182.690 RECHNERSTRUKTUREN – INSTRUCTIONS (2)

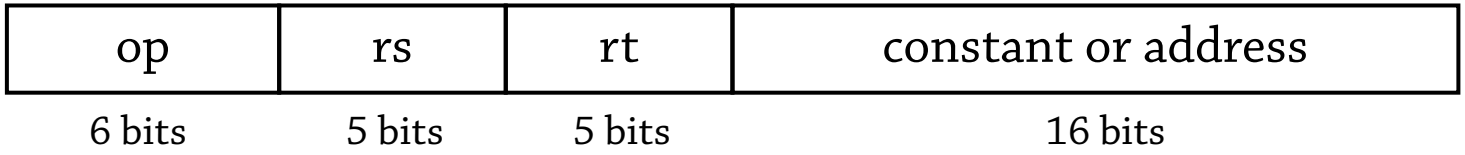
Thomas Polzer
tpolzer@ecs.tuwien.ac.at
Institut für Technische Informatik

Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `beq rs, rt, L1`
  - if (rs == rt) branch to instruction labeled L1
- `bne rs, rt, L1`
  - if (rs != rt) branch to instruction labeled L1
- `j L1`
  - unconditional jump to instruction labeled L1

Branch Addressing

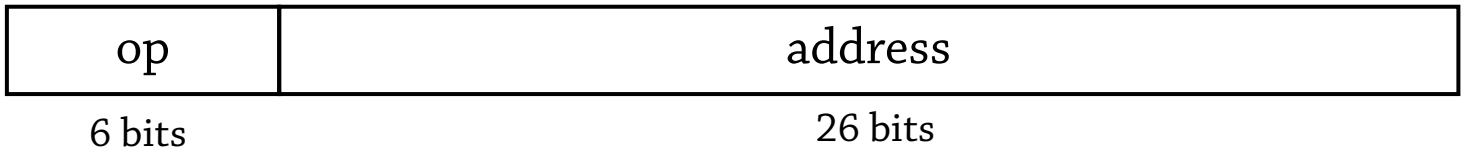
- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward



- PC-relative addressing
  - Target address = PC + 4 + offset × 4
- PC implicitly used
- Limits the branch distance to -2^{15} to $+2^{15}-1$ (word) instructions

Jump Addressing

- Jump (j and jal) targets could be anywhere in text segment
  - Encode full address in instruction



- (Pseudo)Direct jump addressing
  - Target address = PC31…28 : (address × 4)

Compiling If Statements

- C code:

```
if(i == j)
  f = g + h;
else
  f = g - h;
```

Assume that f, g, … in $s0, $s1, …

- Compiled MIPS code:

Assembler calculates addresses

```
      bne $s3, $s4, Else
      add $s0, $s1, $s2
      j   Exit
Else: sub $s0, $s1, $s2
Exit: …
```

Compiling Loop Statements

- C code:

```
while (save[i] == k) i += 1;
```

i in $s3, k in $s5, address of save in $s6

- Compiled MIPS code:

```
Loop:  sll  $t1, $s3, 2
       add  $t1, $t1, $s6
       lw   $t0, 0($t1)
       bne  $t0, $s5, Exit
       addi $s3, $s3, 1
       j    Loop
Exit:  …
```

Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

```
Loop:   sll  $t1, $s3, 2
        add  $t1, $t1, $s6
        lw   $t0, 0($t1)
        bne  $t0, $s5, Exit
        addi $s3, $s3, 1
        j    Loop
Exit:   …
```

80000	0	0	19	9	2	0
80004	0	9	22	9	0	20_h
80008	23_h	9	8	0		
80012	5_h	8	21	2		
80016	8_h	19	19	1		
80020	2_h	20000				
80024						

More Conditional Operations

- Set result to 1 if a condition is true
  - Otherwise, set to 0
- `slt rd, rs, rt`
  - if (rs < rt) rd = 1; else rd = 0;
- `slti rt, rs, constant`
  - if (rs < constant) rt = 1; else rt = 0;
- Use in combination with beq, bne

```
slt $t0, $s1, $s2  # if ($s1 < $s2)
bne $t0, $zero, L  #   branch to L
```

Branch Instruction Design

- Why not blt, bge, …?
- Hardware for <, ≥, … slower than =, ≠
  - Combining with branch involves more work per instruction ➔ requiring a slower clock
  - All instructions penalized!
- beq and bne are the common case
- This is a good design compromise

