Procedures, Assembly Alternatives, and Linking/Loading

- Last Time
  - J, JR, JAL long distance transfers and procedure linkage
  - Stacks (to lower addresses!) and optimizing stack arithmetic
  - symbolic names for registers
- This Time
  - Quiz #3
  - Detailed Procedure call example (recursion, stack discipline, calling conventions)
  - Alternative assembly/machine languages (RISC/CISC)
- Announcements
  - Reminder: Homework #2 due Thursday, February 3rd (beginning of class)

Basics of Procedure Call (review)

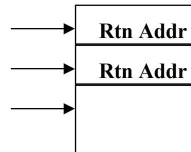
- jal target Jump and Link, $31 <- PC + 4
- jr $31 Jump back to the caller of the procedure
- Recursive procedures: must push return addresses on the stack, pop them before returning

Recursive Procedure (simple)

```
movi    $a0,10
jal     rec_proc
addi    $a3,$a3,1

rec_proc: sw      $31,0($sp)        # push rtn address
          addi    $sp,$sp,-4
          beq     $a0,$0,return     # is arg0 = 0?
          subi    $a0,$a0,1
          jal     rec_proc          # recursive call
return:   lw      $31,4($sp)        # pop rtn address
          addi    $sp,$sp,4
          jr      $31
```

- $a0 passed through
- rtn address pushed on stack
- popped before return
- $a0 at call != $a0 at return



Full MIPS Calling Conventions

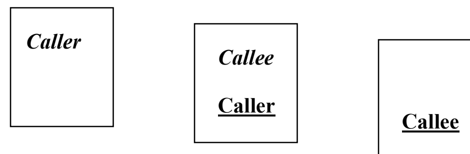
```
$0          Constant 0
$AT         $1    Reserved for Asm and OS
$v0-$v1     $2-$3 Return Values
$a0-$a3     $4-$7 Arguments

$t0-$t9     $8-$16,$24,$25     Caller saved Registers
$s0-$s7     $16-$23            Callee saved Registers
$k0-$k1     $26-$27          Reserved for Asm and OS
$gp         $28                Global Pointer
$sp         $29                Stack Pointer
$fp         $30   Frame Pointer (base of stack frame)
$31               Return address from jal
```

We're down to 8 regs, 10 temps, 4 arg registers, 2 return values!

Calling Conventions

- Caller saved ($t0-$t9), caller must save and restore if want to preserve the values. Callee may use indiscriminately.
- Callee saved ($s0-$s7), caller can assume these are preserved across calls, callee must save values and restore if want to use the registers.



Procedure Call (symbolic regs)

```
. . .
addi $a0,$0,23
sub  $a1,$a0,$t0
and $a2,$a1,$t1
jal  proc
add $s0,$s0,$vl0
. . .
```

```
proc:  sw    $ra,0($sp)
       addi  $sp,$sp,-20        # How many local temps?
       add   $t0,$a0,$a1        # trashes $t0, "caller saves"
       or    $t1,$a2,$a1        # trashes $t1
       sw    $s0,16($sp)        # save the "callee save registers"
       sw    $s1,12($sp)
       sw    $s2,8($sp)
       sw    $s3,4($sp)
       add   .... use $s0-$s3 in here ...
       move  $vl0,$t1           # setup return value
       lw    $s0,16($sp)        # restore the "callee saves"
       lw    $s1,12($sp)
       lw    $s2,8($sp)
       lw    $s3,4($sp)
       lw    $ra,20($sp)        # restore return address
       addi  $sp,$sp,20
       jr    $ra
```

Detailed Explanation

- call + arguments
- caller/callee saves
- return address
- stack management

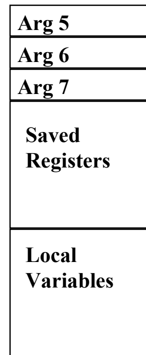
Procedure Call (summary)

- `jal, jr` **for linkage, push $ra if addl calls**
- **$sp=$29 and optimized stack arithmetic**
- **Caller saves don't persist over calls**
- **Callee saves must be saved before trashing**
- **return values in $vl0-$vl1**
- **Other Registers?**

