OAMulator

Online
One Address Machine emulator
and
OAMPL compiler

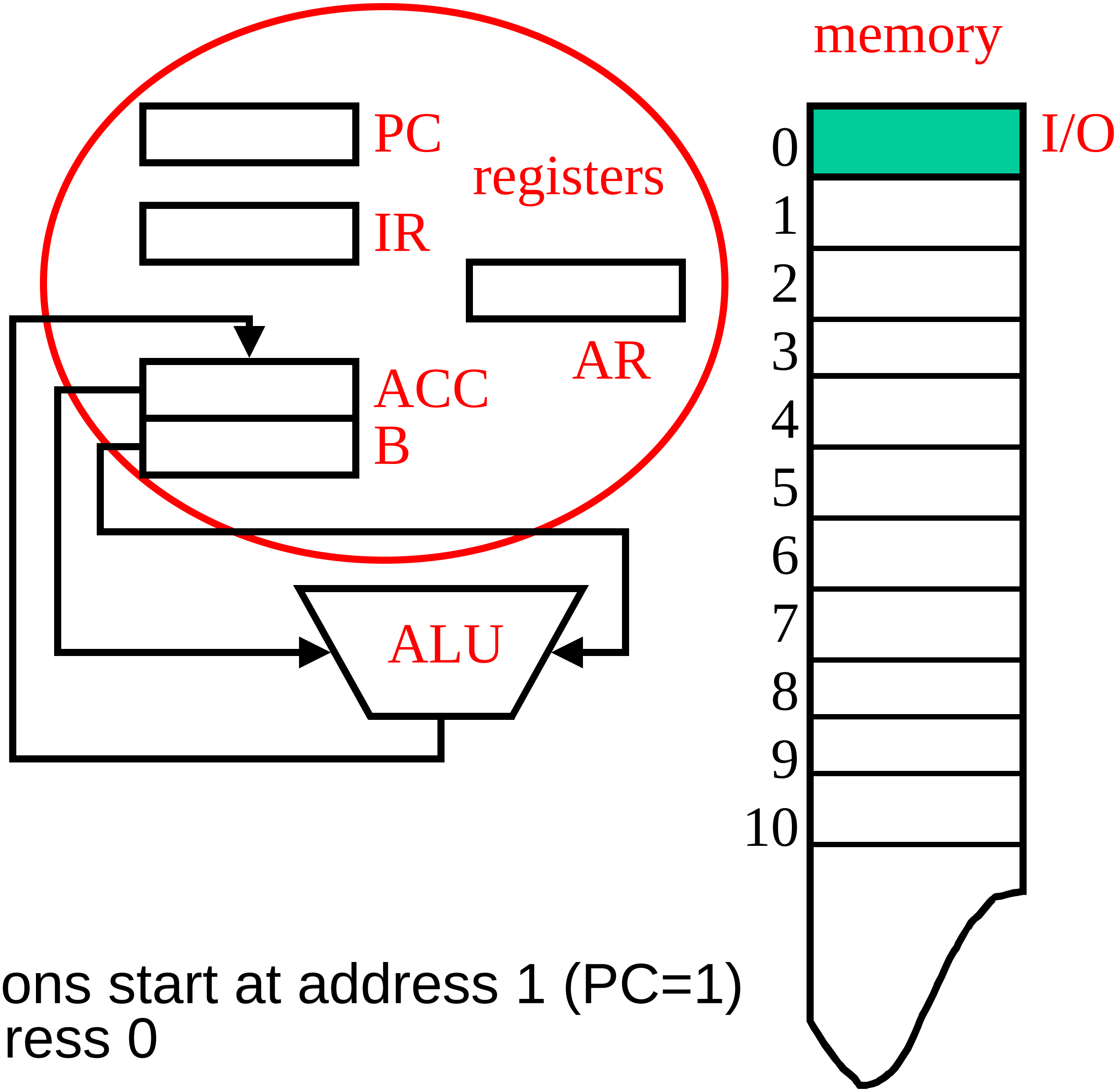
http://myspiders.biz.uiowa.edu/~fil/OAM/

OAMulator educational goals

- OAM emulator concepts
  - Von Neumann architecture
  - Registers, ALU, controller
  - CPU stages of execution
  - Instruction Set Architecture
  - Assembly languages
  - Memory and addressing

- OAMPL compiler concepts
  - High level languages
  - Compilers
  - I/O, assignment, and control statements
  - Variable reference resolution
  - Expressions and parsing
  - Optimization

