

# Pipelined Design

*CPE380, Fall 2024*

**Hank Dietz**

<http://aggregate.org/hankd/>

# Different Implementations

- **Multi-cycle** MIPS, **multiple CPI**:

<http://aggregate.org/EE380/multiv.html>

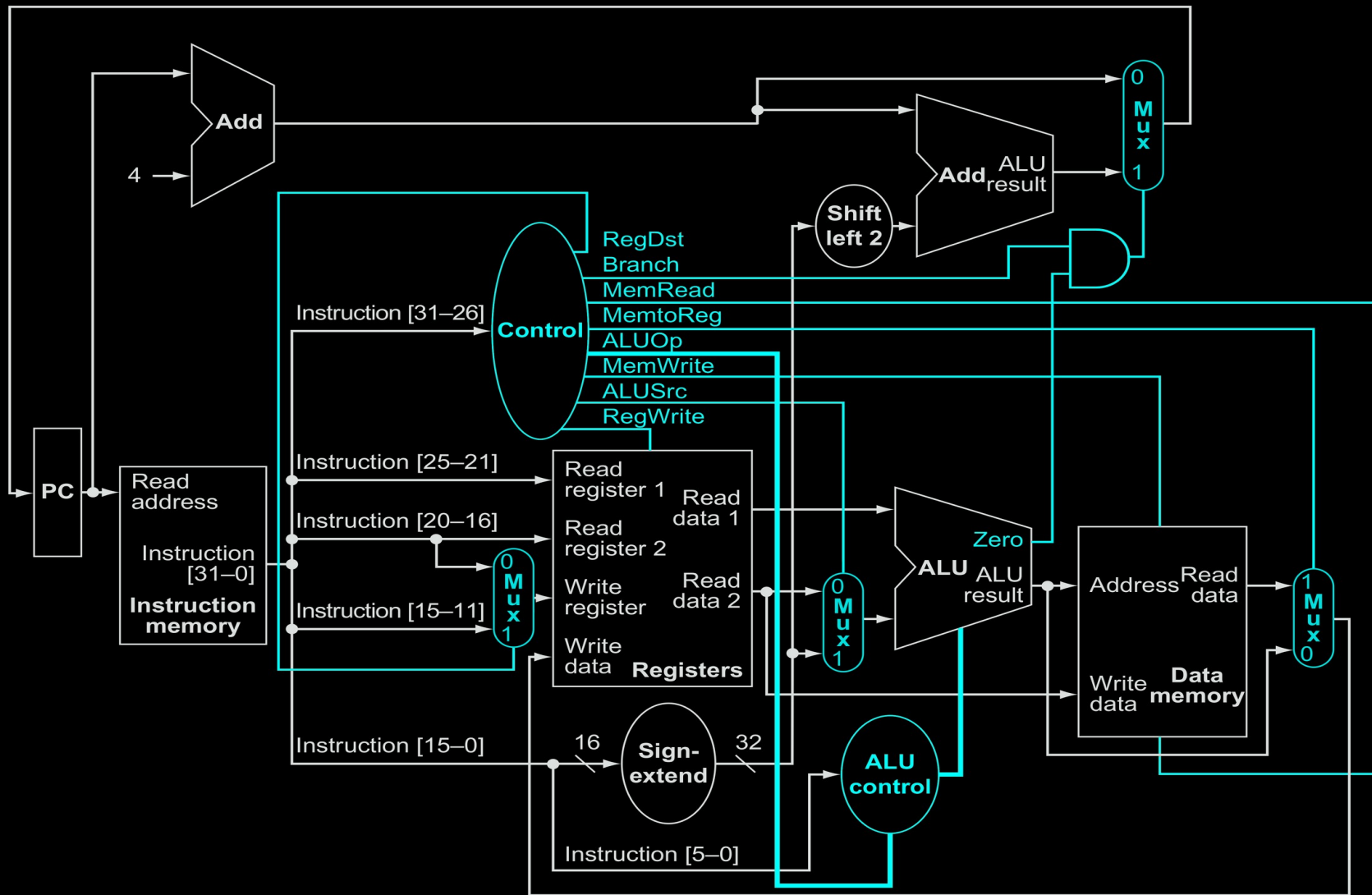
- **Single-cycle** MIPS, **1 CPI**, but **slow clock**:

<http://aggregate.org/EE380/onebeq.html>

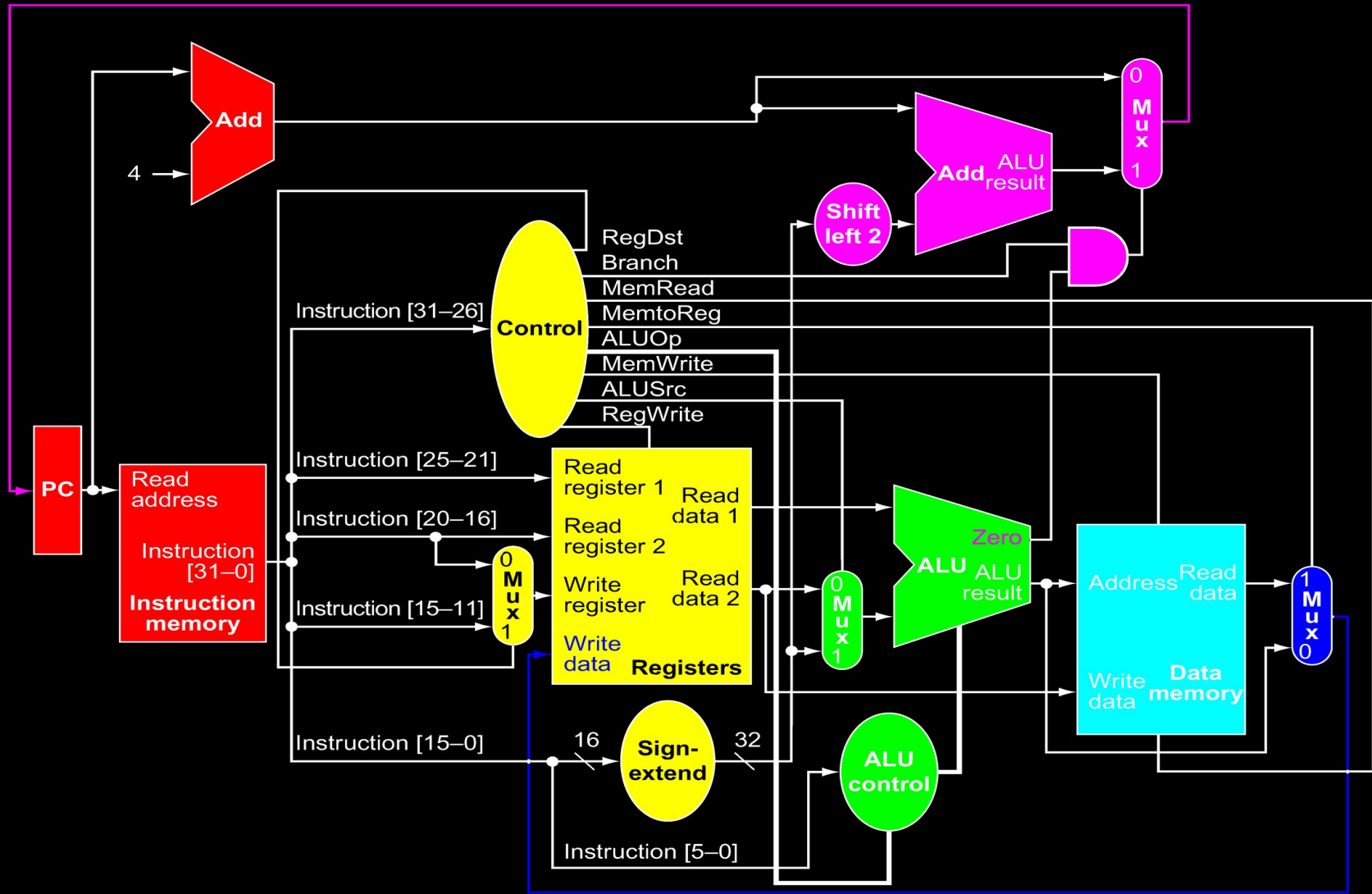
- **Pipelined** MIPS, **multiple CPI**, but **fast clock** and **throughput** up to **1 instruction/cycle**

