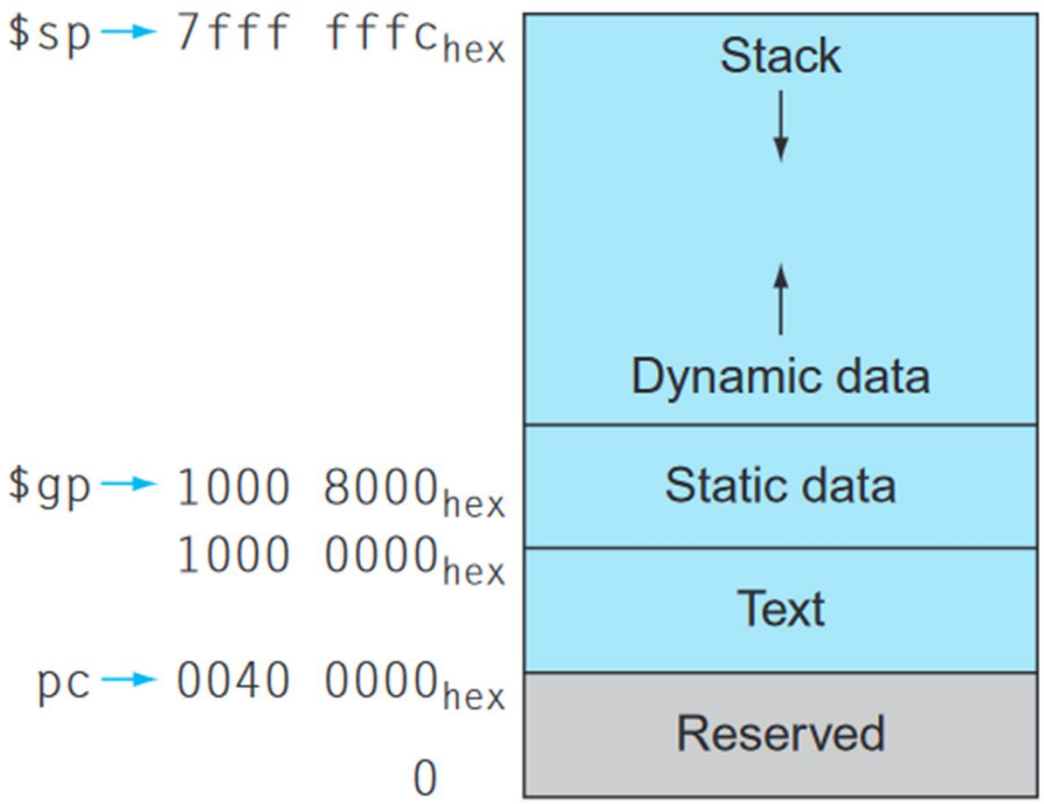
Memory Allocation and Program Translation

S.Venkatesan
Network Security and Cryptography Lab
Department of Information Technology
Indian Institute of Information Technology, Allahabad
venkat@iiita.ac.in

Acknowledgement: The contents and figures are copied from various sources. Thanks to all authors and sources made those contents public and usable for educational purpose

Memory Allocation



Program Counter

- The register containing the address of the instruction in the program being executed.
- Note: ARM instructions are 32 bits long
- Program Counter = Register + Branch Address
- PC-relative addressing – PC + constant in the instruction.

Branch Instruction Encoding

Cond	12	Address
4 bits	4 bits	24 bits

Bits [27:25] identify this as a B or BL instruction – they have values 101 only for these instructions

We can have only 2^24 or 16 MB

To take this to 2^32, we go for PC-relative addressing

Procedures

- Put parameters in a place where the procedure can access them.
- Transfer control to the procedure.
- Acquire the storage resources needed for the procedure.
- Perform the desired task.
- Put the result value in a place where the calling program can access it.
- Return control to the point of origin, since a procedure can be called from several points in a program.

Registers
R0 – R3 for four argument registers
Lr – link register for return address

BL ProcedureAddress

MOV pc, lr

Caller: fill link register (PC+4 in register), parameters value in R0-R3 and jump to procedure with BL
Callee – Compute and place the results into R0 and R1 then return the control to the caller using MOV PC,lr.