Miscellaneous Registers

- **$at,$k0,$k1 -- used by assembler + OS (macros, etc.)**
- **$gp=$28 -- Global pointer points to global static space, static C variables, constants, etc.**
- **$fp = $30 -- Frame pointer points to base of the current procedure frame (can simplify arg addressing)**
- **Approx 18 real, usable registers ($s,$t). All others have some special usage.**

A few more details (stack frames)



- > 4 args, pass on stack
- registers saved atop the stack
- How to know how many args? (convention, fp and sp)
- Registers and locals under compiler control
- Stack grows to LOWER addresses

Detailed Example: Factorial

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

```
fact:        subu $sp,$sp,16
             sw $ra,16($sp)

             beq $a0,$0,no_recurse
             sw $a0,12($sp)     # save argument n
             subi $a0,$a0,1     # n-1
             jal fact
             lw $t0,12($sp)
             mul $vl0,$t0,$vl0  # n * fact(n-1)
             j  return
no_recurse:  addi $vl0,$0,1     # set return val = 1
return:      lw $ra, 16($sp)
             addu $sp,$sp,16
             jr $ra
```

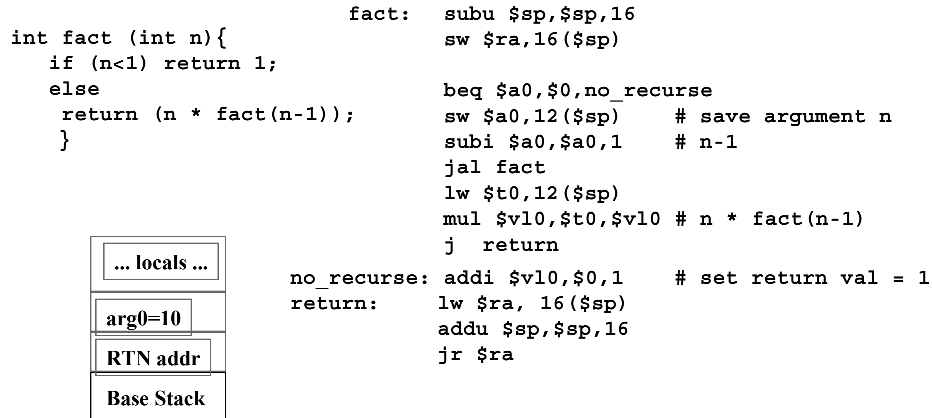
Base Stack

Detailed Example: Factorial

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

```
             fact:    subu $sp,$sp,16
                      sw $ra,16($sp)

                      beq $a0,$0,no_recurse
                      sw $a0,12($sp)    # save argument n
                      subi $a0,$a0,1    # n-1
                      jal fact
                      lw $t0,12($sp)
                      mul $vl0,$t0,$vl0 # n * fact(n-1)
                      j  return
        no_recurse:   addi $vl0,$0,1    # set return val = 1
        return:      lw $ra, 16($sp)
                     addu $sp,$sp,16
                     jr $ra
```



