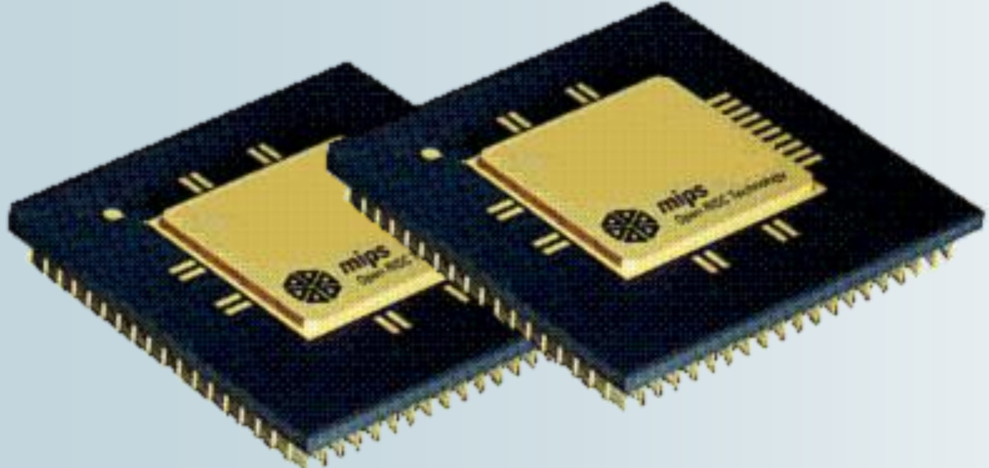
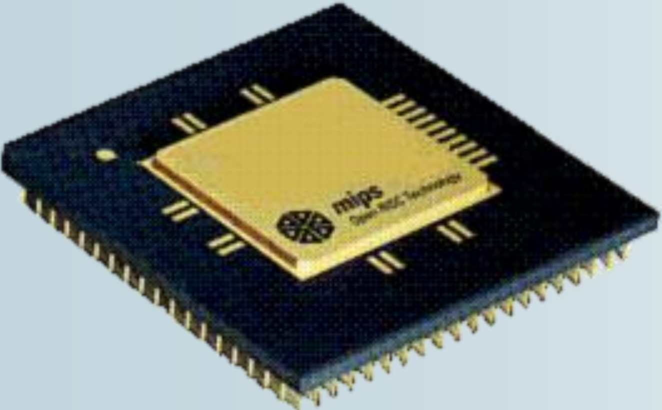


MIPS



Tim Langens,
Jef Neefs,
Elio Struyf,
Luc Verstrepen

Walkthrough

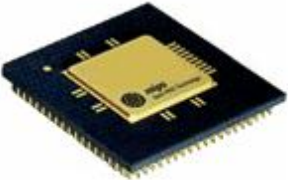
- Introduction
- ISA
  - MIPS description
  - Accessing Data
  - Subroutine Linkage and Stacks
  - Input and Output
- Compilation process
- Datapath and Control
  - Basics of the microarchitecture
  - The MIPS Microarchitecture

Walkthrough

- Introduction
- ISA
  - MIPS description
  - Accessing Data
  - Subroutine Linkage and Stacks
  - Input and Output
- Compilation process
- Datapath and Control
  - Basics of the microarchitecture
  - The MIPS Microarchitecture

Introduction

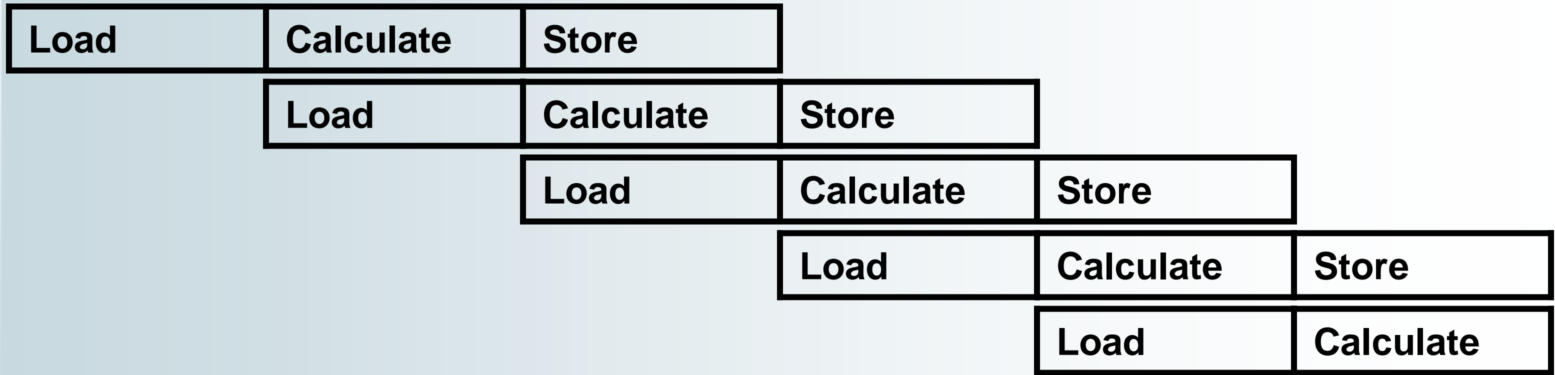
- MIPS (Microprocessor without Interlocked Pipeline Stages)
- Based on the RISC
- All instructions should be able to complete in one cycle
- + faster
- - less instructions

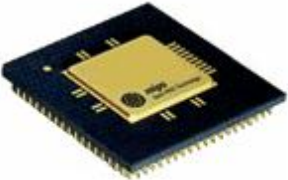


Introduction

No Pipelines (CISC):

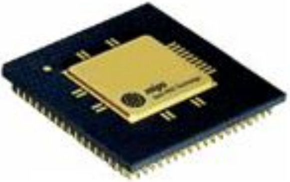
Pipelines (RISC):





Introduction

- Sony Playstation PSX
- Sony PlayStation Portable (PSP)



Walkthrough

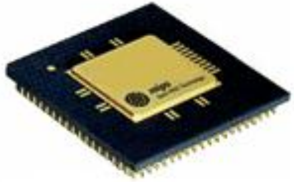
- History and Usage
- ISA
  - MIPS description
  - Accessing Data
  - Subroutine Linkage and Stacks
  - Input and Output
- Compilation process
- Datapath and Control
  - Basics of the microarchitecture
  - The MIPS Microarchitecture

MIPS description

- Memory
- Instruction Set
- Instruction formats
- Data types

MIPS description

- Memory
- Instruction Set
- Instruction formats
- Data types



MIPS memory

- 32-bit
- Fixed memory allocation

0x00400000	Text segment	program instructions
0x10000000	Data segment	
0x7FFFFFFF	decreasing addresses	Stack segment

