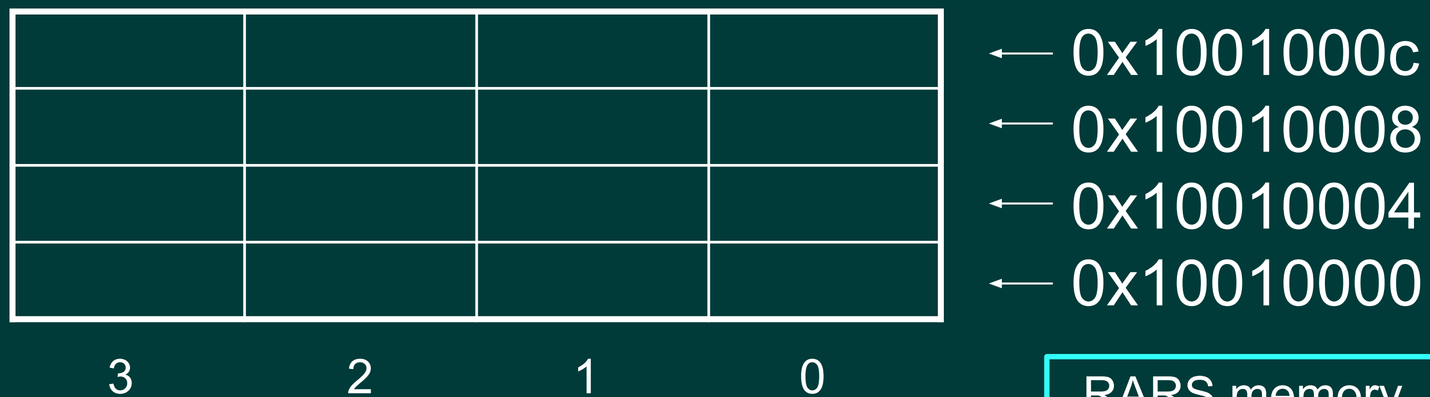
Visualizing Memory

- Logical view of memory is a 2-dimensional matrix
  - The smallest addressable unit is the byte (8 bits)
  - We also need to be able to address halfwords (16 bits), words (32 bits) and double words (64 bits)



RARS memory addresses

  - Each cell stores 1 byte
  - Each row stores 1 word
  - Two rows are required for double words
  - Memory addresses can be used to reference a byte, a halfword, word or double word
    - The size of data to be referenced is defined in the memory reference instruction (more on this to come).

RARS Programming View of Memory

- The programming view of memory is organized by word groups
  - 4-byte groups
  - Rows and columns