Signed vs. Unsigned

- Signed comparison: slt, slti
- Unsigned comparison: sltu, sltui
- Example
  - $s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - $s1 = 0000 0000 0000 0000 0000 0000 0000 0001
  - `slt  $t0, $s0, $s1  # signed`
    - –1 < +1 ⇒ $t0 = 1
  - `sltu $t0, $s0, $s1  # unsigned`
    - +4,294,967,295 > +1 ⇒ $t0 = 0

Bounds Check Shortcut

- Treating signed numbers as if they were unsigned gives a low cost way of checking if $0 \leq x < y$ (index out of bounds for arrays)

```
  sltu $t0, $s1, $t2    # $t0 = 0 if
                        # $s1 > $t2 (max)
                        # or $s1 < 0 (min)
 beq $t0, $zero, IOOB # go to IOOB if
                        # $t0 = 0
```

- The key is that negative integers in two's complement look like large numbers in unsigned notation. Thus, an unsigned comparison of $x < y$ also checks if x is negative as well as if x is less than y

Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

```
        beq $s0,$s1, L1
                ↓
        bne $s0,$s1, L2
        j L1
L2:     …
```

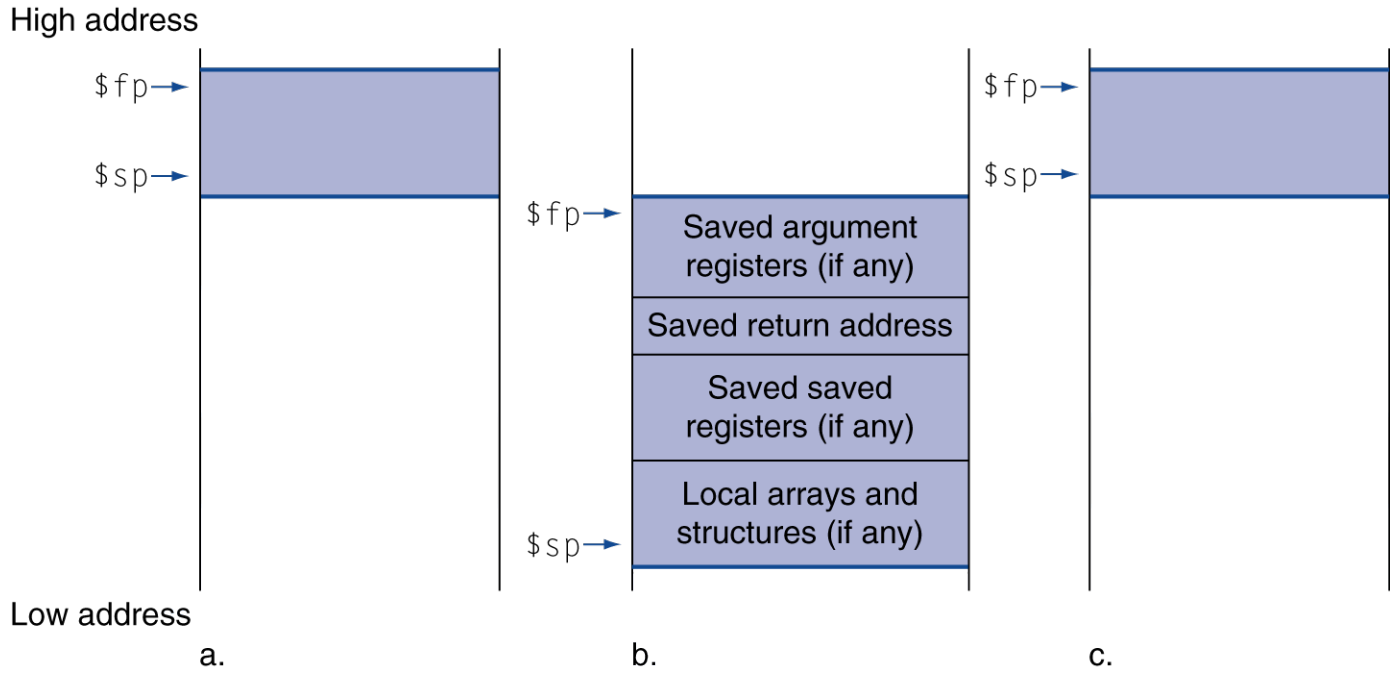
Procedure Calling

- Main routine (caller) places parameters in a place where the procedure (callee) can access them
  - **$a0 - $a3**: four argument registers (convention)
- Caller transfers control to the callee
- Callee acquires the storage resources needed
- Callee performs the desired task
- Callee places the result value in a place where the caller can access it
  - **$v0 - $v1**: two value registers for result values (convention)
- Callee returns control to the caller
  - **$ra**: one return address register to return to the point of origin