# Basic Pipelining

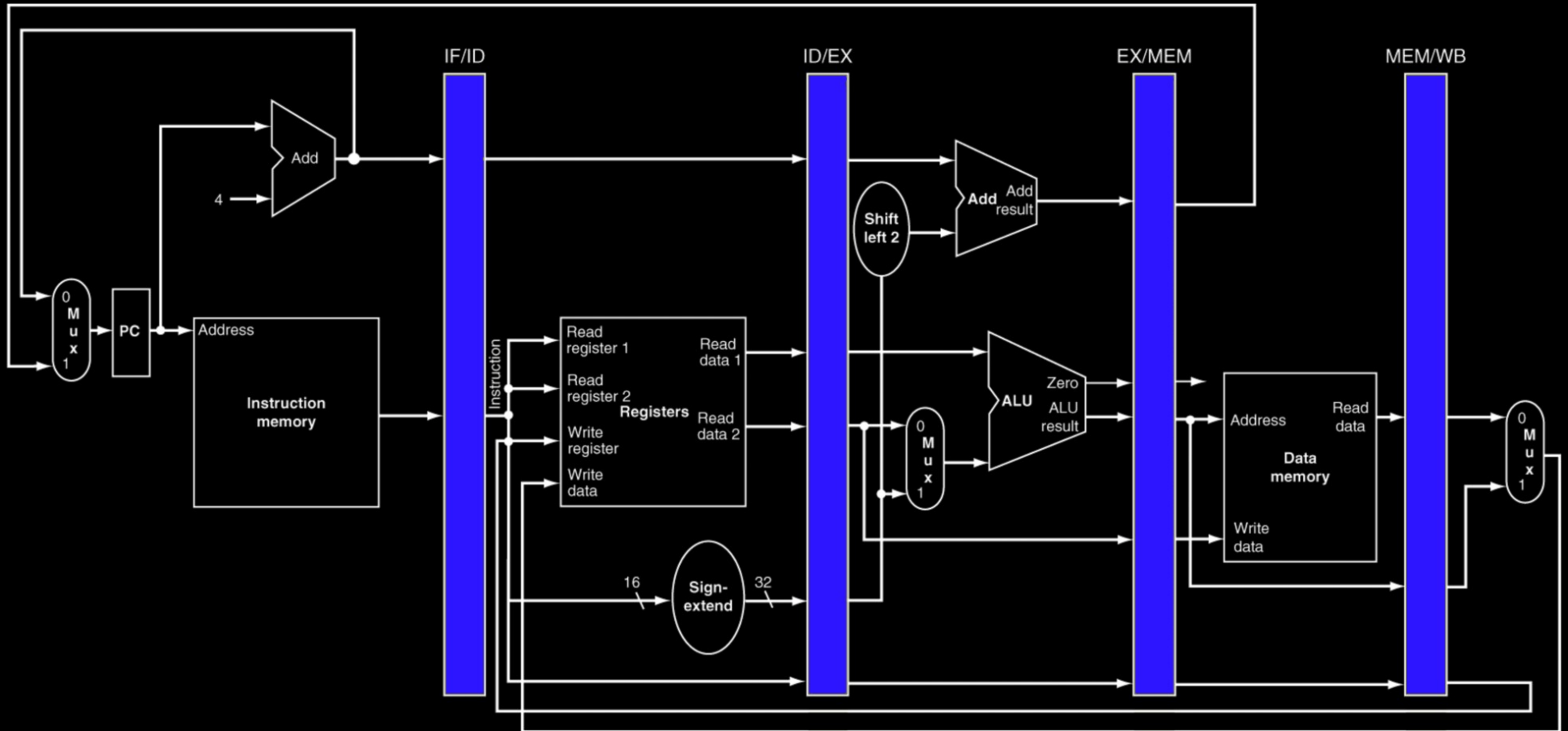
- Single-cycle **control signals move through the pipe** along with the data
- Divide single-cycle design into **equal-delay stages**, adding **buffers between stages**
  - Ideally,  $n$  stages gives  $nX$  throughput
  - Usually  $<nX$ , and  $n$  can't be arbitrarily large
- Throughput comes from having useful work in all stages all the time – avoiding **bubbles**



The single cycle design...



**IF:** Instruction Fetch      **EX:** Execute      **WB:** Write Back  
**ID:** Instruction Decode    **MEM:** Memory Access    what is this?



- Add buffers between pipe stages...