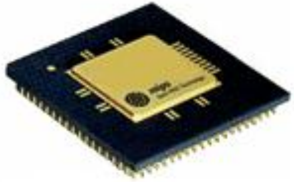
- Little and Big Endian

MIPS description

- Memory
- Instruction Set
- Instruction formats
- Data types

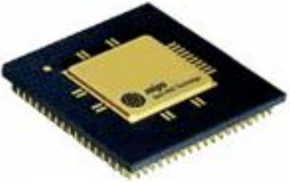
MIPS instruction set

- 32 general purpose registers
- Backwards compatible
- 32 bit wide (even on 64-bit processors)
- Load-store machine
- 3 categories of instructions
  - Load-store
  - Arithmetic and logical
  - Jump and branch



Registers

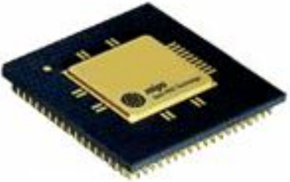
Number	Name	Purpose
$0	$0	Always 0
$1	$at	The *Assembler Temporary* used by the assembler in expanding pseudo-ops.
$2-$3	$v0-$v1	These registers contain the *Returned Value* of a subroutine; if the value is 1 word only $v0 is significant.
$4-$7	$a0-$a3	The *Argument* registers, these registers contain the first 4 argument values for a subroutine call.
$8-$15, $24,$25	$t0-$t9	The *Temporary Registers*.
$16-$23	$s0-$s7	The *Saved Registers*.
$26-$27	$k0-$k1	The *Kernel Reserved registers*. DO NOT USE.
$28	$gp	The *Globals Pointer* used for addressing static global variables.
$29	$sp	The *Stack Pointer*.
$30	$fp	The *Frame Pointer*, if needed
$31	$ra	The *Return Address* in a subroutine call.



Load-store

- Load data from memory into a register
- store register contents in memory.
- Example:

```
lw  $t0, num1   #load word in num1 into $t0
sw  $t0, num2   #store word in $t0 into num2
li  $v0, 4      #load immediate value 4 into $v0
```



Arithmetic and Logical

- perform an operation on data in 2 registers and store the result in a 3rd register.
- Example:

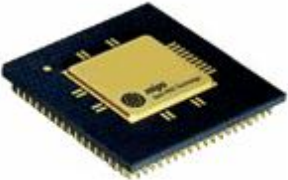
```
add   $t0, $t3, $t4     # $t0 := $t3 + $t4
```

Jump and Branch

- alter the PC to control flow of the program, producing the effects of IF statements and loops

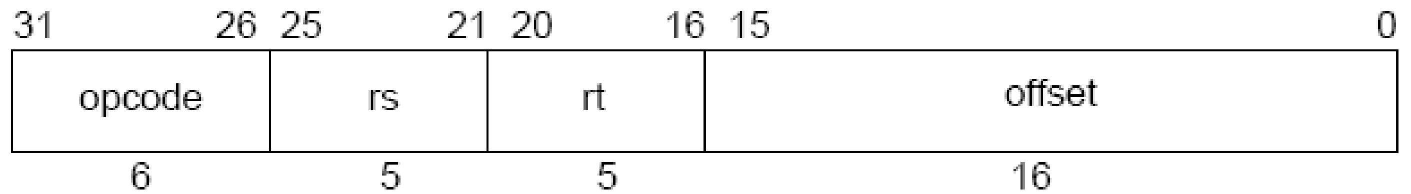
MIPS description

- Memory
- Instruction Set
- Instruction formats
- Data types

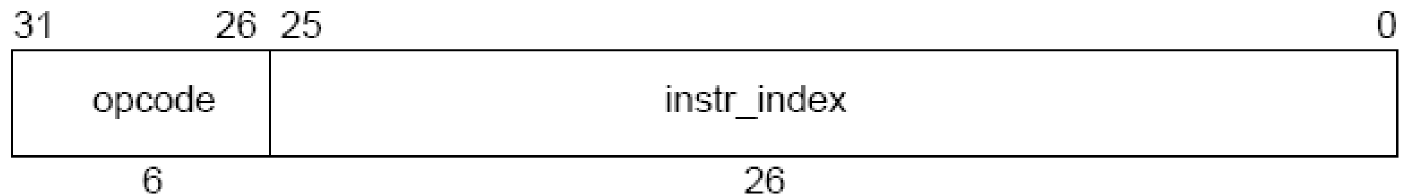


Instruction Formats

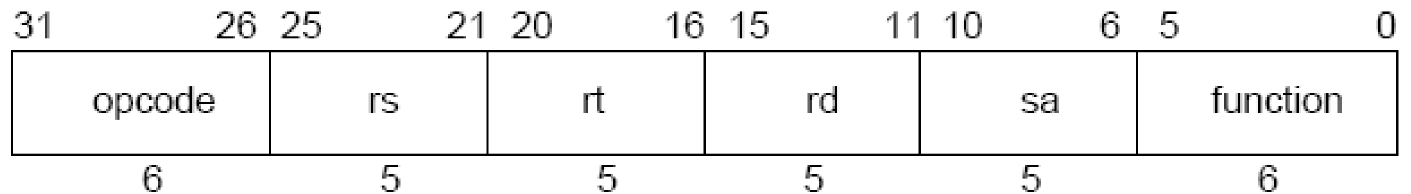
I-Type (Immediate).



J-Type (Jump).



R-Type (Register).

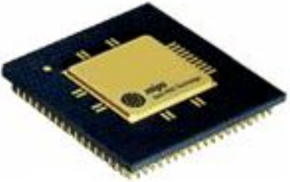


MIPS description

- Memory
- Instruction Set
- Instruction formats
- Data types

MIPS data types

- Integer data types:
  - Byte: 8 bits
  - Half-words: 16 bits
  - Words: 32 bits
  - Double-words: 64 bits (not in basic MIPS I)
- Floating Point:
  - 32-bit single precision
  - 64-bit double precision



Walkthrough

- History and Usage
- ISA
  - MIPS description
  - Accessing Data
  - Subroutine Linkage and Stacks
  - Input and Output
- Compilation process
- Datapath and Control
  - Basics of the microarchitecture
  - The MIPS Microarchitecture

Accessing Data

There are a number of ways to refer to data, either as source operands, or as destination locations for storage.

- Immediate data
- Register access
- Memory access

Immediate Data

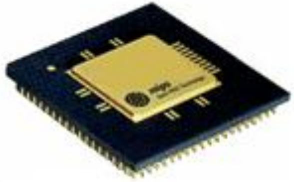
- Source operands can be immediate data (also called constants).
- A constant value is encoded directly in the machine language instruction.
- <16 bits: encoded in lower 16 bits of instruction
- >16 bits: two instructions are needed, a load upper immediate + or instruction

Register Access

- Registers may be used as sources and destination of data.
- MIPS provides 32 registers with a size of 32 bits.
- Access to registers is much faster than accessing data in memory.
- Register n is designated by $n.

