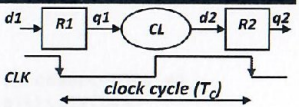
R-type MIPS instructions

Function (Decimal)	Name	Description	Operation
100000 (32)	add	add	[rd] = [rs] + [rt]
100001 (33)	addu	add unsigned	[rd] = [rs] + [rt]
100010 (34)	sub	subtract	[rd] = [rs] - [rt]
100011 (35)	subu	subtract unsigned	[rd] = [rs] - [rt]
100100 (36)	and	and	[rd] = [rs] & [rt]
100101 (37)	or	or	[rd] = [rs] | [rt]
100110 (38)	xor	xor	[rd] = [rs] ^ [rt]
100111 (39)	nor	nor	[rd] = ~([rs] | [rt])
101010 (42)	slt	set less than	([rs]<[rt]) ? [rd]=1 : [rd]=0
101011 (43)	sltu	set less than unsigned	([rs]<[rt]) ? [rd]=1 : [rd]=0

Synchronous digital circuit operation



- The *clocking methodology* defines when data can be written into storage element. On the active edge of the clock *q1* and *q2* are updated with the new values of d1 and d2, respectively
- After q1 is updated, new d2 is computed based on new q1 using the combinational logic (CL), which takes T_C time (d2 is a function of q1). The new d2 will be ready to be written into R2 on the next active edge of the clock
- Registers are written on every clock cycle unless they are disabled using a write enable control signal. Write occurs only when both the write control is asserted and the clock edge occurs
- Another limitation of the single-cycle microarchitecture is that since all the operations of an instruction must be completed within one clock cycle, some functional units (e.g., adders) must be duplicated since they can not be shared during one clock cycle if they are being used. The single-cycle microarchitecture requires three adders (one in the ALU and two for the PC logic)
- In addition, the single-cycle microarchitecture requires separate instruction and data memories because we needed to read the instruction memory and also read from or write onto the data memory in one clock cycle

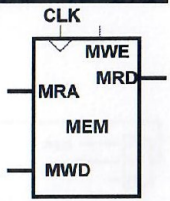
Cycles per instructions

- The *number of instructions* in a program depends on the processor's ISA. Some architectures have complicated instructions that do more work per instruction, thus reducing the number of instructions in a program. However, complicated instructions are often slower to execute in hardware. The number of instructions also depends the programmer and the compiler
- The number of *cycles per instruction* (*CPI*) is the number of clock cycles required to execute an instruction in average. CPI depends on the processor's microarchitecture. CPI also depends on the memory latency
- The reciprocal of the CPI denotes the *instructions per cycle* (*IPC*) or *throughput*
- In a single cycle processor, each instruction takes one clock cycle (i.e., CPI=1). One primary weakness of the single-cycle microarchitecture is that different instructions use different numbers of steps, so simpler instructions can potentially complete faster than more complex ones. However, a single-cycle processor requires a clock cycle long enough to support the slowest instruction. Therefore, the cycle time is limited by the slowest instruction (i.e., `lw`)
- Note that the cycle time to access to the main memory is much longer than CPU's clock cycle

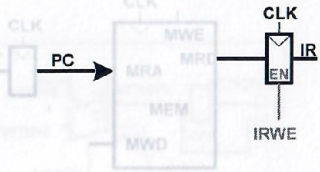
Multi-cycle processor

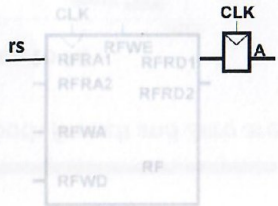
- The multicycle processor addresses the weaknesses of single-cycle microarchitecture by breaking the execution of an instruction into multiple steps. Each step is performed in a clock cycle that is shorter than that of single cycle processor. Note that instructions may take different number of clock cycles
- The motivation for building a multicycle processor architecture is to avoid making all instructions take as long as the slowest one. In a multicycle processor, simpler instructions execute in fewer clock cycles than the complex ones
- The clock cycle is limited by the *slowest instruction step and not the slowest intruction*
- In addition to a *faster clock frequency*, multicycle microarchitecture allows resources, such as functional units, to be used more than once during the execution of an instruction, as long as they are used during different clock cycles. For example one ALU can be used to update the PC in one clock cycle and also to add two operands in another clock cycle. The microarchitecture also uses one memory, but only one memory access per cycle
- While the memory and ALU can be reused for different purposes at various steps, the microarchitecture requires additional internal registers, more multiplexers, and a more complex control unit than that of a single-cycle processor