# Pipeline Throughput

IF	ID	EX	MEM	WB
addu \$t0,\$t1,\$t2				-
	addu \$t0,\$t1,\$t2			-
		addu \$t0,\$t1,\$t2		-
			addu \$t0,\$t1,\$t2	-
and \$t3,\$t4,\$t5				addu \$t0,\$t1,\$t2
	and \$t3,\$t4,\$t5			-
		and \$t3,\$t4,\$t5		-
			and \$t3,\$t4,\$t5	-
...				and \$t3,\$t4,\$t5

IF	ID	EX	MEM	WB
addu \$t0,\$t1,\$t2				-
and \$t3,\$t4,\$t5	addu \$t0,\$t1,\$t2			-
addiu \$t6,\$0,1	and \$t3,\$t4,\$t5	addu \$t0,\$t1,\$t2		-
lw \$t7,0(\$t8)	addiu \$t6,\$0,1	and \$t3,\$t4,\$t5	addu \$t0,\$t1,\$t2	-
sw \$t1,4(\$t9)	lw \$t7,0(\$t8)	addiu \$t6,\$0,1	and \$t3,\$t4,\$t5	addu \$t0,\$t1,\$t2
beq \$t1,\$t2,lab	sw \$t1,4(\$t9)	lw \$t7,0(\$t8)	addiu \$t6,\$0,1	and \$t3,\$t4,\$t5
...	beq \$t1,\$t2,lab	sw \$t1,4(\$t9)	lw \$t7,0(\$t8)	addiu \$t6,\$0,1
...	...	beq \$t1,\$t2,lab	sw \$t1,4(\$t9)	lw \$t7,0(\$t8)
...	...	...	beq \$t1,\$t2,lab	sw \$t1,4(\$t9)
...	...	...	...	beq \$t1,\$t2,lab

# Pipeline Bubble: nop

IF	ID	EX	MEM	WB
addu \$t0,\$t1,\$t2				-
and \$t3,\$t4,\$t5	addu \$t0,\$t1,\$t2			-
addiu \$t6,\$0,1	and \$t3,\$t4,\$t5	addu \$t0,\$t1,\$t2		-
lw \$t7,0(\$t8)	addiu \$t6,\$0,1	and \$t3,\$t4,\$t5	addu \$t0,\$t1,\$t2	-
sw \$t1,4(\$t9)	lw \$t7,0(\$t8)	addiu \$t6,\$0,1	and \$t3,\$t4,\$t5	addu \$t0,\$t1,\$t2
beq \$t1,\$t2,lab	sw \$t1,4(\$t9)	lw \$t7,0(\$t8)	addiu \$t6,\$0,1	and \$t3,\$t4,\$t5
...	beq \$t1,\$t2,lab	sw \$t1,4(\$t9)	lw \$t7,0(\$t8)	addiu \$t6,\$0,1
...	...	beq \$t1,\$t2,lab	sw \$t1,4(\$t9)	lw \$t7,0(\$t8)
...	...	...	beq \$t1,\$t2,lab	sw \$t1,4(\$t9)
...	...	...	...	beq \$t1,\$t2,lab

IF	ID	EX	MEM	WB
addu \$t0,\$t1,\$t2				-
and \$t3,\$t4,\$t5	addu \$t0,\$t1,\$t2			-
addiu \$t6,\$0,1	and \$t3,\$t4,\$t5	addu \$t0,\$t1,\$t2		-
lw \$t7,0(\$t8)	addiu \$t6,\$0,1	and \$t3,\$t4,\$t5	addu \$t0,\$t1,\$t2	-
sw \$t1,4(\$t7)	lw \$t7,0(\$t8)	addiu \$t6,\$0,1	and \$t3,\$t4,\$t5	addu \$t0,\$t1,\$t2
beq \$t1,\$t2,lab	sw \$t1,4(\$t7)	lw \$t7,0(\$t8)	addiu \$t6,\$0,1	and \$t3,\$t4,\$t5
beq \$t1,\$t2,lab	sw \$t1,4(\$t7)	nop	lw \$t7,0(\$t8)	addiu \$t6,\$0,1
beq \$t1,\$t2,lab	sw \$t1,4(\$t7)	nop	nop	lw \$t7,0(\$t8)
beq \$t1,\$t2,lab	sw \$t1,4(\$t7)	nop	nop	nop
...	beq \$t1,\$t2,lab	sw \$t1,4(\$t7)	nop	nop
...	...	beq \$t1,\$t2,lab	sw \$t1,4(\$t7)	nop
...	...	...	beq \$t1,\$t2,lab	sw \$t1,4(\$t7)
...	...	...	...	beq \$t1,\$t2,lab



# Setting The Stages

Instruction	IF	ID	EX	MEM	WB	Circuit delay
addu	250	100	300	–	100	750ps
and	250	100	100	–	100	550ps
addiu	250	100	300	–	100	750ps
lw	250	100	300	250	100	1000ps
sw	250	100	300	100	100	850ps
beq	250	100	300	–	–	750ps

- Single-cycle limited by **lw** to **1GHz clock**
- 5-stage pipeline limited by **EX**
  - 300ps latency might allow **3.33GHz** clock
  - If buffer between stages is 100ps, **2.5GHz**
- Multi-cycle might be **5 cycles @2.5GHz**

# Why A Bubble?

- **Structural hazards**: can't do 2 things on one unit of HW simultaneously – **single-cycle cures this!**
- **Data dependence**: need results from previous computations, which might not be done yet
- **Control dependence**
  - Computation of branch target address
  - Conditional branch/jump taken vs. not taken

# Dependence Analysis

- Use or R: reads the value bound to a name
- Def or W: binds a new value to a name
- True dependence: carries a value,  $D \rightarrow U$ , RAW  
add  $\$t0, \$t1, \$t2$  or  $\$t3, \$t0, \$t4$
- Anti-dependence: kills a value,  $U \leftarrow D$ , WAR  
add  $\$t0, \$t1, \$t2$  or  $\$t1, \$t3, \$t4$
- Output dependence: kills a value,  $D \rightarrow D$ , WAW  
add  $\$t0, \$t1, \$t2$  or  $\$t0, \$t3, \$t4$

# When Dependence Matters

- True dependence causes a delay

```
add $t0, $t1, $t2  
or  $t3, $t0, $t4 //wait for $t0
```

- Other types don't

# How To Handle Dependence

- Have programmer/assembler **pad with NOPs**
  - **Violates ISA concept** (how much padding?); too little padding gives wrong answers, too much wastes time
  - **Makes program bigger**

IF	ID	EX	MEM	WB
add \$t0, \$t1, \$t2	...	...	...	...
nop	add \$t0, \$t1, \$t2	...	...	...
nop	nop	add \$t0, \$t1, \$t2	...	...
nop	nop	nop	add \$t0, \$t1, \$t2	...
or \$t3, \$t0, \$t4	nop	nop	nop	add \$t0, \$t1, \$t2
...	or \$t3, \$t0, \$t4	nop	nop	nop
...	...	or \$t3, \$t0, \$t4	nop	nop
...	...	...	or \$t3, \$t0, \$t4	nop
...	...	...	...	or \$t3, \$t0, \$t4