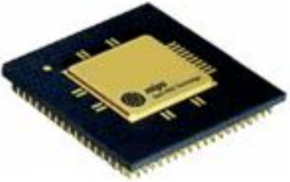
Memory Access

- With 32-bit addresses the user can access data in the memory up to 4 GB.
- Large amounts of storage compared to registers, but access is slower.
- MIPS restricts memory access to load and store instructions (like other RISC processors and most supercomputers).



Addressing Modes

Name	Format	Address Computation
Direct	Label	Address of label
Indexed	Label(register)	Address of label + contents of register
Indirect	(register)	Contents of register
Base	Imm(register)	Contents of register + immediate



Walkthrough

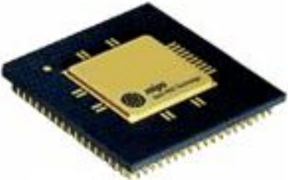
- History and Usage
- ISA
  - MIPS description
  - Pseudo-Ops
  - Accessing Data
  - Subroutine Linkage and Stacks
  - Input and Output
- Compilation process
- Datapath and Control
  - Basics of the microarchitecture
  - The MIPS Microarchitecture

MIPS function call and return

- After calling a function (subroutine, procedure) the program will return to the next instruction after the call. This is the return address.
- MIPS uses register 31 as the return address register.
- Instructions: jal + jr $ra

MIPS function call and return

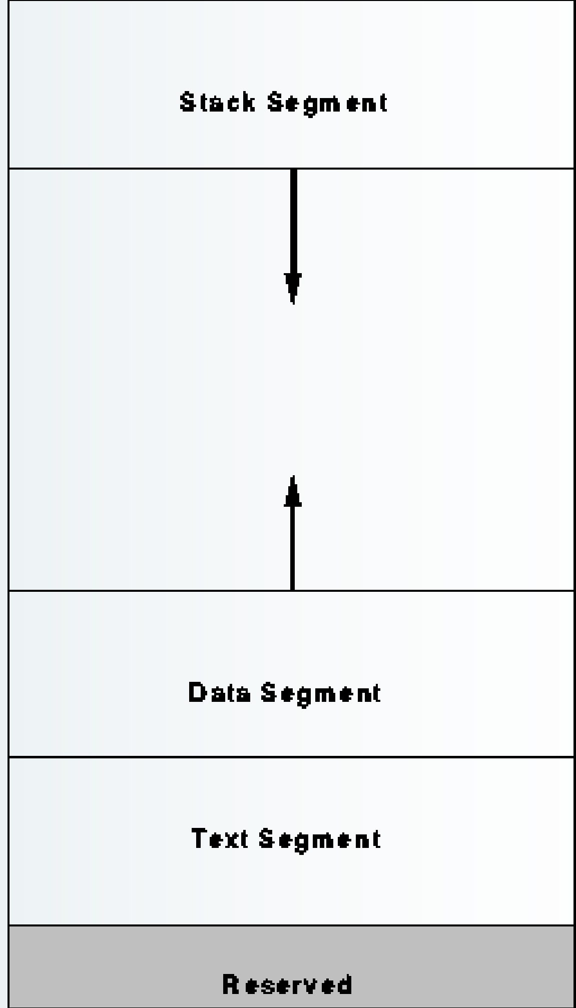
- These instructions work well for main programs that call functions sequentially.
- For example: print an integer, then print an end-of-line character.
- When a function calls another function, we have to save return addresses in a systematic way -> <u>use of a stack</u>

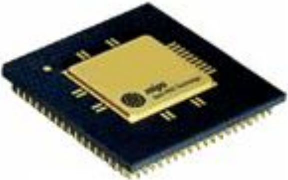


Principles of the stack

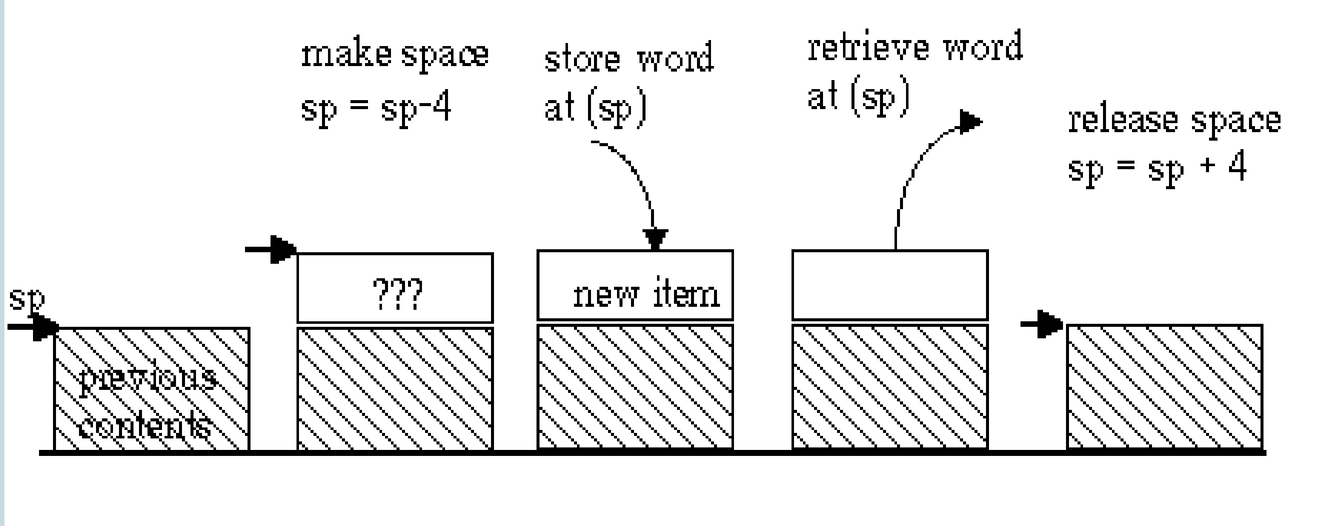
The address space of a program consists of:

- Text segment
- Static data
- Dynamic data
- Stack segment



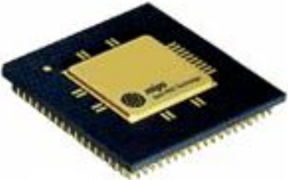


Principles of the stack




```
sub   $sp, 4       # PUSH on the stack
sw    $ra, ($sp)  # our return address

lw    $ra, ($sp)  # POP our return address from the stack
add   $sp, 4       # top of stack back as before the PUSH
```



MIPS calling convention

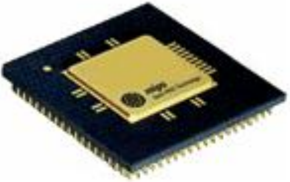
The stack frame for a subroutine may contain space for:

- Local data storage
- Padding
- Return address
- Values of saved registers
- Arguments passed to new subroutines

