CSCI 210: Computer Architecture
Lecture 4: Assembly Programming

Stephen Checkoway
Oberlin College
Slides from Cynthia Taylor

Announcements

- Problem Set 0 due Friday 11:59 p.m.

Why you should learn (a little) assembly

- Learn what your computer is fundamentally capable of
- By learning about how high-level mechanisms are created in assembly, we learn what is fast, what is slow
- Might use it for reverse engineering, embedded systems, compilers

CS History: Sophie Wilson

Developed the ARM Instruction Set Architecture

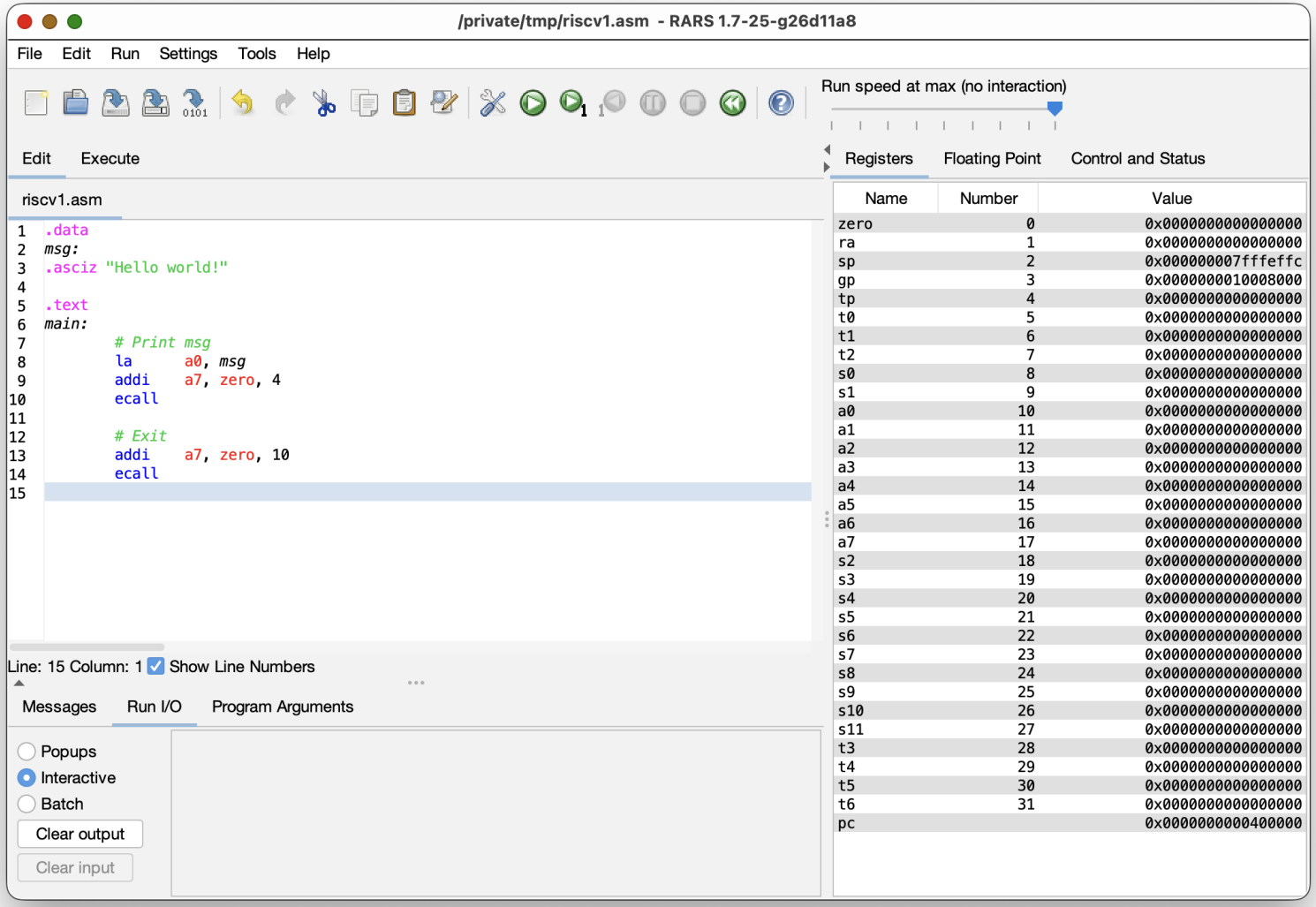
Our first RISC-V assembly language program

```
1   .data
2   msg:
3   .asciz "Hello world!"
4
5   .text
6   main:
7           # Print msg
8           la      a0, msg
9           addi    a7, zero, 4
10          ecall
11
12          # Exit
13          addi    a7, zero, 10
14          ecall
```