BL instruction saves the PC+4 in register lr

Using More Registers

- If there is a need of more registers than four argument and two return value registers then we need to go for the spill registers (register to memory).
- Data structure of spilling is Stack
- Stack Pointer is a register (no.13) that holds the top of the stack. --> uses Push and Pop
- The growth of the stack is in reverse.

Example [with MIPS]

```
int leaf_example (int g, int h, int i, int j)
{
      int f;

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

Assume g, h, i and j are in register $a0, $a1, $a2, $a3 and f in $S0

```
addi $sp, $sp, -12  # adjust stack to make room for 3 items
sw  $t1, 8($sp)     # save register $t1 for use afterwards
sw  $t0, 4($sp)     # save register $t0 for use afterwards
sw  $s0, 0($sp)     # save register $s0 for use afterwards
```

Procedure – Stack Utility

To return the value of f, we copy it into a return value register:

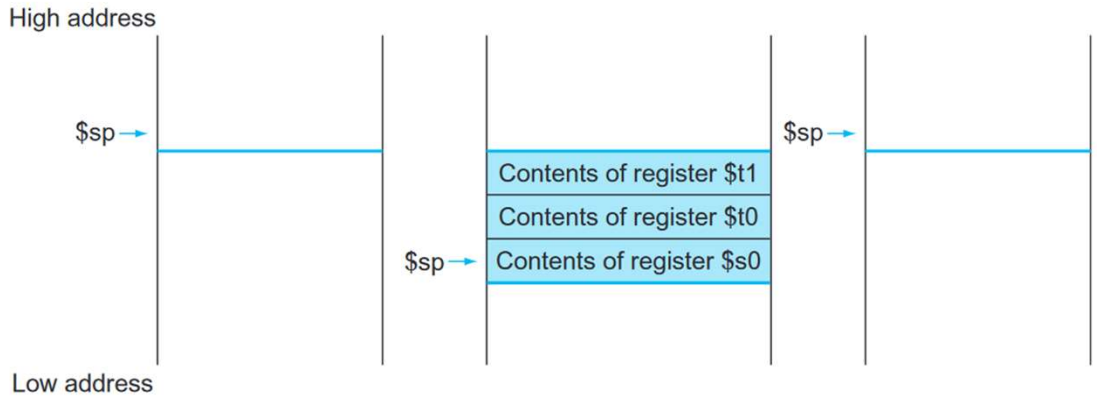
```
add $v0,$s0,$zero # returns f ($v0 = $s0 + 0)
```

Before returning, we restore the three old values of the registers we saved by "popping" them from the stack:

```
lw $s0, 0($sp)  # restore register $s0 for caller
lw $t0, 4($sp)  # restore register $t0 for caller
lw $t1, 8($sp)  # restore register $t1 for caller
addi $sp,$sp,12 # adjust stack to delete 3 items
```

The procedure ends with a jump register using the return address:

```
jr   $ra    # jump back to calling routine
```



Nested Procedures

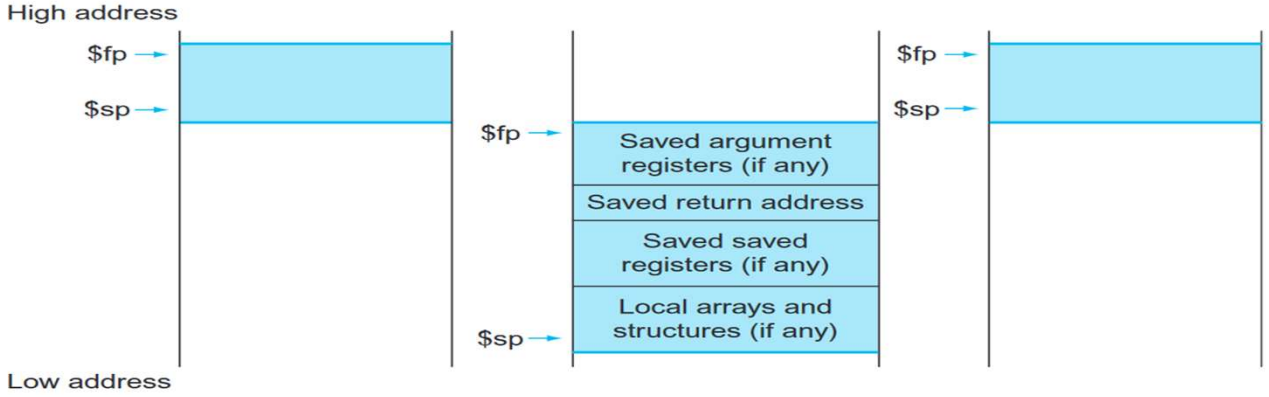
```
int fact (int n)
{
        if (n < 1) return (1);
        else return (n * fact(n – 1));
}
```

Preserved	Not preserved
Saved registers: $s0–$s7	Temporary registers: $t0–$t9
Stack pointer register: $sp	Argument registers: $a0–$a3
Return address register: $ra	Return value registers: $v0–$v1
Stack above the stack pointer	Stack below the stack pointer

Activation Record

- Stack has to store the local array or structures of the procedure but do not fit in registers.

- The segment of stack containing a procedure's saved registers and local variables is called a procedure frame or activation record.



ASCII Code

- American Standard Code for Information Interchange.

- Not efficient for represent because of size. For example, to represent 1 Billion we need 10 ASCII digits and each 8 bits long.

Immediate Operands

- 12 bit is reserved for operand 2.
- In that case, number bigger than that size cannot be accommodated.
- Hence, 12 bit is converted to 8-bit constant field and 4 bit rotate right field.

X * 2^2i

- X is between 0 and 255 and i is between 0 and 15.

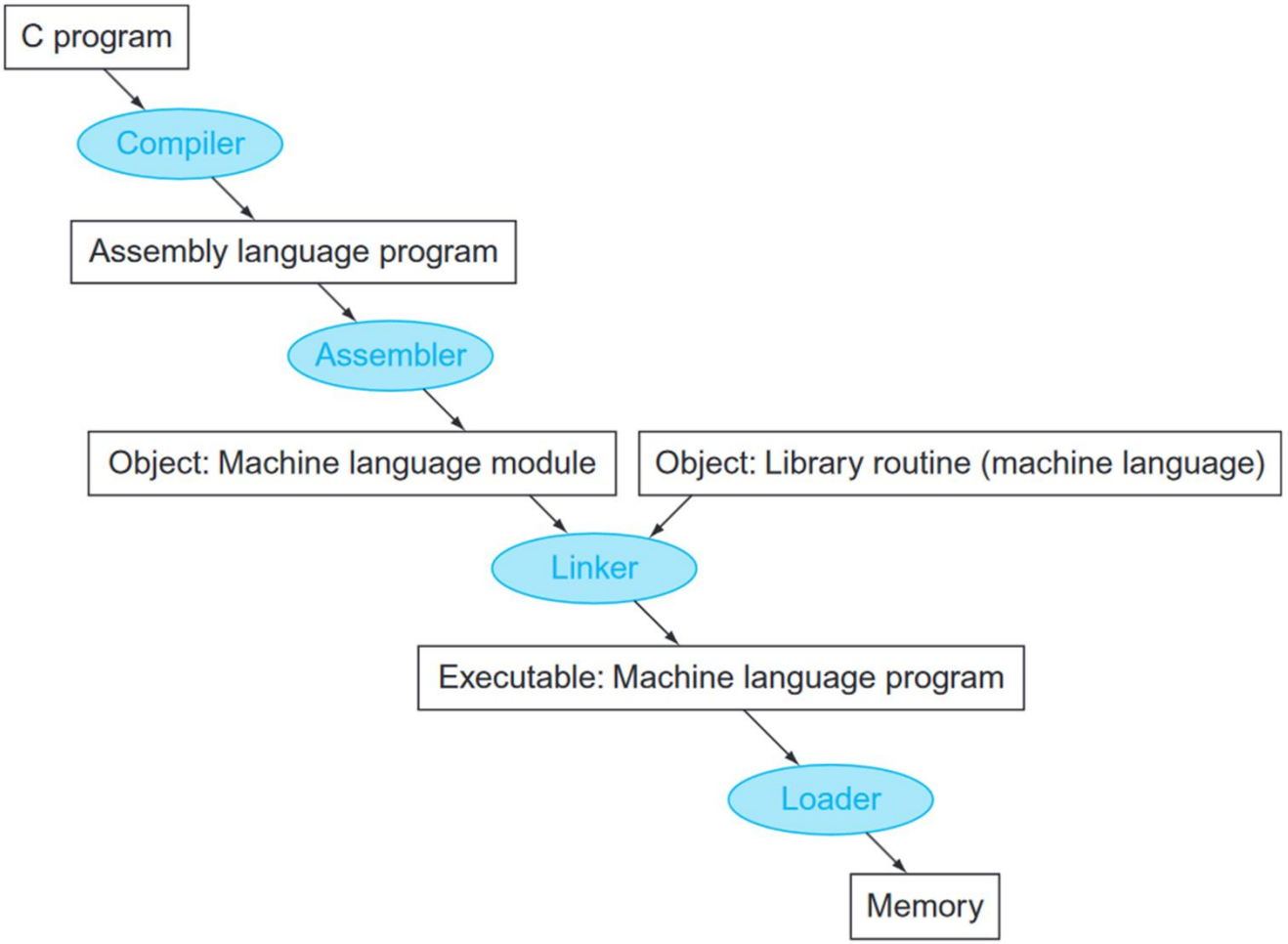
Addressing

- Register offset
- Scaled register offset
- Immediate Pre-Indexed
- Immediate Post-Indexed
- Register Pre-Indexed
- Scaled Register Pre-Indexed
- Register Post-Indexed

Parallelism

- Data Race – Two memory accesses form a data race if they are from different threads to same location, atleast one is a write, and they occur one after another.
- Lock and Unlock.
- Mutual Exclusion.
- Atomicity
- Synchronize (Swap)

Translation Hierarchy



Translators

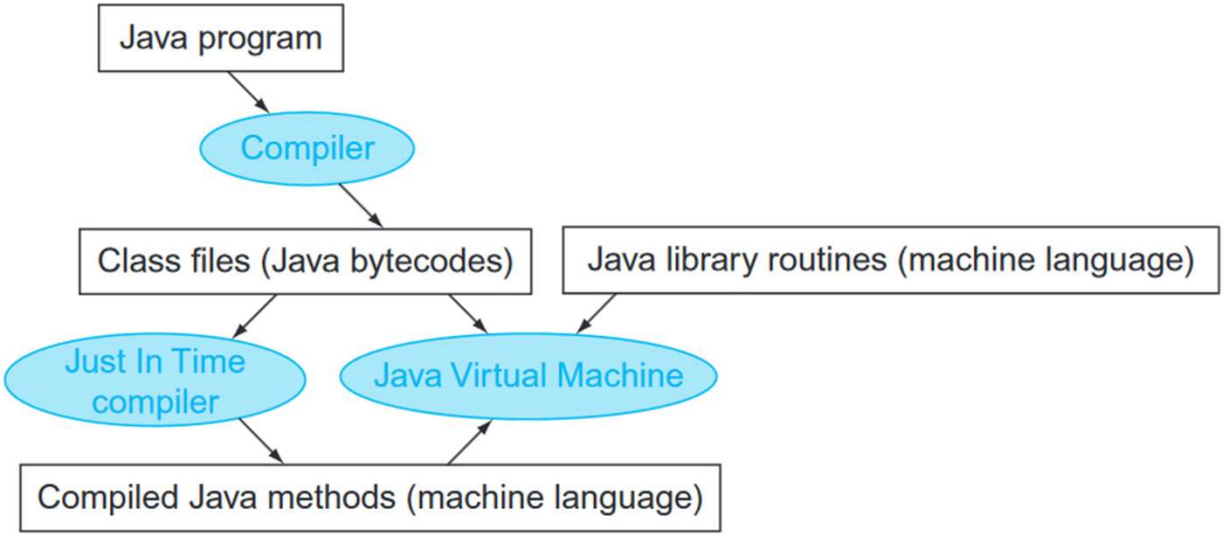
- Assembler – Pseudoinstructions, Symbol Table
- Linker – Executable File
  - Place code and data modules symbolically in memory.
  - Determine the addresses of data and instruction labels.