Previous Stack Frame

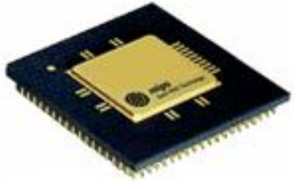
sp+fsize →
sp+fsize-4 → local data m-1
sp+… →
local data 0
empty
return address
saved reg k-1
saved reg 0
arg n-1
…
sp+16 → arg 4
sp+12 → (arg3)
sp+8 → (arg2)
sp+4 → (arg1)
sp → (arg0)

Current Stack Frame



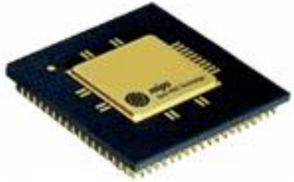
Walkthrough

- History and Usage
- ISA
  - MIPS description
  - Pseudo-Ops
  - Accessing Data
  - Subroutine Linkage and Stacks
  - Input and Output
- Compilation process
- Datapath and Control
  - Basics of the microarchitecture
  - The MIPS Microarchitecture



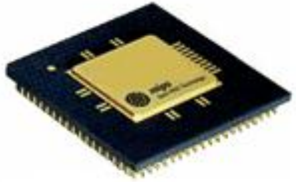
Input and output

Service	Call code	Arguments	Result
Print integer	1	$a0 = integer	Integer printed
Print float	2	$f12 = float	Float printed
Print double	3	$f12 = double	Double printed
Print string	4	$a0 = address of string	String printed



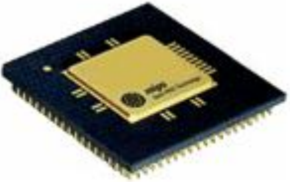
Input and output

Service	Call code	Arguments	Result
Read integer	5	none	$v0 holds integer
Read float	6	none	$f0 holds float
Read double	7	none	$f0 holds double
Read string	8	$a0 = address to store,$a1 = length limit	Characters stored



Input and output

Service	Call code	Arguments	Result
Sbrk	9	$a0 = amount	$v0 holds address
Exit	10	none	Ends the program

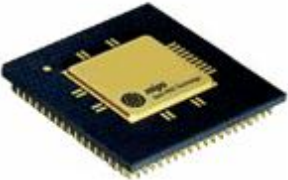


Walkthrough

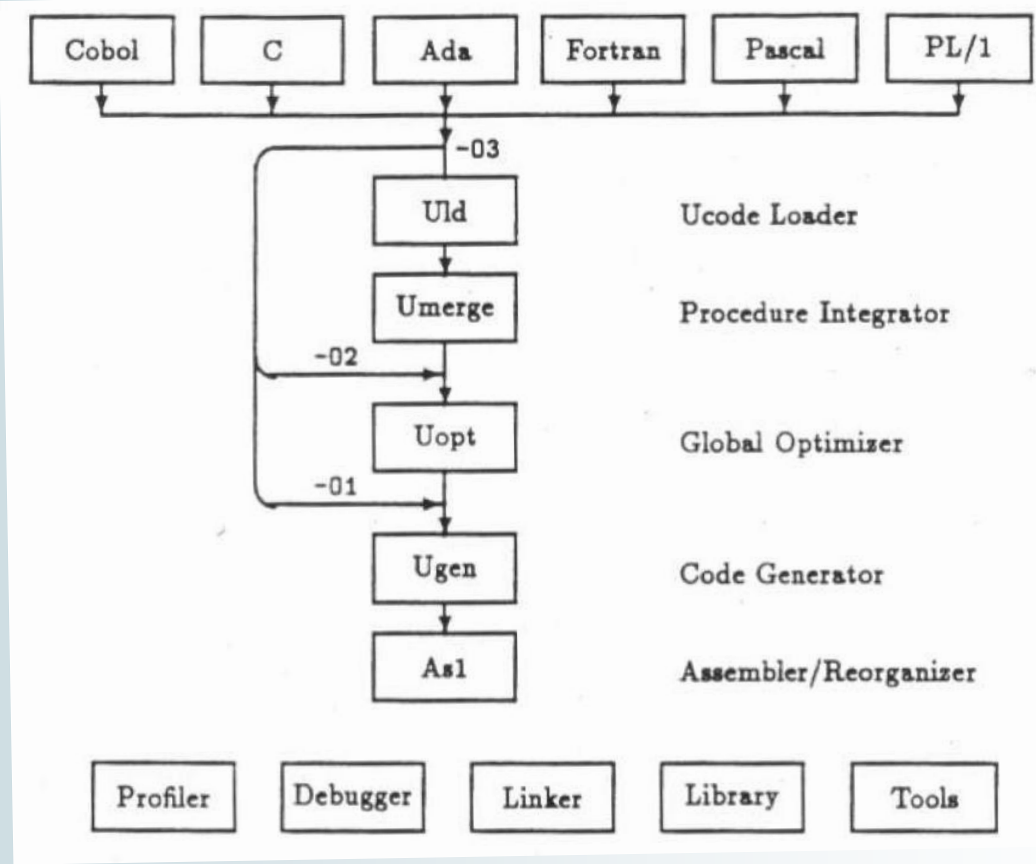
- History and Usage
- ISA
  - MIPS description
  - Pseudo-Ops
  - Accessing Data
  - Subroutine Linkage and Stacks
  - Input and Output
- Compilation process
- Datapath and Control
  - Basics of the microarchitecture
  - The MIPS Microarchitecture

C code

```
#include <stdio.h>
int main (int argc, char *argv[])
{
    int I;
    int sum = 0;
    for (I=0; I<=100; I++) sum += I * I;
            printf (“The sum 0..100=%d\n”,sum);
}
```

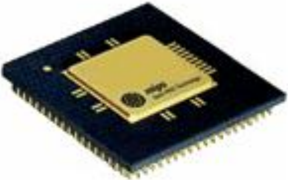


Steps of Compilation



Ucode or Unicode

- Intermediate language: Translates the source code of a program into a form more suitable for the machine.
- Ucode will use a stack
- Is a read only storage for instructions