# How To Handle Dependence

- **Hardware interlock**
  - rs or rt in ID same as dest in EX, MEM, WB
  - Detects dependence & stalls until satisfied
  - `nop` is really `or` with **side-effects disabled**:  
MemRead, MemWrite, RegWrite, & Branch

IF	ID	EX	MEM	WB
add <code>\$t0, \$t1, \$t2</code>	...	...	...	...
or <code>\$t3, \$t0, \$t4</code>	add <code>\$t0, \$t1, \$t2</code>	...	...	...
...	or <code>\$t3, \$t0, \$t4</code>	add <code>\$t0, \$t1, \$t2</code>	...	...
...	or <code>\$t3, \$t0, \$t4</code>	<code>nop</code>	add <code>\$t0, \$t1, \$t2</code>	...
...	or <code>\$t3, \$t0, \$t4</code>	<code>nop</code>	<code>nop</code>	add <code>\$t0, \$t1, \$t2</code>
...	or <code>\$t3, \$t0, \$t4</code>	<code>nop</code>	<code>nop</code>	<code>nop</code>
...	...	or <code>\$t3, \$t0, \$t4</code>	<code>nop</code>	<code>nop</code>
...	...	...	or <code>\$t3, \$t0, \$t4</code>	<code>nop</code>
...	...	...	...	or <code>\$t3, \$t0, \$t4</code>

# How To Handle Dependence

- **Value forwarding**: use interlock circuitry to find value & forward it to ID stage output buffer
  - ALU result ready from EX, so **NO DELAY!**
  - **lw** result ready from MEM, so **1 cycle delay**
  - Value still gets stored in register in WB
  - Works great, but adds lots of datapath...

IF	ID	EX	MEM	WB
lw \$t0, 0(\$t1)	...	...	...	...
or \$t3, \$t0, \$t4	lw \$t0, 0(\$t1)	...	...	...
...	or \$t3, \$t0, \$t4	lw \$t0, 0(\$t1)	...	...
...	or \$t3, \$t0, \$t4	nop	lw \$t0, 0(\$t1)	...
...	...	or \$t3, \$t0, \$t4	nop	lw \$t0, 0(\$t1)
...	...	...	or \$t3, \$t0, \$t4	nop
...	...	...	...	or \$t3, \$t0, \$t4

# Using Dependence Information

- True dependence causes a delay

```
lw    $t0, 0($t1)
or    $t3, $t0, $t4    //wait for $t0
xor   $t5, $t6, $t7    //takes a cycle
```

- Use **code motion** or **out-of-order execution** to **execute useful instructions while waiting!**

```
lw    $t0, 0($t1)
xor   $t5, $t6, $t7    //this is free!
or    $t3, $t0, $t4    //wait for $t0
```



# Reordering Instructions

- **Code motion** by compiler/assembler:
  - + Sees whole program, has time for analysis
  - Limited number of registers, imperfect info (e.g., does not know function unit details)
- **Out-of-order execution** by hardware:
  - + Perfect info, can use **register renaming** (with more registers than an instruction can name)
  - Limited to instructions buffered
- **Modern systems do both**

# Computing “May Move”

- No dependence allows code to move
- Anti-dependence and output-dependence impose an ordering constraint... but renaming can remove that constraint:

```
add $t0, $t1, $t2  
or  $t1, $t3, $t4
```

*becomes*

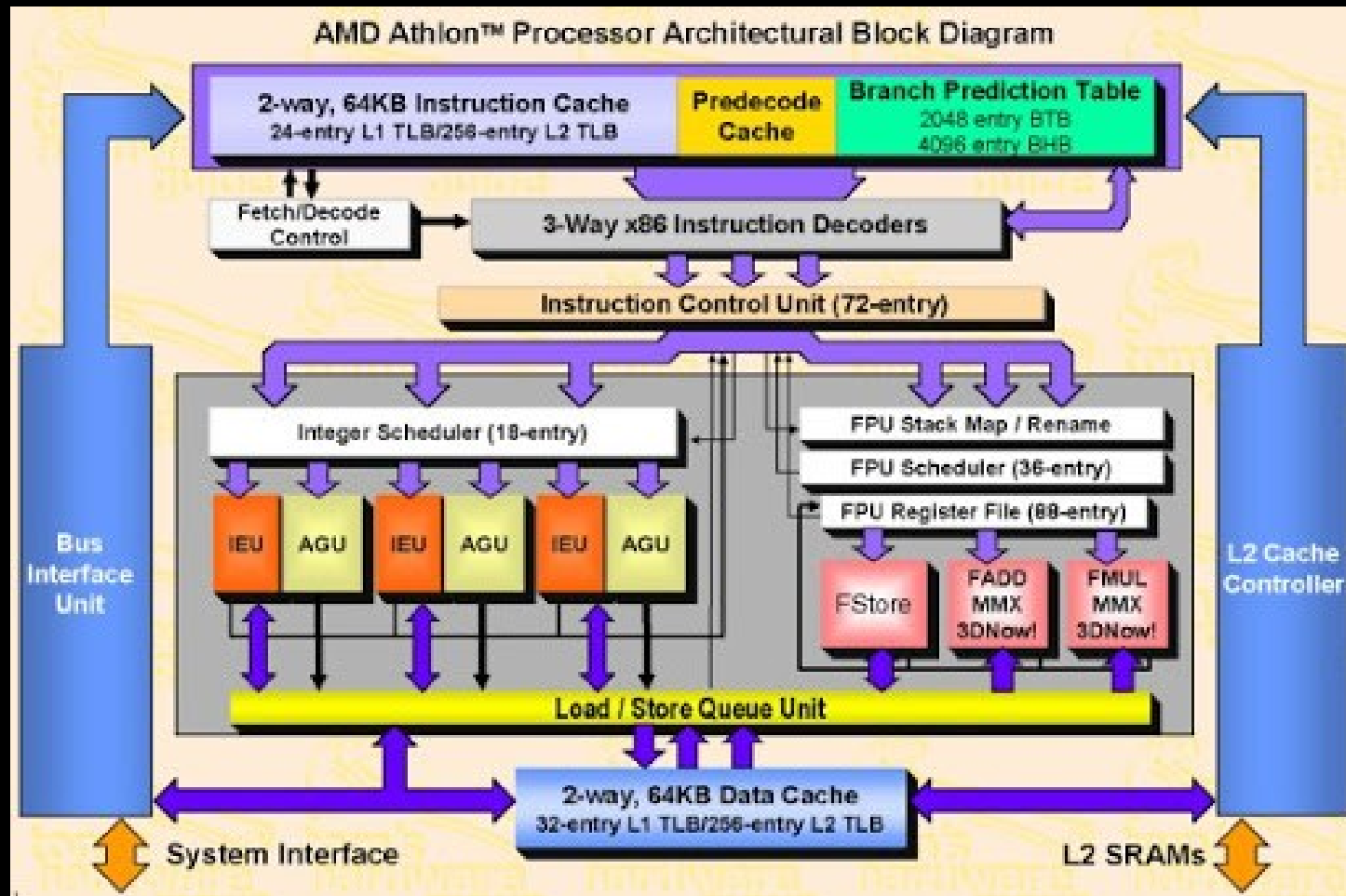
```
add $t0, $t1, $t2  
or  $t1', $t3, $t4
```

```
add $t0, $t1, $t2  
or  $t0, $t3, $t4
```

