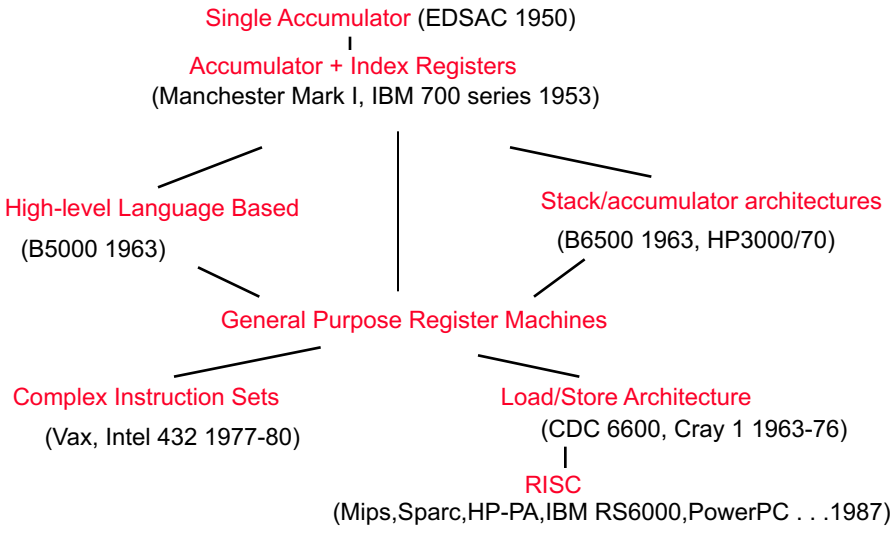
Review

- Last Class:
  - Moore's Law and Dennard scaling
  - Bandwidth and latency.
  - Performance measurement (CPU time, CPI, Amdahl's Law)
- Today's class:
  - Review of instruction set architecture (ISA)
  - Instruction format
  - Addressing mode
- Announcement and reminder
  - TA officer hour: Thu 11am to 1pm. Will update syllabus.
  - https://pitt.zoom.us/j/6327445850
  - Quiz (no credit) dues this Friday midnight
  - No class next Monday (Jan 17th)

(1)

Evolution of Instruction Sets



(2)

Basic CPU storage options

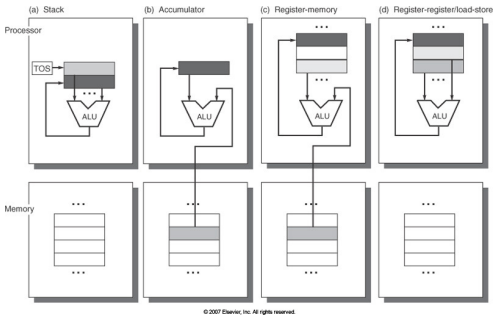
C = A + B in different storage schemes:

1. ***Stack***

```
push  A
push  B
add
pop   C
```

2. ***Accumulator***

```
load  A
add   B
Store C
```



3. ***Memory-Memory***

```
add  C, A, B
```

4. ***Register-Memory***

```
load  R1, A
add   R2, R1, B
store R2, C
```

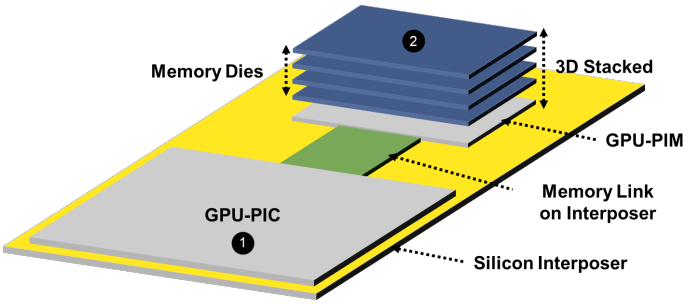
5. ***Register-Register***

```
load  R1, A
load  R2, B
add   R3, R2, R1
store R3, C
```

What is the effect on: speed, memory traffic, encoding, program length?
What determines the number of registers?