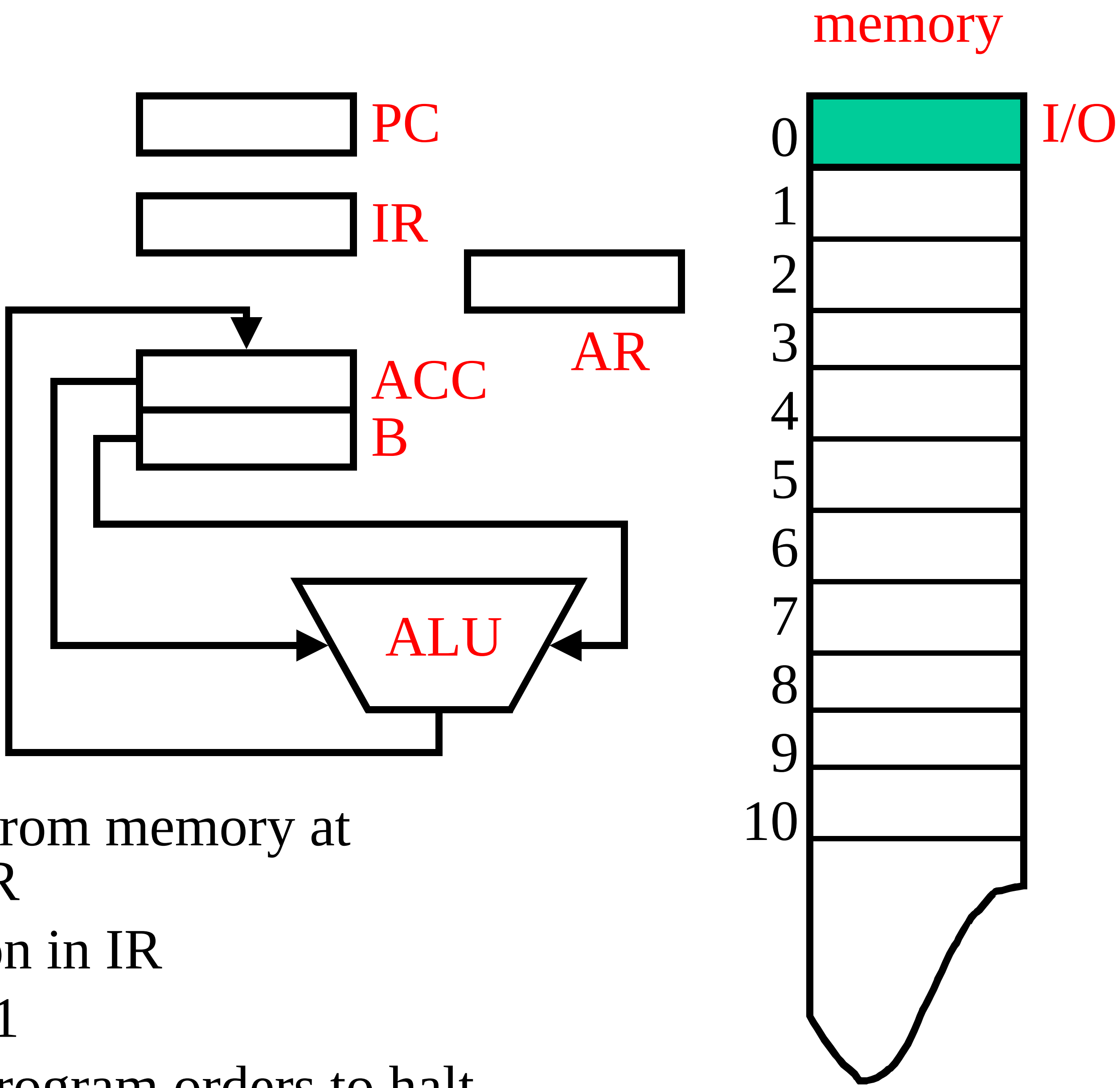
The One Address Machine (OAM)



Assumptions:

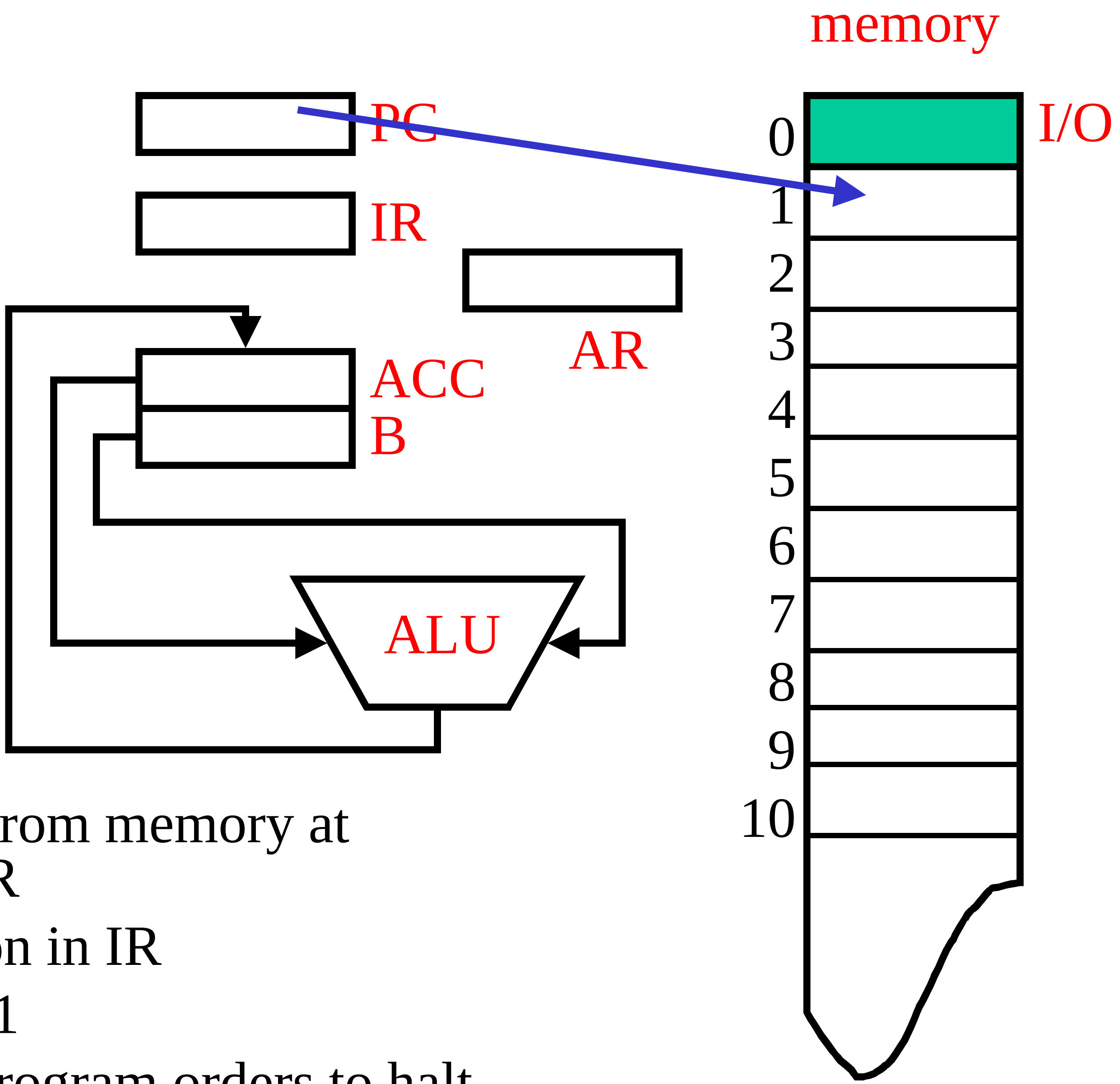
- Infinite memory
- Integers only
- Program instructions start at address 1 (PC=1)
- I/O occurs at address 0