*becomes*

```
add $t0, $t1, $t2  
or  $t0', $t3, $t4
```

# A Real Processor: AMD Athlon

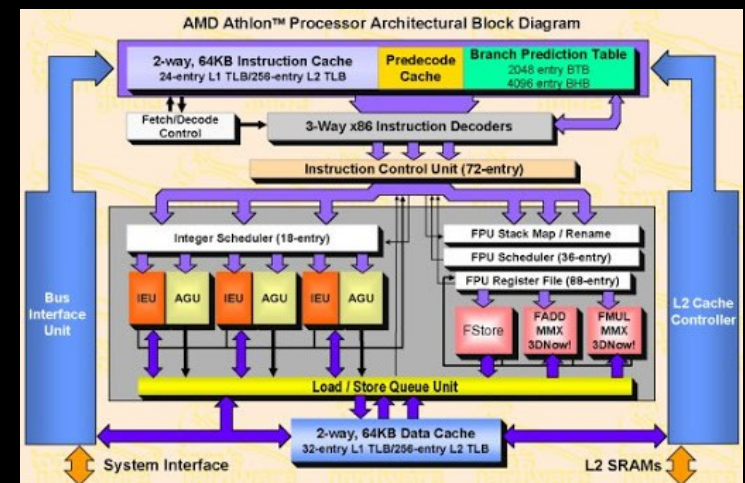


E.g., IA32 has 8 float registers, but this has 88!

# Parallel Execution Orders

- IF stage can fork to feed multiple pipes
  - SIMD: GPU, SWAR, and Vector
  - VLIW: Very Long Instruction Word
  - EPIC: Explicitly Parallel Instruction Computer
  - Superscalar: instructions grouped at runtime

Athlon can sustain **3 IPC**:  
**3 Integer Execution Units**  
**3 Address Generation Units**  
Float store, add, & multiply



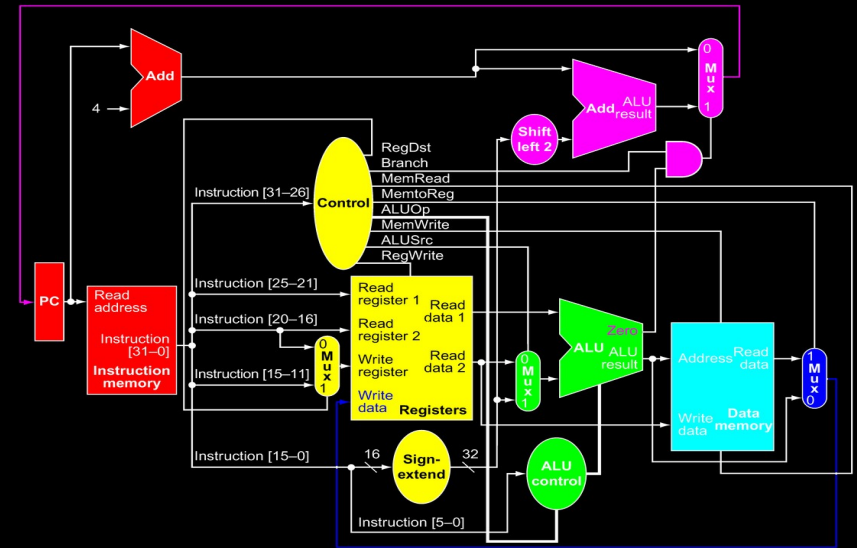
# Computing The Branch Target

- Branch instructions encode offset, not address
  - Add PC + offset *typically* takes a cycle
  - What do we do while waiting for address?
- Hardware interlock & stall
- **Delayed branch**: branch *after* next instruction
- **BTB: Branch Target Buffer**
  - Caches branch {PC: target} pairs and looks 'em up at same time as fetching instruction
  - No help 1<sup>st</sup> time, but no delay on repeats