(3)

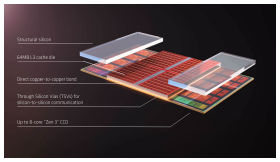
Processing-in-memory



Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities

Ashutosh Pattnaik[1] Xulong Tang[1] Adwait Jog[2] Onur Kayıran[3]
Asit K. Mishra[4] Mahmut T. Kandemir[1] Onur Mutlu[5,6] Chita R. Das[1]
[1]Pennsylvania State University [2]College of William and Mary
[3]Advanced Micro Devices, Inc. [4]Intel Labs [5]ETH Zürich [6]Carnegie Mellon University

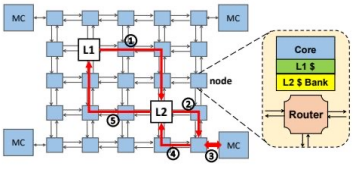
PACT, 2016

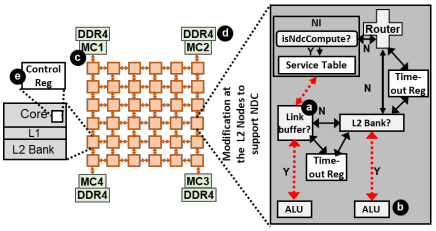


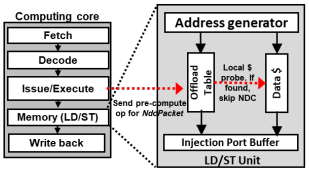
AMD V-cache (announces 17th Nov 2021)

(4)

Near-data Computing







Compiler Support for Near Data Computing

Mahmut Taylan Kandemir
Penn State University
USA
mtk2@cse.psu.edu

Jihyun Ryoo
Penn State University
USA
jihyunryoo@gmail.com

Xulong Tang
University of Pittsburgh
USA
tax6@pitt.edu

Mustafa Karakoy
TUBITAK-BILGEM
Turkey
m.karakoy@yahoo.co.uk

PPOPP, 2021

(5)

- Design decisions must take into account:
  - technology
  - machine organization
  - programming languages
  - compiler technology
  - operating systems

- Issues in instruction set design:
  - operand storage in CPU (stack, registers, accumulator)
  - number of operands in an instruction (fixed or variable number)
  - type and size of operands (how is operand type determined)
  - addressing modes,
  - allowed operations and the size of op-codes,
  - size of each instruction.

Op-code	operand	operand	operand	Other fields

(6)

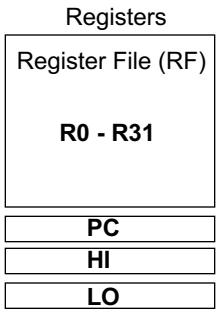
RISC vs CISC

- RISC = Reduced Instruction Set Computer
  - MIPS, SPARC, PowerPC, ARM (Cortex), etc.
- CISC = Complex Instruction Set Computer
  - X86 is the only surviving example
- Goals in the 1980s – reduce design time, faster/smaller implementation, ISA processor/compiler co-design
- ISAs are measured by how well compilers use them, not by how well or how easily assembly language programmers use them
- There are (or, at least, it's believed there are) many old and useful programs that only exist as machine code, so supporting old ISAs has economic value

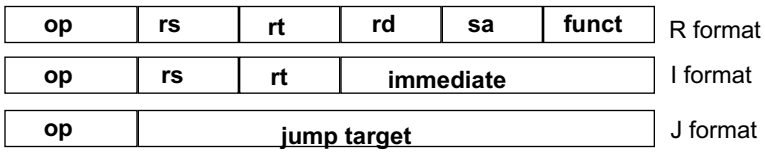
(7)

MIPS-32 ISA

- **Instruction Categories**
  - **Computational**
  - **Load/Store**
  - **Jump and Branch**
  - **Floating Point**
    - » **coprocessor**
  - **Memory Management**
  - **Special**



3 Instruction Formats: all 32 bits wide



(8)

Aside: MIPS Register Convention

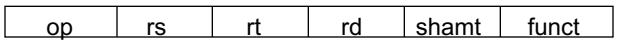
Name	Register Number	Usage	Preserve on call?
$zero	0	constant 0 (hardware)	n.a.
$at	1	reserved for assembler	n.a.
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	yes
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return addr (hardware)	yes
$cr0-cr8	9	X86 reserved	n.a.