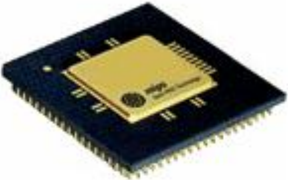


Ucode or Unicode

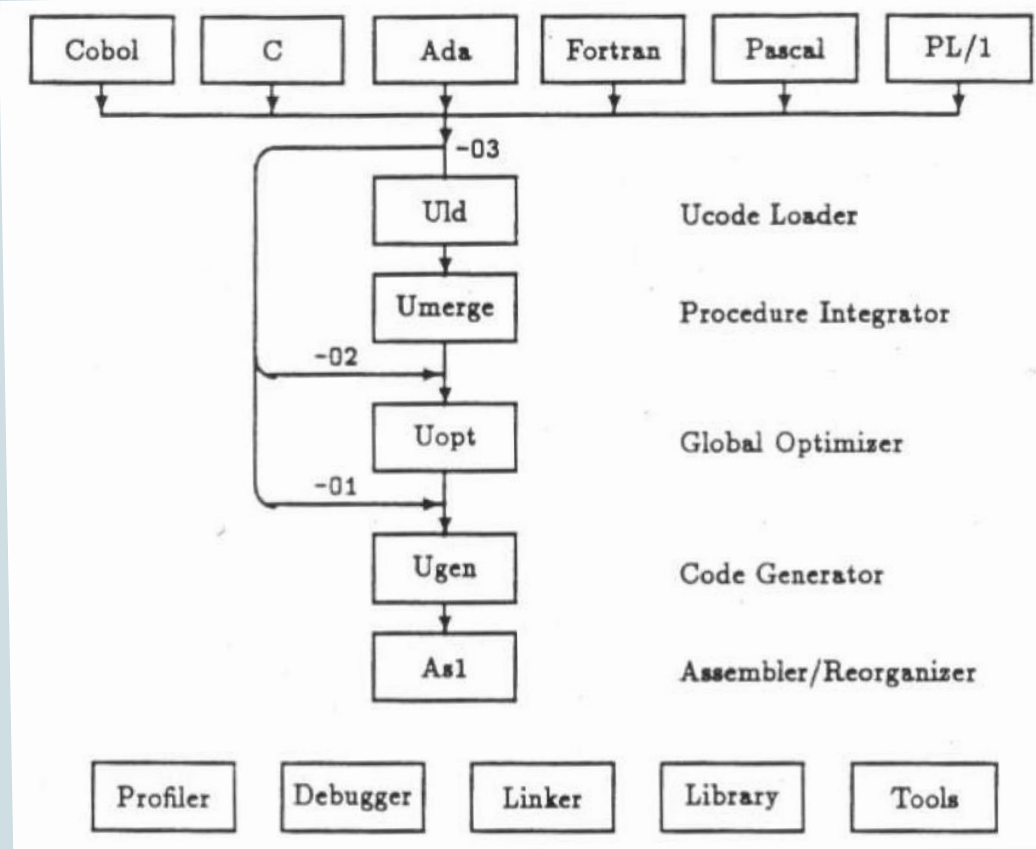
```
.text
.align    2
.globl    main
.ent      main 2
main:
    subu$sp, 32
    sw    $31, 20($sp)
    sd    $4, 32($sp)
    sw    $0, 24($sp)
    sw    $0, 28($sp)
loop:
    lw    $14, 28($sp)
    mul   $15, $14, $14
    lw    $24, 24($sp) addu        $25, $24, $15
    sw    $8, 28($sp)
```

Ucode or Unicode

```
    ble     $8, 100, loop
    la      $4, str
    lw      $5, 24($sp)
    jal     printf
    move            $2, $0
    lw      $31, 20($sp)
    addu$sp, 32
    j       $31
    .end main
    .data
    .align          0
str:
    .asciiz “The sum 0..100=%d\n”
```



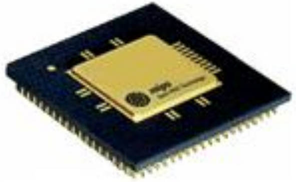
Optimization levels



Assembler process

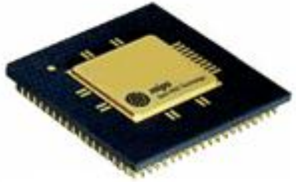
Two steps are taken for the process

1. Associate memory locations with labels.
   - Record the name and position of each label in the symbol table.
2. Translate each statement to the machine code.
   - Three different types of machine code format.



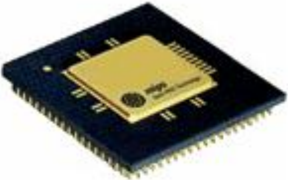
Assembler process

```
addiu   sp, sp,-32
sw      ra, 20(sp)
sw      a0, 32(sp)
sw      a1, 36(sp)
sw      zero, 24(sp)
sw      zero, 28(sp)
lw      t6, 28(sp)
lw      t8, 24(sp)
multu   t6, t6
addiu   t0, t6, 1
slti    at, t0, 101
sw      t0, 28(sp)
```



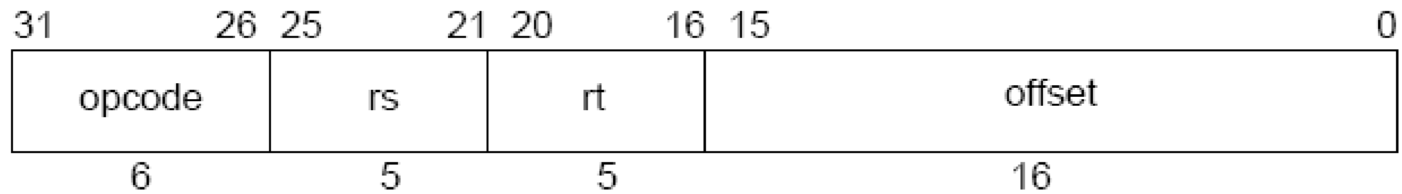
Assembler process

```
mflo    t7
addu    t9, t8, t7
bne     at, zero, -9
sw      t9, 24(sp)
lui     a0,4096
lw      a1, 24(sp)
jal     1048812
addiu   a0, a0, 1072
lw      ra, 20(sp)
addiu   sp, sp, 32
jr      ra
move    v0, zero
```

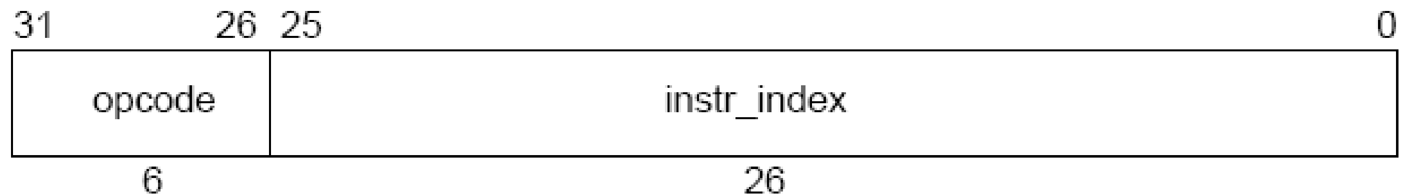


Machine code format

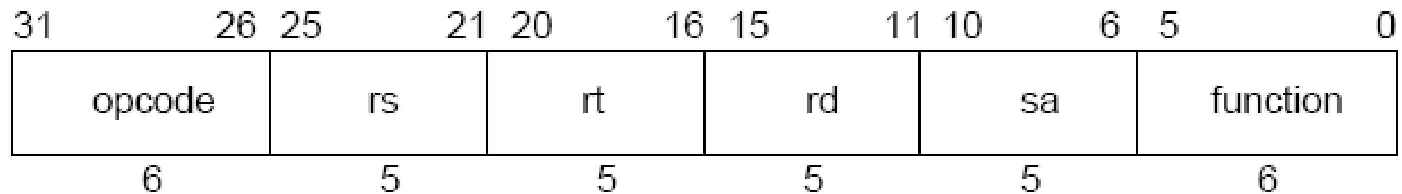
I-Type (Immediate).

