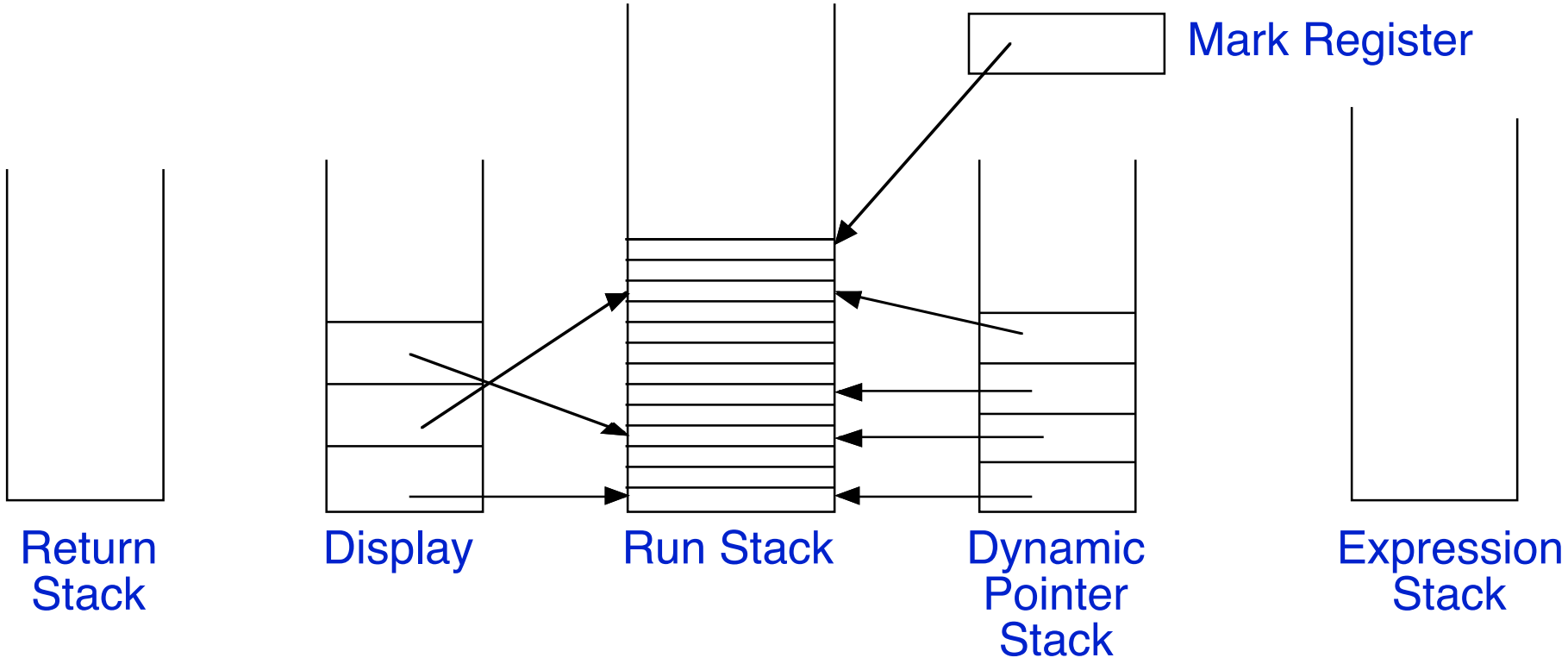
Implementing the Run Time Model

- Before we can proceed to the final stages of compilation, storage allocation and code generation, we need to consider some general questions of representation of the Run Time Model on real machines
- The basic problem will be designing efficient representations of each of the model's structures using real machine resources such as registers, memory management facilities and built-in stacks of the target machine architecture
- The choices we make here will determine, to a large extent, how efficient the code we generate will be in its use of memory and CPU time

Recall ...

- Run Time Model – 5 stacks
  - Expression Stack – expression evaluation
  - Run Stack – storage allocation
  - Display – scope management
  - Dynamic Pointer Stack – restoring Run Stack and Display
  - Return Stack – remember program counter for call/return



Implementation of the Run Time Model

- Depending on the language we are compiling, we may have more or less freedom in our choices
- Some languages, like C and C++, have features that are very close to the hardware, such as variables at absolute addresses, register variables, and so on, that severely limit our choices
- Example:

```
int x = 1;
char *p = (char *)&x;
if (*p == 0){
      printf("big endian");
} else {
      printf("little endian");
}
```

byte address	0	1	2	3		3	2	1	0
byte value	0	0	0	1		0	0	0	1

- But most modern languages, like Turing, Java, Modula 3, Python, Perl, and so on, have a semantics that is intended to be machine independent, and leave us a lot of freedom in our choices