What is CR3 for?

(9)

MIPS Instruction Fields

- **MIPS fields are given names to make them easier to refer to**



op	6-bits	opcode that specifies the operation
rs	5-bits	register file address of the first source operand
rt	5-bits	register file address of the second source operand
rd	5-bits	register file address of the result's destination
shamt	5-bits	shift amount (for shift instructions)
funct	6-bits	function code augmenting the opcode

(10)

Example

- **MIPS instruction**
  - **Add $8, $9, $10**

0	9	10	8	0	32

000000	01001	01010	01000	00000	100000

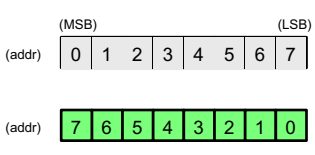
  - **Hex representation: 012A 4020**
  - **Decimal representation: 19,546,144**

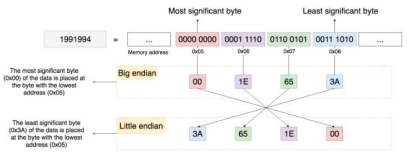
(11)

Memory addressing

- Most modern machines are *byte-addressable* -- yet most memory and cache traffic is in terms of words.
- A natural alignment problem (the start of a word or a double word).
  - Compiler is responsible
  - hardware does the checking
- How are bytes addressed within a word?
  - Big Endian -- address byte 0 is the MSB (IBM, MIPS, SPARC)
  - Little Endian -- address byte 0 is the LSB (vax, intel 80x86)

  Problem when we deal with serial communication and I/O devices.





(12)

Source:https://uynguyen.github.io/2018/04/30/Big-Endian-vs-Little-Endian/

Addressing Modes

1) Register
   operand = content of register = (R) — Add R1, R2
2) Immediate
   operand = in instruction = C — Add R1, #54
3) Register indirect
   operand = in memory = Mem[(R)]
   address = content of register — Add R1, (R2)
4) Displacement (or base)
   operand in memory = Mem[(R) + Base]
   address = content of register + base — Add R1, 54(R2)
5) Indexed
   operand in memory = Mem[(R) + (IR)]
   address = content of R+ content of IR — Add R1, (R2+R3)

Note: (R) means content of R and Mem[A] means content of memory address A. (13)

6) Direct (absolute)
   operand in memory = Mem[C]
   address = a constant in the instruction — Add R1, (1000)
7) Memory indirect
   operand in memory = Mem[Mem[(R)]]
   address = the content of Mem[(R)] — Add R1, @(R2)
8) Auto-increment (or decrement)
   operand in memory = Mem[(R)]
   The content of R is incremented — Add R1, (R2)+
9) Scaled
   operand in memory = Mem[C+(R)+(IR)*d] — Add R1, 100(R2)(R3)

Which addressing mode is most suitable for:
local variables, stack operations, array operations, pointers, branch addresses, branch condition evaluation, address translation.

(14)

Operations

- **Arithmetic/logical**: add, sub, mult, div, shift (arith,logical), and, or, not, xor …
- **Data movement**: copy, move, load, store, ..
- **Control:** branch, jump, call, return, trap, …
- **System**: OS and memory management (ignore for now)
- Floating point:
- **Decimal**: legacy from COBOL
- **String**: move, copy, compare, search
- **Graphics**: pixel operations, compression, ...

- In Media and Signal processing, **partitioned** (or **paired**) operations are common (example: add the two half-words of a word).
- **Saturating arithmetic** avoids interruption for overflow or underflow conditions
- **Multiply-accumulate** operations are very useful for dot products.