Fetch-execute-increment cycle



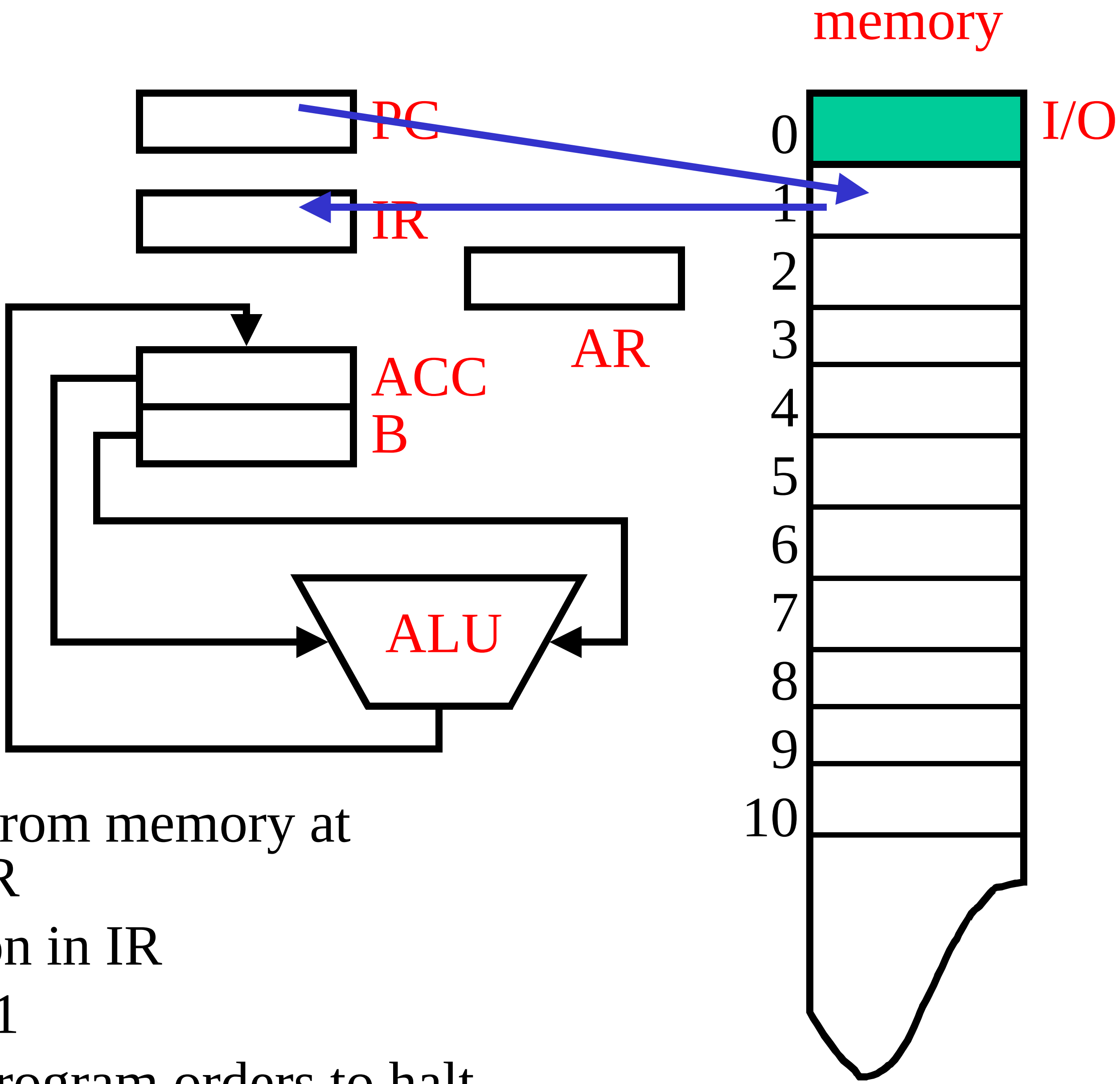
- Read instruction from memory at address PC into IR
- Execute instruction in IR
- Increment PC by 1
- Repeat until the program orders to halt