- Data segment (starting with `.data`) contains data definitions, in this case an ASCII-encoded string
- Code segment (starting with .text) contains assembly instructions
- Data and code can be given symbolic names using a label (see `msg` and `main`)

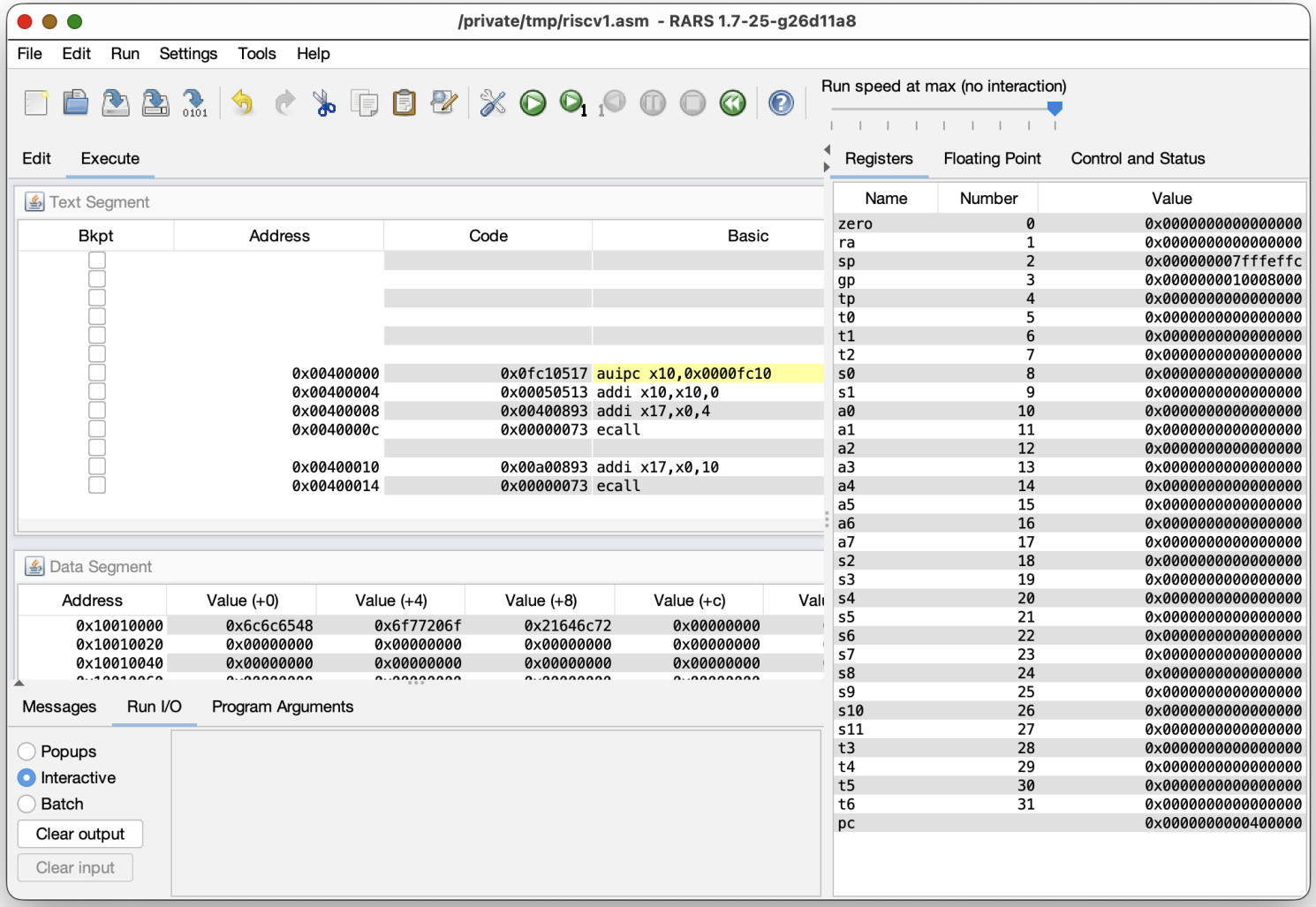
Running this code in RARS

1. Create a new file
2. Type the program
3. Click the Assemble button



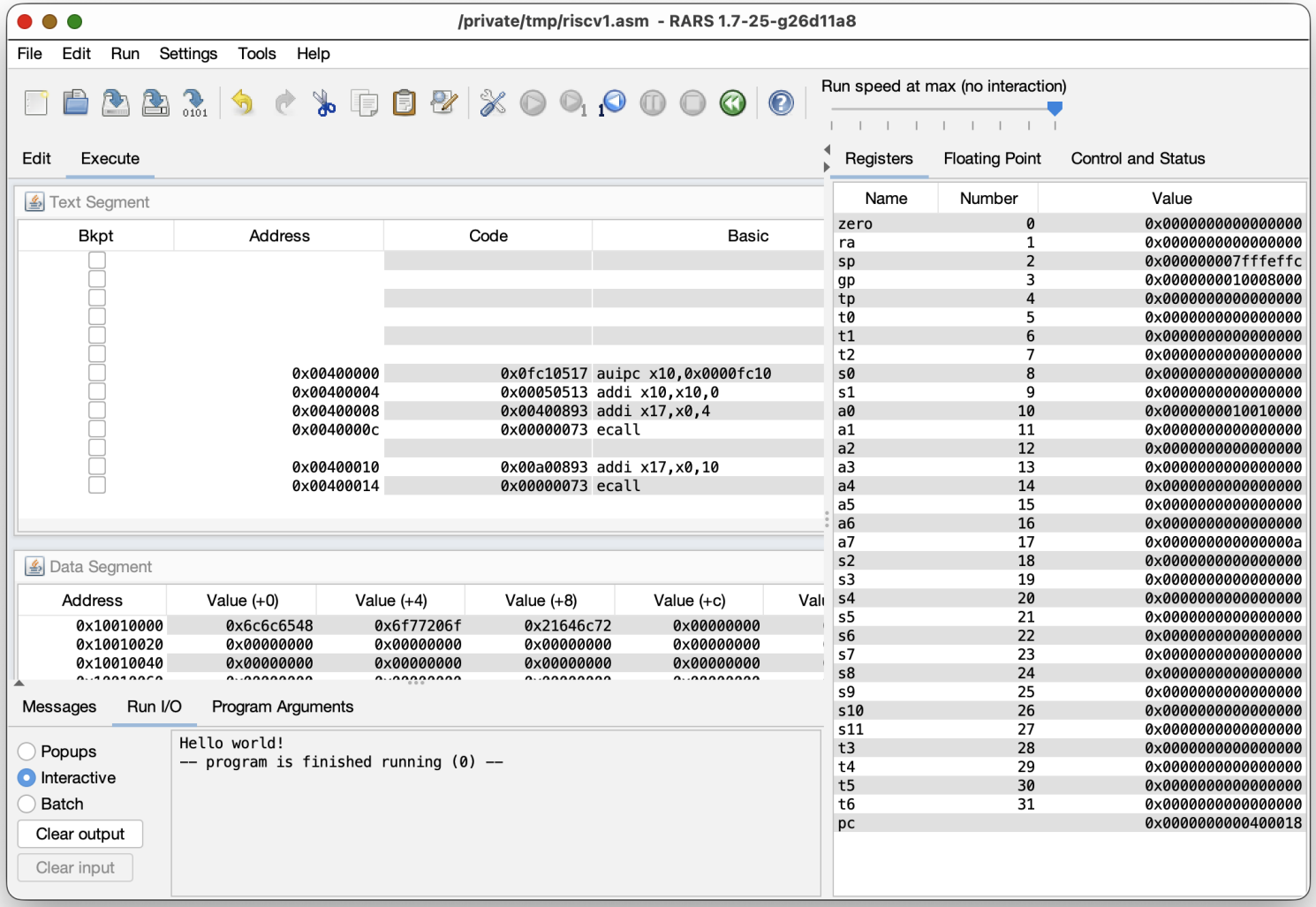
After assembly

1. View switches to “Execute” tab
2. Click the Run button



After Running

1. Output appears at the bottom



The RISC-V Instruction Set

- Used as the example throughout the book
- Newish computer architecture developed at UC Berkeley
  - It's the 5th in a series of Berkeley RISC architectures, RISC-I, RISC-II, SOAR, and SPUR are the preceding 4
- Typical of many modern ISAs
  - Most similar to MIPS and ARM

Three Basic Types of Instructions

- Arithmetic and logical
  - Operates on data entirely in registers
- Immediate
  - One of the values (operand) used by the instruction is encoded directly in the instruction
- Branch/jump
  - Changes the pc to a new location