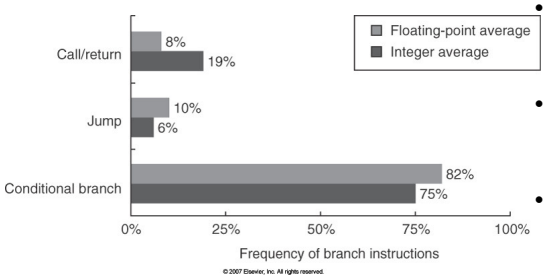
(19)

Frequent Operations

Rank	80x86 instruction	Integer average (% total executed)
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
Total		96%

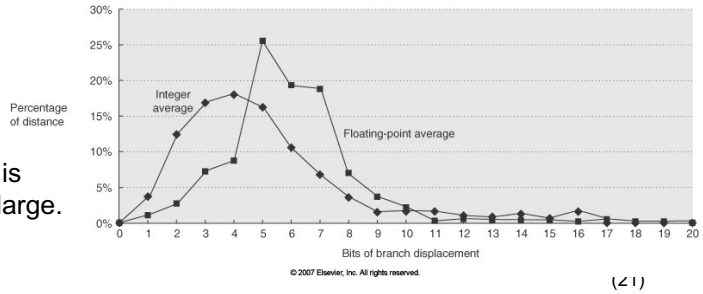
(20)

Branch instructions



- Branch conditions are usually simple equality/inequality comparisons
- More than 80% of comparisons use immediate constants,
- A majority of comparisons is with “zero”

- Branch distance is usually not very large.



(21)

Branch address specification

1) PC relative: -- makes the code position independent,
-- reduces the number of bits for target specification
-- target should be known at compile-time

2) put the target address in a register:
-- less restrictions on the range of branch address
-- useful for “switch” statements and function pointers
-- loaded at run-time (shared libraries and virtual functions)

(22)

Specification of branch conditions

- Use condition codes (flags usually set by hardware)
- Use condition registers
- compare and branch instructions
- Predicated instructions (operations guarded by a predicate)

C Source:

if (a < b) c++ else c+=1+b

Assume a → r1
b → r2
c → r3

Unpredicated

```
    cmp r1,r2
    blt L1
L0: add r3,r2,r3
L1: add r3,1,r3
```

Predicated

```
cmplt r1,r2,p1
add    r3,1,r3
add_p p1,r3,r2,r3
```

(23)

Procedure call and Return

- At a minimum the return address should be saved
  - Use branch and link instructions
  - Need to use a stack for nested calls
- Registers may have to be saved (by hardware or software)
- Registers can be saved by the caller or the callee,
- May mark some registers as “temporary”,
- Pass arguments in registers or on stack

- CPU vs GPU
  - CPU register file is small (dynamic register renaming)→expensive context switch → CPU is latency sensitive
  - GPU register file is large (static register allocation at compile time) → lightweight context switch → GPU is latency in-sensitive

(24)

Encoding the instruction set

- Need to include op-code, operands, and maybe other fields
- Variable # of operands may call for variable instruction length
- Variable instruction length may reduce the code size,
- Fixed instruction length is easier to decode and faster to execute
- May use variable length op-code
- How do you specify the addressing mode?

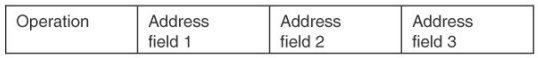
Examples:

- The VAX:
  - – can have any number of operands, each may use any addressing modes,
  - – Each operand uses a 4-bit specifier + 4-bit register address + one possible byte or word for displacement/immediate.
- RISC instructions use a fixed # of operands and specific addressing modes,
- Intel and IBM 360/370 use a hybrid approach (a few instruction lengths)

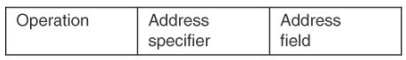
(25)

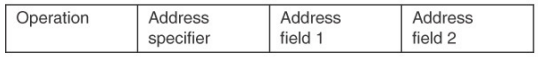
Encoding the instruction set

(a) Variable (e.g., Intel 80x86, VAX)



(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)





(c) Hybrid (e.g., IBM 360/370, MIPS16, Thumb, TI TMS320C54x)

© 2007 Elsevier, Inc. All rights reserved.

