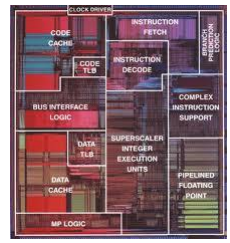
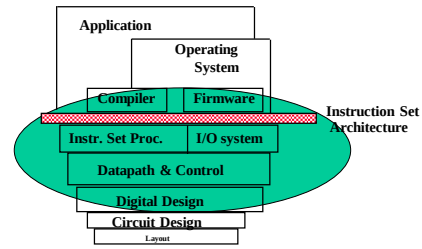
UTD

CS/SE 3340
Computer Architecture





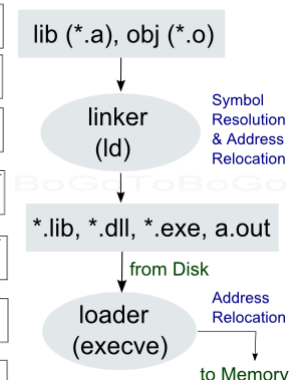
Addressing Modes & Linker

Adapted from slides by Profs. D. Patterson and J. Hennessey

8086 ADDRESS MODES

TYPE	INSTRUCTION	SOURCE	ADDRESS GENERATION	DESTINATION
1) REGISTER	MOV AX, BX	REGISTER BX		REGISTER AX
2) IMMEDIATE	MOV CH,3AH	DATA 3AH		REGISTER CH
3) DIRECT	MOV [1234], AX	REGISTER AX	(DS x 10H) + DISPLACEMENT 10000H + 1234	MEMORY 11234H
4) REGISTER INDIRECT	MOV [BX], CL	REGISTER CL	(DS x10H) + BX 10000H + 0300H	MEMORY 10300H
5) BASE PLUS INDEX	MOV [BX + SI], BP	REGISTER BP	(DS x 10H) + BX + SI 10000H + 0300H + 0200H	MEMORY 10500H
6) REGISTER RELATIVE	MOV CL, [BX +4]	MEMORY 10304H	(DS x 10H) + BX + 4 10000H + 0300H + 4	REGISTER CL
7) BASE RELATIVE PLUS INDEX	MOV ARRAY [BX + SI], DX	REGISTER DX	(DS x 10H) + ARRAY + BX + SI 10000H+1000H+0300H+0200H	MEMORY 11500H

ASSUME BX = 0300H, SI = 0200H, ARRAY = 1000H, DS = 1000H



2

Addressing Modes

- An instruction encodes data manipulation: **op code** and **operands**
- How does the processor get **operands** to operate on?
  - Addressing mode specifies how the processor interprets operand fields to get **operands**
- An operand field of an instruction either contains the actual operand value (immediate) or a reference to the operand location
  - In the register file (register #) or in the main memory (memory address)

3

Addressing Mode – cont'd

- The addressing mode of the operand field determines the ultimate value of the operand
- Common addressing modes in modern processors
  - *Immediate*
  - *Register*
  - *Direct*
  - *Indirect*
  - *Register indirect*
  - *Displacement*
- MIPS supports only a subset of these addressing modes

4

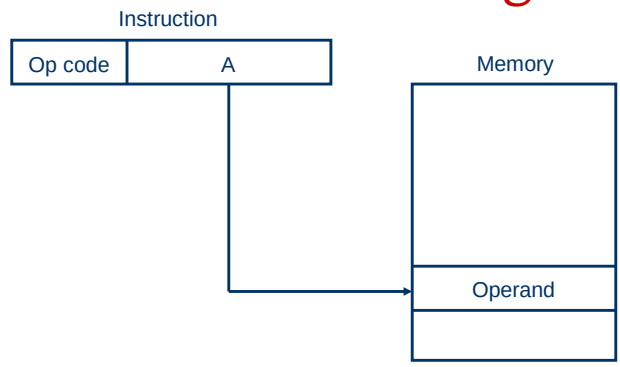
Addressing Modes - Notations

Typical notations used during discussion of addressing modes

- A — content of an operand field in the instruction that refers to a memory address
- R — content of an operand field in the instruction that refers to a register in the register file
- (X) — content of memory location X or register X
- EA — Effective Address of the location containing the referenced operand