Fetch-execute-increment cycle



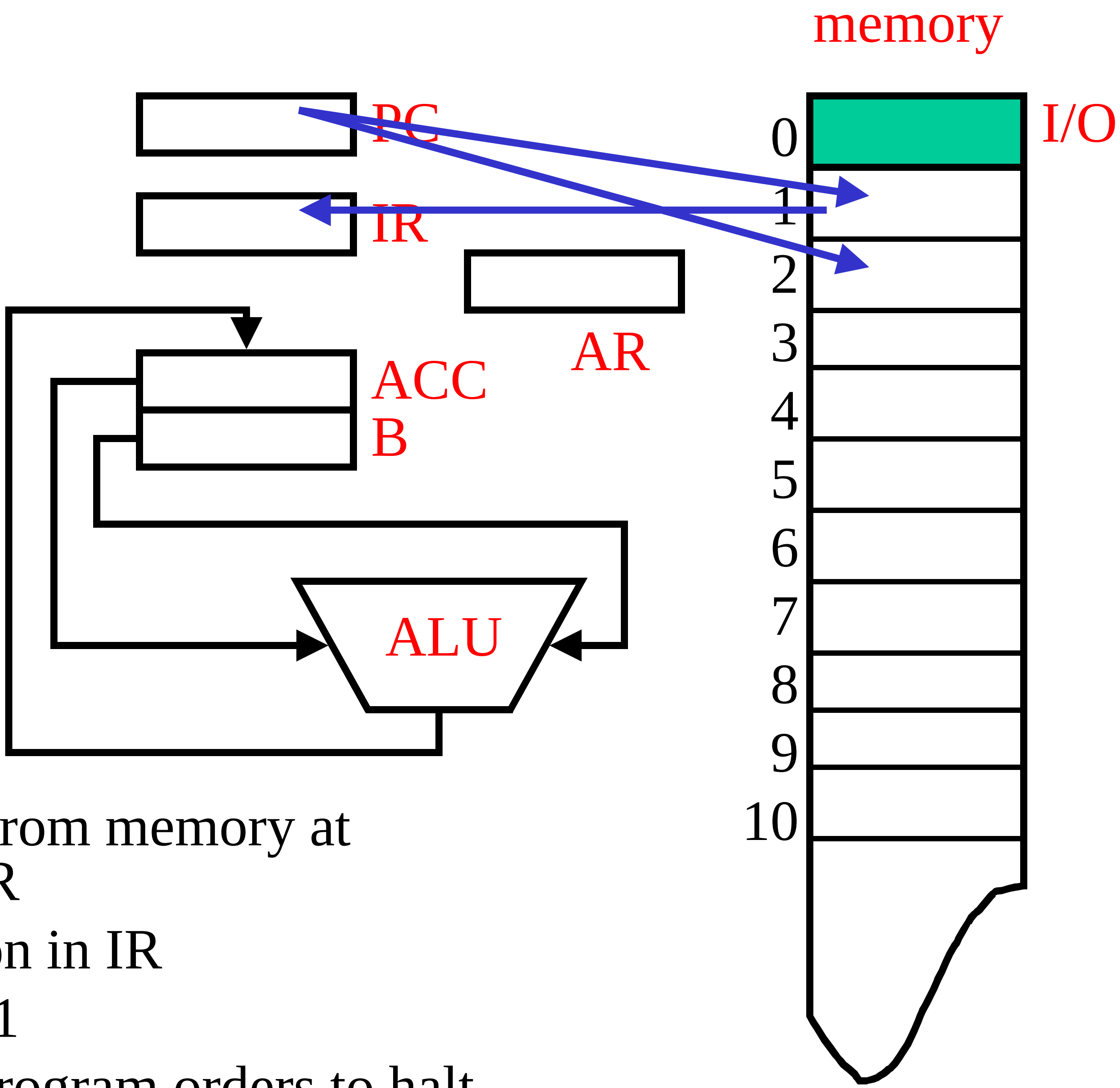
- Read instruction from memory at address PC into IR
- Execute instruction in IR
- Increment PC by 1
- Repeat until the program orders to halt

Fetch-execute-increment cycle



- Read instruction from memory at address PC into IR
- Execute instruction in IR
- Increment PC by 1
- Repeat until the program orders to halt

Fetch-execute-increment cycle



- Read instruction from memory at address PC into IR
- Execute instruction in IR
- Increment PC by 1
- Repeat until the program orders to halt

OAM assembly language

- OAM Instruction Set: the set of instructions understood by the assembler/controller
- ALU instructions: one address

```
ADD address ;; load B, add ACC and B, store result in ACC
SUB address ;; load B, subtract B from ACC, store result in ACC
MLT address ;; load B, multiply ACC and B, store result in ACC
DIV address ;; load B, divide ACC by B, store result in ACC
```

  - OP address means take the value at address, put it in register B, then do OP on contents of ACC and B, and finally store result in ACC

```
SET value ;; set ACC to value
NEG       ;; negate ACC
INC       ;; increment ACC value by 1
DEC       ;; decrement ACC value by 1
```

OAM assembly language

- Memory and I/O instructions: one address

```
LDA address ;; load ACC with value stored in memory at address
STA address ;; store contents of ACC in memory at address
```