Register Usage

- $a0 – $a3: arguments (reg's 4 – 7)
- $v0, $v1: result values (reg's 2 and 3)
- $t0 – $t9: temporaries
  - Can be overwritten by callee
- $s0 – $s7: saved
  - Must be saved/restored by callee
- $gp: global pointer for static data (reg 28)
- $sp: stack pointer (reg 29)
- $fp: frame pointer (reg 30)
- $ra: return address (reg 31)

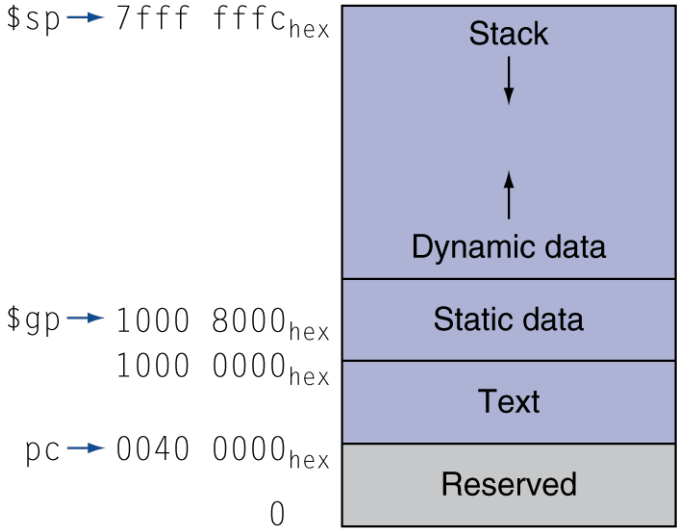
Local Data on the Stack



- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage

Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - e.g., malloc in C, new in C++
- Stack: automatic storage



Procedure Call Instructions

- Procedure call: jump and link

```
jal ProcedureLabel
```

  - Address of following instruction put in **$ra**
  - Jumps to target address
- Procedure return: jump register

```
jr $ra
```

  - Copies **$ra** to program counter

Leaf Procedure Example

- C code:

```
int leaf_example (int g, int h,
                  int i, int j)
{
   int f;
   f = (g + h) - (i + j);
   return f;
}
```

- Arguments g, …, j in **$a0**, …, **$a3**
- f in **$s0** (need to save **$s0** on stack)
- Result in **$v0**

Leaf Procedure Example

- MIPS code:

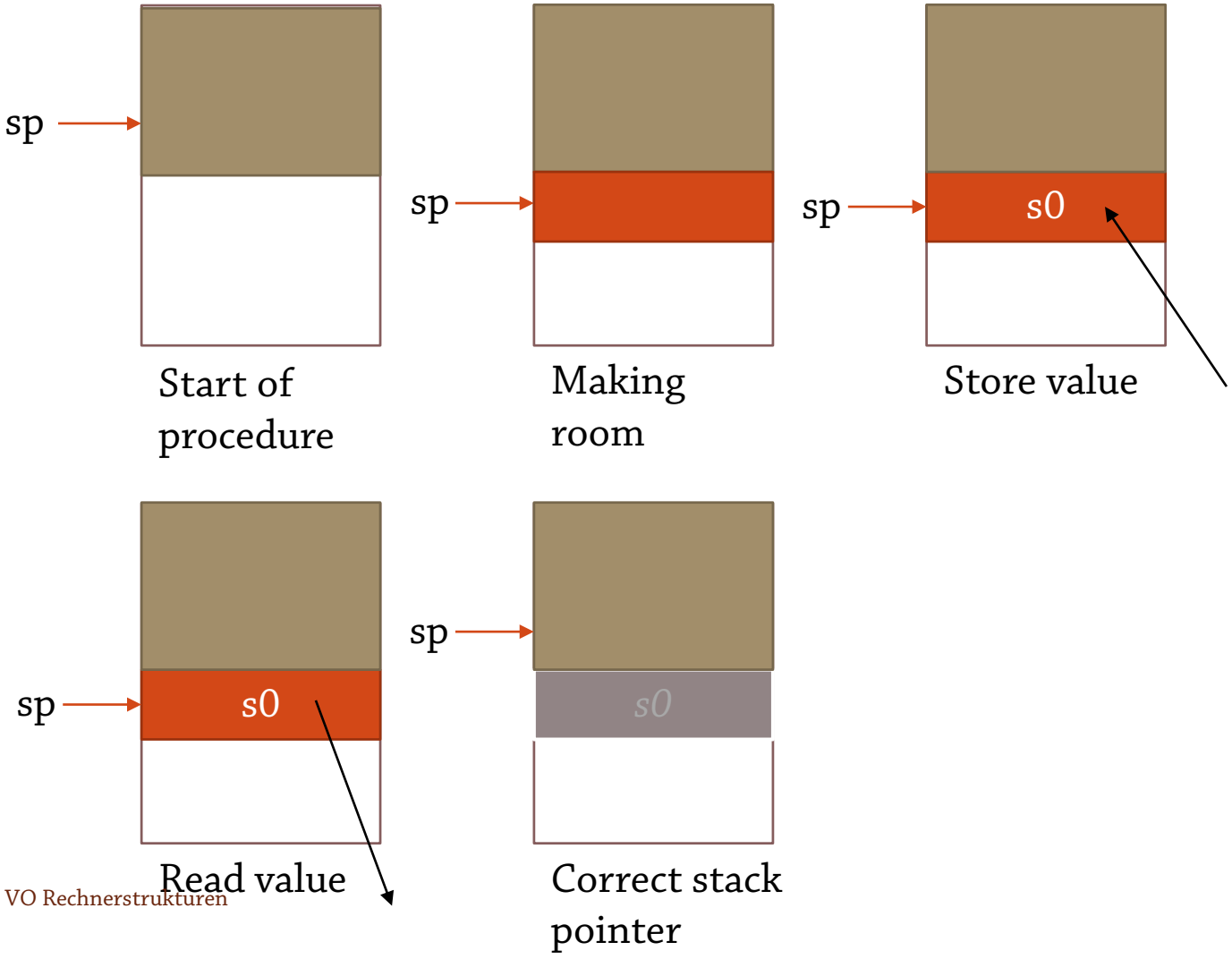
`leaf_example:`	`addi $sp, $sp, -4` `sw $s0, 0($sp)`	Save **$s0** on the stack