  - Patch both the internal and external references.
- Loader
  - Reads the executable fi le header to determine size of the text and data segments.
  - Creates an address space large enough for the text and data.
  - Copies the instructions and data from the executable file into memory.
  - Copies the parameters (if any) to the main program onto the stack.
  - Initializes the machine registers and sets the stack pointer to the first free location.
  - Jumps to a start-up routine that copies the parameters into the argument registers and calls the main routine of the program. When the main routine returns, the start-up routine terminates the program with an exit system call.

Linker

Object file header			
	Name	Procedure A	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	Instruction	
	0	`lw $a0, 0($gp)`	
	4	`jal 0`	
	...	...	
Data segment	0	(X)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	`lw`	X
	4	`jal`	B
Symbol table	Label	Address	
	X	–	
	B	–	
Object file header			
	Name	Procedure B	
	Text size	200_{hex}	
	Data size	30_{hex}	
Text segment	Address	Instruction	
	0	`sw $a1, 0($gp)`	
	4	`jal 0`	
	...	...	
Data segment	0	(Y)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	`sw`	Y
	4	`jal`	A
Symbol table	Label	Address	
	Y	–	
	A	–	

Executable file header		
	Text size	300_{hex}
	Data size	50_{hex}
Text segment	Address	Instruction
	$0040\ 0000_{hex}$	`lw $a0,` 8000_{hex}`($gp)`
	$0040\ 0004_{hex}$	`jal 40 0100`$_{hex}$
	...	...
	$0040\ 0100_{hex}$	`sw $a1,` 8020_{hex}`($gp)`
	$0040\ 0104_{hex}$	`jal 40 0000`$_{hex}$
	...	...
Data segment	Address	
	$1000\ 0000_{hex}$	(X)
	...	...
	$1000\ 0020_{hex}$	(Y)
	...	...

Java Translation Hierarchy



Dynamic Linking

- Linked at the time of execution.
- Overcomes the issues of static linking.

Array

array RN 0 ; 1st argument address of array
n RN 1 ; 2nd argument size (of array)
i RN 2 ; local variable i
Zero RN 3 ; temporary to hold constant 0