Supporting `lw` *instruction*



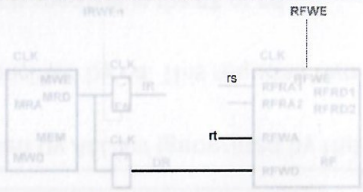
- In a multi-cycle microarchitecture, the processor uses one memory MEM for instructions and data. The first step to execute an instruction is to fetch the instruction from the memory (*fetch state*). The fetched instruction is stored in a *non-architectural instruction register* IR. The non-architectural registers are used to temporarily hold the results of each step
- The IR write enable signal *IRWE* is asserted when the IR should be updated with a new instruction in the fetch state
- The fetched instruction remains in IR until the execution of the current instruction is completed
- To execute the `lw` instruction, the next step is to read the base address stored in the register, addressed by the `rs` field of the instruction. Thus `rs` is connected to the address input *RFRA1* of the RF. The RF reads the register onto *RFRD1* port and the value is stored in nonarchitectural register A



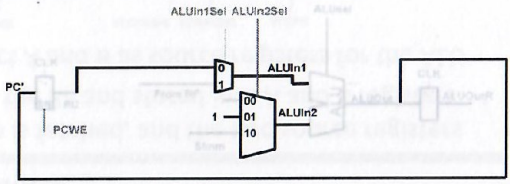


lw *instruction datapath*

- Finally, the data is written back to the RF
- The destination register is specified by the **rt** field of the instruction connected to *RFWA*. *RFWE* is asserted by the control unit

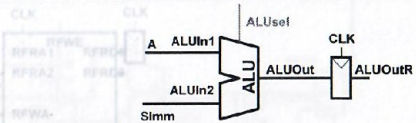


- To update the **PC** in the single-cycle processor, a separate adder was required. In the multicycle processor, the existing ALU can be used in one of the clock cycles of instruction execution when it is not used
- To update the **PC**, we use the input multiplexers at the ALU to choose the **PC** and the constant 1 as ALU inputs. The multiplexer controlled by *ALUIn1Sel* chooses either the **PC** or register A as *ALUIn1* and the A multiplexer controlled by *ALUIn2Sel* chooses either 1 or *SImm* as *ALUIn2*
- The *PCWE* control signal is asserted by the control unit to enable the **PC** register to be written only during a certain clock cycle
- The *ALUOut* is written directly to **PC**

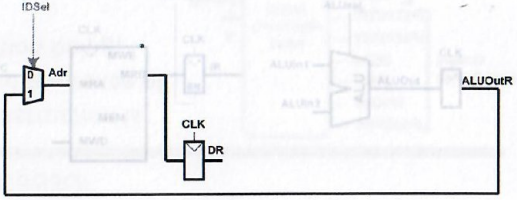


`lw` *instruction datapath*

- In the next clock cycle, *ALUsel* is set to 010 to perform the summation of the base address and *Simm*. *ALUOut* is stored in a nonarchitectural register *ALUOutR*

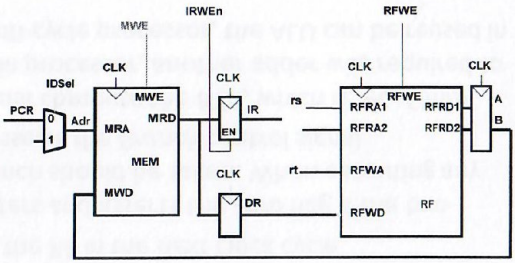


- Note that *SImm* is a combinational function of **`Inst`** and will not change while the current instruction is being executed (since IR does not change until the next instruction is fetched), so there is no need to dedicate a register to hold *SImm*
- The next step is to load the data from the calculated address in the memory. A multiplexer is added in front of the memory address bus to choose the memory address *Adr* from either the **PC** (for fetching an instruction in the fetch state) or from *ALUOutR* for reading data for **`lw`**. The multiplexer allows reusing the memory over two different states
- The data read from the memory is stored in the non-architectural register DR