	`add $t0, $a0, $a1`
	`add $t1, $a2, $a3`
	`sub $s0, $t0, $t1`
		Procedure body

	`add $v0, $s0, $zero`	Move result to **$v0**
	`lw $s0, 0($sp)` `addi $sp, $sp, 4`	Restore **$s0**
	`jr $ra`	Return

Leaf Procedure Example

- Stack:



Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Non-Leaf Procedure Example

- C code:

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

- Argument n in **$a0**
- Result in **$v0**

Non-Leaf Procedure Example

- MIPS code:

```
1    fact:   addi $sp, $sp, -8
2            sw $ra, 4($sp)
3            sw $a0, 0($sp)
4            slti $t0, $a0, 1
5            beq $t0, $zero, L1
6            addi $v0, $zero0, 1
7            addi $sp, $sp, 8
8            jr $ra
9    L1:     addi $a0, $a0, -1
10           jal fact
11           lw $a0, 0($sp)
12           lw $ra, 4($sp)
13           addi $sp, $sp, 8
14           mul $v0, $a0, $v0
15           jr $ra
```

Save return address (**$ra**) and parameter (**$a0**) onto the stack

If statement

If body, result **$v0** = 1

Correct stack pointer (**$sp**)

Return

Else body, decrement **$a0** (= n)

Recursive call, sets new value to **$ra**

Restore parameter (**$a0**)

Restore parameter (**$a0**)