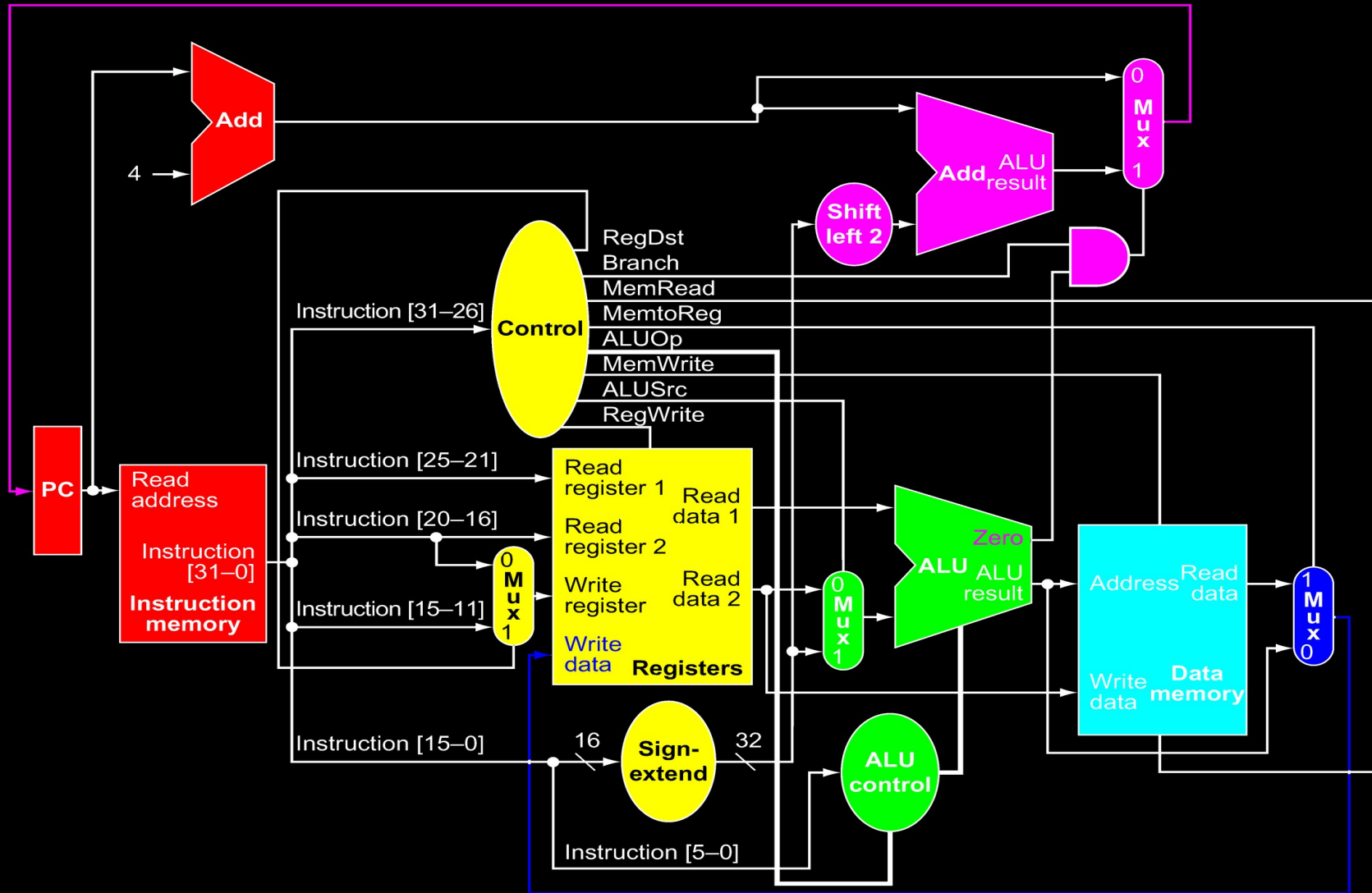
# To Take, Or Not To Take?

- When do we know if conditional jump or branch is taken or not taken?

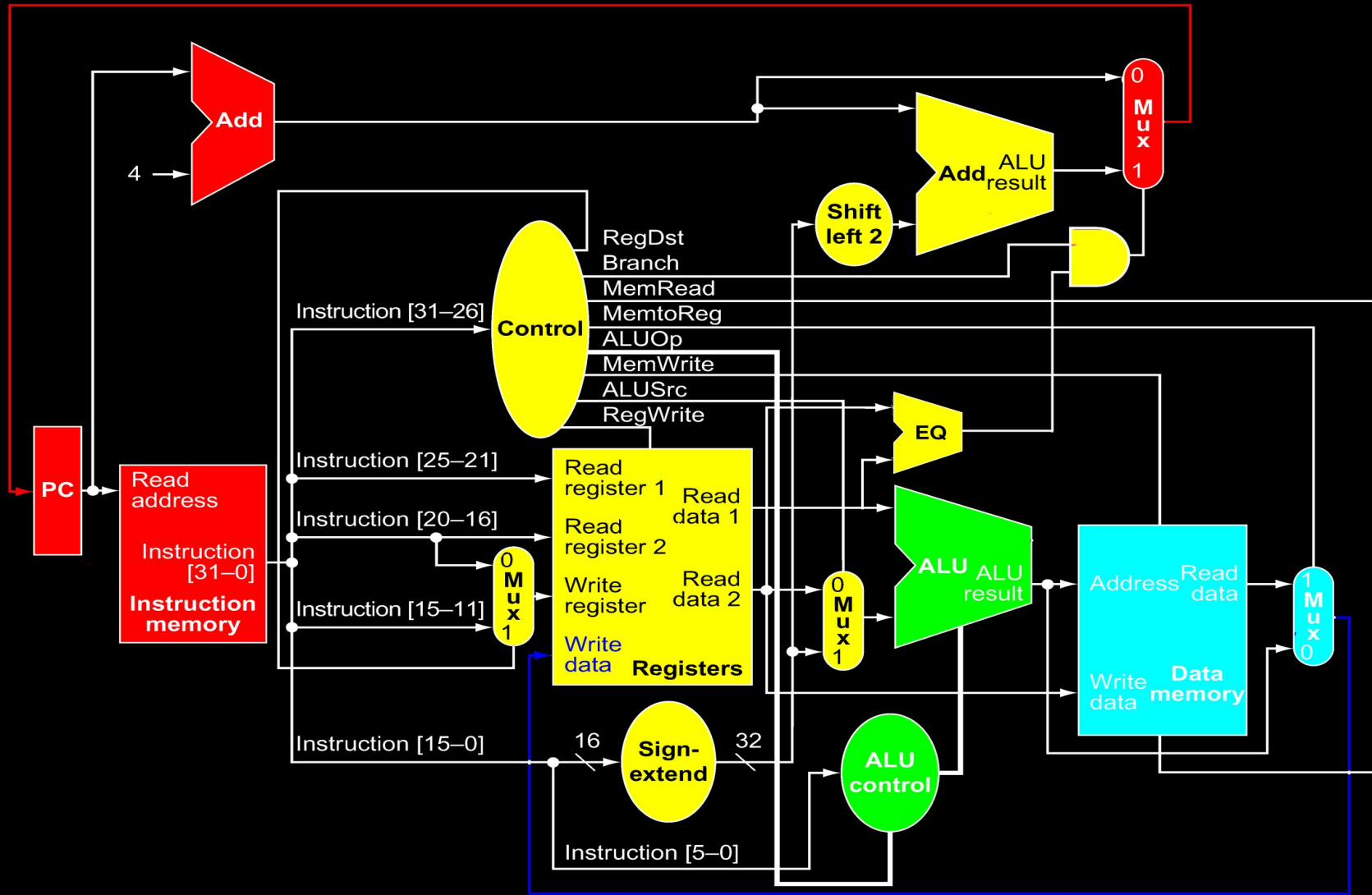
Determined only after EX stage?



