

What were in CS120B

- ❑ **General purpose processors (GPP), application specific processors (ASIP), single purpose processor (SPP)**
- ❑ **How to build SPP**
 - **State machine implementation**
- ❑ **Basics about GPP**
 - **Fetch, decode, read operands, execute, store results**
 - **Actions in each cycle**
 - **Sample instruction set**
 - **8051 – 8-bit processor**
- ❑ **Memory hierarch**
- ❑ **Bus architecture, interrupts**
- ❑ **A digital camera example**

What we have in CS161

- ❑ There will be overlaps. But CS161 goes into more details that are up-to-date.
- ❑ A specific instruction set architecture (Chapter 2)
- ~~❑ Arithmetic and how to build an ALU (Chapter 3)~~
- ❑ Performance issues (Chapter 4)
- ❑ Constructing a processor to execute our instructions, the data path and control (Chapter 5)
- ❑ Pipelining to improve performance (Chapter 6)
- ❑ Memory: caches and virtual memory (Chapter 7)
- ~~❑ Peripherals (Chapter 8)~~

Chapter 1 – Computer Abstractions and Technology

Introduction

Intel Pentium 4 Northwood

Buffer Allocation & Register Rename

Instruction Queue (for less critical fields of the uOps)

General Instruction Address Queue & Memory Instruction Address Queue (queues register entries and latency fields of the uOps for scheduling)

Floating Point, MMX, SSE2 Renamed Register File
128 entries of 128 bit.

uOp Schedulers

FP Move Scheduler:
(8x8 dependency matrix)

Parallel (Matrix) Scheduler for the two double pumped ALU's

General Floating Point and Slow Integer Scheduler:
(8x8 dependency matrix)

Load / Store uOp Scheduler:
(8x8 dependency matrix)

Load / Store Linear Address Collision History Table

Integer Execution Core

- (1) uOp Dispatch unit & Replay Buffer
Dispatches up to 6 uOps / cycle
- (2) Integer Renamed Register File
128 entries of 32 bit + 6 status flags
12 read ports and six write ports
- (3) Databus switch & Bypasses to and from the Integer Register File.
- (4) Flags, Write Back
- (5) Double Pumped ALU 0
- (6) Double Pumped ALU 1
- (7) Load Address Generator Unit
- (8) Store Address Generator Unit
- (9) Load Buffer (48 entries)
- (10) Store Buffer (24 entries)

Execution Pipeline Start

Register Alias History Tables (2x126)
Register Alias Tables uOp Queue

Instruction Trace Cache

Micro code Sequencer
Micro code ROM & Flash
Trace Cache Fill Buffers
Distributed Tag comparators
24 bit virtual Tags

Trace Cache Access, next Address Predict

Trace Cache Branch Prediction Table (BTB), 512 entries.
Return Stacks (2x16 entries)
Trace Cache next IP's (2x)
Miscellaneous Tag Data

Instruction Decoder

Up to 4 decoded uOps/cycle out.
(from max. one x86 instr/cycle)
Instructions with more than four are handled by Micro Sequencer
Trace Cache LRU bits
Raw Instruction Bytes in Data TLB, 64 entry fully associative, between threads dual ported (for loads and stores)