Run Time Model - Machine Dependencies

- One of the main purposes of the standard compiler structure is to isolate target machine dependencies to the back end
- There are two main phases of the back end:
  - *Storage Allocation*, which determines the representation of variables and data in the target machine, and
  - *Code Generation*, which chooses the instruction sequences to implement computations and statements
- In PT, Storage Allocation is simple since it allocates all variables as static (i.e., as if they were all in the global scope) - so it is done as part of Semantic Analysis
- This introduces a slight machine dependency in the PT Semantic phase - it needs to know the sizes of integers, characters and addresses in the memory of the target machine

Run Time Model - Machine Architectures

- Before we continue with the issues in the text, we'll do a quick review of some of the characteristics of the kinds of target machines we might be compiling for
- Given the recent prevalence of the Intel x86 and related architectures as general purpose computers, it's easy to forget the range of other possible target computer structures - but many other machines are in active use that we must compile for
- Examples:
  - MS Xenon / IBM PowerPC (Xbox 360),
  - ARM Snapdragon / Apple A8
    (almost all smartphones, automobiles, appliances, etc.),
  - IBM Cell (Sony Playstation 3)
  - IBM Z mainframe (every business enterprise in the world)
- Some main differences to consider are:
  - stack vs. register architecture
  - addressing modes
  - CISC vs. RISC instruction set
  - general purpose vs. special purpose vs. windowed register sets

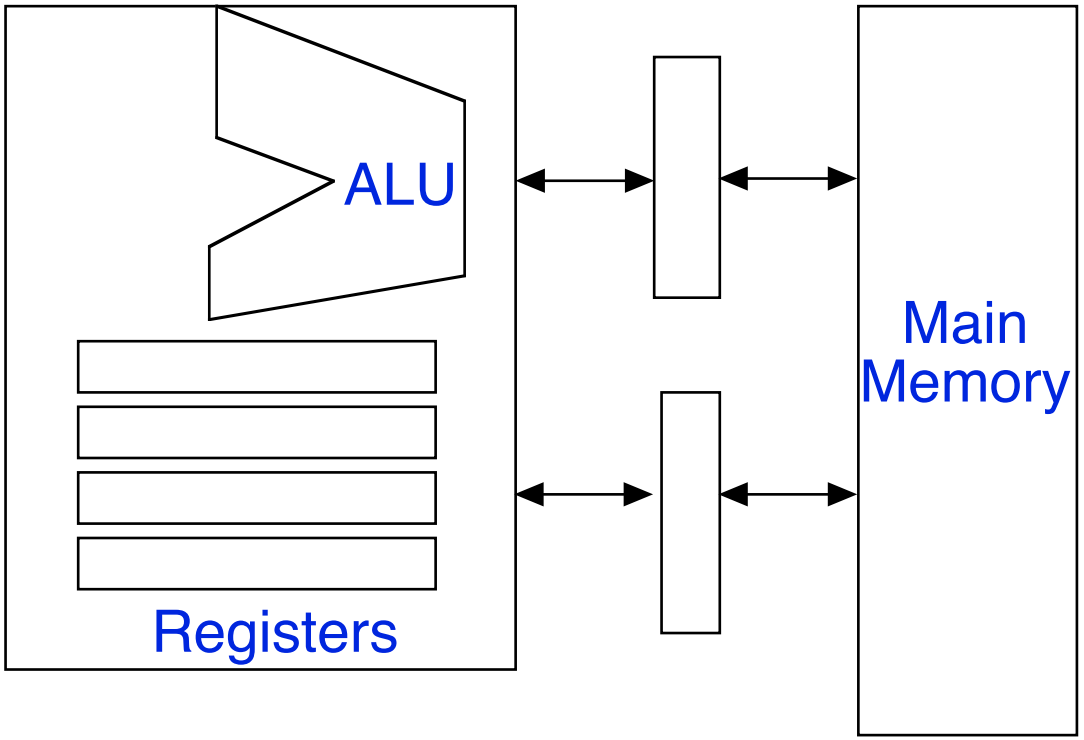
Stack vs Register Machines

- The first distinction is between stack-based and register-based machines
- In Stack Based machines (e.g., Burroughs B5500, Lilith M Machine, Symbolics Lisp Computers, Forth and Postscript machines) the architecture is based entirely on a fast hardware stack to hold all memory and data - all instructions act directly on the elements on top of the stack (no registers!)
- In this sense they are very similar to the PT virtual machine, but implemented directly in hardware
- These were very popular in the early 1980's, when the speed of processors was not so different from the speed of memory, and are enjoying a bit of a resurgence in embedded systems as large, fast static caches make them faster again
- The first computer on the Hubble space telescope was a Forth machine, and Postscript machines are common in printers