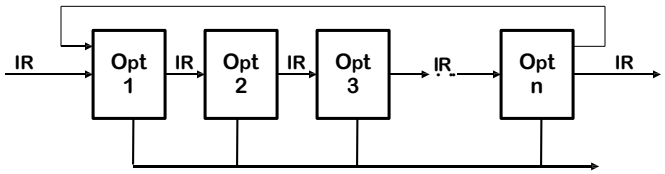
(26)

Role of compilers

- Compilers are multi-phase: Front-end, high-level optimization, global optimization and code generation.
- The goals of a compiler are: correctness, speed of compiled code, speed of compilation, debugging support, …
- Compiler can do better optimization when instructions are simple
- Allocation of variables:
  - registers are used for temporaries, and possibly parameter passing
  - stacks are used for activation records and local variables
  - a global data area (may be bottom of stack) is used for globals
  - a heap is used for dynamically declared data

(27)

The Optimizer (or Middle End)

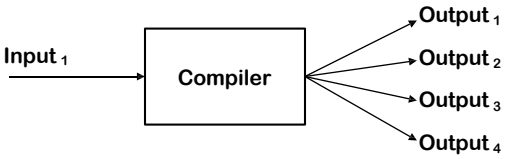


Modern optimizers are structured as a series of passes

- Typical Transformations
  - Discover & propagate some constant value
  - Move a computation to a less frequently executed place
  - Specialize some computation based on context
  - Discover a redundant computation & remove it
  - Remove useless or unreachable code
  - Encode an idiom in some particularly efficient form

(28)

What does optimization do?



- The compiler can produce many outputs for a given input
  - The user might want the fastest code
  - The user might want the smallest code
  - The user might want the code that pages least
  - The user might want the least power consuming code
  - The user might want the code that …
- Optimization tries to reshape the code to better fit the
- user's goal

(29)

A "Typical" RISC

- Fixed format instruction
- General purpose registers -- some have overlapping register windows.
- 3-address, reg-reg arithmetic instruction
- Single address mode for load/store:
  base + displacement (no indirection)
- Simple branch conditions -- use PC relative mode for branching.
- Hardwired control (as opposed to micro-programmed control)
- Pipelined execution (one instruction issue every clock tick),
- Delayed Branches and pipeline stalls.

RISC II (Berkeley) had 39 instructions, 2 addressing modes and 3 data types,
Vax had 304 instructions, 16 addressing modes, 14 data types,

RISC II programs were 30% larger than Vax programs but 5 times as fast.
The RISC compiler were 9 times faster than the Vax compiler.

(30)

The RISC-V architecture

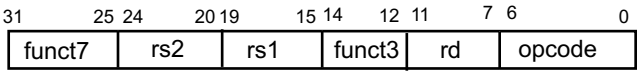
- 32, 64-bit general purpose registers (GPRs)
  - called x0, … , x31 (x0 is hardwired to the value 0).
- 32, 64-bit floating point registers - FPRs (each can hold a 32-bit single precision or a 64-bit double precision value)
  - called f0, f1, … , f31 (or f0, f2, … , f30).
- A few special purpose registers (example: floating point status),
- Byte addressable memories with 64-bit (or 48-bit) addresses.
- 32-bit instructions
- Only immediate and displacement addressing modes (12-bit field)

- **Data transfer operations**: ld, lw, lb, lh, flw, sd, sw, sb, sh, fsw, …
- **Arithmetic/logical operations**: add, addi, sub, subi, slt, and, andi, xor, mul, div, …
- **Control operations**: beq, bne, blt, jal, jalr, …
- **Floating point operations**: fadd, fsub, fmult, fsqrt, …

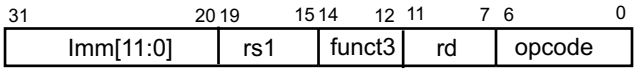
(31)

RISC-V instruction format

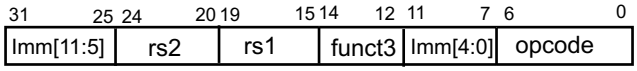
Register-Register (R-type) – used mainly for ALU instructions



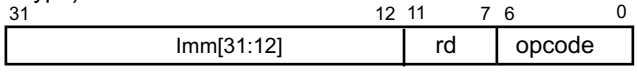
Register-Immediate (I-type) – used mainly for load instructions



Register-Immediate (S-type) – used mainly for Store and Branch instructions



Jump (U-type)



(32)