  - OP address means do OP between values at address and ACC
  - Write <=> Output and Read <=> Input if address=0 (memory-mapped I/O)

OAM assembly language

- Control instructions: alter the flow of control from the sequential model

```
BR  address ;; set PC=address (next instruction from address+1)
BRP address ;; set PC=address if ACC is positive, else ignore
BRZ address ;; set PC=address if ACC is zero, else ignore
HLT         ;; stop OAM
```

  - Conditional and unconditional branch
    - The unconditional branch is a one-address instruction that instructs the computer to "take the next instruction from memory location address instead of current address +1”
    - The conditional branch does the same but only if the given condition on ACC is satisfied

Simple OAM assembly language programs

```
1. LDA 0
2. STA 100
3. LDA 0
4. ADD 100
5. STA 100
6. MLT 100
7. STA 0
8. HLT
```

```
1. SET 10
2. STA 0
3. DEC
4. BRP 1
5. HLT
```

- What do they do?

Simple OAM assembly language programs

```
1. LDA 0
2. STA 100
3. LDA 0
4. ADD 100
5. STA 100
6. MLT 100
7. STA 0
8. HLT
```

```
1. SET 10
2. STA 0
3. DEC
4. BRP 1
5. HLT
```

Initial internal state:
PC=1; AR=?; IR=?;
ACC=?; B=?

- What do they do?

Simple OAM assembly language programs

```
1. LDA 0
2. STA 100
3. LDA 0
4. ADD 100
5. STA 100
6. MLT 100
7. STA 0
8. HLT
```

```
1. SET 10
2. STA 0
3. DEC
4. BRP 1
5. HLT
```

Initial internal state:
PC=1; AR=?; IR=?;
ACC=?; B=?

After fetch:
PC=1; AR=1; IR=SET 10;
ACC=?; B=?

- What do they do?

Simple OAM assembly language programs

```
1.  LDA 0
2.  STA 100
3.  LDA 0
4.  ADD 100
5.  STA 100
6.  MLT 100
7.  STA 0
8.  HLT
```

```
1.  SET 10
2.  STA 0
3.  DEC
4.  BRP 1
5.  HLT
```

Initial internal state:
PC=1; AR=?; IR=?;
ACC=?; B=?

After fetch:
C=1; AR=1; IR=SET 10;
ACC=?; B=?

After execute:
PC=1; AR=1; IR=SET 10;
ACC=10; B=?

- What do they do?

Simple OAM assembly language programs

```
1. LDA 0
2. STA 100
3. LDA 0
4. ADD 100
5. STA 100
6. MLT 100
7. STA 0
8. HLT
```

```
1. SET 10
2. STA 0
3. DEC
4. BRP 1
5. HLT
```

Initial internal state:
PC=1; AR=?; IR=?;
ACC=?; B=?

After fetch:
C=1; AR=1; IR=SET 10;
ACC=?; B=?

After execute:
=1; AR=1; IR=SET 10;
ACC=10; B=?

After increment:
PC=2; AR=1; IR=SET 10;
ACC=10; B=?

- What do they do?

Programming and compilation

- Let's see how your applications written in some high-level language are compiled
  - OAMPL (Simple PL for One Address Machine)
  - OAMPL compiler: from OAMPL to OAM machine code
- OAMPL instructions are more complex than (correspond to many) OAM instructions
- Example:

```
WRITE "Input a B value."
READ B
WRITE "Input an A value."
READ A
ASSIGN A (- A (* B A))
ASSIGN C (* A A)
WRITE "The value of (A - AB) squared is"
WRITE C
```

OAMPL compiler

- The job of the compiler is to translate OAMPL instructions into a corresponding set of OAM instructions without compromising the integrity of the algorithm the program implements
- Many different, equally valid, such translations exist. All things being equal, we'd prefer a shorter translation, since fewer OAM instructions that accomplish the same thing imply the program will execute faster

OAMPL instructions

- I/O, assignment, and control statements
- OAMulator's OAMPL compiler is case-independent for keywords and case-dependent for variables
- I/O statements: READ & WRITE
  - `WRITE const | variable | exp`
  - `READ variable`

```
WRITE 5
WRITE "Foo!"
WRITE A
WRITE (+ A (* 5 B))
```

```
READ A
```

Resolving variable references

- OAM only groks memory locations
- We need a map from variable names to memory locations
  - Compilers generally do a first pass to figure out all variables so that variable locations will not clash with the loaded program instructions
  - We use a less sophisticated approach: skip a location whenever a variable is first encountered (intersperse variables with code)

```
1. LDA 0   ;; Read A into ACC
2. STA 4   ;; Place A into location 4
3. BR 4    ;; Skip memory location where A is stored
4. NOOP    ;; Place holder; this instruction will NEVER be executed
…
12. LDA 0   ;; Read A again (not new: reuse location for consistency!)
13. STA 4   ;; Place A into location 4
```

Parsing OAMPL expressions

- OAMPL expressions consist of nested expressions using the operators +, –, *, /
- ***`(operator operand1 operand2)`***

  Prefix notation make parsing easier

  +, *, / take 2 expressions as arguments

  – may take one or two expressions as arguments
- At the lowest level, an expression may be a constant or a variable name
- Example: if C=4, what should the following print to the screen?

```
WRITE (* 3 (- C))
```

Parsing OAMPL expressions

- OAMulator's OAMPL compiler only allows at most one level of nesting in expressions
- Examples:
  - GOOD: `(+ a (/ b 2))`
  - GOOD: `(* (- x1 x2) (- x1 x2))`
  - BAD: `(- 1 (* 3 (- c)))`

