NWEN 242

3. Language of the computer

Agenda

- Operations of the computer hardware
- Signed and unsigned numbers
- Representing instructions in the computer
- Logic operations
- Instructions for making decisions
- Hardware support for procedure calls
- Character encoding and manipulation
- MIPS addressing schemes
- Translating and starting a program
- Arrays and pointers

Introduction

- To command a computer's hardware, you must speak its language.
- The words of a computer's language are called instructions.
- The collection of words, the vocabulary, is called the instruction set.

- Chosen technology: MIPS
  - Designed since 1980s.
  - 32 bit architecture (word = 4 bytes)
  - 32 bit addressable space

MIPS assembly language

- MIPS stands for Microprocessors without Interlocked Pipeline Stages
  - Not "million instructions per second"
- MIPS assembly language was originally designed during 1980s (Hennessy)
- Used by NEC, Nintendo, SGI, Sony, …
- MIPS processors are RISC processors
  - Restricted Instruction Set Computer
  - Different from Complex Instruction Set Computer (CISC), such as Intel x86
  - Promoted by Patterson
    - Simple, elegant, fast
    - Successfully exemplifies four main design principles (to be covered later)

Start from machine language

- Computers use the binary alphabet to store both program instructions and data
  - The only two letters available are 0 and 1
    - $5_{10} = 101_2$, $1000001_2 = 65_{10}$
  - Add instruction
    - $00000000010001010011000000100000_2$

base 10
base 2

- A computer understands only binary (machine) instructions.

- In the beginning, programmers use **binary instructions** to write programs

- But this was **hard** and **error prone**

Assembly language

- To make programming easier, programmers started to use a notation closer to human way of thinking
  - add C, A, B # C=A+B
- To translate the new notation to machine language, special programs named assemblers were needed
- An assembly language instruction corresponds to a machine instruction
  - Except pseudoinstructions

Example: language translation

- High-level programming languages are closer to human thinking:

C = A + B

Compiler

```
lw $a1, mem[A]
lw $a2, mem[B]
add $v0, $a1, $a2
```

Assembler

```
10001101 11100101 00000000 00000000
10001101 11100110 00000000 00000100
00000000 01000101 00110000 00100000
```

Abstraction

- Abstraction is a technique for hiding lower level details of hardware and software to provide a simpler higher level view
- One of the most important abstractions is the interface between the hardware and the lowest level software
  - Instruction set architecture
  - Includes machine instructions, I/O devices, etc.

- Abstraction hides complexity but also hides any issues involved in using the hardware
- Understanding what lies beneath the abstraction allows us to maximise our use of the hardware

Quick question

- What is the main difference between 32-bit processors and 64-bit processors?
  - A. The maximum amount of memory supported.
  - B. 32-bit processors are cheaper to build.
  - C. Programs will run faster on 64-bit processors.
  - D. 64-bit processor can only run 64-bit computer programs.

Operations of the computer hardware

- Operation: an action to be performed by the computer hardware.

- Among all possible operations, arithmetic operation must be supported by all processors.

```
add a, b, c     # The sum of b and c is placed in a.
add a, a, d     # The sum of b, c, and d is now in a.
add a, a, e     # The sum of b, c, d, and e is now in a.
```

operation

Variables/operands

comments

  - Each operation has exactly three variables.

MIPS assembly in brief

- MIPS operands
  - 32 registers
    - $s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at
  - 2^{30} memory words
  - Immediate (or constants)
- Types MIPS instructions
  - Arithmetic
    - add, subtract, add immediate
  - Data transfer
    - load word, store word, etc.
  - Logic
    - and, or, shift left logical, shift right logical, etc.
  - Conditional branch
    - branch on equal, branch on not equal, etc.
  - Unconditional jump
    - Jump, jump and link, etc.

Why? Why not 2^{32} ?

See slide 19

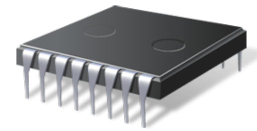
The number of operands

- The natural number of operands for an operation like addition is three

- Requiring every instruction to have exactly three operands, no more or no less, conforms to the philosophy of keeping the hardware simple

Design principle 1: Simplicity favors regularity.

Registers



- A processor register is a small amount of storage available as part of a processor
  - Registers are built in processor's datapath
  - Processor registers are normally at the top of the memory hierarchy.

- Type of registers
  - General purpose registers (GPRs): store both data and addresses and can be directly accessed in a user program.
  - Floating point registers (FPRs): store floating point numbers.
  - Constant registers: hold read only values such as $zero.
  - Special purpose registers (SPRs): instruction register, status register, cause register, etc.