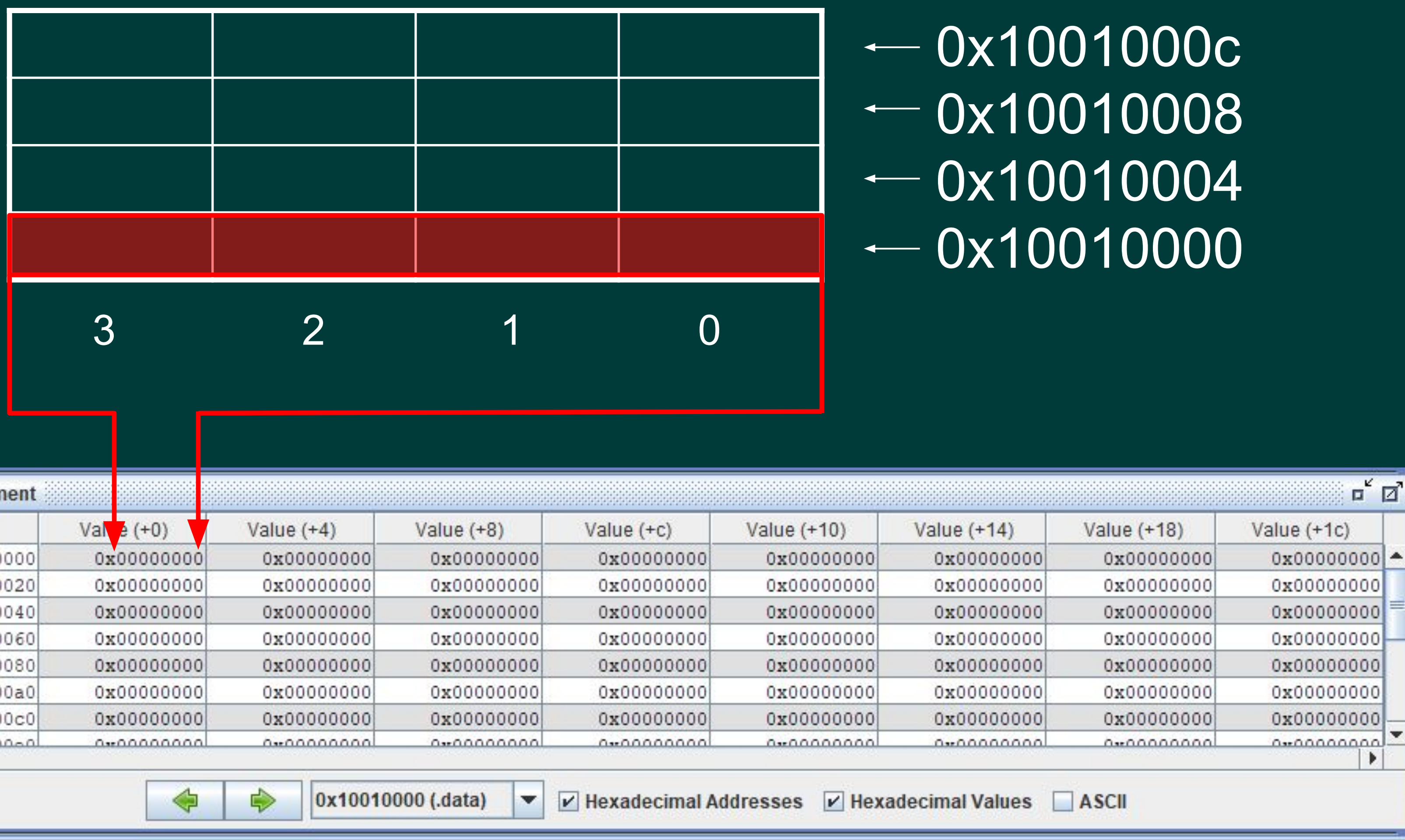
This is what memory will look like when we are programming in RARS.

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00281e11	0x0034bf15	0x63626161	0x33323164	0x00000034	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

- Each row contains 8 word cells – 4 bytes each cell
- Bytes are ordered “little endian” – low order address is rightmost byte within word
  - First data item is hex 11 shown in yellow highlighting at address 0x10010000
  - Subsequent bytes from right to left, then continue in next cell
  - Columns label offset of low order byte in each cell