Parsing OAMPL expressions

```
WRITE (* 3 (- C))
```

- Compiler strategy:
  - Evaluate first operand
  - Store this intermediate result in a temporary location
  - Evaluate second operand
  - Multiply by intermediate result
  - Write to screen
- So if we start at 11 and assume C is in 7:

```
;; instructions to place value of first operand for MLT instruction in ACC...
n   STA n+2  ;; Place intermediate result in location n+2
n+1 BR  n+2  ;; Skip intermediate result
n+2 NOOP     ;; Place holder for intermediate result
;; instructions to place value of second operand for MLT instruction in ACC...
m   MLT n+2  ;; Multiply by intermediate result from n+2
m+1 STA 0    ;; Write result to output
```

Parsing OAMPL expressions

- Compiler strategy:
  - Evaluate first operand
  - Store this intermediate result in a temporary location
  - Evaluate second operand
  - Multiply by intermediate result
  - Write to screen

```
WRITE (* 3 (- C))
```

- So if we start at 11 and assume C is in 7:

```
11. SET 3    ;; Load a 3 into ACC
12. STA 14   ;; Store intermediate result
13. BR 14    ;; Skip intermediate result
14. NOOP     ;; Place holder
15. LDA 7    ;; Load value of C into ACC
16. NEG      ;; Negate
17. MLT 14   ;; Multiply by intermediate result
18. STA 0    ;; Write ACC to screen
```

```
;; instructions to place value of first operand for MLT instruction in ACC...
n   STA n+2  ;; Place intermediate result in location n+2
n+1 BR  n+2  ;; Skip intermediate result
n+2 NOOP     ;; Place holder for intermediate result
;; instructions to place value of second operand for MLT instruction in ACC...
m   MLT n+2  ;; Multiply by intermediate result from n+2
m+1 STA 0    ;; Write result to output
```

OAMPL assignment statement

- ASSIGN is the most common OAMPL statement
  - **`ASSIGN variable const|var|exp`**
  - As with READ, ASSIGN should handle both the situation where the first argument is a brand new variable, and the situation where the first argument is a preexisting variable

OAMPL control statements

- The OAMPL statements IF and ENDIF support conditional execution
  - **`IF const|var|exp … ENDIF`**
- Notes:
  - Can be nested!
  - Why can we do without ELSE?

```
IF 3          ;; true if != 0
WRITE ”Bar!” ;; body
ENDIF         ;; continue
```

OAMPL control statements

- The OAMPL statements IF and ENDIF support conditional execution
  - `IF const|var|exp … ENDIF`
- Notes:
  - Can be nested!
  - Why can we do without ELSE?

```
IF 3           ;; true if != 0
WRITE ”Bar!” ;; body
ENDIF          ;; continue
```

```
20. SET 3
21. BRZ 23
22. SET ”Bar!"
23. STA 0
```

OAMPL control statements

- The OAMPL control statements LOOP and END support the notion of iteration
  - **`LOOP const|var|exp … END`**
- Notes for proper implementation:
  - Maybe zero times!
  - Can have nested loops!

```
LOOP 10        ;; repeat 10 times
WRITE "Foo!"   ;; body
END            ;; go back unless done
```

OAMPL control statements

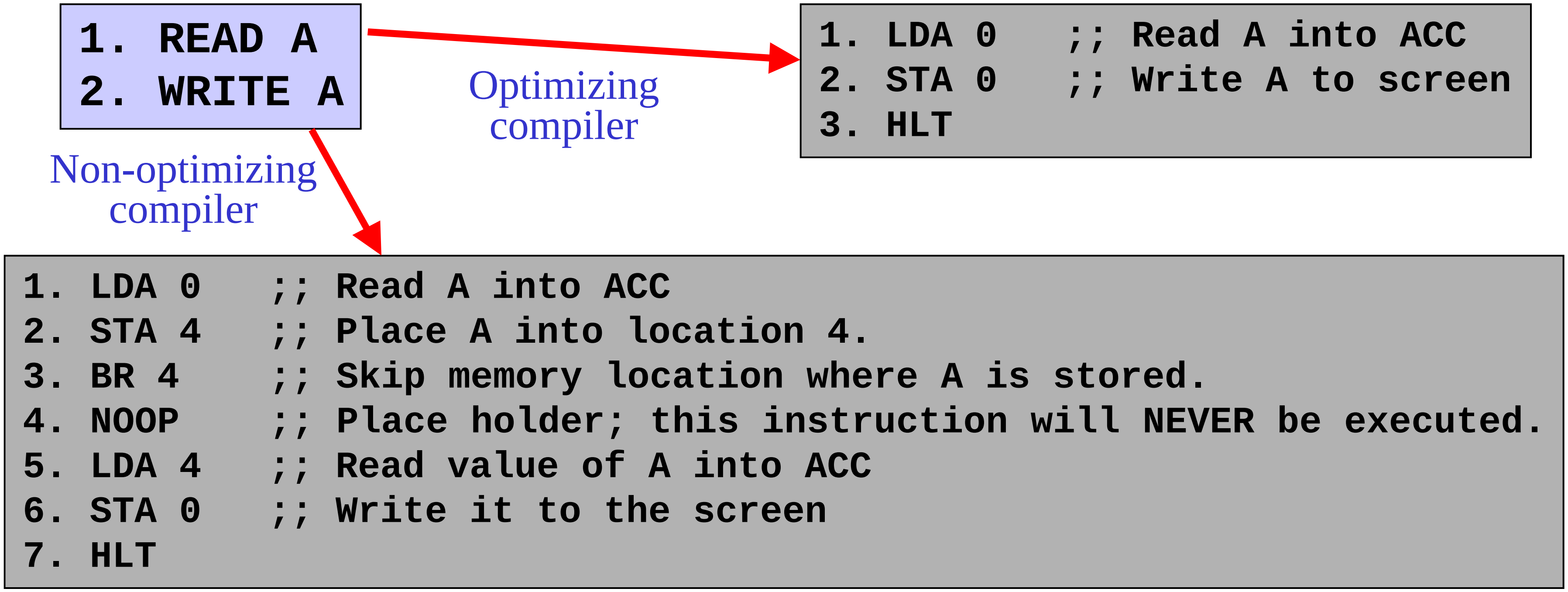
- The OAMPL control statements LOOP and END support the notion of iteration
  - **`LOOP const|var|exp … END`**
- Notes for proper implementation:
  - Maybe zero times!
  - Can have nested loops!

```
LOOP 10        ;; repeat 10 times
WRITE "Foo!"  ;; body
END            ;; go back unless done
```

```
18. SET 10
19. BR 25
20. NOOP
21. STA 20
22. SET "Foo!"
23. STA 0
24. SET -1
25. ADD 20
26. BRP 20
```