The number of registers

Register Number	Mnemonic Name	Conventional Use	Register Number	Mnemonic Name	Conventional Use
\$0	zero	Permanently 0	\$24, \$25	\$t8, \$t9	Temporary
\$1	\$at	Assembler Temporary (reserved)	\$26, \$27	\$k0, \$k1	Kernel (reserved for OS)
\$2, \$3	\$v0, \$v1	Value returned by a subroutine	\$28	\$gp	Global Pointer
\$4–\$7	\$a0–\$a3	Arguments to a subroutine	\$29	\$sp	Stack Pointer
\$8–\$15	\$t0–\$t7	Temporary (not preserved across a function call)	\$30	\$fp	Frame Pointer
\$16–\$23	\$s0–\$s7	Saved registers (preserved across a function call)	\$31	\$ra	Return Address

Design principle 2: Smaller is faster.

Example 1

- Compile two c assignment statements in MIPS.

```
a = b + c;
d = a - e;
```

Which is correct?

- A. add a, b, c
  sub d, a, e

- B. add b, c, a
  sub a, e, d

Example 2

- Compile a complex c assignment into MIPS

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

- MIPS code:

```
add t0,g,h # temporary variable t0 contains g + h
add t1,i,j  # temporary variable t1 contains i + j
sub f,t0,t1 # f gets t0 - t1, which is (g + h) - (i + j)
```

Example 3

- Compile a c assignment using registers

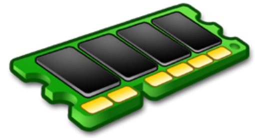
```
f = (g + h) - (i + j);
```

  - Allocate registers to hold variables
    - f ($s0), g ($s1), h ($s2), i ($s3), j ($s4), t0 ($t0), t1 ($t1)

- MIPS code:

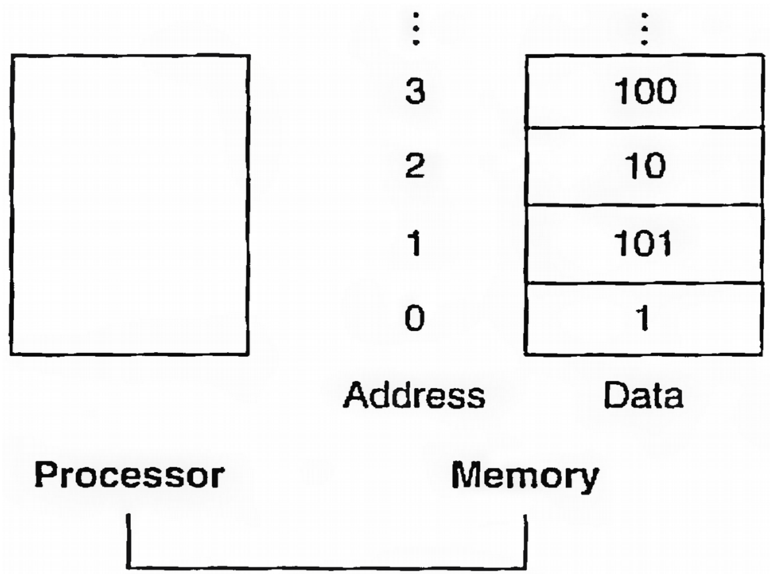
```
add $t0,$s1,$s2 # register $t0 contains g + h
add $t1,$s3,$s4 # register $t1 contains i + j
sub $s0,$t0,$t1 # f gets $t0 - $t1, which is (g + h)-(i + j)
```

Memory operands



- The processor can keep only a small amount of data in registers.
- Large data structure should be kept in memory.