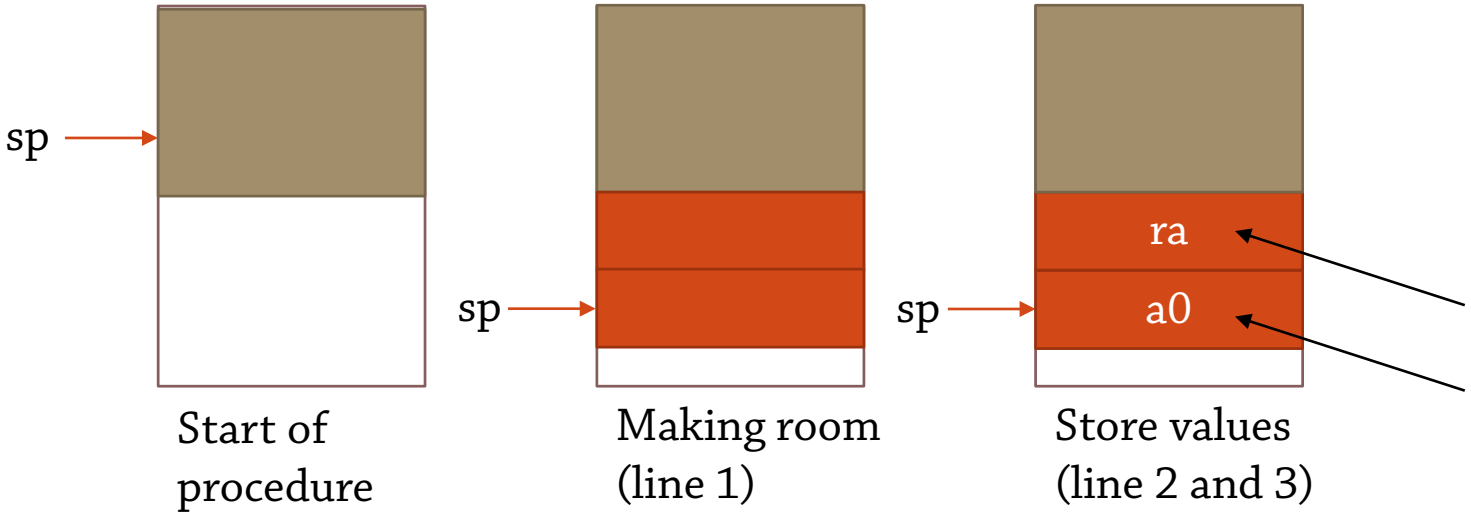
Correct stack pointer (**$sp**)

Multiply to get result **$v0**

Return

Leaf Procedure Example

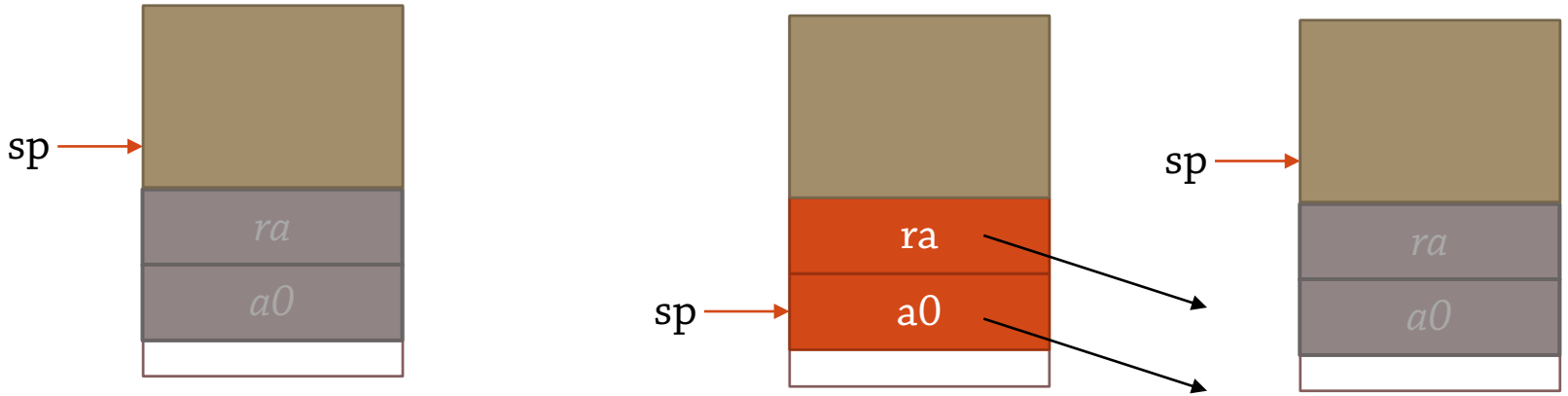
- Stack:



Start of procedure

Making room (line 1)

Store values (line 2 and 3)



Correct stack pointer (line 7)

Read values (line 11 and 12)

Correct stack pointer (line 13)

Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, …
  - Most of the world's alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

Byte/Halfword Operations

- Could use bitwise operations
- MIPS byte/halfword load/store
  - String processing is a common case
- `lb rt, offset(rs)`
  - Sign extend to 32 bits in rt
- `lh rt, offset(rs)`
- `lbu rt, offset(rs)`
  - Zero extend to 32 bits in rt
- `lhu rt, offset(rs)`
- `sb rt, offset(rs)`
- `sh rt, offset(rs)`
  - Store just rightmost byte/halfword

32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant

```
lui rt, constant
```

  - Copies 16-bit constant to left 16 bits of rt
  - Clears right 16 bits of rt to 0

```
lui $s0, 61
```