Instruction Fetch from L2 cache and Branch Prediction

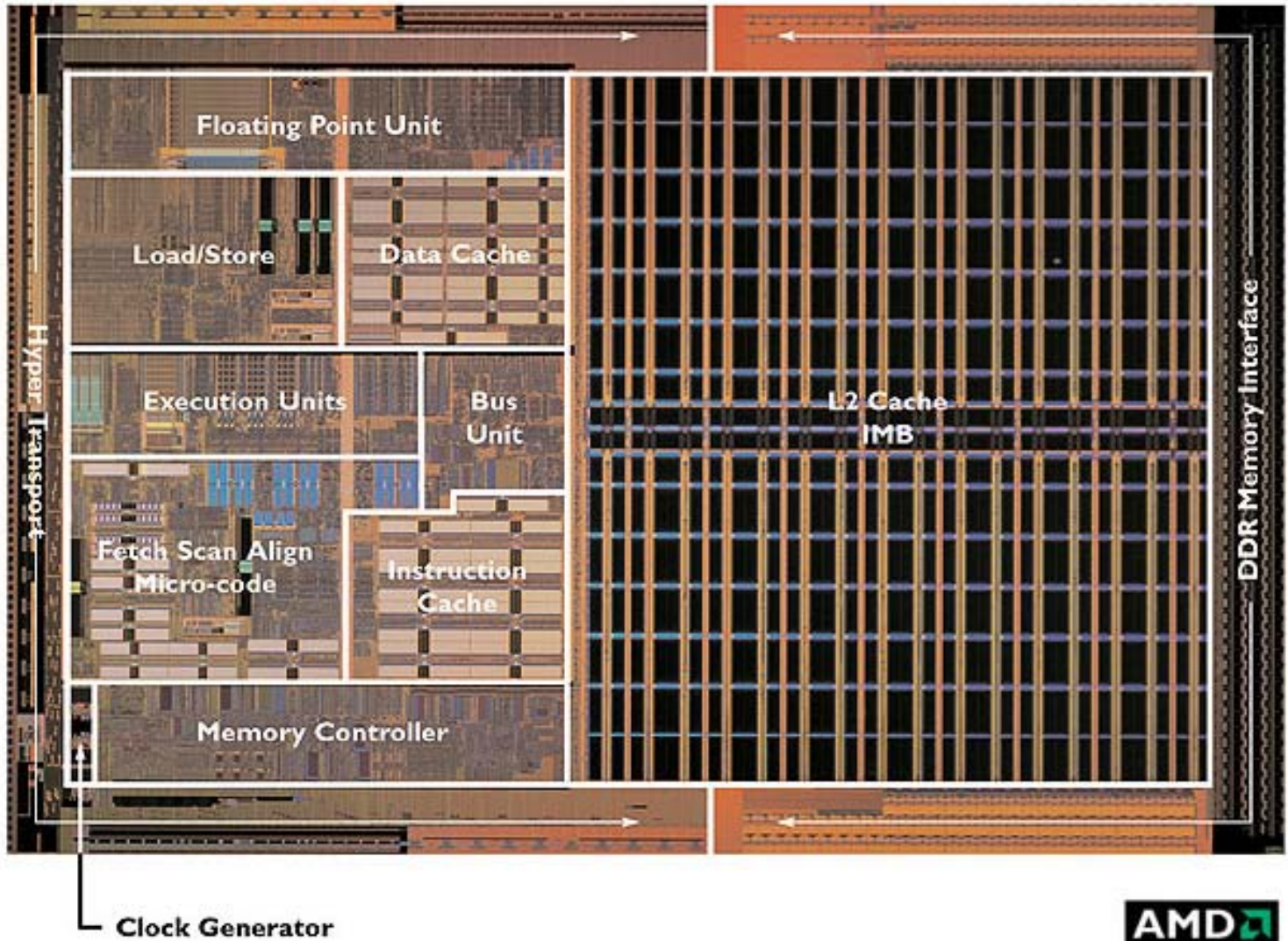
Front End Branch Prediction Tables (BTB), shared, 4096 entries in total
Instruction TLB's 2x64 entry, fully associative for 4k and 4M pages. In: Virtual address [31:12] Out: Physical address [35:12] + 2 page level bits

Front Side Bus Interface, 400..800 MHz



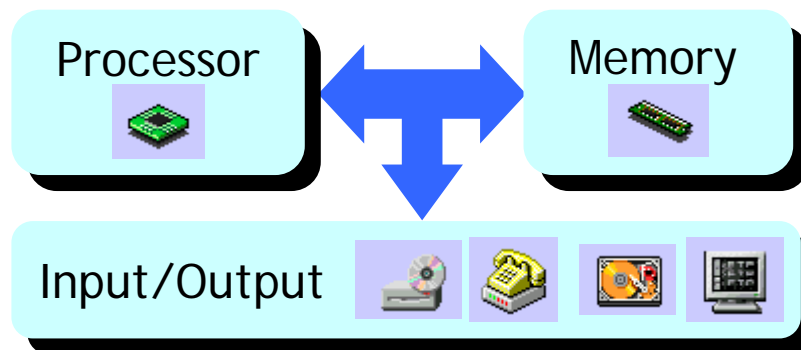
- (11) ROB Reorder Buffer 3x42 entries
- (12) 8 kByte Level 1 Data cache
- (13) Summed Address Index decode and Way Predict
- (14) Cache Line Read / Write Transferbuffers and 256 bit wide bus to and from L2 cache