Logical View vs. RARS View



Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

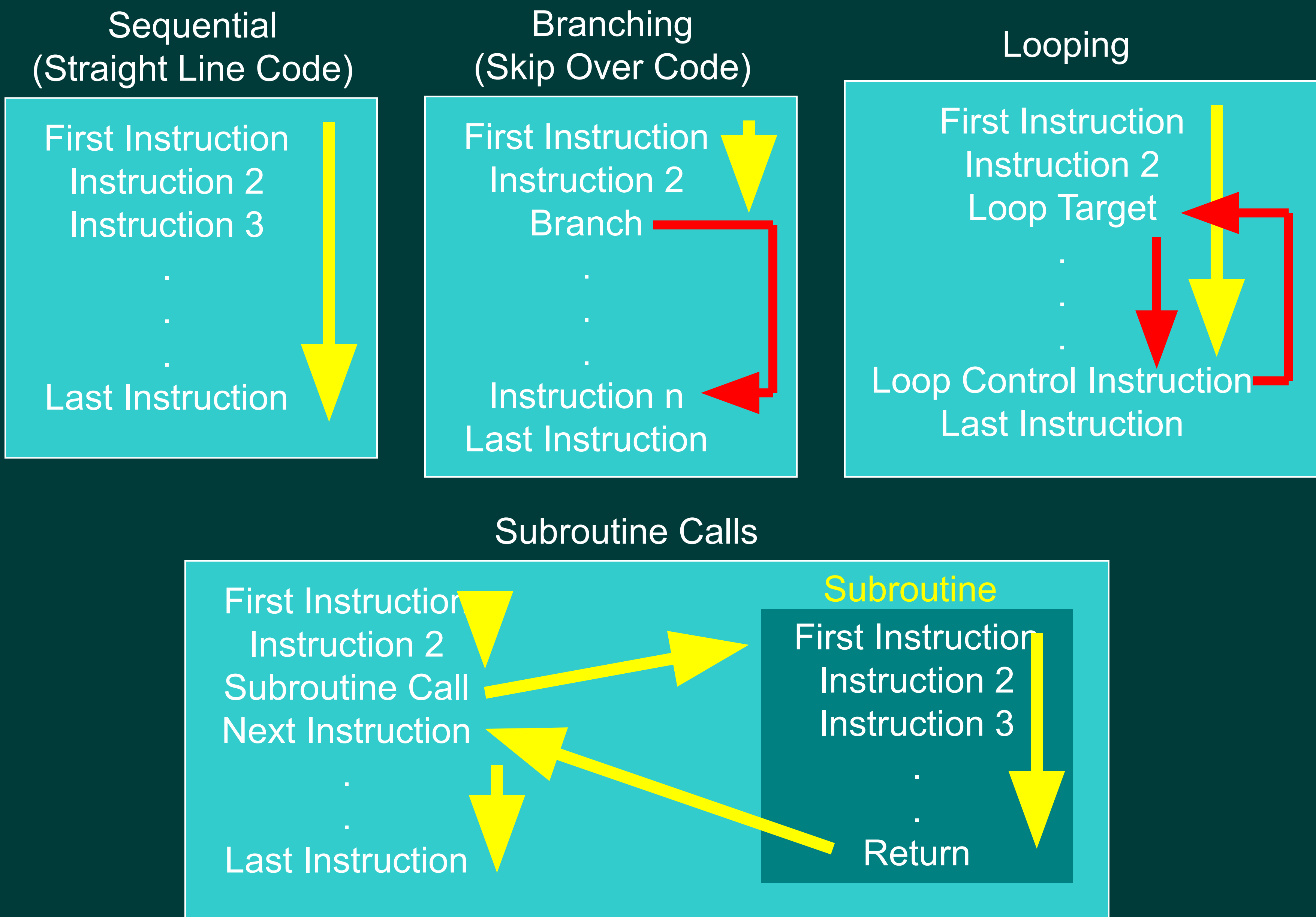
Assembly Language

- Hardware level programming language
  - Each instruction equates to one machine operation
- Instructions are executed sequentially
  - Each instruction is read from memory, decoded and executed (abstract view – more details in Ch. 4)
- The operations defined in a single HLL statement may result in several individual machine operations
- Assembly language has a rigid syntax because it is designed for specific hardware

Instructions

- Instruction is the basic unit of work in the CPU.
- Represents a single task (operation) to perform.
- Instructions have two parts:
  - The operation mnemonic (every instruction has one)
  - The operands to be used in performing the operation
    - Exceptions: Not all instructions need data – most do
- Types of operations
  - Memory access (data transfers - reading/writing data to/from memory)
  - Arithmetic/logic (add, subtract, and, or, etc.)
  - Control (decision making – branching, looping)
  - System calls (call on OS to perform a function)

Program Patterns in Assembly Language



RISC-V Assembly Language Program Structure

```
        .data

        … data allocation in memory

        .text
main:

        … instructions

exit:
```

Define data (data segment)

Perform operations on data (text segment)

.data and .text are *assembler directives*.
Assembler directives begin with a period and are used to convey information to the assembler and tell it how to translate an assembly language program to machine code.

main: and exit: are labels.
Labels define references to locations in memory. A memory address is associated with every label by the assembler.

Data Types

- Refers to a classification of data that specifies what kind of information a variable can hold
  - Usually determines the operations that can be performed on it
  - And how the data is stored in memory
  - It tells the computer how to interpret the value of a piece of data
- Primitive
  - Built-in, intrinsic, basic, single values
- Non-primitive
  - Data types that store a collection of values in various formats, rather than just a single value

Java Data Types

byte (p) 8-bit signed int
short (p) 16-bit signed int
int (p) 32-bit signed int
long (p) 64-bit signed int
char (p) 16-bit Unicode
String (n-p) class

p = primitive
n-p = non-primitive
(all statically typed)

