University of Glasgow

Programming Languages H

Dr Ornela Dardha

University of Glasgow

4 Interpretation

- Overview
- Virtual machine interpretation
- Case study: SVM
- Case study: SVM interpreter in Java

Overview

- An ***S* interpreter** accepts code expressed in language *S*, and *immediately* executes that code.
- Assume that the code to be interpreted is just a sequence of simple instructions (including conditional/unconditional jumps).
- The interpreter works as follows:
  - First it initializes the state.
  - Then it repeatedly *fetches*, *analyses*, and *executes* the next instruction.
  - Executing an instruction updates the state as required.

University of Glasgow

Virtual machine interpretation

- Virtual machine code typically consists of:
  - load/store instructions
  - arithmetic/logical instructions
  - conditional/unconditional jumps
  - call/return instructions
  - etc.
- The virtual machine state typically consists of:
  - storage (code, data)
  - registers (status, program counter, stack pointer, etc.).

Case study: SVM *(1)*

- **SVM** (Simple Virtual Machine) will be used as a case study in this course.
- SVM is suitable for executing programs in *simple imperative* PLs.
- For a full description, see *SVM Specification* (available from the PL Moodle page).

Case study: SVM *(2)*

- Source code and corresponding SVM code:

```
p = 1;
while (p < n)
   p = 10*p;
```

```
 0: LOADC 1
 3: STOREG 2
 6: LOADG 2
 9: LOADG 1
12: COMPLT
13: JUMPF 29
16: LOADC 10
19: LOADG 2
22: MULT
23: STOREG 2
26: JUMP 6
29: HALT
```

assume that `n` and `p` are located at global addresses 1 and 2

code to evaluate ‘`p < n`’

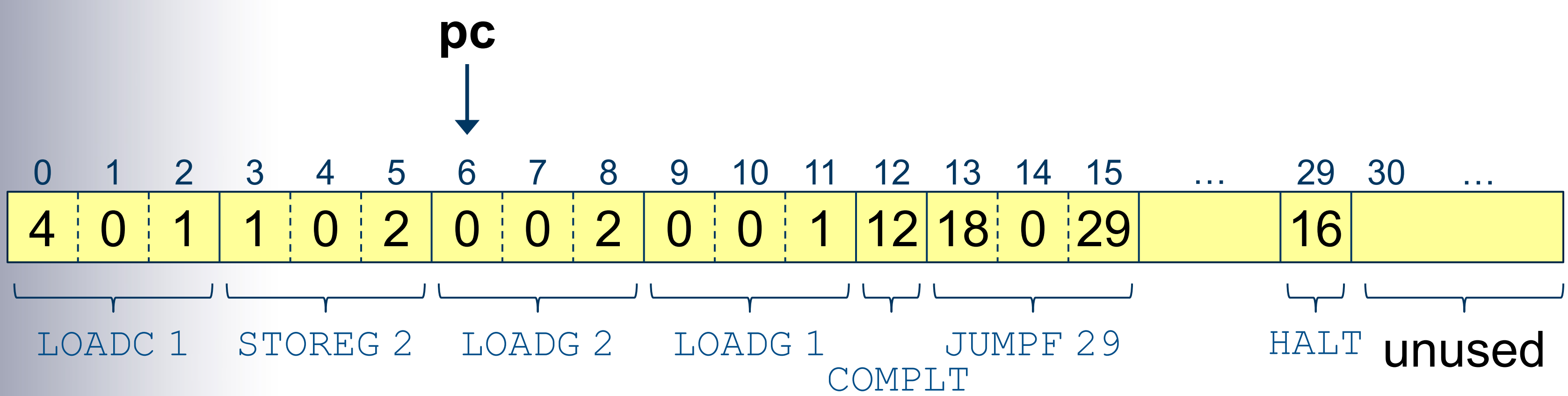
code to execute ‘`p = 10*p;`’

Case study: SVM *(3)*

- SVM storage:
  - the **code store** is a fixed array of *bytes* (32,768 *bytes*) providing space for **instructions**
  - the **data store** is a fixed array of *words* (32,768 *words*) providing a stack to contain *global* and *local* **data**.
- SVM main registers:
  - **pc** (program counter) points to the next instruction to be executed
  - **sp** (stack pointer) points to the top of the stack
  - **fp** (frame pointer) points to the base of the topmost frame *(see §14)*
  - **status** indicates whether the programming is running, failed, or halted.