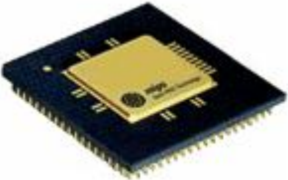


J-Type (Jump).

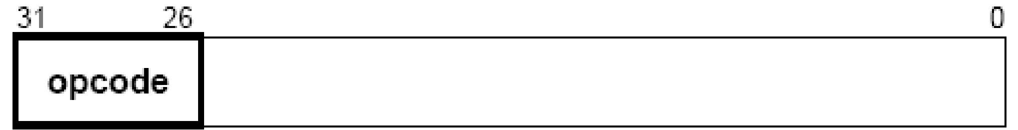


R-Type (Register).



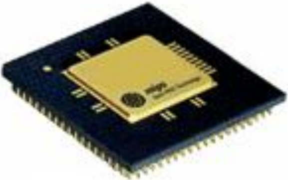


Opcode



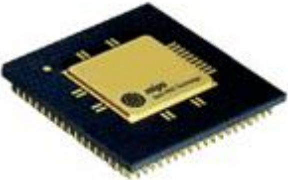
opcode bits 28..26 Instructions encoded by **opcode** field.

bits 31..29		0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	000	*SPECIAL* δ	*REGIMM* δ	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	010	*COP0* δ,π	*COP1* δ,π	*COP2* δ,π	*COP3* δ,π,κ	*	*	*	*
3	011	*	*	*	*	*	*	*	*
4	100	LB	LH	LWL	LW	LBU	LHU	LWR	*
5	101	SB	SH	SWL	SW	*	*	SWR	*
6	110	*	LWC1 π	LWC2 π	LWC3 π,κ	*	*	*	*
7	111	*	SWC1 π	SWC2 π	SWC3 π,κ	*	*	*	*



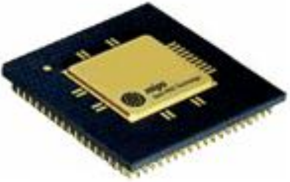
Translated into machine code

00100111101111011111111111100000
10101111101111110000000000010100
10101111101001000000000000100000
10101111101001010000000000100100
10101111101000000000000000011000
10101111101000000000000000011100
10001111101011100000000000011100
10001111101110000000000000011000
00000001110011110000000000011001
00100101110010000000000000000001
00101001000000010000000001100101
10101111101010000000000000011100



Translated into machine code

00000000000000000111100000010010
00000011000011111100100000100001
00010100001000001111111111110111
10101111101110010000000000011000
00111100000001000001000000000000
10001111101001010000000000011000
00001100000100000000000011101100
00100100100001000000010000110000
10001111101111110000000000010100
00100111101111010000000000100000
00000011111000000000000000001000
00000000000000000001000000100001

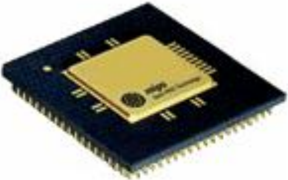


Walkthrough

- History and Usage
- ISA
  - MIPS description
  - Pseudo-Ops
  - Accessing Data
  - Subroutine Linkage and Stacks
  - Input and Output
- Compilation process
- Datapath and Control
  - Basics of the microarchitecture
  - The MIPS Microarchitecture

Basics of the microarchitecture

- Fetch next instruction from the memory
- Decode the Opcode
- Read the operands
- Execute the instruction and store the results
- Start all over



Basics of the microarchitecture

ALU

Registers

Control Unit

Data Path

Control Section

System Bus

Walkthrough

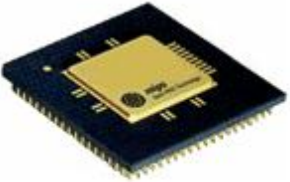
- History and Usage
- ISA
  - MIPS description
  - Pseudo-Ops
  - Accessing Data
  - Subroutine Linkage and Stacks
  - Input and Output
- Compilation process
- Datapath and Control
  - Basics of the microarchitecture
  - The MIPS Microarchitecture

The MIPS microarchitecture

- Datapath Design and Implementation
- Single-Cycle and Multicycle Datapaths
- Control
- Exception Handling

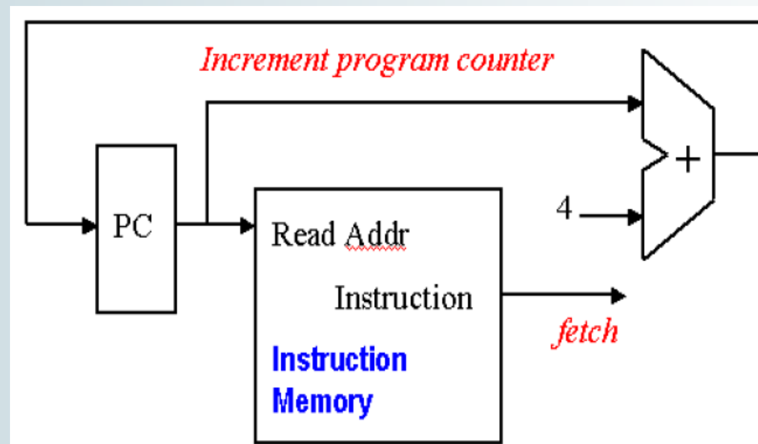
The MIPS microarchitecture

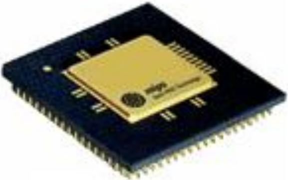
- Datapath Design and Implementation
- Single-Cycle and Multicycle Datapaths
- Finite state Control
- Microprogrammed Control



Datapath

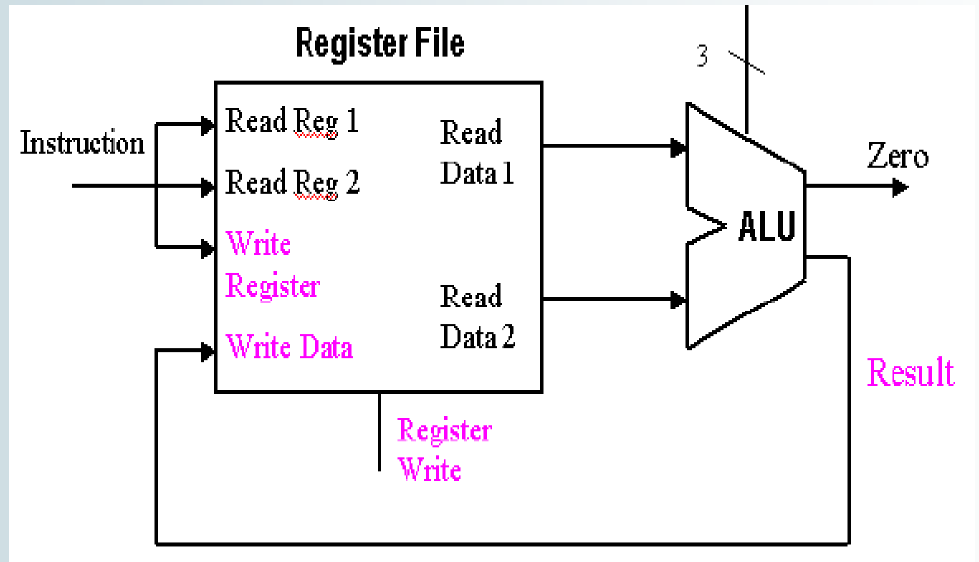
- The datapath is the power of a processor
- Simple datapath components include
  - *Memory* (stores the current instruction)
  - *PC* or program counter (stores the address of current instruction)
  - A*LU* (executes current instruction)