Operands

- Assembly instructions operate on **operands**
  - You can think of each instruction as a mini-function and the operands are the arguments to the function
- Instructions are written in the form:

```
name   op1, op2, op3
```

- There are different types of operands including
  - Register operands (data is in the register)
  - Immediate operands (data is directly in the instruction)
  - Memory operands (data is in memory)

Arithmetic and Logical Operations

- Add and subtract, three operands
  - Two sources and one destination

```
add a, b, c   # a = b + c
sub a, b, c   # a = b – c
and a, b, c   # a = b & c (bit-wise AND)
```

- All arithmetic and logical operations have this form with 3 operands

Convert to pseudo RISC-V: f = (g + h) - (i + j);

A.

```
add    f, g, h
sub    f, i, j
```

```
add a, b, c  # a = b + c
sub a, b, c  # a = b - c
```

B.

```
add    t0, g, h
add    t1, i, j
sub    f, t0, t1
```

C.

```
sub    f, (add g,h), (add i,j)
```

D. More than one of these is correct

Register Operands

- Arithmetic instructions use register operands
- RISC-V has thirty-two 64-bit* general purpose registers
  - Numbered 0 to 31 with names `x0` through `x31`
  - Each has an "ABI name" (I called them friendly two classes ago) that reflects their conventional usage, e.g., `zero`, `t0`, `t1`, `s0`, `s1`, and `a0`
  - 32-bit data is called a word; 64-bit data is called a doubleword
- MIPS has thirty-two 32-bit general purpose registers
  - The numbers are different, but these are basically the same as RISC-V, e.g., `$zero`, `$t0`, `$t1`, `$s0`, `$s1`, and `$a0`.
- ARM has thirty 32-bit general purpose registers
- x86-64 has 16 general purpose registers, around 40 named registers used by the processor
  - Can be used as 8, 16, 32, or 64 bit registers

*RISC-V has 32-bit and 128-bit variants as well which have thirty-two 32-bit or 128-bit registers, respectively

RISC-V Register Convention

Register	ABI Name	Description
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5–7	t0–2	Temporaries
x8	s0/fp	Saved register/frame pointer
x9	s1	Saved register
x10–11	a0–1	Function argument/return values
x12–17	a2–7	Function arguments
x18–27	s2–11	Saved registers
x28–31	t3–6	Temporaries