5

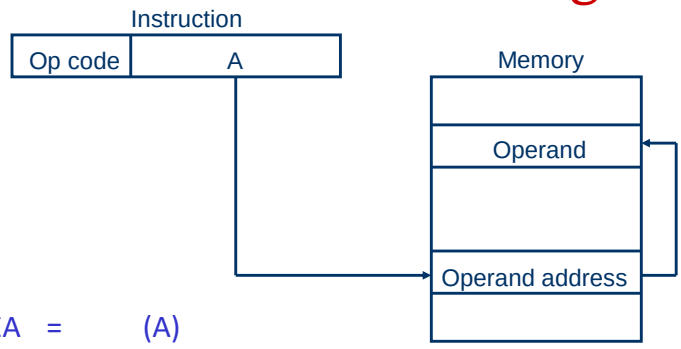
Direct Addressing



- EA = A
- Operand is the content of memory at address A
- Example: **`add [20CE40FE], edx`**
- What is the main performance disadvantage of this mode? Why old ISAs (e.g. x86) has this mode?

6

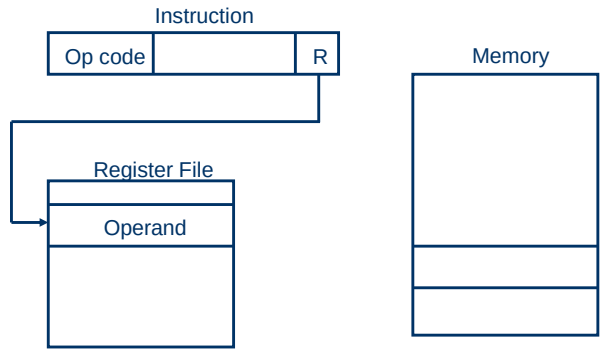
Indirect Addressing



- EA = (A)
- Content of memory at the address A is the effective address (EA) of the operand
- Useful for pointers manipulation
- Example: `call [01FE2CD0]`
- Note: same main disadvantage as direct addressing mode
- MIPS does not support this addressing mode

7

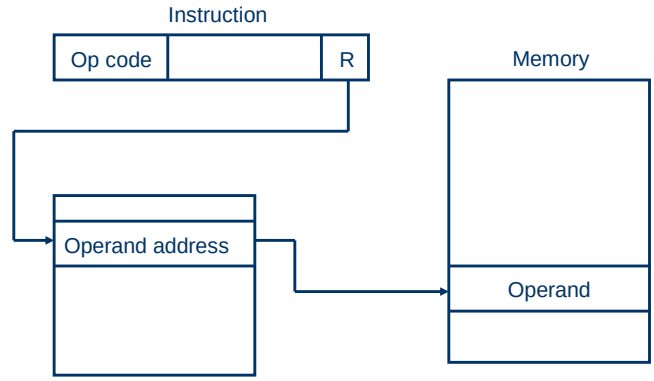
Register Addressing



- EA = R (register number)
- Content of register R is the operand
- Example: `add $t0, $s1, $s2`
- Performance advantage?

8

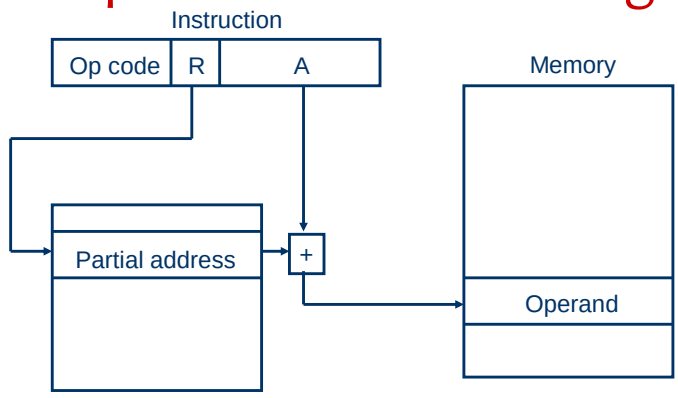
Register Indirect Addressing



- EA = (R)
- Register R contains the effective address of operand
- Example: `lw $s0, ($t1)`
- Require data memory access to get the effective address?

9

Displacement Addressing



- EA = A + (R)
- Effective address of the operand is calculated from a base (R) and a displacement A
- There are several variations of displacement addressing

10

Displacement Addressing – Base Addressing

- R contains the base address, A is the displacement from the base address
- Useful to access data of a structure (e.g. struct, object, array,…)
- e.g. `lb $t0, 3($t2)`
  - What is base and what is displacement in this example?
- What if the displacement is not a constant?

11

Displacement Addressing – Relative Addressing

- Effective address is relative to the content of R
- MIPS's hardware branching instructions (**`bne, beq`**) uses the addressing mode
  - Effective address is relative to the PC (PC is implicitly used as R)
  - e.g. `bne $t0, $zero, else` (else is assembled into an immediate value relative to the PC!)