Case study: SVM *(4)*

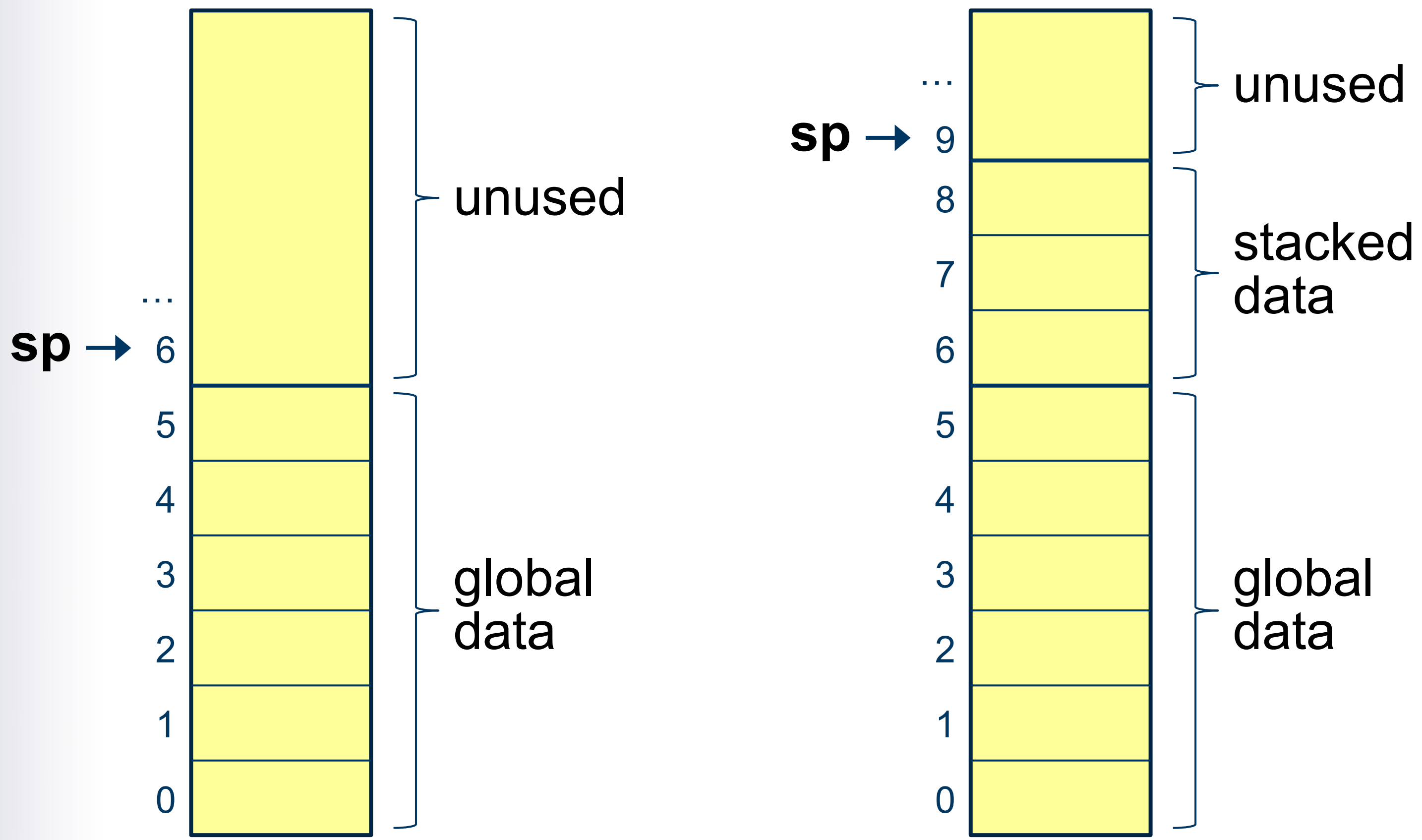
- Illustration of code store:



- Each instruction occupies 1, 2, or 3 bytes.