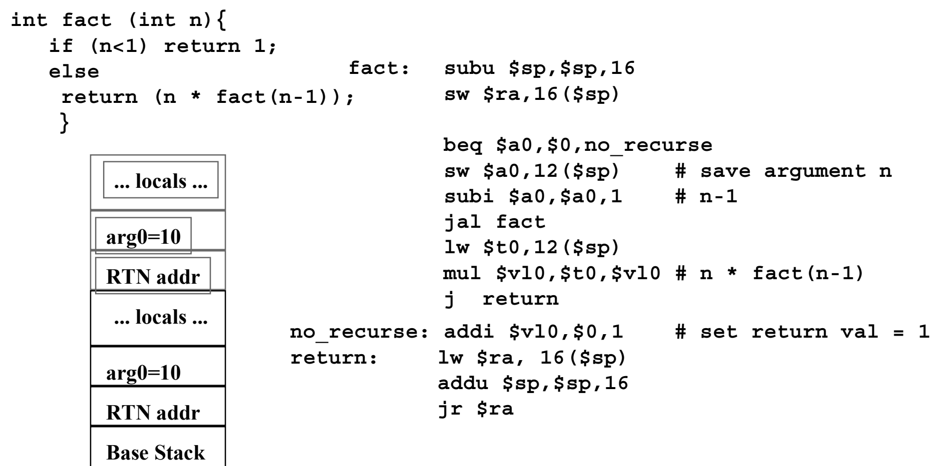
CSE 141 Chien 11 January 27, 2000

Detailed Example: Factorial

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

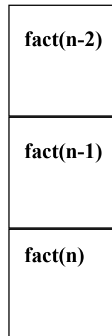
```
             fact:    subu $sp,$sp,16
                      sw $ra,16($sp)

                      beq $a0,$0,no_recurse
                      sw $a0,12($sp)    # save argument n
                      subi $a0,$a0,1    # n-1
                      jal fact
                      lw $t0,12($sp)
                      mul $vl0,$t0,$vl0 # n * fact(n-1)
                      j  return
        no_recurse:   addi $vl0,$0,1    # set return val = 1
        return:      lw $ra, 16($sp)
                     addu $sp,$sp,16
                     jr $ra
```



CSE 141 Chien 12 January 27, 2000

Stack After Recursive calls



- stack grows to lower addresses
- calls allocate/deallocate according to stack discipline (clean up your mess!)
- Where are the caller saves regs for fact(n) in stack? Where are the callee saves?
- Return links?

Summary: Procedure Call

- Procedure Linkage
- Calling conventions and register usage
- Stack structure and organization
- Now, you should be able to write assembly programs that interoperate with C and make recursive calls.

Perspective on the MIPS Instruction Set

- definitely a "RISC" = Reduced Instruction Set Computer
  - Instructions are simple and regular
  - There are only a modest number of operation codes
  - There are VERY few addressing modes (displacement, some PC-relative)
  - Instructions are all the same size
  - A few fixed formats for the instructions
- One of the original academic RISC projects that went commercial... (the other is the SPARC -- Sun architecture)

Assembly Overview for MIPS architecture

- Approximately....
  - 40 Add class opcodes
  - 20 control transfer (Jump and Branch)
  - 25 memory instructions
  - 25 Floating point instructions
  - + 5 miscellaneous
  - => still nearly 150 instructions
- 3 formats, all 32 bits
- only 2 addressing modes
- 32 general purpose registers

MIPS Overview (cont.)

- => Most of these features aid the implementation (fast, simple)
- Simple instruction sets are good for exposition of the minimal functionality required.
- => Not all instruction sets are this way...
- Why?
  - Evolution and learning how to do things better.
  - Previously, lots of assembly code was written by programmers, so focus on providing “higher level” interfaces
- What about others? VAX? PowerPC? 80x86 family? (most common computer processor)

A high-end design point: DEC VAX

- VAX (Virtual Address Extension)
  - 11/730-780, 1978-1987
  - 8xxx series, 1988 - present, 16 data registers, a few special
  - “Orthogonal Architectures” -> opcodes and all addressing modes
  - Many operations 250+, many addressing modes (8), all possible combinations, hardware must deal with these
  - Many formats, alignment of fields dependent on the addressing modes
  - Complex operations like “polynomial evaluate”, “block move”
  - Special call/return instructions
  - Execution time from 5 - 5,000 cycles, extremely complex implementations
  - Wulf “subsetting” argument, compilers

A Modern Alternative: the IBM/Motorola PowerPC Architecture

- Designed 1989 / 1993
- Leverages RISC ideas
- 32 general purpose registers, segment registers for larger address space
- simple instruction set, few "compound operations" added for critical inner loop operations
  - autoincrement/decrement modes
  - multiply/accumulate
- branch architecture: condition codes, multiple sets, separate test and use of information, distinct namespace
- Designed for multiple issue, a modern "RISC"?
- All of these machines are getting quite complex.

80x86 Architecture: Computer Evolution

- Why don't computer instruction sets always reflect the "state of the art" in understanding?
  - Differences of workload & opinion, "Binary Compatibility" requirements
  - Periodic opportunities for technology inclusion, ISA revision
- Binary compatibility vs. assembly language programming
  - BC: old programs and executables should run and run well
  - ALP: assembly language should be understandable and easy to use
  - => distinct constraints. BC admits "compatible extensions" to add new features, complex interfaces, and translation for execution. Few of the constraints of "lower level" programming.