12

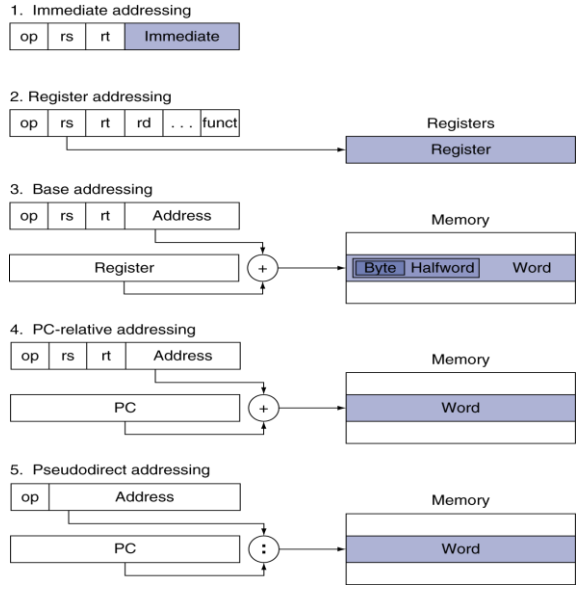
Displacement Addressing – Indexed

- **A** can be a fixed base address and **R** can be used as an *index*
  - Useful to access data in an array
- Example:

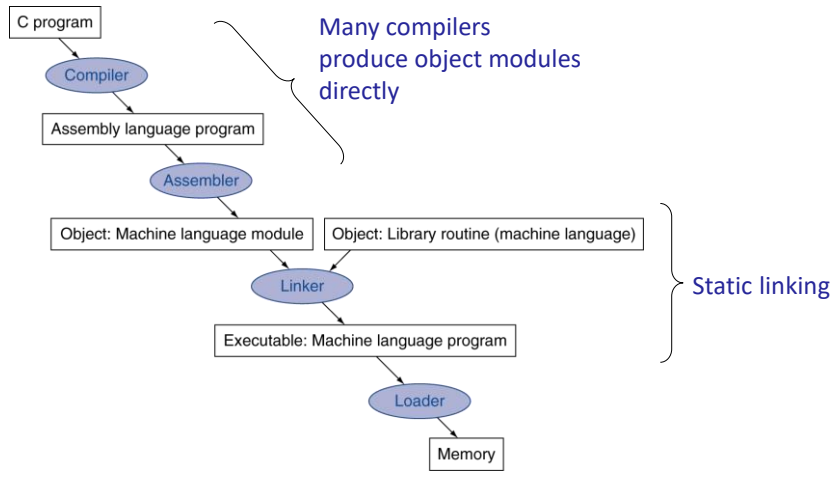
```
myarray:  .word 10, 20, 30, 40, 50, 60, 70
          ...
          li   $t1, 3
          sll  $t1, $t1, 2
          lw   $s0, myarray($t1)
```

- What is loaded into $s0?

MIPS Addressing Modes Summary



Translation and Startup



15

Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
  - *Header*: described contents of object module
  - *Text segment*: translated instructions
  - *Static data segment*: data allocated for the life of the program
  - *Relocation info*: for contents that depend on absolute location of loaded program
  - *Symbol table*: global definitions and external refs
  - *Debug info*: for associating with source code

16

Linking Object Modules

- Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external refs
- Could leave location dependencies for fixing by a relocating loader
  - But with virtual memory, no need to do this
  - Program can be loaded into absolute location in virtual memory space

17

Loading a Program

- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space
  3. Copy text and initialized data into memory
     - Or set page table entries so they can be faulted in
  4. Set up arguments on stack
  5. Initialize registers (including `$sp, $fp, $gp`)
  6. Jump to startup routine
     - Copies arguments to `$a0,` … and calls main
     - When main returns, do exit `syscall`