```
clearl(int array[], int size)
{
    int i;
    for (i = 0; i < size; i += 1)
        array[i] = 0;
}
```

MOV i,0 ; i = 0
MOV zero, 0 ; zero = 0

Loop1 : STR zero, [array,i, LSL #2] ; array[i] = 0

This instruction is the end of the body of the loop, so the next step is to increment i:

ADD, i, i, # 1

CMP i, size ; i < size
BLT Loops ; if (i < size) go to loop1

Pointer

array RN 0 ; 1st argument address of array
n RN 1 ; 2nd argument size (of array)
p RN 2 ; local variable i
Zero RN 3 ; temporary to hold constant 0
arraySize RN12 ; address of array[size]

MOV p, array ; p = address of array[0]
MOV Zero, #0 ; zero = 0
Loop2: STR zero, [p], #4 ; Memory[p] = 0; p = p + 4
ADD arraySize, array, size, LSL # 2 ; arraySize = address of array[size]
CMP p, arraySize ; p < & array[size]
BLT Loop2 ; if (p<&array[size]) go to loop2

```
clear2(int *array, int size)
{
    int *p;
    for (p = &array[0]; p < &array[size]; p = p + 1)
        *p = 0;
}
```

Array Vs Pointer

MOV i,0 ; i = 0

MOV zero, 0 ; zero = 0

Loop1 : STR zero, [array,i, LSL #2] ; array[i] = 0

ADD, i, i, # 1

CMP i, size ; i < size

BLT Loops ; if (i < size) go to loop1

MOV p, array ; p = address of array[0]

MOV Zero, #0 ; zero = 0

Loop2: STR zero, [p], #4 ; Memory[p] = 0; p = p + 4

ADD arraySize, array, size, LSL # 2 ; arraySize = address of array[size]

CMP p, arraySize ; p < & array[size]

BLT Loop2 ; if (p<&array[size]) go to loop2

Reference

- Computer Organization and Design (ARM edition) - The Hardware and Software Interface by David A. Patterson and John L. Hennessy