We'll talk more about ra, sp, and the distinction between temporaries (t0–t6) and saved registers (s0–s11) when we talk about function calls

Register Operand Example

- C code:

```
f = (g + h) - (i + j);
```

  - f, g, h, i, and j in registers s0, s1, s2, s3, and s4

- Compiled RISC-V code:

```
add  t0, s1, s2
add  t1, s3, s4
sub  s0, t0, t1
```

Some R-type instructions

- `add dest, src1, src2`
- `sub dest, src1, src2`
- `mul dest, src1, src2`
- `div dest, src1, src2`
- `and dest, src1, src2`
- `or  dest, src1, src2`
- `xor dest, src1, src2`

Assume registers initially have the following values

a0	a1	t0	t1	s0
2	100	5	6	7

What values do they have after running this code?

```
add t0, a0, zero
add t1, a0, a0
add t1, t1, t1
sub t0, t1, t0
add s0, t0, a1
```

	a0	a1	t0	t1	s0
A	2	100	5	6	7
B	2	100	6	8	106
C	5	-10	-17	22	7
D	5	100	15	20	115
E	None of the above				

`add dest, src, zero`

- Previous slide used
  `add t0, a0, zero`
  to copy the contents of `a0` to `t0`
- This is such a common operation, there's a RISC-V pseudoinstruction—an instruction that the assembler expands to one or more real RISC-V instructions—for it: `mv`
- `mv  dest, src`

Questions about Arithmetic Operations?

Memory operands

- Memory operands are only used for instructions that load data from memory into a register or store data from a register into memory
- Memory operands have the form `offset(register)`
  - E.g., `0(t0)`, `32(s0)`, `-8(t1)`
- The value of the operand is a **memory address** and it is computed by taking the value of the register and adding the offset
  - E.g., if register `t1` holds the value 1016, then `-8(t1)` refers to memory address 1016 + -8 = 1008

Memory Instructions

- `ld  t0, 0(t1)`
  - `t0 = Mem[t1+0]`
  - Loads 8-byte double word from memory starting at address `t1+0` into register `t0`
- `sd  t0, 8(t1)`
  - `Mem[t1+8] = t0`
  - Stores 8-bytes double word in register `t0` into memory starting at address `t1+8`
- These instructions are the cornerstones of our being able to move data to and from memory

Load instructions

- **ld** — Load doubleword. Loads 8 bytes from memory into a register
  - `ld t0, 0(s0)`
- **lw** — Load word. Loads 4 bytes from memory into a register
  - `lw t0, 8(t4)`
- **lh** — Load halfword. Loads 2 bytes from memory into a register
  - `lh t2, 6(t1)`
- **lb** — Load byte. Loads 1 byte from memory into a register
  - `lb t3, 3(t0)`

- `ld`, `lw`, and `lb` are more common than `lh`

Store instructions

- **sd** — Store doubleword. Stores 8 bytes from a register into memory
  - `ld t0, 0(s0)`
- **sw** — Store word. Stores 4 bytes from a register into memory
  - `lw t0, 8(t4)`
- **sh** — Store halfword. Stores 2 bytes from a register into memory
  - `lh t2, 6(t1)`
- **sb** — Store byte. Stores 1 byte from a register into memory
  - `lb t3, 3(t0)`

- `sd`, `sw`, and `sb` are more common than `sh`

Accessing the Operands

There are typically two locations for nonconstant operands – registers (internal storage e.g., `t0` or `a0`) and memory. In each column we have which—reg or mem—is better for the metric specified in the column header. Which row is correct?