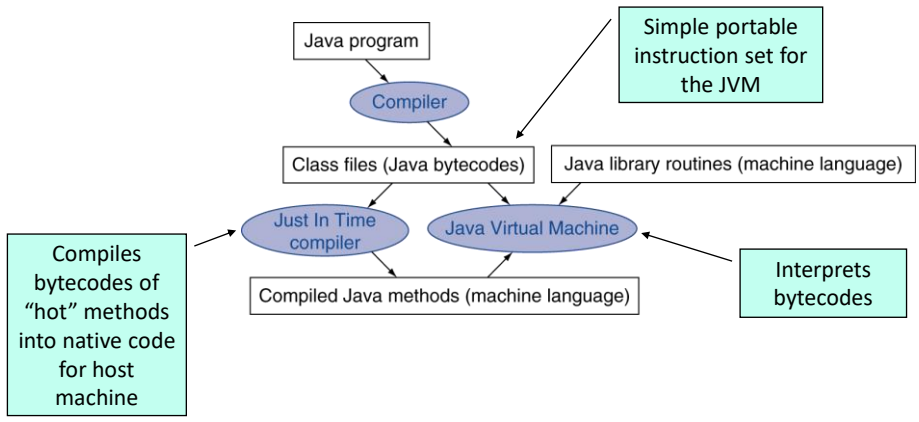
18

Dynamic Linking

- Only link/load library procedure when it is called
  - Requires procedure code to be relocatable
  - Avoids image bloat caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions

19

Java Applications



20

C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function
- Swap procedure (leaf)

```
void swap(int v[], int k)
{
  int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

  - v in $a0, k in $a1, temp in $t0

The Procedure Swap

```
swap: sll $t1, $a1, 2    # $t1 = k * 4
      add $t1, $a0, $t1  # $t1 = v+(k*4)
                         #   (address of v[k])
      lw $t0, 0($t1)     # $t0 (temp) = v[k]
      lw $t2, 4($t1)     # $t2 = v[k+1]
      sw $t2, 0($t1)     # v[k] = $t2 (v[k+1])
      sw $t0, 4($t1)     # v[k+1] = $t0 (temp)
      jr $ra             # return to calling routine
```

The Sort Procedure in C

- Non-leaf (calls swap)

```
void sort (int v[], int n)
{
  int i, j;
  for (i = 0; i < n; i += 1) {
    for (j = i - 1;
         j >= 0 && v[j] > v[j + 1];
         j -= 1) {
      swap(v,j);
    }
  }
}
```

  - v in $a0, k in $a1, i in $s0, j in $s1

23

The Procedure Body

```
        move $s2, $a0           # save $a0 into $s2
        move $s3, $a1           # save $a1 into $s3
        move $s0, $zero         # i = 0
for1tst: slt  $t0, $s0, $s3     # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
        beq  $t0, $zero, exit1  # go to exit1 if $s0 ≥ $s3 (i ≥ n)
        addi $s1, $s0, -1       # j = i - 1
for2tst: slti $t0, $s1, 0       # $t0 = 1 if $s1 < 0 (j < 0)
        bne  $t0, $zero, exit2  # go to exit2 if $s1 < 0 (j < 0)
        sll  $t1, $s1, 2        # $t1 = j * 4
        add  $t2, $s2, $t1      # $t2 = v + (j * 4)
        lw   $t3, 0($t2)        # $t3 = v[j]
        lw   $t4, 4($t2)        # $t4 = v[j + 1]
        slt  $t0, $t4, $t3      # $t0 = 0 if $t4 ≥ $t3
        beq  $t0, $zero, exit2  # go to exit2 if $t4 ≥ $t3
        move $a0, $s2           # 1st param of swap is v (old $a0)
        move $a1, $s1           # 2nd param of swap is j
        jal  swap               # call swap procedure
        addi $s1, $s1, -1       # j -= 1
        j    for2tst            # jump to test of inner loop
exit2:  addi $s0, $s0, 1        # i += 1
        j    for1tst            # jump to test of outer loop
```

Move params

Outer loop

Inner loop

Pass params & call

Inner loop

Outer loop

24

The Full Procedure

```
sort:   addi $sp,$sp, -20      # make room on stack for 5 registers
        sw $ra, 16($sp)        # save $ra on stack
        sw $s3,12($sp)         # save $s3 on stack
        sw $s2, 8($sp)         # save $s2 on stack
        sw $s1, 4($sp)         # save $s1 on stack
        sw $s0, 0($sp)         # save $s0 on stack
        …                      # procedure body
        …
        exit1: lw $s0, 0($sp)  # restore $s0 from stack
        lw $s1, 4($sp)         # restore $s1 from stack
        lw $s2, 8($sp)         # restore $s2 from stack
        lw $s3,12($sp)         # restore $s3 from stack
        lw $ra,16($sp)         # restore $ra from stack
        addi $sp,$sp, 20       # restore stack pointer
        jr $ra                 # return to calling routine
```

25

Summary

- Addressing modes specify how operands data are obtained
  - From the instruction, register file or from memory
- MIPS ISA supports minimal set of addressing modes
- Linker produces an executable by linking several object modules and necessary routines from libraries
- Loader reads executable files from disk to main memory to run a program
- These are typical system software necessary to run programs

26