0000 0000 0011 1101	0000 0000 0000 0000

```
ori $s0, $s0, 2304
```

0000 0000 0011 1101	0000 1001 0000 0000

Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Result depends of order of accesses
- Hardware support required
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write
- Could be a single instruction
  - E.g., atomic swap of register ↔ memory (in one cycle)
  - Or an atomic pair of instructions

Synchronization in MIPS

- Load linked: `ll rt, offset(rs)`
- Store conditional: `sc rt, offset(rs)`
  - Succeeds if location not changed since the ll
    - Returns 1 in rt
  - Fails if location is changed
    - Returns 0 in rt

Synchronization in MIPS (Example)

- Atomic swap (to test/set lock variable)

```
try: add $t0,$zero,$s4 #copy exchange value
     ll  $t1,0($s1)    #load linked
     sc  $t0,0($s1)    #store conditional
     beq $t0,$zero,try #branch store fails
     add $s4,$zero,$t1 #put load value in $s4
```

Assembler Pseudo-Instructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudo-instructions: figments of the assembler's imagination

```
move $t0, $t1    →   add $t0, $zero, $t1

blt $t0, $t1, L  →   slt $at, $t0, $t1
                     bne $at, $zero, L
```

  - **$at** (register 1): assembler temporary

Decoding Machine Code

- Convert into binary representation
- Use opcode field to find instruction format
- Split the instruction into the corresponding fields
- Interpret the fields of the instruction and create the corresponding assembler representation

Decoding Machine Code (Example)

Machine code: $008A4820_h$

0000 0000 1000 1010 0100 1000 0010 0000

0000 00 → Opcode: 0 → R-Type

Opcode	rs	rt	rd	shamt	func
000000	00100	01010	01001	00000	100000
0	4	A_h	9	0	20_h
spec	$a0	$t2	$t1	0	add

→ Assembler instruction: add $t1,$a0,$t2

Decoding Machine Code (Example)