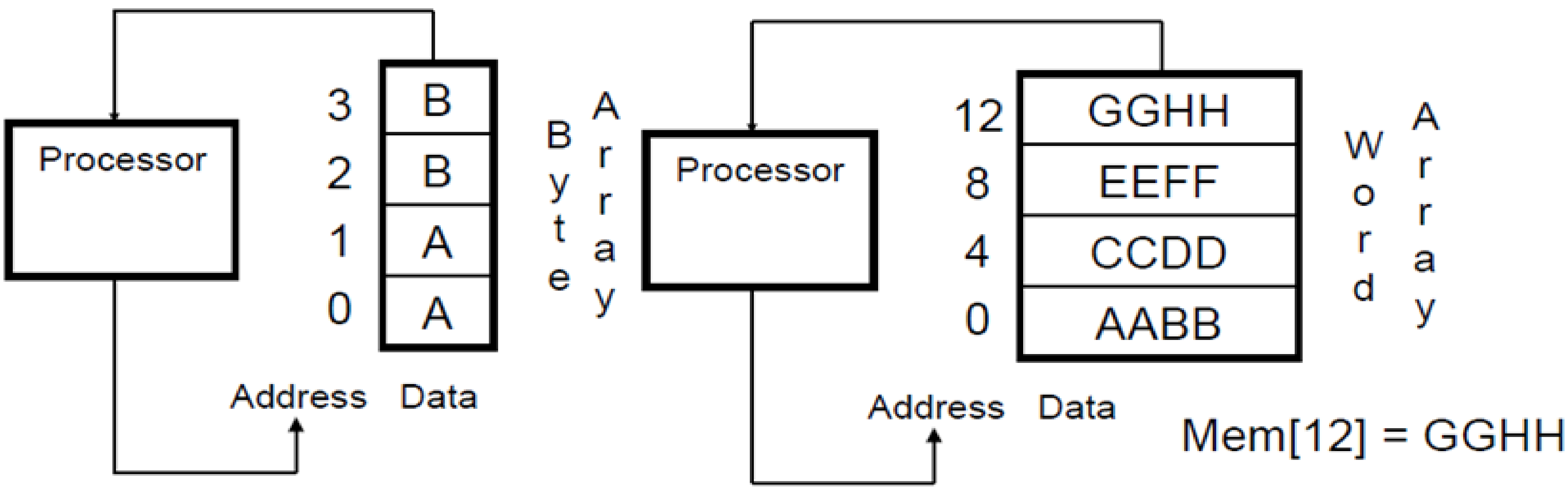
- Memory can be viewed as a large, single-dimensional array



- Data transfer instructions: **lw** and **sw**

Memory address space

Sequential addresses	Byte array	Word array
Start at	0	0
Up to the top of memory	2^{32} = 4 GB	2^{30} = 1 GW
Step by	1 byte	4 bytes



Memory vs. registers

- Memory is much larger
  - Up to 1GW compared to 32 W (32 registers of 1W)
- Registers are much faster
  - At least 100 times
- Using main memory for manipulating operands would be agonizingly slow
- So, we:
  - Transfer data from memory into registers when needed
  - Spill registers back if needed

How to access memory?

- A is an array of 100 words.
- The compiler knows two variables g and h, which are associated respectively with $s1 and $s2.
- Base address of array A is in $s3.

- Compile the c assignment statement below:

```
g = h + A[8];
```

Answer

- Copy the value of element A[8] into a register $t0
  - Determine the address of element A[8]
    - Base address + offset
  - Move the value of the memory unit at that address into $t0

```
lw    $t0,8($s3) # Temporary reg $t0 gets A[8]
```

instruction

Offset
Should be 32… why?

base

- Perform the addition and save the result into g (represented by $s1)

```
add   $s1,$s2,$t0 # g = h + A[8]
```

An example

- Compile c assignment statement below using lw and sw.
  - A is an array of words.
  - Memory is an array of bytes.
  - The base address of A is in $s3.
  - Variable h is associated with $s2.

```
A[12] = h + A[8];
```

- Determine the offset of A[12]
  - A. 12
  - B. 3
  - C. 48
  - D. 0

MIPS code

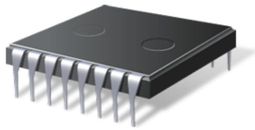
- Load the value of A[8] into a register and perform addition

```
lw    $t0,32($s3)   # Temporary reg $t0 gets A[8]
add   $t0,$s2,$t0   # Temporary reg $t0 gets h + A[8]
```

- Save the addition result into A[12].

```
sw    $t0,48($s3)   # Stores h + A[8] back into A[12]
```

Understand use of registers



- Compilers try to keep the most frequently used variables in registers and places the rest in memory.
- Spilling registers: the process of putting less commonly used variables (or those needed later) into memory.
- Data is more useful in registers
  - Data in register is both faster to access and simpler to use with less energy demand.
  - Data usually needs to be moved into a register before use.
  - MIPS data transfer instruction only reads one operand or writes one operand, without operating on it.

Constant or immediate operands

- Many times a program will use a constant in an operation

- One simple solution

```
lw $t0, AddrConstant4($s1)   # $t0 = constant 4
add $s3,$s3,$t0              # $s3 = $s3 + $t0 ($t0 == 4)
```

- A more efficient solution

```
addi    $s3,$s3,4            # $s3 = $s3 + 4
```

Design principle 3: Make the common case fast.

Check yourself

- For a given function, which programming language likely takes the most lines of code? Which programming language likely takes the least lines of code?
  - A. Java
  - B. C
  - C. MIPS assembly language

Check yourself

- Given the importance of registers, what is the rate of increase in the number of registers in a chip over time?
  - A. Very fast: they increase as fast as Moore's law, which predicts doubling the number of transistors on a chip every 18 months or two years.
  - B. Very slow: there is inertia in instruction set architecture, so the number of registers increase only as fast as new instruction sets become viable.