Case study: SVM *(5)*

- Illustration of data store (simplified):



Case study: SVM *(6)*

- SVM instruction set (simplified):

Op-code	*Mnem-onic*	*Behaviour*
6	`ADD`	pop w_2; pop w_1; push (w_1+w_2)
7	`SUB`	pop w_2; pop w_1; push (w_1–w_2)
8	`MUL`	pop w_2; pop w_1; push (w_1×w_2)
9	`DIV`	pop w_2; pop w_1; push (w_1/w_2)
10	`CMPEQ`	pop w_2; pop w_1; push (if w_1=w_2 then 1 else 0)
11	`CMPLT`	pop w_2; pop w_1; push (if w_1<w_2 then 1 else 0)
14	`INV`	pop w; push (if w=0 then 1 else 0)

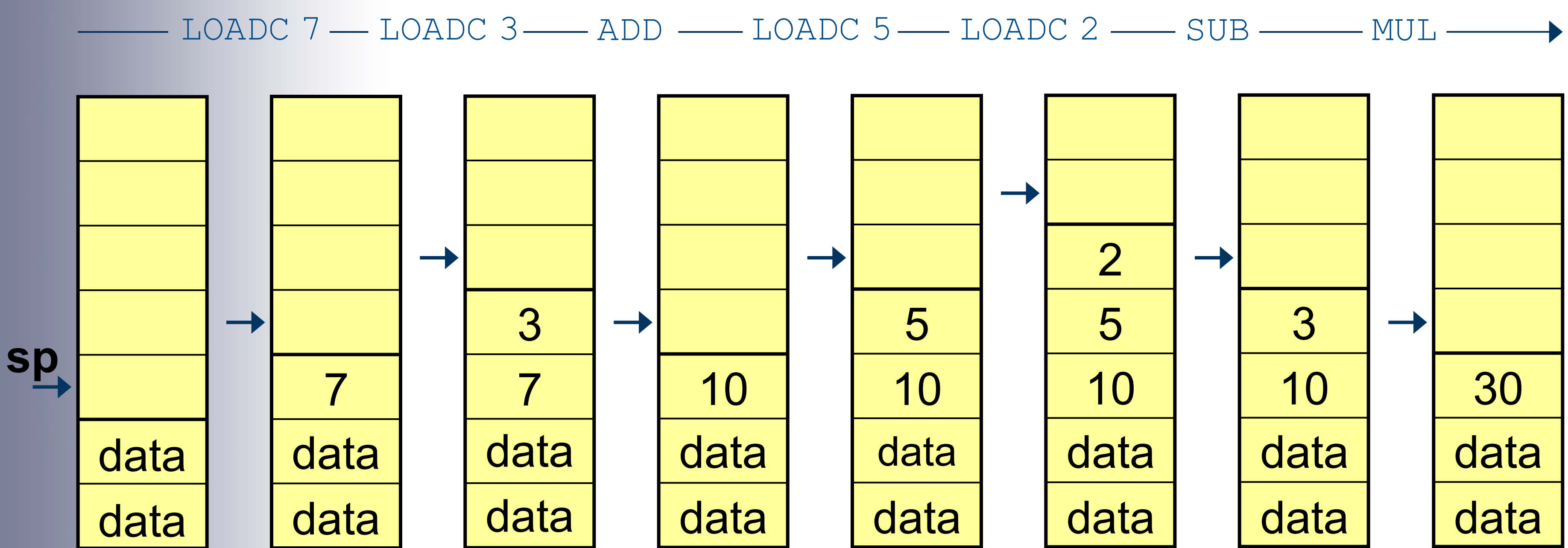
Case study: SVM *(7)*

- SVM instruction set *(continued)*:

Op-code	*Mnemonic*	*Behaviour*
0	`LOADG` ***d***	$w \leftarrow$ word at address d; push w
1	`STOREG` ***d***	pop w; word at address $d \leftarrow w$
4	`LOADC` ***v***	push v
16	`HALT`	status $\leftarrow$ halted
17	`JUMP` ***c***	pc $\leftarrow c$
18	`JUMPF` ***c***	pop w; if w = 0 then pc $\leftarrow c$
19	`JUMPT` ***c***	pop w; if $w \neq 0$ then pc $\leftarrow c$

Case study: SVM *(8)*

- The top of the stack is used for evaluating expressions.
- E.g., evaluating `(7+3)*(5-2)`:



University of Glasgow

Writing an interpreter

- Interpreters are commonly written in C or Java.
- In such an interpreter:
  - the virtual machine *state* is represented by a group of variables
  - each instruction is executed by *inspecting* and/or *updating* the virtual machine state.

Case study: SVM interpreter in Java *(1)*

- Representation of *instructions*:

```
final byte
    LOADG  =  0,   STOREG =  1,
    LOADL  =  2,   STOREL =  3,
    LOADC  =  4,
    ADD    =  6,   SUB    =  7,
    MUL    =  8,   DIV    =  9,
    CMPEQ  = 10,
    CMPLT  = 12,   CMPGT  = 13,
    INV    = 14,   INC    = 14,
    HALT   = 16,   JUMP   = 17,
    JUMPF  = 18,   JUMPT  = 19,
    …;
```

Case study: SVM interpreter in Java *(2)*

- Representation of the virtual machine *state*:

```
byte[] code;                // code store

int[] data;                 // data store

int pc, cl, sp, fp,         // registers
       status;

final byte
    RUNNING = 0,
    FAILED  = 1,
    HALTED  = 2;
```

Case study: SVM interpreter in Java *(3)*

- The interpreter initializes the state, then repeatedly fetches and executes instructions:

```
void interpret () {
    // Initialize the state:
    status = RUNNING;
    sp = 0;  fp = 0;
    pc = 0;
    do {
        // Fetch the next instruction:
        byte opcode = code[pc++];
        // Execute this instruction:
        …
    } while (status == RUNNING);
}
```

Case study: SVM interpreter in Java *(4)*

- To execute an instruction, first inspect its opcode:

```
// Execute this instruction:
switch (opcode) {
    case LOADG: …
    case STOREG: …
    …
    case ADD: …
    case CMPLT: …
    …
    case HALT: …
    case JUMP: …
    case JUMPT: …
    …
}
```

Case study: SVM interpreter in Java *(5)*

- Executing arithmetic/logical instructions:

```
case ADD: {
    int w2 = data[--sp];
    int w1 = data[--sp];
    data[sp++] = w1 + w2;
    break; }

case CMPLT: {
    int w2 = data[--sp];
    int w1 = data[--sp];
    data[sp++] = (w1 < w2 ? 1 : 0);
    break; }
```

Case study: SVM interpreter in Java *(6)*

- Executing load/store instructions:

```
case LOADG: {
    int d = code[pc++]<<8 | code[pc++];
    data[sp++] = data[d];
    break; }

case STOREG: {
    int d = code[pc++]<<8 | code[pc++];
    data[d] = data[--sp];
    break; }
```

fetch 2-byte operand

Case study: SVM interpreter in Java *(7)*

- Executing jump/halt instructions:

```
case HALT: {
    status = HALTED;
    break; }

case JUMP: {
    int c = …;
    pc = c;
    break; }

case JUMPT: {
    int c = …;
    int w = data[--sp];
    if (w != 0) pc = c;
    break; }
```

fetch 2-byte operand