Machine code: $23bd0014_h$

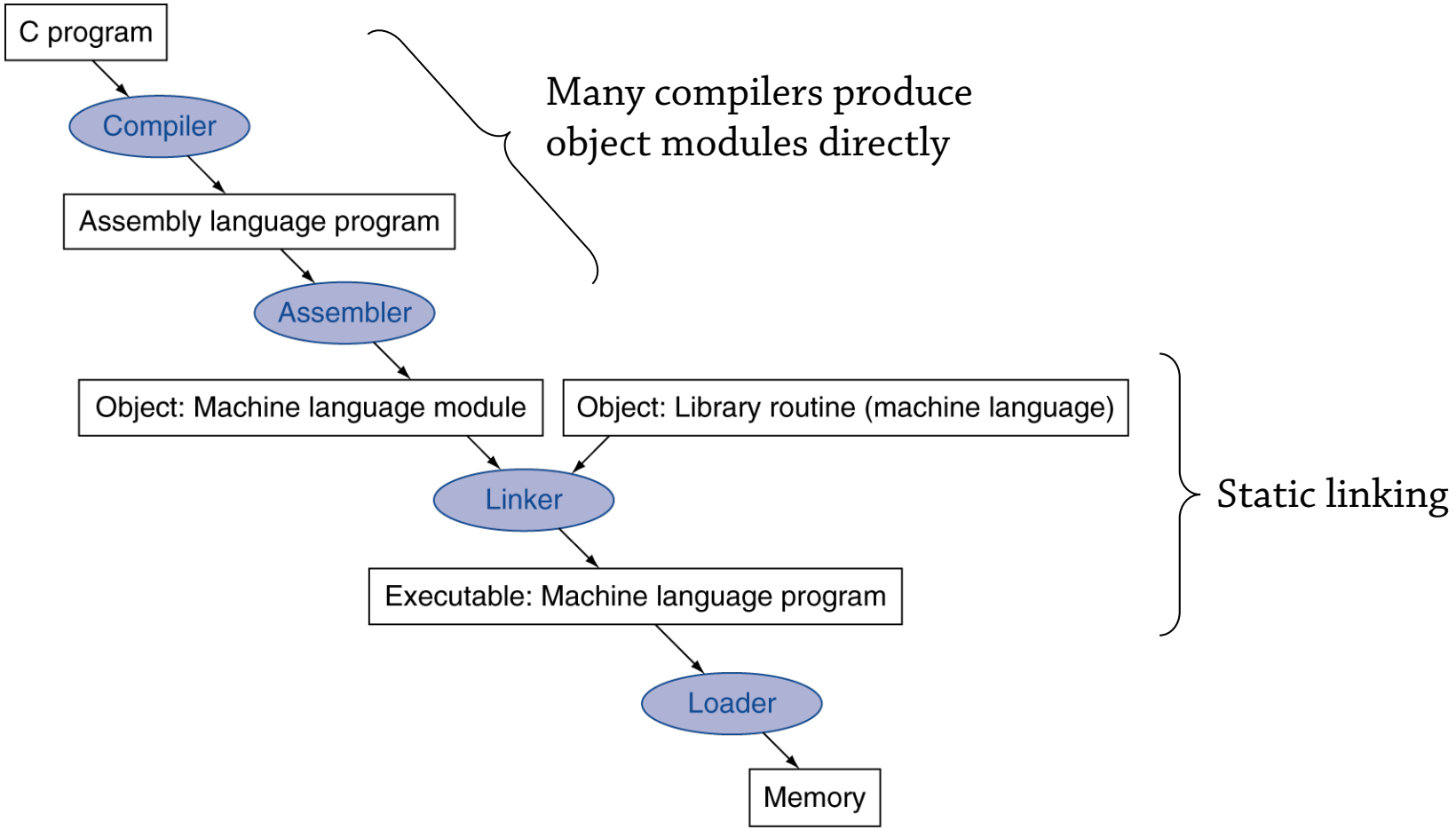
0010 0011 1011 1101 0000 0000 0001 0100

0010 00 → Opcode: 08_h → I-Type

Opcode	rs	rt	imm
001000	11101	11101	0000000000010100
08_h	$1D_h$	$1D_h$	14_h
08_h	29	29	20
addi	$sp	$sp	20

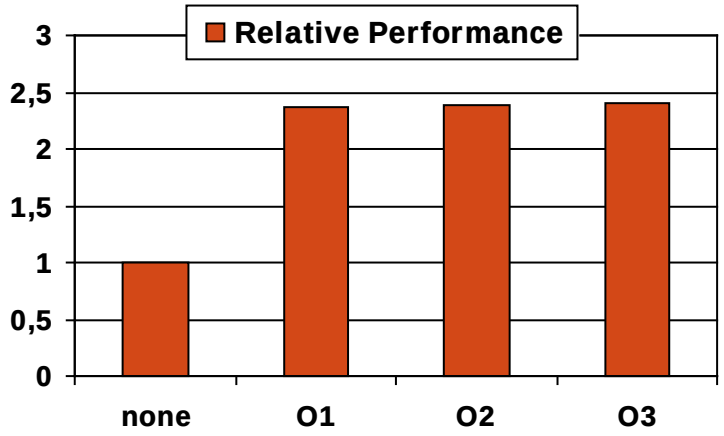
→ Assembler instruction: addi $sp,$sp,20