Register Based Machines

- Register Based machines are now much more common - in this model, there is a limited amount of very fast memory located in the CPU, specially addressed called registers

- It is the compiler's responsibility to manage register use such that data used in instructions is in registers when needed, and such that frequently accessed data stays there - modern static caches help with this, but don't solve the whole problem

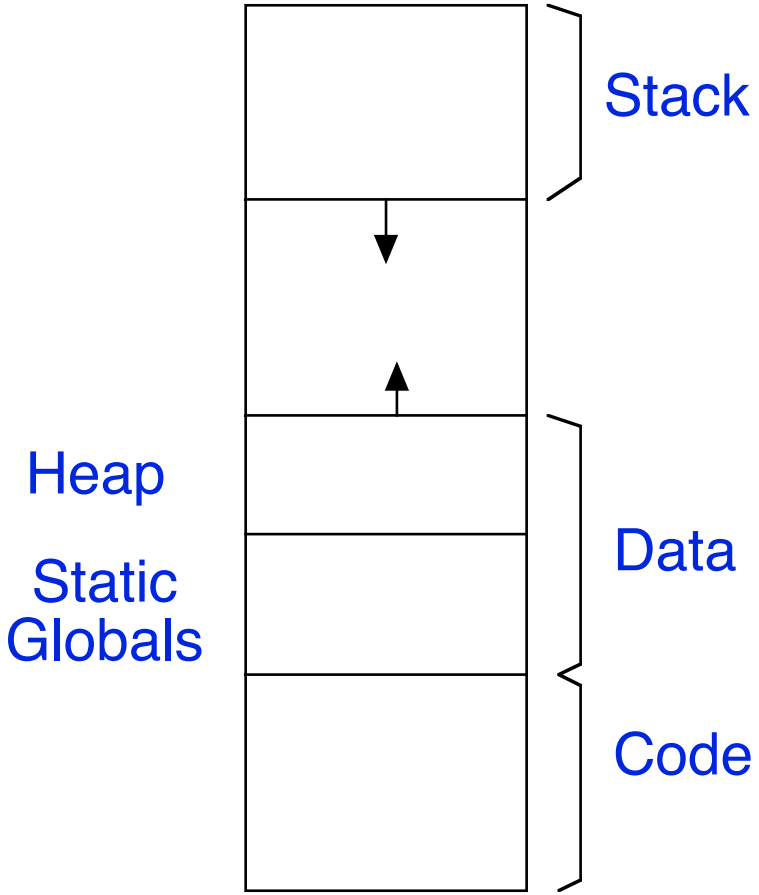


Memory in a Register Machine

- Register machines typically treat memory as a single linear array of bytes - however, some machines distinguish more than one address space, for example instructions may have a different address space than data, which makes the machine look like two separate memories to the compiler

- In a concession to the stack architecture, modern register machines typically support a memory-based hardware stack as well - using special addressing modes and dedicated registers

- Thus the compiler has the choice on these machines of what to represent in registers and what to represent using the hardware stack



Addressing Modes

- Data in memory may be accessed in various ways, called addressing modes - depending on the style of machine, there may be few or many, and they may be simple or complex

- Examples:

Immediate - value of the operand is stored in the instruction
Register - the operand is in a register e.g., move #3,r1
Static - absolute address of the operand is in the instruction
PC Relative - the operand address relative to the instruction address is in the instruction
Register Indirect - the address of the operand is in a register e.g., move (r1),r2
Base Register + Offset - a base address is in a register, a constant offset of the operand is added e.g., move 4(r1),r2
Base Register + Offset + Index - a base address is in a register, a constant offset is added and the contents of another register (the index) is added, scaled by the operand size (subscripting)

Addressing Modes

- Any given machine will have a subset of all possible addressing modes - e.g., RISC machines are typically load-store machines, with only simple modes to load or store from registers
- On these machines operands (with a few small exceptions) must be loaded from memory into registers before they can be used
- Some instructions on some machines require the use of particular registers - a common approach is to have some registers specifically for use as arithmetic operands (*data registers*) and some reserved for use as bases and indexes (*address registers*)
- Other restrictions may apply - e.g., on the IBM Z series (a "mainframe" computer), offsets from a base address are limited to 4096 bytes, and most machines have limits on the ranges of immediate values - e.g., a constant operand on the Sun SPARC is limited to 13 bits
- The compiler must enforce and adapt to these limitations in the code it generates, while still implementing the semantics of the programming language

Example Architecture - IBM 360 "Mainframe"

- Now called the IBM "Z-series", one of the oldest still operating architectures - and one of the most prevalent (you don't see them much, but you use several every day)