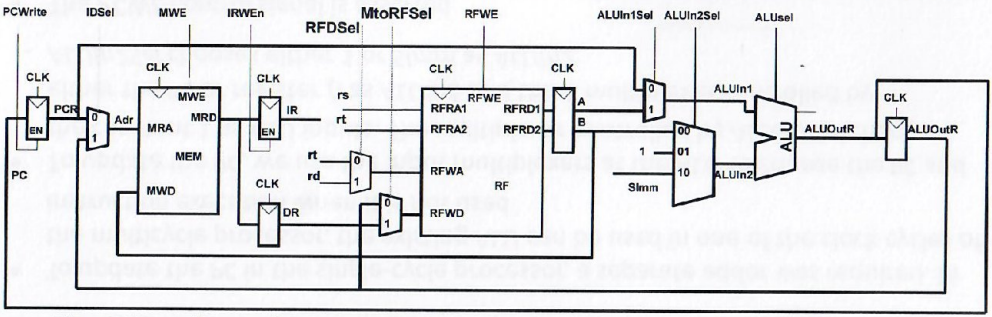
sw *instruction datapath*

- Similar to the **lw** instruction, the **sw** instruction reads a base address from *RFRD1* of the RF addressed by **rs** and sign-extends the immediate. Then the ALU adds the base address to *SImm* to find the effective memory address. These steps are already supported by existing **LW** datapath and control unit
- A register from the RF addressed by **rt** should be read and written it into the memory. The **rt** field of the instruction is connected to the RFRA2 of the RF. When the register is read, it is stored in a non-architectural register B. On the next clock, it is written into MEM
- The memory write enable control signal *MWE*=1 enables the write operation
- Note that **rt** is the destination register for **lw** and the source register for **sw** instruction



R-type instructions muicroarchitecture

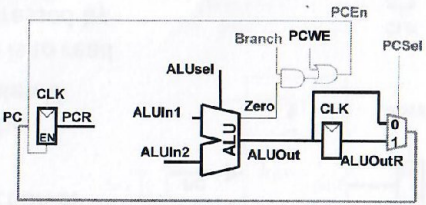
- For R-type instructions, the instruction is fetched, and the two source registers addressed by **rs** and **rt** are read from the RF and stored into A and B registers
- *ALUIn2Sel*=1 and *ALUIn2Sel*=00 to select A and B as source registers for the ALU



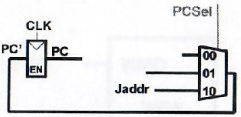
- The ALU performs the appropriate operation chosen by *ALUSel* (generated by the control unit) and stores the result in *ALUOutR*
- On the next clock, *ALUOutR* is written to the RF specified by **rd**. This requires two new multiplexers
  - The *RFDSel* selects whether the destination register is specified in the **rt** or **rd**
  - The *MtoRFSel* multiplexer selects whether *RFWD* comes from *ALUOutR* (for R-type instructions) or from *DR* (for **lw**)

Updating **PC** *for branch instructions*

- The **PC** should be updated either when *PCWE*=1 or when both *Branch* and *Zero* are asserted
- The *PCSel* multiplexer chooses what value should be written to **PC**
  - For non-branch instructions, *PCSel*=0 and for **beq** instructions that should be taken (i.e., **rs=rt**), first *PCSel*=0 to store **PC**+1 in the PC and then in another clock cycle *PCSel*=1 to store the updated **PC**+*SImm* (the final value of **PC** after executing **beq** instruction is **PC**+1+*SImm*) if branch is taken



- In case of a `j` instruction, the jump destination address is formed by appending the six most significant bits of the already incremented **PC** to the 26-bit **addr** field of the instruction. The *PCSel* multiplexer is extended to take this address as a third input (*PCSel*=10) for the `J` instruction