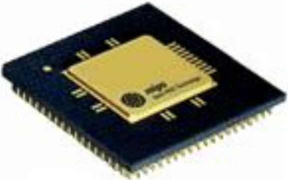




Datapath: Register instruction

- The ALU accepts its input from the DataRead ports of the register file
- The register file is written to by the ALUresult output of the ALU and the RegWrite signal.





Datapath: Load/store instruction

Instruction

Read register 1

Read register 2

Write register

Write data

Registers

Read data 1

Read data 2

RegWrite

16

Sign extend

32

3

ALU operation

ALU

Zero

ALU result

MemWrite

Address

Write data

Data memory

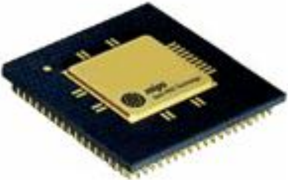
Read data

MemRead

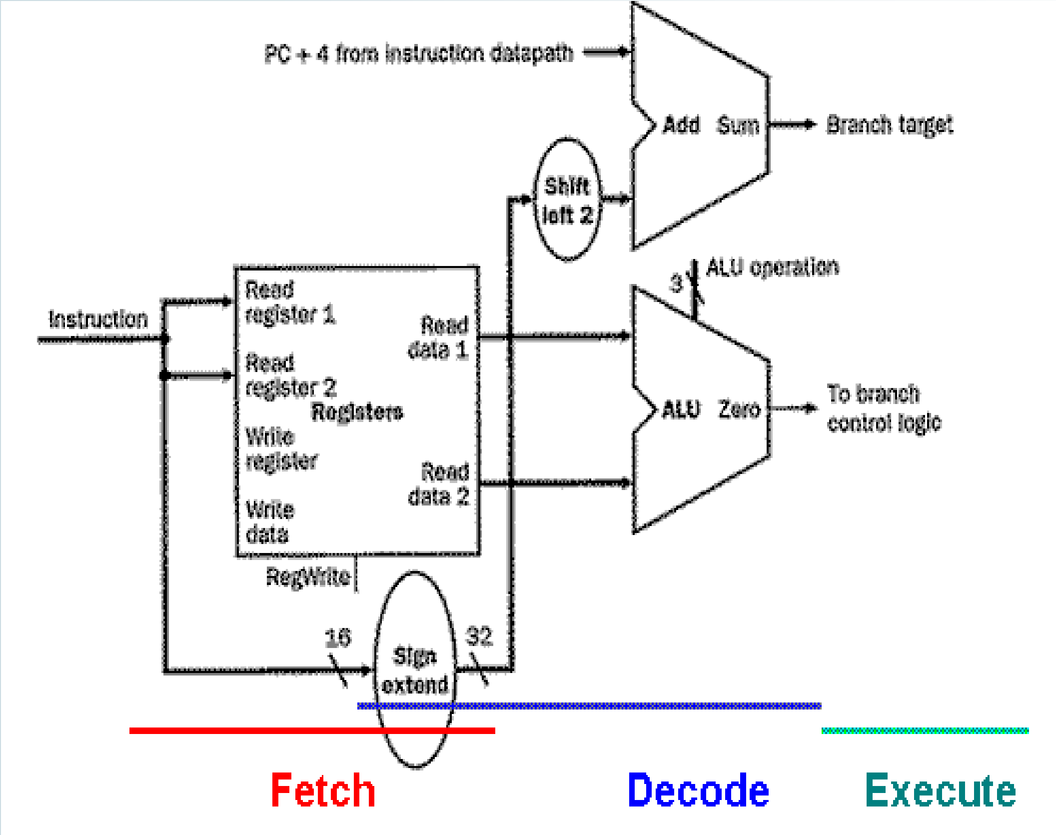
Fetch

Decode

Execute



Datapath: Branch/Jump

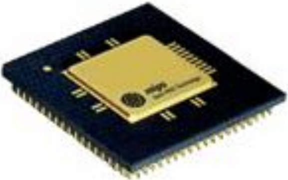


The MIPS microarchitecture

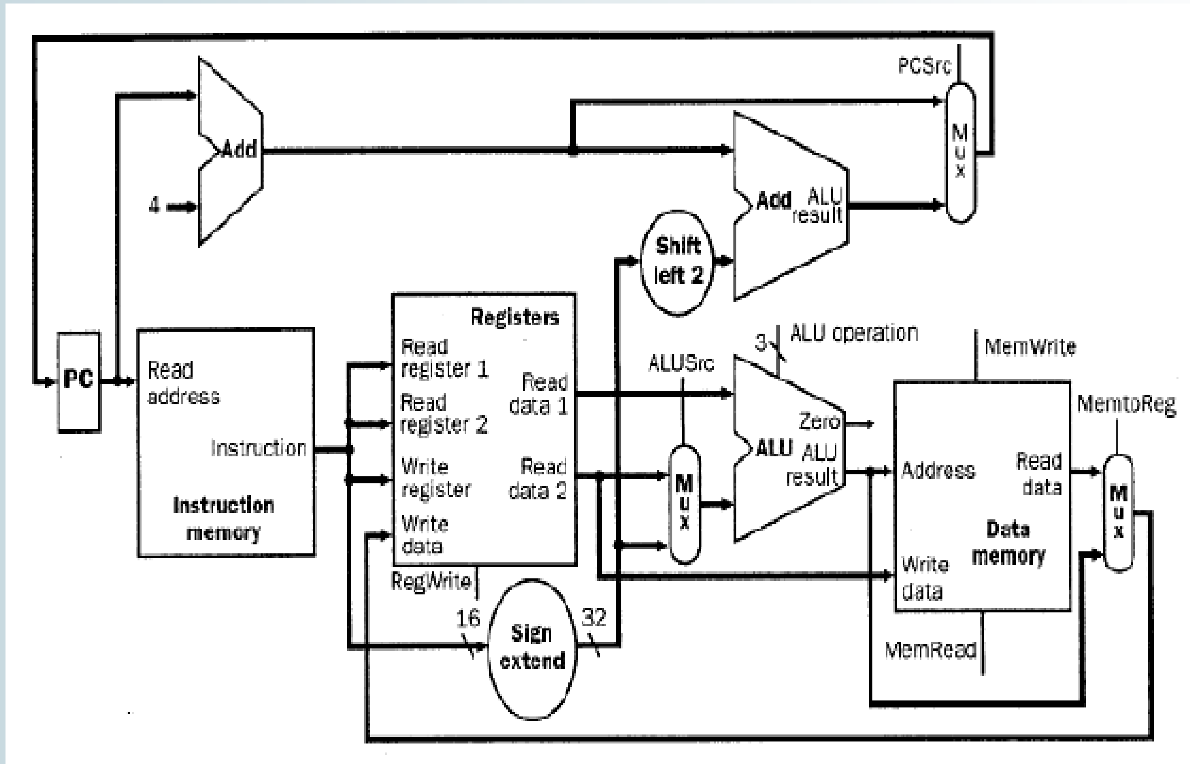
- Datapath Design and Implementation
- Single-Cycle and Multicycle Datapaths
- Finite state Control
- Microprogrammed Control

