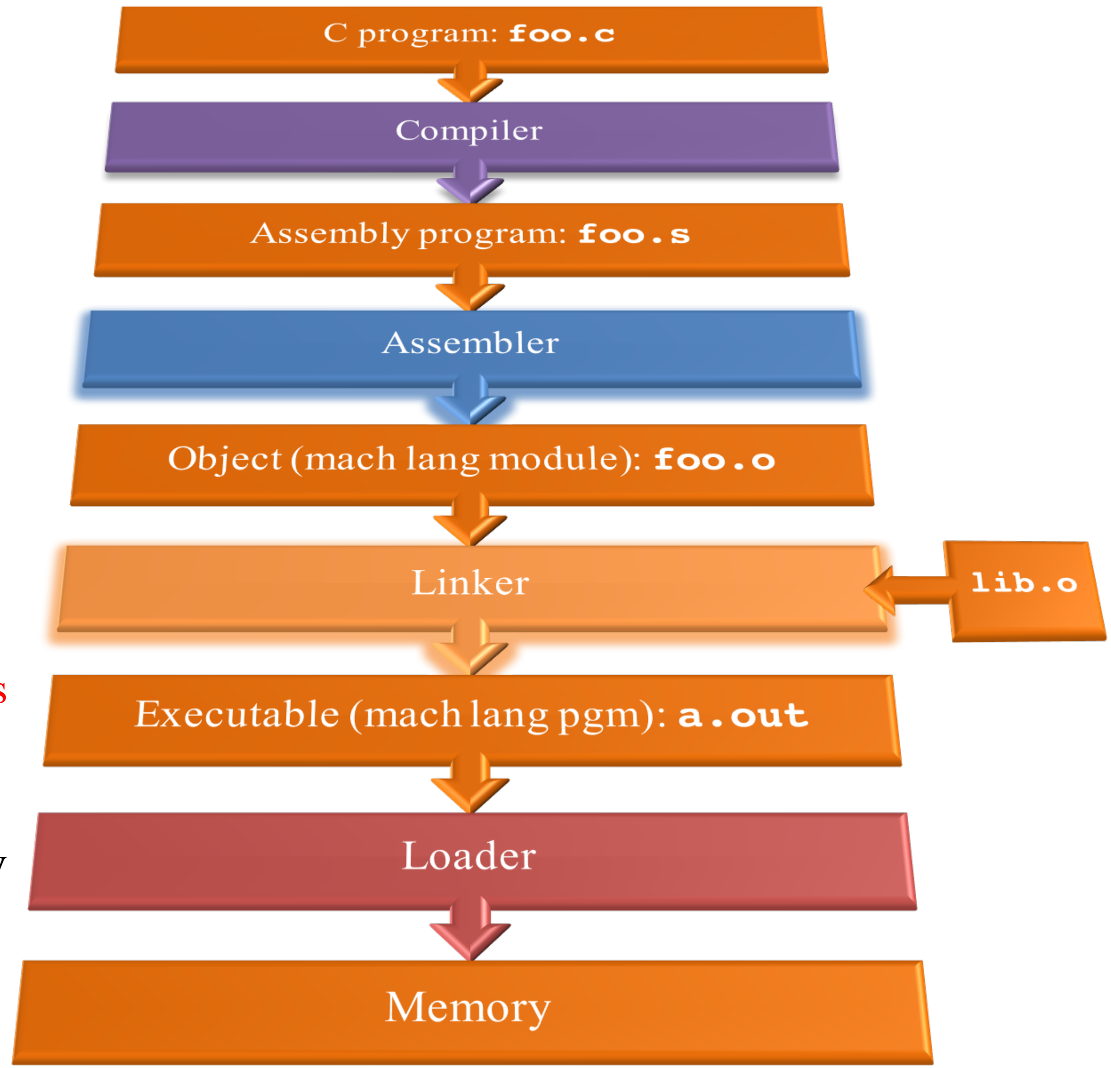
2)
```
jal LOOP
LOOP: addu $v0, $ra, $zero
```

3)
```
la $v0, LOOP
```

	1	2	3
A)	T,	T,	T
B)	T,	T,	F
C)	F,	T,	T
D)	F,	T,	F
E)	F,	F,	T

Steps in compiling a C program

- Compiler converts a single HLL file into a single assembly language file.
- Assembler removes pseudo-instructions, converts what it can to machine language, and creates a checklist for the linker (relocation table). A `.s` file becomes a `.o` file.
  - Does 2 passes to resolve addresses, handling internal forward references
- Linker combines several `.o` files and resolves absolute addresses.
  - Enables separate compilation, libraries that need not be compiled, and resolves remaining addresses
- Loader loads executable into memory and begins execution.



Pseudo-instruction Replacement

- Assembler treats convenient variations of machine language instructions as if real instructions

Pseudo:	Real:
`subu $sp,$sp,32`	`addiu $sp,$sp,-32`
`sd $a0, 32($sp)`	`sw $a0, 32($sp)` `sw $a1, 36($sp)`
`mul $t7,$t6,$t5`	`mult $t6,$t5` `mflo $t7`
`addu $t0,$t6,1`	`addiu $t0,$t6,1`
`ble $t0,100,loop`	`slti $at,$t0,101` `bne $at,$0,loop`
`la $a0, str`	`lui $at,left(str)` `ori $a0,$at,right(str)`

Question

At what point in process are all the machine code bits generated for the following assembly instructions:

1) `addu $6, $7, $8`

2) `jal fprintf`

A: 1) & 2) After compilation

B: 1) After compilation, 2) After assembly

C: 1) After assembly, 2) After linking

D: 1) After assembly, 2) After loading

E: 1) After compilation, 2) After linking