Introduction



What is computer architecture about

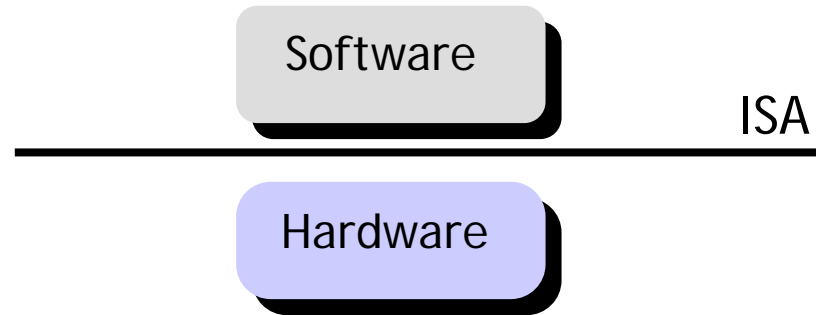
- ❑ **Computer architecture** is the study of building computer systems.



- ❑ **CS161 is roughly split into three parts.**
 - The first third discusses **instruction set architectures**—the bridge between hardware and software.
 - Next, we introduce more advanced processor implementations. The focus is on **pipelining**, which is one of the most important ways to improve performance.
 - Finally, we talk about **memory** systems, **I/O**, and how to connect it all together.

Chapter 2 – Instructions: Language of the Computer

Instruction set architecture



- ❑ **We'll talk about several important issues that we didn't see in the simple processor from CS120B.**
 - **The instruction set in CS120B lacked many features, such as support for function calls. We'll work with a larger, more realistic processor.**
 - **We'll also see more ways in which the instruction set architecture affects the hardware design.**

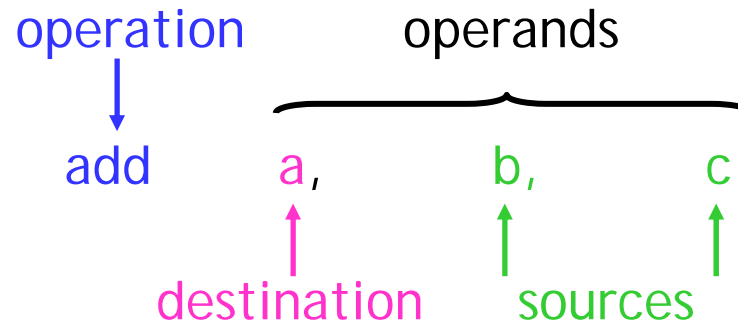
MIPS

- ❑ In this class, we'll use the MIPS instruction set architecture (ISA) to illustrate concepts in assembly language and machine organization
 - Of course, the concepts are not MIPS-specific
 - MIPS is just convenient because it is real, yet simple (unlike x86)
- ❑ The MIPS ISA is still used in many places today. Primarily in embedded systems, like:
 - Various routers from Cisco
 - Game machines like the Nintendo 64 and Sony Playstation 2
- ❑ You must become “fluent” in MIPS assembly:
 - Translate from C to MIPS and MIPS to C



MIPS: register-to-register, three address

- ❑ MIPS is a **register-to-register**, or **load/store**, architecture.
 - The destination and sources must all be registers.
 - Special instructions, which we'll see later, are needed to access main memory.
- ❑ MIPS uses **three-address** instructions for data manipulation.
 - Each ALU instruction contains a **destination** and two **sources**.
 - For example, an addition instruction ($a = b + c$) has the form:



MIPS register names

- ❑ MIPS register names begin with a \$. There are two naming conventions:

- By number:

\$0 \$1 \$2 ... \$31

- By (mostly) two-character names, such as:

\$a0-\$a3 \$s0-\$s7 \$t0-\$t9 \$sp \$ra

- ❑ Not all of the registers are equivalent:

- E.g., register **\$0** or **\$zero** always contains the value **0**
 - (go ahead, try to change it)

- ❑ Other registers have special uses, by convention:

- E.g., register **\$sp** is used to hold the “stack pointer”

- ❑ You have to be a little careful in picking registers for your programs.