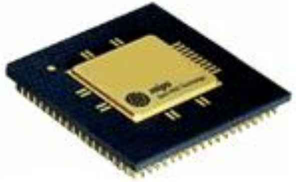
Single cycle datapath

- Executes in one cycle all instructions that the datapath is designed to implement
- Impacts CPI in a beneficial way
  - CPI = 1 cycle for all instructions
- Find similarities among instruction types



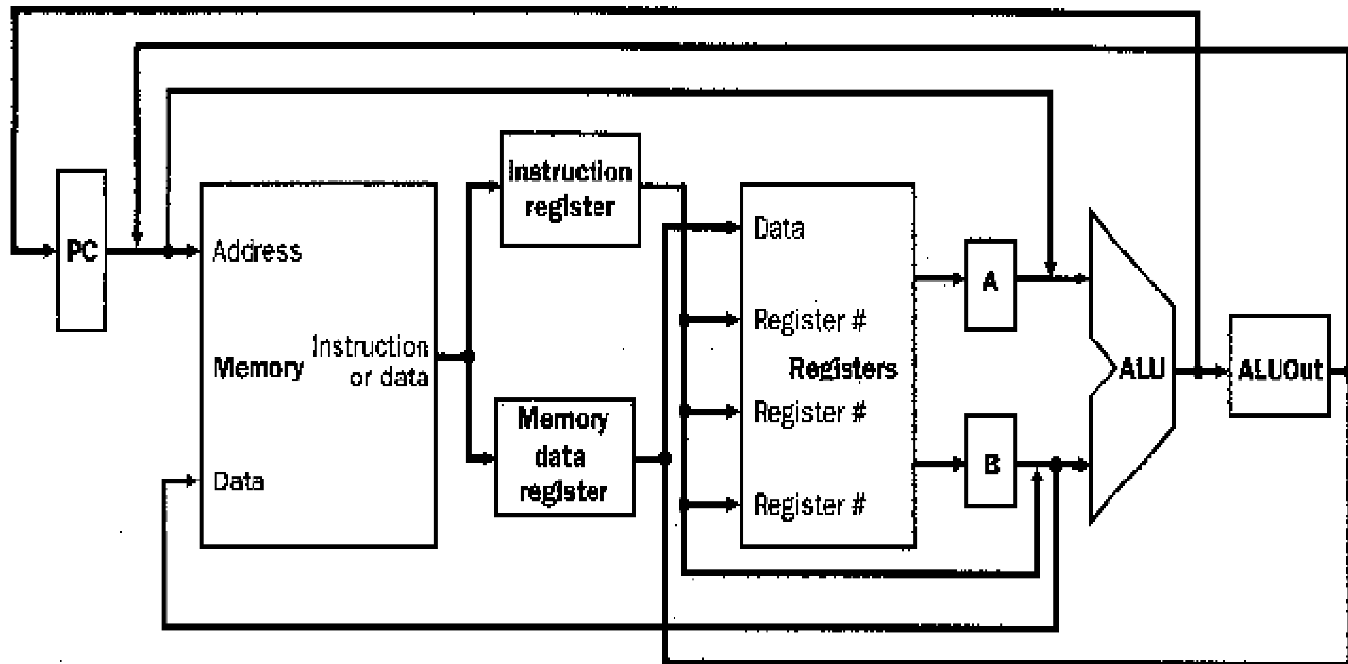
Single cycle datapath





Multi cycle datapath

- Each functional unit can be used more than once
- Each instruction step takes one cycle, so different instructions have different execution times

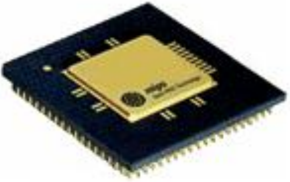


The MIPS microarchitecture

- Datapath Design and Implementation
- Single-Cycle and Multicycle Datapaths
- Finite state Control
- Microprogrammed Control

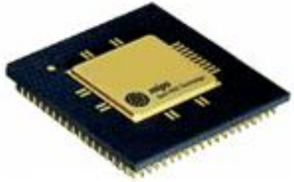
Control

- *FSM* predicts actions appropriate for datapath's next computational step
- *Microprogramming*, uses a programmatic representation to implement control



FSM and Multicycle Datapath Performance

- Each state is one clock cycle
- The critical path for each instruction type
  - Load: 5 states
  - Store: 4 states
  - R-format ALU instructions: 4 states
  - Branch: 3 states
  - Jump: 3 states
- CPI = [#Loads · 5 + #Stores · 4 + #ALU-instr's · 4 + #Branches · 3 + #Jumps · 3] / (Total Number of Instructions)



Implementation of Finite-State Control

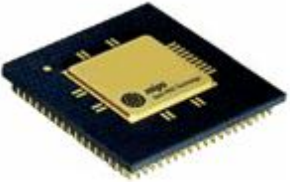
- Read-only memory (ROM)
- Programmable logic array (PLA)
- Combinatorial logic implements the transition function and a state register stores the current state of the machine
- Inputs are the IR opcode bits
- Outputs are the various datapath control signals

The MIPS microarchitecture

- Datapath Design and Implementation
- Single-Cycle and Multicycle Datapaths
- Finite state Control
- Microprogrammed control

Microprogrammed Control

- Abstractions from the programming language.
- Microinstruction Format:
  - Initialize what structure and content the microinstructions fields must have.
- Sequencing Mechanism:
  - Determine which instruction will be used next.
- Exception Handling:
  - Determine what actions must be taken when there are errors.



Exception Handling

There are two types of exceptions

1. Exception is an event from within the processor, such as arithmetic overflow.
2. Interrupt is an event that causes an unexpected change in control flow.
   - From outside the processor

Hardware support for exception handling

You need two registers for the exception handling

- Exception counter: 32-bit register holds the address of the exception-causing instruction.
- *Cause*: 32-bit register contains a binary code that describes the cause or type of exception.

Questions?