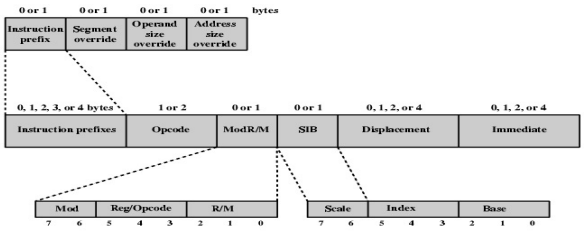
More to read: https://inst.eecs.berkeley.edu/~cs61c/resources/su18_lec/Lecture7.pdf

The Intel 80x86 architecture

- 1971 - Intel 4004 (4-bit architecture)
- 1972 - Intel 8008 (8-bit architecture)
- 1974 - Intel 8080 (larger ISA, 16-bit address space, single accumulator, only 6 VLSI chips)
- 1974 - Intel 8086 (16-bit architecture, 16-bits dedicated registers)
- 1980 - Intel 8087 (floating point co-processor)
- 1982 - Intel 80286 (24-bit address space but has a compatible mode)
- 1985 - Intel 80386 (32-bit architecture and address space, 32 GPRs, paging and segmentation hardware),
- 1989 - Intel 80486
- 1992 - Intel Pentium
- 1996 - Pentium 2 (233-366 MHz, 512 KB L2 cache)
- 1999 - Pentium 3 (100-133 MHz, 512KB L2 cache)
- 2000 - Pentium 4 (1.3-3.6 GHz, 256KB – 2MB L2 cache)
- 2005 - Pentium D (2.66-3.73 GHz, 2–4 MB L2 cache)
- 2007 - Pentium dual core (1.6-2.7 GHz, 1-2MB L2 cache)
- 2010 - Nahalem (up to 3GHz, up to 8 cores, up to 30MB L3)

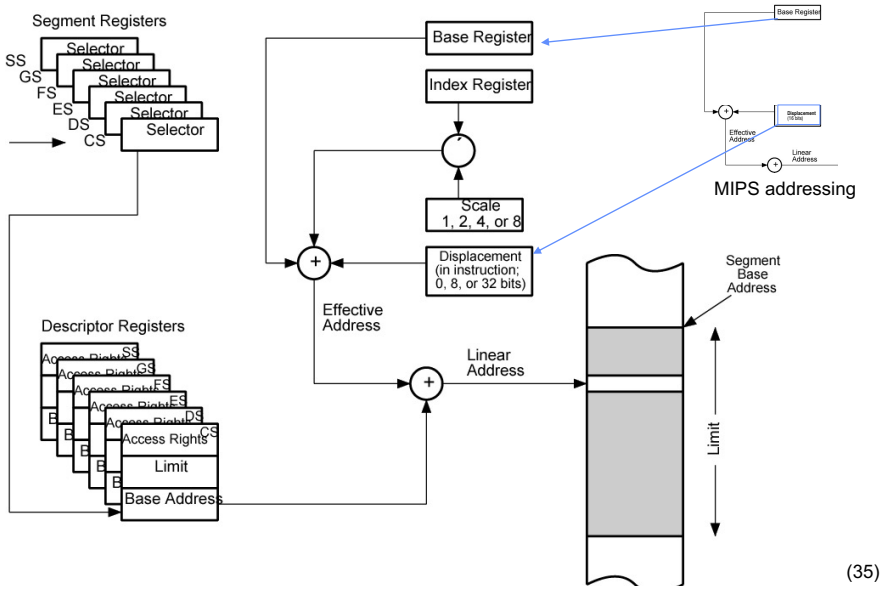
The Intel original 80x86 architecture

- Variable length op-code and instructions



- 16-bit architecture, but can get 20-bit address using segmentation
- addressing modes:
  - absolute
  - register indirect (BX, SI, DI in 16-bit modes, extended registers in 32-bit mode)
  - base mode (BX, SI, DI, SI + displacement which is 8, 16 bits , or 8, 16, 32 bits)
  - indexed BX+SI, BX+DI, BP+SI, BP+DI
  - based indexed (indexed+ 8 or 16 bit displacement)
  - based plus scaled indexed (on 386, scale = 0,1,2,3 , restrictions on register use is removed)
  - based with scaled index and displacement.

x86 Address Calculation



(35)