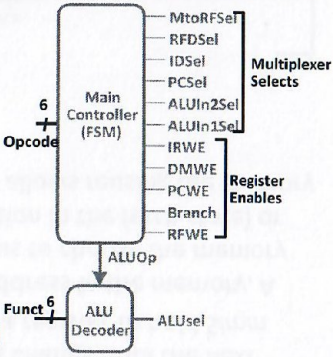


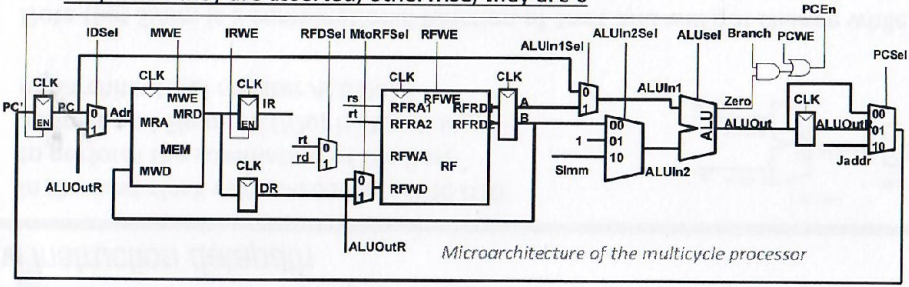
beq *instruction datapath*

- After fetching the instruction, the two source registers addressed by the **rs** and **rt** fields of the instruction are read from the RF in the next clock cycle
- The ALU subtracts the **rs** and **rt** registers and asserts the *Zero* flag if the two registers are equal and hence, the branch should be taken. When executing any branch instructions, the control unit asserts the *Branch* control signal
- If the branch is taken, the datapath must compute the BTA, which is **PC**+*SImm* (relative addressing). In the single-cycle processor, another adder was required to compute the BTA, however, in the multi-cycle processor, the ALU can be reused in different clock cycles:
  1. In one clock cycle, the ALU computes **PC+1** by selecting *ALUIn1Sel*=0, *ALUIn2Sel*=01 and writes the result back to the **PC** (*PCWE*=1)
  2. In another clock cycle, the ALU uses this updated **PC** value to compute **PC**+*SImm* by selecting *ALUIn1Sel*=0, *ALUIn2Sel*=10 , *PCWE*=0 and stores it in *ALUOutR*

Control unit of the multi-cycle processor

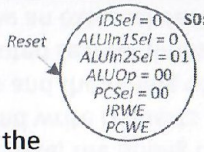
- In the multi-cycle processor, since an instruction is executed over multiple clock cycles, the sequence of control signals depends on the instruction being executed
- The main controller can be implemented using a FSM that generates the proper control signals for the multiplexers select signals and register enable signals over every clock cycle of instructions execution
  - Select signals are listed only when their value matters; otherwise, they are don't cares. Enable signals are listed only when they are asserted; otherwise, they are 0





Microarchitecture of the multicycle processor

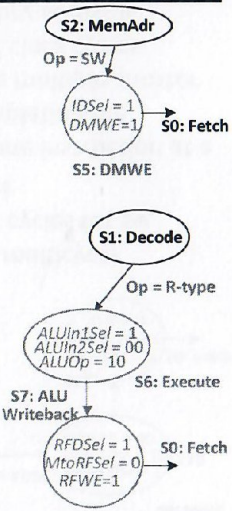
Instruction fetch, updating the **PC** *, and instruction decode*



- The FSM enters the *fetch state* upon reset to fetch the instruction from MEM at the address stored in the **PC**
- *IDSel=0*, so the memory address is taken from the **PC**
- On the positive edge of the clock, *IRWE* is asserted to write the instruction into the instruction register IR
- In the same state, while fetching an instruction, the **PC** should be incremented by 1 to point to the next instruction. The ALU is used in this state to compute **PC**+1; *ALUIn1Sel*=0, so *ALUIn1* comes from the **PC** ; *ALUIn2Sel*=01, so *ALUIn2* is the constant *1; ALUOp*=00, so the ALU decoder produces *ALUsel*=010 and the ALU adds
- To update the **PC** with the new value, *PCSel*=00, and *PCWE* is asserted. When *PCSel*=00, the output of ALU is written directly to PC
- The next step is to decode the instruction. Decoding involves examining the **opcode** of the instruction to determine the correct value of the control signals for the execution of the fetched instruction. In this clock cycle, the decode state, the processor always reads the two sources from the RF at **rs** and **rt** and also the immediate is sign-extended. *PCWE*=0 so the PC is not update