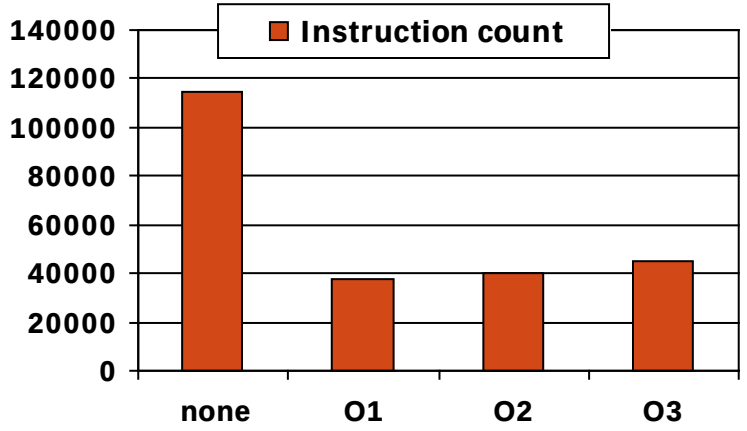
Translation and Startup

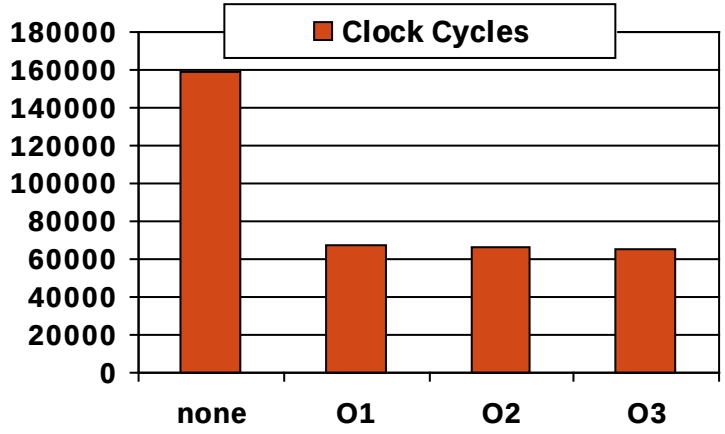


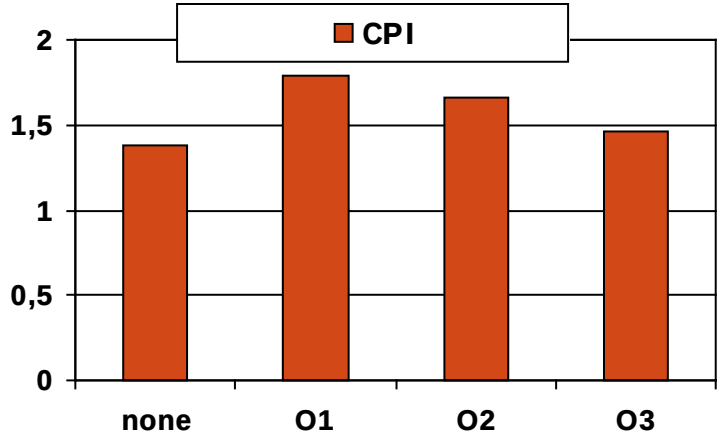
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux









ARM & MIPS Similarities

- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

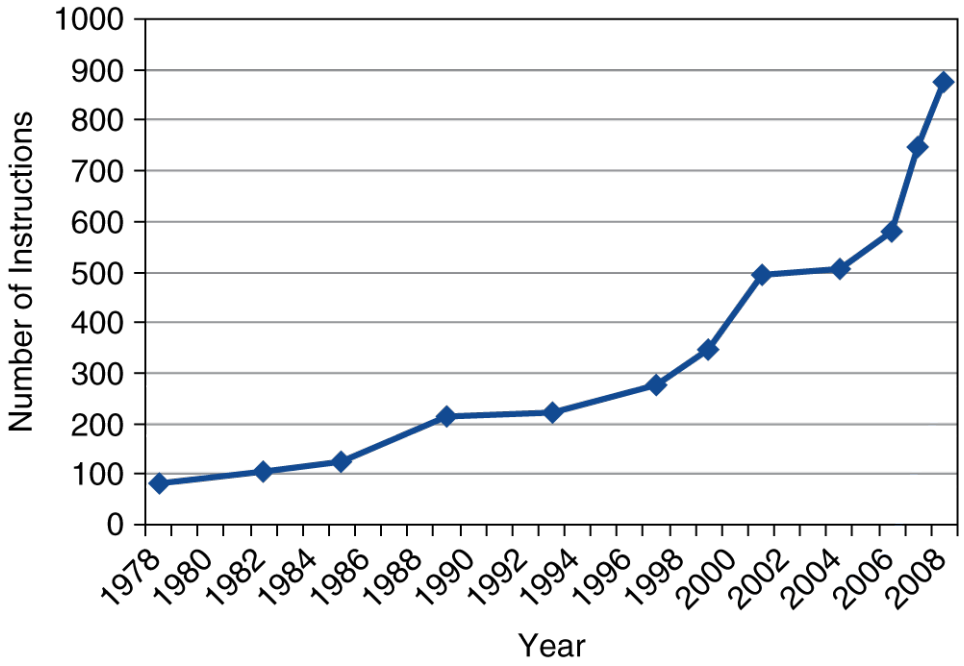
	ARM	MIPS
Date announced	1985	1985
Instruction size	32 bits	32 bits
Address space	32-bit flat	32-bit flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Registers	15 × 32-bit	31 × 32-bit
Input/output	Memory mapped	Memory mapped

Fallacies

- Powerful instruction ⇒ higher performance
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions
- Use assembly code for high performance
  - But modern compilers are better at dealing with modern processors
  - More lines of code ⇒ more errors and less productivity

Fallacies

- Backward compatibility ⇒ instruction set doesn't change
  - But they do accrete more instructions



x86 instruction set

Pitfalls

- Reading sequential words
  - Addresses not sequential!
  - Increment by 4, not by 1!

Concluding Remarks

- Design principles
  - 1. Simplicity favors regularity
  - 2. Smaller is faster
  - 3. Make the common case fast
  - 4. Good design demands good compromises
- Layers of software/hardware
  - Compiler, assembler, hardware
- MIPS: typical of RISC ISAs

182.690 RECHNERSTRUKTUREN – INSTRUCTIONS (2)

Thomas Polzer
tpolzer@ecs.tuwien.ac.at
Institut für Technische Informatik