RISC-V Data Elements

byte 8-bit signed int
half 16-bit signed int
word 32-bit signed int
dword 64-bit signed int
ascii 8-bit UTF-8 Unicode
asciz null-term char

Assembly level primitives;
not statically typed; use is dependent on context

Defining Data in Memory

- RISC-V assembler directives in .data segment
  - .byte - 8-bit numeric value
  - .half - 16-bit numeric value
  - .word- 32-bit numeric value
  - .dword - 64-bit numeric value
  - .ascii - ASCII character (1 or more)
  - .asciz - null terminated ASCII character (1 or more)
  - .space - specify number of bytes to allocate
- Most data items are labeled for reference (symbol name)
  - Synonymous with user-defined variable names in HLLs
  - Symbol names must be unique
    - Can't use same label for multiple data items
- Data items should be initialized to some value
- Multiple values can be allocated with a single directive
  - Ex. arrays (more on this later)

Example Data Definition in RISC-V

```
Code                                    Comments

        .data                     # begins data segment
num:     .byte    17               # 8-bit number
array:   .byte    30  40  50  60   # four 8-bit numbers (array)
lrgnum:  .half    600              # 16-bit number
bignum:  .word    3456789          # 32 bit number
char:    .ascii   "a"              # single character
strng:   .asciz   "abcd1234"       # null terminated string (9 bytes)
extra:   .space   10               # memory space for 10 bytes
```

Notes:

1. Each data value has a symbol name. Symbol names are user defined (just like variable names in HLLs).
2. Multiple values can be delimited by a blank or by a comma (arrays).
3. Characters and strings must be enclosed in double quotation marks.
4. The extra definition allocates 10 bytes of memory. Anything can be stored in this space (10 bytes or character data).
5. The numeric values specified are by default base 10. You can also specify hexadecimal values by using the prefix 0x and the hex number.

RARS Memory Allocation

```
        .data
num:    .byte   17
array:  .byte   30  40  50  60
lrgnum: .half   600
bignum: .word   3456789
char:   .ascii  "a"
strng:  .asciz  "abcd1234"
extra:  .space  10
```

3	2	1	0	Address
space (4 bytes)				0x1001001c
space (4 bytes)				0x10010018
space (2 bytes)		"null"	"4"	0x10010014
"3"	"2"	"1"	"d"	0x10010010
"c"	"b"	"a"	"a"	0x1001000c
3456789				0x10010008
600		⊗	60	0x10010004
50	40	30	17	0x10010000

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x32281e11	0x0258003c	0x0034bf15	0x63626161	0x3332[illegible]164	0x00000034	0x00000000	0x00000000

Labels

Label	Address ▲
DataSegmentExampleFro...	
num	0x10010000
array	0x10010001
lrgnum	0x10010006
bignum	0x10010008
char	0x1001000c
strng	0x1001000d
extra	0x10010016

Empty memory location
- Memory hole
- Could not be filled due to data values defined & sequencing

Alignment Restriction

- Defines where numeric values are placed when defined in memory
- Offset restriction based on size of data
- RISC-V default:
  - Halfword values must begin at memory addresses that are a multiple of 2
  - Word values must begin at memory addresses that are a multiple of 4
  - Double-word values must begin at memory addresses that are a multiple of 4

Memory Offset

- The memory offset is a value that is added to the base address forming the actual memory address of the memory location to be referenced
- The offset value is determined by the order and types of data defined in the data segment
- Example: our previous data definition

	.data		
num:	.byte	17	← offset 0 (always for 1st data value)
array:	.byte	30 40 50 60	← offset 1 (30), 2 (40), 3 (50), 4 (60)
lrgnum:	.half	600	← offset 6 (extending through 7)
bignum:	.word	3456789	← offset 8 (extending through 11)
char:	.ascii	"a"	← offset 12
strng:	.asciz	"abcd1234"	← offset 13 (extending through 21)
extra:	.space	10	← offset 22 (extending through 31)

Ponder This …

```
      .data
one:    .byte   97 98 99          # 3-element array
two:    .word   0x00636261        # int defined in hex
three:  .asciz "abc“              # 3-char null term string
four:   .word   6513249           # int defined in decimal
```

In memory:

0x10010000	0x00636261	0x00636261	0x00636261	0x00636261
↑	↑	↑	↑	↑
Starting address of memory	one	two	three	four