  - 16 general purpose registers
  - address offset limited to 12 bits (0 to 4095)
  - limited immediate operands, constants normally stored in static memory locations and accessed as constant variables
  - addressing modes: Register, Base+Offset, Base+Offset+Index
  - multi-byte decimal operands - up to 18 digits long

Example Architecture - PDP-11

- The machine that launched UNIX, C, and the model for many modern computers
  - 8 general purpose registers, each 16 bits
    one reserved as stack pointer (SP),
    one for program counter (PC)
  - 8 general addressing modes: Register, Register Indirect, Indexed, Indexed Indirect, AutoIncrement/Decrement (for push/pop), AutoIncrement/Decrement Indirect
- The autoincrement/autodecrement modes are where C gets its ++ and -- operations from
- A very general and powerful set of addressing modes, allowing many nonobvious operand paradigms
  (e.g., PC autoincrement allows immediate operand values directly in the code)
- Similar machines - NS32000, VAX, MC68000 (original Mac, Palm, cellphones)

Example Architecture - x86

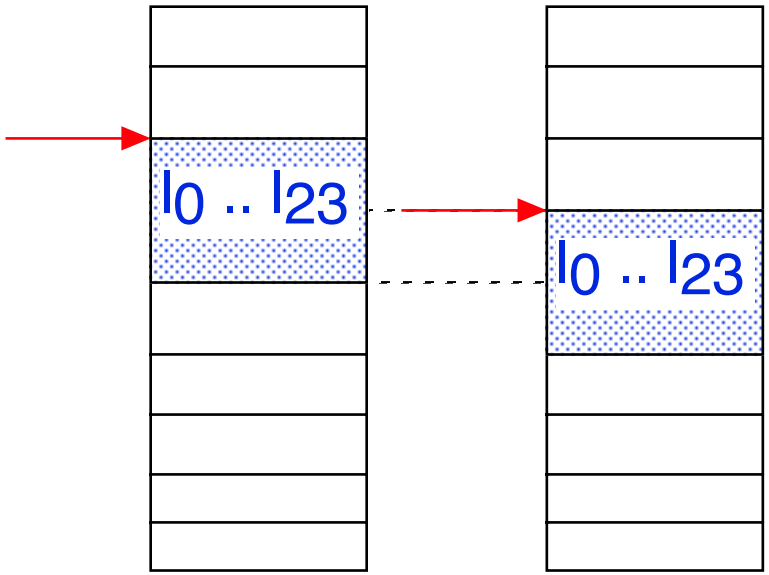
- Among the most common machines today
  - 4 general purpose registers (A, B, C, D), 4 address registers one of which is the stack pointer (SI, DI, BP, SP)
  - one operand of every instruction must be a register
  - A,B,C,D registers are accumulators, C,D counters, some instructions require particular registers or pairs
  - special index registers
  - addressing modes: Register, Immediate, Static, Base, Base+Offset, Index + Offset, Scaled Index + Offset, Base + Index, Base +Index +Offset, Base + Offset + Scaled Index, Base + Scaled Index
- Later versions (starting with i386) have generalized the registers, but introduced a performance penalty for using them incorrectly

Fixed Register Set RISC Architectures

- Modern computers are increasingly RISC (Reduced Instruction Set Computer) architectures (e.g, MIPS, Sun Sparc, DEC Alpha, IBM PowerPC, ARM Snapdragon, Intel Itanium, IBM Cell)
- There are two general classes of RISC machines - fixed register set and windowed register set
- All RISC machines are characterized by
  - a large set (at least 32, and up to about 1,024) of general purpose registers
  - load/store architecture - arithmetic operations accept only registers as operands
  - limited set of addressing modes, often only two, Register +Base and Indirect Register + Base - all other modes must be programmed using these
- In a fixed register set RISC computer, all registers are global and must be managed by the compiler globally

Windowed Register Set RISC Architectures

- Windowed register set RISC computers (e.g., Sun Sparc, Pyramid) also have a large number of registers, but only a subset of them can be used at a time
- Some of the registers (the global registers, typically 8) are usable everywhere - the rest are arranged into a register file
- A special internal register (not program controllable) is a pointer to the current *window* in the register file
- The window gives access to a number of registers (typically 24) that can currently be used
- On procedure call, the pointer is advanced 16 registers, providing an 8 register overlap, allowing fast parameter passing



Summary

- Two tasks left - storage allocation and code generation
- In order to allocate storage and generate code for a target machine, we need to choose representations of the abstract machine structures - this depends on the target machine
- Choice of representation is the single biggest factor in determining the quality of code we can generate
- There are several classes of target machines, for which we need to make different representation decisions - stack machines, register machines, RISC machines, fixed vs windowed registers
- **Next**:

  Representation of the ES, RS and Display on different classes of machines