Optimizing compilers

- The complexity of the compiler increases:
  - With the complexity of the language
  - With the desired efficiency of the output code



```
1. READ A
2. WRITE A
```

Optimizing compiler

```
1. LDA 0    ;; Read A into ACC
2. STA 0    ;; Write A to screen
3. HLT
```

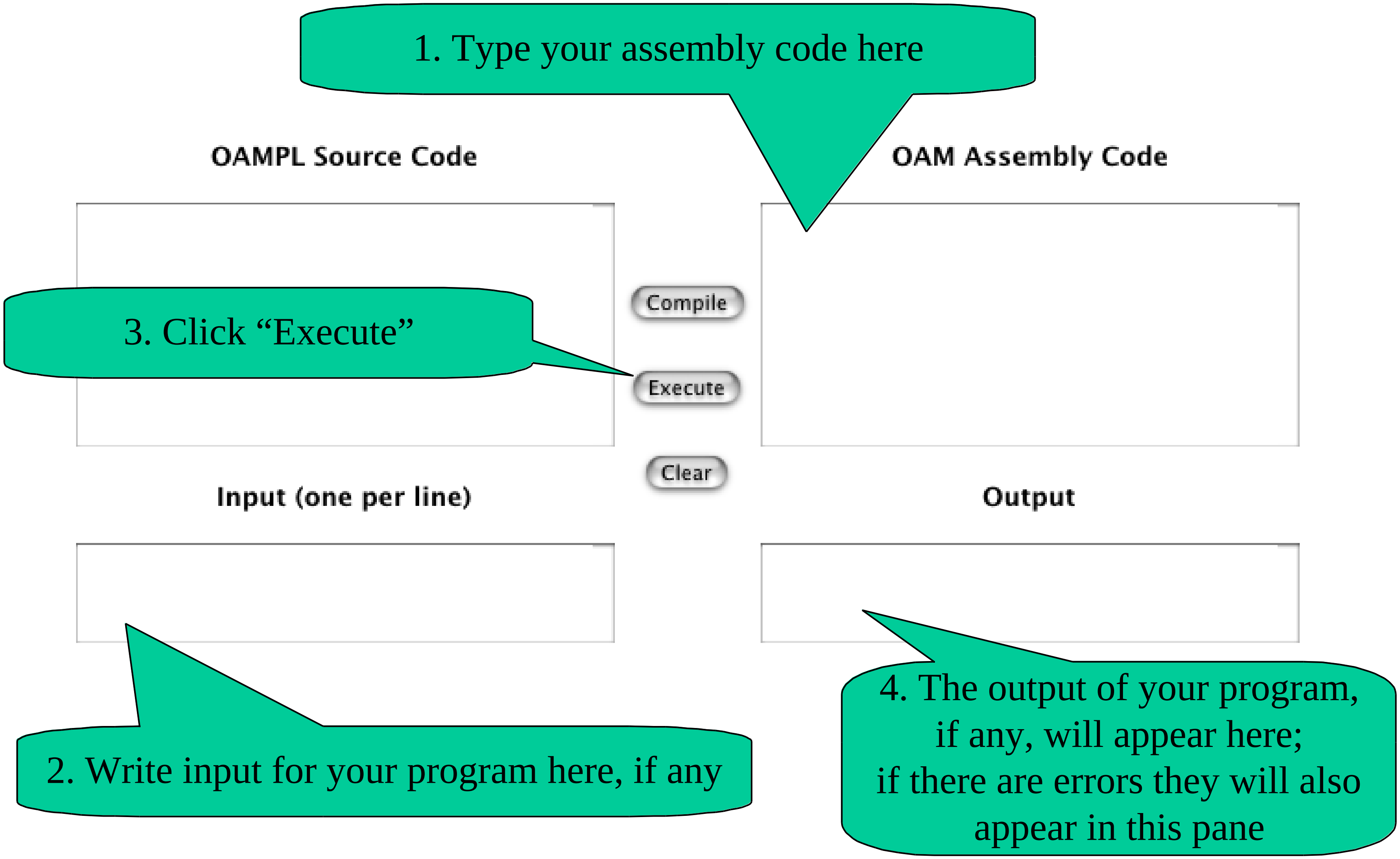
Non-optimizing compiler

```
1. LDA 0    ;; Read A into ACC
2. STA 4    ;; Place A into location 4.
3. BR 4     ;; Skip memory location where A is stored.
4. NOOP     ;; Place holder; this instruction will NEVER be executed.
5. LDA 4    ;; Read value of A into ACC
6. STA 0    ;; Write it to the screen
7. HLT
```