Intel Microprocessor History

- 4004 First single-chip microprocessor, 1974
  - 4-bit machine, era of rack-sized machines, utility unclear, target was pocket calculator market...
- 8008, 8080,8085 First 8-bit micros, “bug book”
  - Birth of hobbyist microcomputers, Z80’s
- 8086,8088 8/16 bit micro, 1979
  - Basis of IBM PC’s, lots of microcontroller usage
- 80286 24-bit addresses, PC/AT
  - little native code, “real mode”

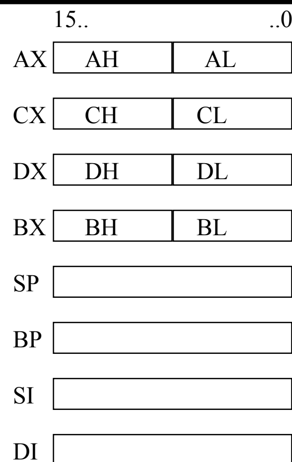
Intel Micro History (cont.)

- 80386 32-bit addressing, modern PC
  - still lots of 16-bit code, Win16, Win32
- 486, Pentium, P6 (minor ISA changes)
- Merced Epic (IA64)
  - Major changes, 64-bit addressing, new instruction formats, 256 registers, Epic
  - Significant hardware for compatibility

Implications of Historical ISA's

- Popular microprocessors -> significant software base
- Competitive advantage, preserve software base, prevent defections
- Cost-Benefits favor compatibility at each step, though exponential growth means??? eventually phase out?
- "Evolution of an architecture", looks like layers of revisions.
- Resource limits of early microprocessors similar to first computers in 50's... so... Intel's 8080 was a single accumulator architecture...

Initial Architecture: 8086

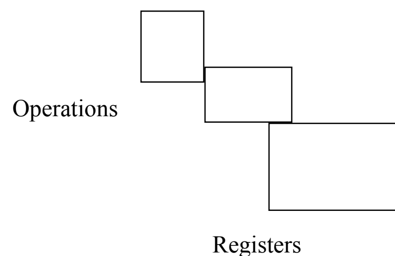


- 8/16 bit architecture (8008 "compatibility")
- Accumulator and special registers (vs 32 GPR's) -- fewer interconnects and "wired operations" into registers
- 8-bit and 16-bit register names
- IP (program counter), not shown
- Why special regs? Instructions w/ fewer operands
- 2-address, architecture, but registers not GPR, so addressing modes linked to registers (e.g. stack offset, base offset, etc.) MUST get values into the right registers...

Addressing modes and single accumulator architecture

- 8080 was a true single accumulator machine
- 8086 extended with many registers, most were special use
- instructions used special registers (DH, DL), etc.
- addressing modes use special registers (SP, BX, etc.)
- procedure call linkage instructions use special registers
- asymmetric tangle of operations/addressing modes/registers
- => lots of fun hacking!

Asymmetry and Orthogonality



- Asymmetric architectures couple the use of registers and selection of operations.
- Harder to generate good code.
- 17 ways to clear accumulator A...

How do you clear accumulator A?

- Load constant 0
- subtract AX from AX
- xor AX with itself
- shift left 16 positions
- ...
- Which is fastest and why?
  - Compilers needed to choose (few)
  - Programmers knew this well.

24-bit and 32-bit extensions

- 80286 -> 24 bit external addresses
- Segment register usage changed
  - Info in segment register used to produce 24 bit addresses from 16-bit registers (24 bit segment base)
  - little change to basic programming model
- 80386 -> 32 bit external address, 32-bit registers
  - updated register files
  - addition of operations which make "more orthogonal" sets of operations
  - modifiers of opcodes to extend the functionality

Updated Register Files

32-bit	16-bit	High	Low
EAX	AX	AH	AL
ECX	CX	CH	CL
EDX	DX	DH	DL
EBX	BX	BH	BL
ESP	SP		
EBP	BP		
ESI	SI		
EDI	DI		
	CS		
	SS		
	DS		
	ES		
EIP	IP		
EFlags	Flags		