- **NOP padding** or **HW interlock** very inefficient!
  - Usually determined in a late pipe stage
  - **Blocks ALL progress** (including superscalar)



**IF:** Instruction Fetch      **EX:** Execute      **WB:** Write Back  
**ID:** Instruction Decode    **MEM:** Memory Access    **what is this?**



**IF:** Instruction Fetch      **EX:** Execute      **WB:** Write Back  
**ID:** Instruction Decode    **MEM:** Memory Access



# Branch Prediction. Do I Feel Lucky?



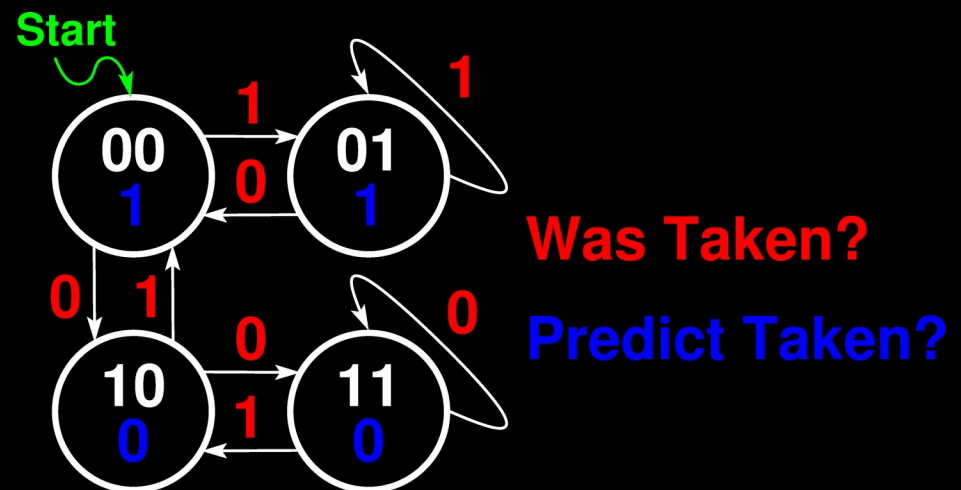
*Guess wrong and some instructions are gonna die  
(well, we actually say they're **squashed**)*

- **Always not taken** – the easiest guess
- **Always taken** – more often right for do loops
- **Always BOTH not taken and taken**
  - Was tried using dual pipe of Pentium Pro
  - **Always wastes 50% of instruction fetch!**
- **Forward not, backward taken**

Those who cannot remember the past are condemned to repeat it. – *G. Santayana*

- **BHB: Branch History Buffer**
  - Jump & branch taken/not-taken “history,” but actually records a prediction state, *not* history
  - Indexed by PC; can be merged with BTB

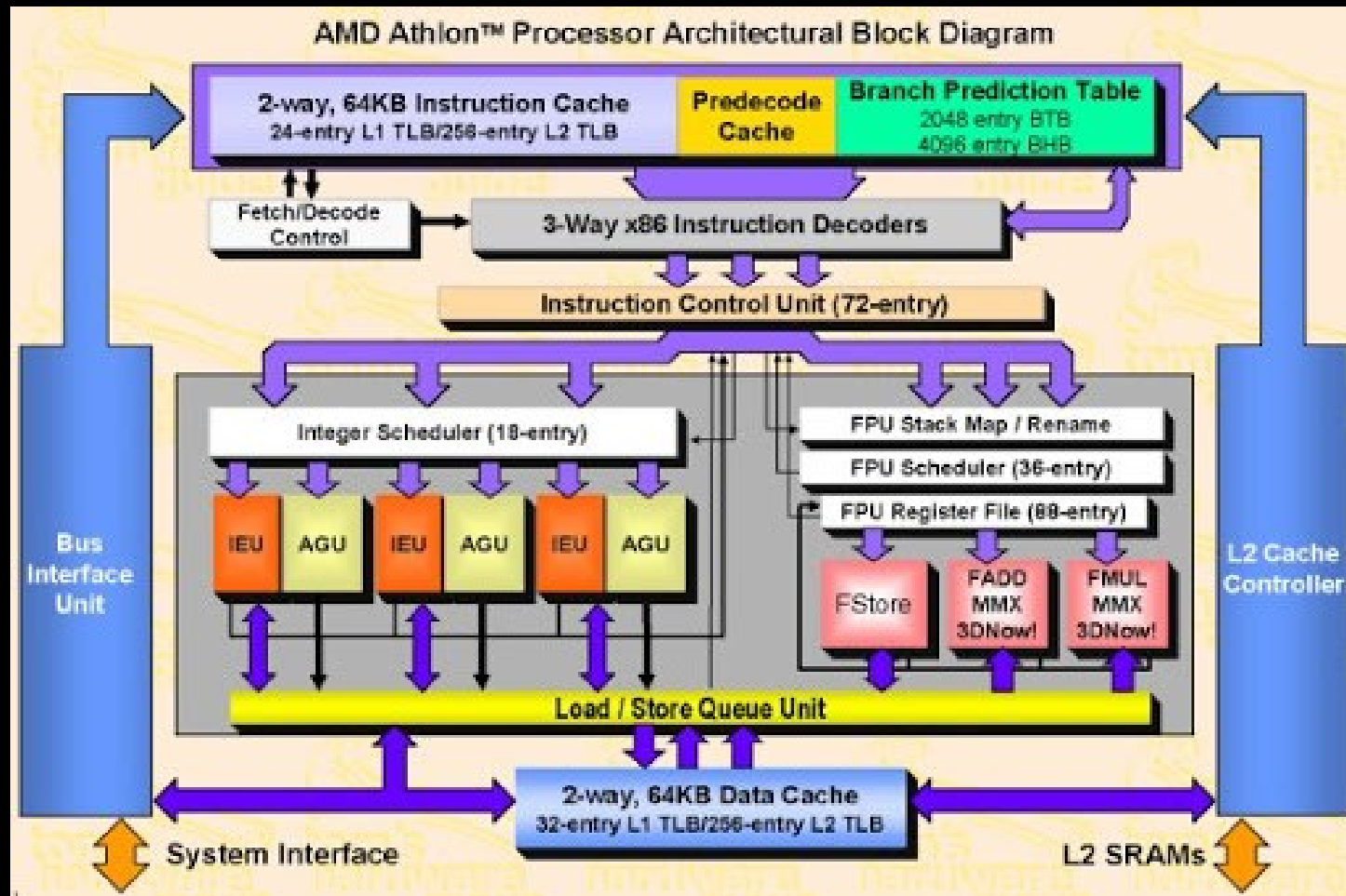
Simplest is 4-state,  
just 2 bits/entry:



# Fancier Prediction Methods

- Two types of prediction history recorded:
  - History of *this particular* branch/jump
  - History of *last K* branches/jumps anywhere
- More complex state machines can recognize *much* more complex patterns; also can combine multiple predictors as a **tournament predictor**
- Some systems also allow compiler hints by marking **usually taken** vs. **usually not taken**