Optimizing compilers

- How could we optimize this sample code?
    - Note that A is not used elsewhere
    - Note that the integrity of A is not compromised by other operations
- Optimizers make multiple passes through the code to check for many such conditions and transform the code to more and more efficient forms, without altering the semantics!
    - Some optimizations are machine dependent
    - Optimization is hard and time consuming
- OAMulator's compiler does *not* optimize

OAMulator tutorial: OAM



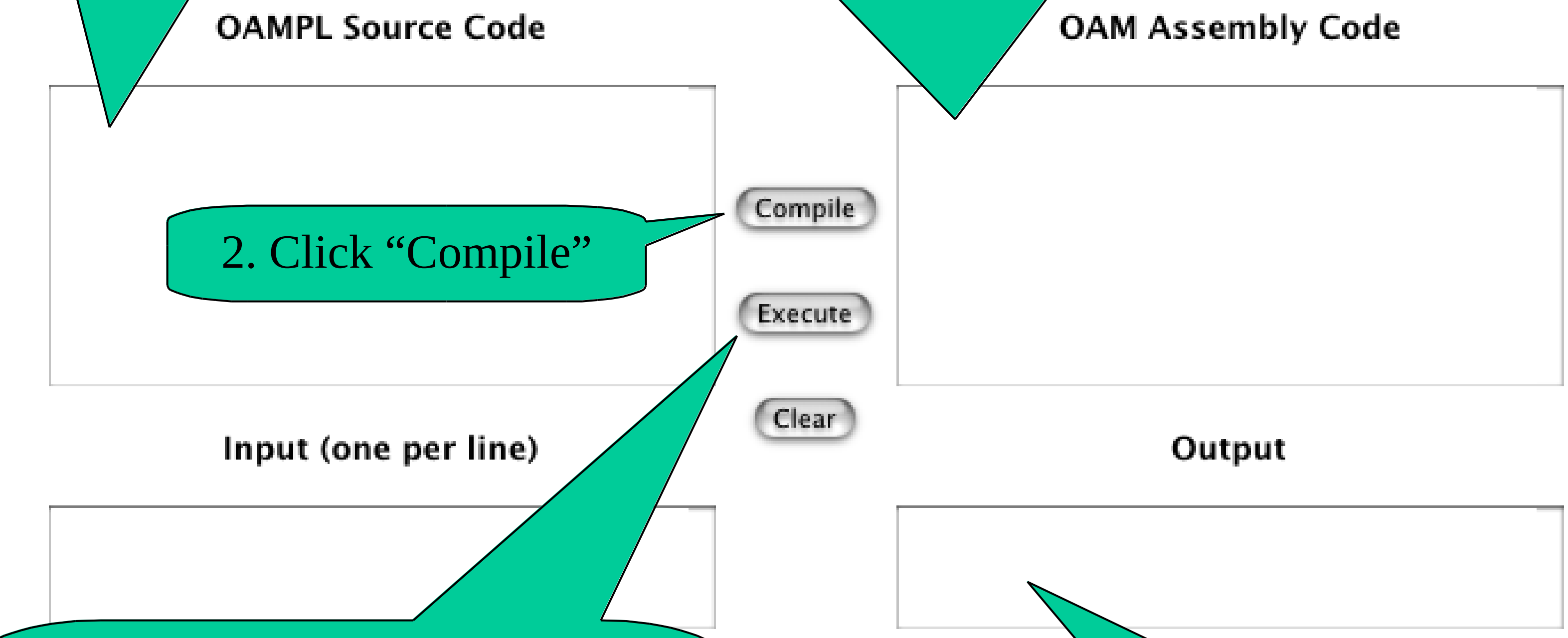
1. Type your assembly code here

3. Click “Execute”

2. Write input for your program here, if any

4. The output of your program, if any, will appear here; if there are errors they will also appear in this pane

OAMulator tutorial: OAMPL



1. Type your program here

2. Click “Compile”

3. Your program, compiled into assembly, will appear here

4. If there are error messages they will appear here

5. If compilation was successful, you can now run your compiled program by clicking “Execute” (don’t forget input if necessary)

History and credits

- OAM and OAMPL were developed in the early 1990's by Prof. Alberto Maria Segre at Cornell University
  - Part of a suite of simple instruction sets and computer architectures designed to support instruction in an introductory computer science course for non-majors
  - First came the SM (Stack Machine) and SMPL
  - Then came OAM and OAMPL
  - Last came the TAM (Two Address Machine) and TAMPL
  - Emulators & compilers for these machines & PLs were written in Scheme
- OAM and OAMPL have been used by Proff. Segre and Menczer to support teaching of computer hardware and software concepts in the *Introduction to Information Systems* course, part of the MIS Master program at the University of Iowa
- OAMulator was developed in Perl by Prof. Menczer
  - It is hosted on the *myspiders* server, funded in part by an Instructional Improvement Award from the University of Iowa Council on Teaching