Basic arithmetic and logic operations

- ❑ The basic integer arithmetic operations include the following:

`add` `sub` `mul` `div`

- ❑ And here are a few logical operations:

`and` `or` `xor`

- ❑ Remember that these all require three register operands; for example:

```
add    $t0, $t1, $t2    # $t0 = $t1 + $t2
xor    $s1, $s1, $a0    # $s1 = $s1 xor $a0
```

Immediate operands

- The ALU instructions we've seen so far expect register operands. How do you get data into registers in the first place?

- Some MIPS instructions allow you to specify a signed constant, or “immediate” value, for the second source instead of a register. For example, here is the immediate add instruction, **addi**:

```
addi    $t0, $t1, 4    # $t0 = $t1 + 4
```

- Immediate operands can be used in conjunction with the **\$zero** register to write constants into registers:

```
addi    $t0, $0, 4    # $t0 = 4
```

- Data can also be loaded first into the memory along with the executable file. Then you can use load instructions to put them into registers

```
lw      $t0, 8($t1)   # $t0 = mem[8+$t1]
```

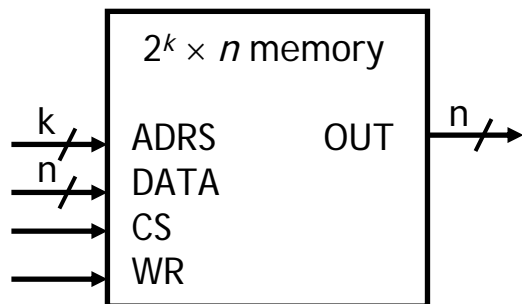
- MIPS is still considered a load/store architecture, because arithmetic operands cannot be from arbitrary memory locations. They must either be registers or constants that are embedded in the instruction.

We need more space – memory

- ❑ **Registers are fast and convenient, but we have only 32 of them, and each one is just 32-bits wide.**
 - That’s not enough to hold data structures like large arrays.
 - We also can’t access data elements that are wider than 32 bits.
- ❑ **We need to add some main memory to the system!**
 - RAM is cheaper and denser than registers, so we can add lots of it.
 - But memory is also significantly slower, so registers should be used whenever possible.
- ❑ **In the past, using registers wisely was the programmer’s job.**
 - For example, C has a keyword “register” that marks commonly-used variables which should be kept in the register file if possible.
 - However, modern compilers do a pretty good job of using registers intelligently and minimizing RAM accesses.

Memory review

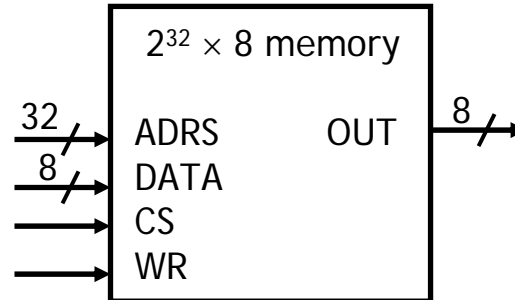
- ❑ Memory sizes are specified much like register files; here is a $2^k \times n$ RAM.



| CS | WR | Operation |
|----|----|------------------------|
| 0 | x | None |
| 1 | 0 | Read selected address |
| 1 | 1 | Write selected address |

- ❑ A chip select input **CS** enables or “disables” the RAM.
- ❑ **ADRS** specifies the memory location to access.
- ❑ **WR** selects between reading from or writing to the memory.
 - To read from memory, WR should be set to 0. **OUT** will be the n -bit value stored at ADRS.
 - To write to memory, we set $WR = 1$. **DATA** is the n -bit value to store in memory.

MIPS memory



- ❑ MIPS memory is **byte-addressable**, which means that each memory address references an 8-bit quantity.
- ❑ The MIPS architecture can support up to 32 address lines.
 - This results in a 2³² x 8 RAM, which would be 4 GB of memory.
 - Not all actual MIPS machines will have this much!