	Faster access	Smaller number to specify a reg/mem location	More locations
A	Mem	Mem	Reg
B	Mem	Reg	Mem
C	Reg	Mem	Reg
D	Reg	Reg	Mem
E	None of the above		

Load-store architectures

can do:

```
load r3, M(address)
add r1 = r2 + r3
```

can't do

```
add r1 = r2 + M(address)
```

$\Rightarrow$ forces heavy dependence on registers, which is exactly what you want in today's CPUs

- more instructions

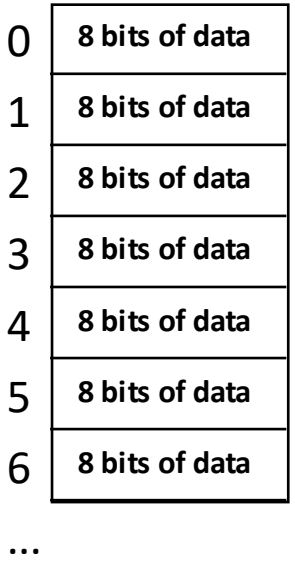
+ fast implementation

Memory

- Main memory used for composite data
  - Arrays, structures, dynamic data
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Data are (typically) **aligned** in memory
  - Address of a doubleword must be a multiple of 8; address of a word must be a multiple of 4; address of a halfword must be a multiple of 2
  - Data whose addresses are not a multiple of their size are **misaligned**
  - Misaligned memory accesses cause hardware exceptions in many architectures like MIPS and some implementations of RISC-V; others, like x86 support misaligned memory accesses but they are slower (for reasons we'll discuss at the very end of the semester!)

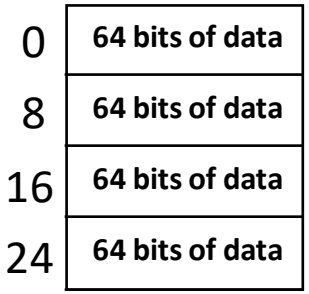
Memory Organization

- Viewed as a large, single-dimension array
- A memory address is an index into this array
- “Byte Addressing" means that the index points to a byte of memory.



Memory Organization

- Bytes are nice, but most data items use larger sizes, e.g., a word or a doubleword
- For RISC-V, a doubleword is 64 bits or 8 bytes



Registers hold 64 bits of data

- 2^{64} bytes with byte addresses from 0 to 2^{64} - 1
- 2^{61} doublewords with byte addresses 0, 8, 16, ... 2^{64} - 8
- 2^{64} is 18,446,744,073,709,551,616; computers don't really have that much memory!

If you have a pointer to address 1000 and you increment it by one to 1001. What does the new pointer point to, relative to the original pointer?

A) The next doubleword in memory

B) The next byte in memory

C) Either the next doubleword or byte – depends on if you use that address for a load byte or load doubleword

D) Pointers are a high level construct – they don't make sense pointing to raw memory addresses

E) None of the above

If an 8-byte doubleword is in memory at address 4203088, what is the address of the next doubleword in memory?

A) 4203089
B) 4203096
C) 14203084
D) It depends on the value of the doublewords in memory
E) Since a doubleword is 8 bytes, it’s not possible to have one at address 4203088

Getting the address of data in the first place

- Three main locations for data in a program
  - **Global variables (these live in the data segment)**
  - Local variables (function call stack)
  - Dynamically allocated memory (memory allocated at runtime)

Global variables in assembly code

- Global variables live in the data segment
- We can use assembler directives to
  1. Switch to the data segment
  2. Allocate space for the globals
  3. Switch back to the text (code) segment

```
.data                       # Switches to data segment
nums:                       # Label for address of following array
.dword 37, -42, 806         # Allocates space for 3 dwords
.text                       # Switches to text (code) segment
```

`la` — Load address pseudo instruction

Sets a register to the address pointed to by the symbolic label

```
.data
nums:
.dword 37, -42, 806

.text
main:
        la      s0, nums
        ld      t0, 0(s0)
        ld      t1, 8(s0)
        ld      t2, 16(s0)
```

After this code, `t0` holds `37`, `t1` holds `-42` and `t2` holds `806`

Reading

- Next lecture: Assembly continued
  - 2.3

- Problem Set 0: Due Friday at 11:59 p.m.