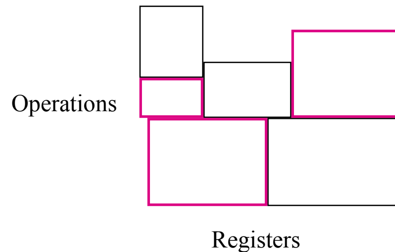
- **16->32 bit data registers**
- **Segment registers same size**
- **IP, Flags extension**
- **Modes (addressing and operations), datatype sizes and segment registers/modes**
- **Exploiting “mode bits” and modifier bytes to “condition” the operations**

Extending an Instruction Set Encoding

prefix	add	A	op2

- **Basic Instruction (old) encoding**
- **Extended operation (modifier), “32-bit opns”**
- **=> trick works because modifier bytes previously supported**
- **Complex decoding and instruction formats (1-17 bytes)**
- **Prefix, opcode, addr specifier(s), displacements, immediates**
- **How does this compare to flexibility in DLX?**

Extending usability of Registers

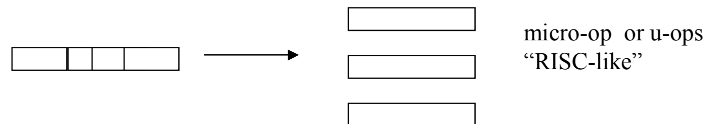


- How to make special-purpose registers more general purpose?
- Fill in the gaps with new instruction encodings (prefixes, modes, mod bits)

Encoding Results

- Complex and varied instruction set
- "Breaks all the rules", but not as complex as the VAX
- Intel able to make it fast, and competitive with other architectures (with 2x)... how?
- P6/PII/PIII: Aggressive prefetching and instruction "translation" into RISC-like micro-ops
- Simple fast pipeline to support these micro-operations, similar to a RISC processor.
- "on the fly" translation -- pros? cons?

Fast 80x86 Implementation



- **Basic Structure; when one u-op?**
- **Pros: code density, compatibility**
- **Cons: design effort, complexity, compilation, "latency"**
- **Similar strategy pursued by other CISC-ish architectures such as VAX**

80x86 Statistics and Usage

	Compress	Eqntott	espresso
Data Opns (x86)	615	230	751
DRatio to DLX	1.03	1.25	1.58
Total Insts (x86)	2226	1203	2216
Inst Ratio to DLX	0.61	1.74	0.85

- **Instruction Set design affects usage**
- **Total instruction counts, proportion (counts in millions)**
  - **Instruction counts are comparable, but varied; 1.03 overall**
  - **Data Accesses are greater for 80x86, 1.23 overall**
  - **lack of registers stresses the memory system**

80x86 Statistics and usage (cont)

	doduc	hydro2d	su2cor
Data Opns (x86)	778	8212	4782
Dratio to DLX	2.40	4.44	3.44
Insts (x86)	1223	13,342	6197
Inst Ratio to DLX	1.19	2.53	1.62

- **Floating Point Codes (millions)**
  - **Significantly more instructions (8087 stack architecture), 1.73 overall**
  - **Significantly more data accesses, 3.35 overall**
- **=> lack of registers stresses the memory system**

X86 Summary

- **Evolution of architectures which are important in the software base**
- **Historical architectures are much more complex.**
- **Creativity, hackery, and aggressive implementations can keep these alive, but ISA does have implications for performance and system characteristics.**

What is RISC vs. CISC?

- Debate about instruction set complexity
- Mostly over, since compilers deal best with simple instruction sets, and eases implementation, everyone subscribes.
- Many shades of grey in "RISC" iness
- Most implementations are extremely complex (1M to 50M transistors)
  - Simple implementation is not a strong argument
  - Ability to achieve high clock speed, high performance is the major motivation

Summary

- Procedure Calling
  - Complex process
  - Constrained use of registers
  - Stack discipline and memory traffic
- Machine Language Alternatives
  - RISC vs. CISC
    - Driven by programming and implementation issues
      - Can see the programming issues
      - Will explore implementation issues in second third of the class
  - Historical Architectures