# A Real Processor: AMD Athlon



Note 2048-entry BTB and 4096-entry BHB

# Verilog Implementation

- Like you'd expect:
  - Can reuse basic single-cycle design
  - Each stage becomes it's own `always`
  - Need multiple copies of some signals, one for each stage that uses them
- Not like you'd expect:
  - Some things don't follow pipe flow
  - Some non-stages should be separate things

# Owner Computes

- For example, **who updates the PC?**
  - IF sets  $PC = PC + 4$
  - ID sets  $PC = \text{branch target}$
  - EX, MEM, or WB forces  $PC = PC$  because data dependence blocks the pipe
- How to coordinate access to shared data?
  - Multiple readers is fine, with one writer
  - Could use locks, semaphores, etc.
  - Easiest is only one entity can update each:  
the **owner picks the value and is only writer**

# A Pipelined Verilog Version

- Organized as parallel-executing chunks for:
  - **IF** stage: reads and writes **IF\_**
  - **ID** stage: reads **IF\_**, writes **ID\_**
  - **EX** stage: reads **ID\_**, writes **EX\_**
  - **MEM** stage: reads **EX\_**, writes **MEM\_**
  - **WB** stage: reads **MEM\_**
  - Are we **running**?
  - Are mispredicted instructions **squashed**?
  - Are we **blocked** by data dependence or forwarding values?
  - Simulation **tracing** support





# IF: Instruction Fetch stage

- Really simple...
  - Memory is `WORD, not byte, so address>>2
  - The only thing setting IF\_ir and IF\_pc

```
// IF: Instruction Fetch stage
always @(posedge clk) if (running && !blocked) begin
    IF_ir <= m[(squash ? target : IF_pc) >> 2];
    IF_pc <= (squash ? target : IF_pc) + 4;
end
```

# ID: Instruction Decode stage

- Decodes the instruction
- Reads from register file  $r[]$
- Computes `beq` comparison & target

```
// ID: Instruction Decode stage
always @(posedge clk) if (running && !ID_Bad) begin
  if (blocked) ... else begin
    case (squashed `OP)
      `RTYPE: begin
        case (squashed `FUNCT)
          `ADDU:   begin RegDst=1; Branch=0; MemRead=0;
                     ALUOp=`ALUADD; MemWrite=0; ALUSrc=0;
                     RegWrite=1; Bad=0; end
        ... endcase ... endcase
      ... ID_s <= s; ID_t <= t; ... ID_ALUOp <= ALUOp; ...
    end end
```

# EX: Execute stage

- Contains the ALU for integers & addresses

```
// EX: EXecute stage
always @(posedge clk) if (running) begin
  case (ID_ALUOp)
    `ALUAND: alu = ID_s & ID_src;
    `ALUOR:  alu = ID_s | ID_src;
    `ALUADD: alu = ID_s + ID_src;
    `ALUSUB: alu = ID_s - ID_src;
    `ALUSLT: alu = ID_s < ID_src;
    `ALUXOR: alu = ID_s ^ ID_src;
    default: alu = (ID_src << 16);
  endcase
  EX_alu <= alu;
  EX_t <= ID_t;
  EX_rd <= ID_rd;
  EX_MemRead <= ID_MemRead;
  EX_MemWrite <= ID_MemWrite;
  EX_Bad <= ID_Bad;
end
```

# MEM: MEMory access stage

- Does a memory read or write
- Uses `EX_MemRead` for both read and mux `v`

```
// MEM: data MEMory access stage
always @(posedge clk) if (running) begin
    if (EX_MemRead) v = m[EX_alu >> 2]; else v = EX_alu;
    if (EX_MemWrite) m[EX_alu >> 2] <= EX_t;

    MEM_v <= v;
    MEM_rd <= EX_rd;
    MEM_Bad <= EX_Bad;
end
```

# WB: Write Back stage

- Writes result into register...
  - Register \$0 is read only
  - Not writing? Say we write to register 0...
- An instruction isn't done until it's here, so this is where halting really happens

```
// WB: register Write Back stage
always @(posedge clk) if (running) begin
    if (MEM_rd) r[MEM_rd] <= MEM_v;
    if (MEM_Bad) halt <= 1;
end
```

# Running?

- Enables normal operation of stages
- Not running normally if:
  - Halted
  - There's a reset in progress; reset writes into stuff it doesn't own, so we need owners off

```
// Running state?  
wire running;  
assign running = ((!halt) && (!reset));
```

# Squashed?

- For `beq`, we'll predict `not taken`, so `IF_pc+4`
- If we were right, no bubble
- If wrong, need to squash fetched instructions
  - `No side-effects to undo yet`
  - Just convert into ``NOP` to `prevent future s-e`

```
`define NOP `OR    // Null Operation is or $0,$0,$0
```

```
// Squash instruction fetched on a mispredicted branch  
wire `INST squashed;  
assign squashed = (squash ? `NOP : IF_ir);
```





# Simulation Tracing

- Yet another (complex!) parallel-executing thing:

```
// State-by-state trace
`ifdef TRACE
always @(posedge clk) if (running) begin
    ...
    $display("IF ir=%x pc=%1d", IF_ir, IF_pc);
    case (IF_ir `OP)
        `RTYPE: begin
            case (IF_ir `FUNCT)
                `ADDU: $display("IF addu $%1d,$%1d,$%1d",
                    IF_ir `RD, IF_ir `RS, IF_ir `RT); ...
            endcase endcase
        if (ID_Bad) $display("ID illegal instruction");
        else $display("ID s=%1d t=%1d src=%1d rd=%1d
            MemRead=%b ALUOp=%b MemWrite=%b", ID_s, ID_t,
            ID_src, ID_rd, ID_MemRead, ID_ALUOp, ID_MemWrite);
    ...
end
`endif
endmodule
```



# Three Verilog Implementations

- **Multi-cycle** MIPS, **multiple CPI**:

<http://aggregate.org/EE380/multiv.html>

- **Single-cycle** MIPS, **1 CPI**, but **slow clock**:

<http://aggregate.org/EE380/onebeq.html>

- **Pipelined** MIPS, **fast clock**, **~1 CPI throughput**:

<http://aggregate.org/EE380/pipe.html>

# Out-Of-Order Implementation?

- *Beyond the scope of this class*, but it's **dataflow**
- CDC6600 **Scoreboard**
  - Need **both operands in registers**
  - Instruction **waits @ function unit**
- **Tomasulo Scheduling**
  - Copies operands **when each one is available**
  - Instruction **waits @ reservation station**
  - **Common data bus** broadcasts each result
- Now rename registers, no serial bottleneck