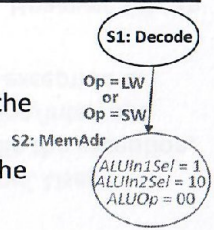
sw *instruction and R-type instructions*

- After calculating the effective address in S2, if the instruction is **sw**, the data read from the register file at **rt** is written into the memory. In S5, *IDSel*=1 to select the effective address computed in S2 and saved in *ALUOutR*. *DMWE* is asserted to write onto the memory
- For R-type instructions, after an instruction is decoded in S1, in S6 the ALU inputs, A and B registers are selected by *ALUInSel*=1 and *ALUIn2Sel*=00 and the ALU operation is performed indicated by the **funct** field of the instruction. *ALUOp*=10 for all R-type instructions and the *ALUOut* is stored in *ALUOutR*
- In S7, *ALUOutR* is written to the RF at **rd** by setting *RFDSel*=1, *MtoRFSel*=0, and *RFWE*=1



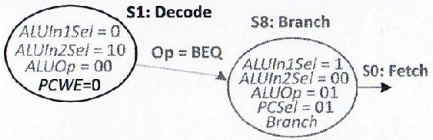
`lw` *instruction*

- After the decode state, the FSM proceeds to one of several possible states, depending on the **opcode**
- If the instruction is **`lw`** or **`sw`**, the multicycle processor computes the effective address by setting *ALUIn1Sel*=1 to select register A, *ALUIn2Sel*=10 to select *SImm* , and *ALUOp*=00, so the ALU adds. The effective address is stored in the *ALUOutR* register for use in the next step
- In the next step S3, *IDSel*=1 to select the effective memory address that was computed in the previous state. In this state, the memory is read from this address and saved in the DR register
- In state S4, DR is written to the RF. *MtoRFSel*=1 to select DR; *RFDSel*=0 to select the destination register from **`rt`**; *RFWE* is asserted
- Finally, the FSM returns to the initial state S0 to fetch the next instruction



beq *instruction*

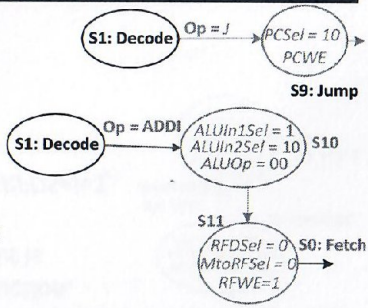
- The processor compares the two source registers to determine whether the branch should be taken and also calculate the BTA. This requires two uses of the ALU



- Since the ALU was not used during S1 when the processor decodes the instruction and reads two registers **rs** and **rt**, it can use ALU to calculate the branch target address **PC**+*Simm*. *ALUIn1Sel*=0 to select the incremented **PC**, *ALUIn2Sel*=10 to select *SImm*, and *ALUOp*=00 to add. The calculated BTA is stored in *ALUOutR*. Note that if the instruction is not a conditional branch instruction, the computed BTA will not be used in the subsequent state. So not every computation is utilized
- Then in S8, the ALU subtracts the two registers *A* and *B* read from registers addressed by the **rs** and **rt** fields of the instruction, and determines whether the result is zero. If it is, the *ALUOutR*, which holds the BTA, is written into the **PC** and the new output of the ALU, which is zero in case the two input registers are equal, is stored in the register *ALUOutR*. *PCSel*=1 to take the BTA from *ALUOutR*, and *Branch*=1 to update the **PC** with this address. The *PCSel* multiplexer chooses **PC** from either *ALUOutR* (in S0) or *ALUOutR* (in S8)

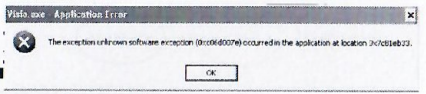
`J` *and* **`addi`** *instruction*

- After decoding the `J` instruction in S1, in S9 *PCSel* = 10 and *PCWE*=1 to store *Jaddr* into the **PC**
- For the **`addi`** instruction, the datapath is already capable of adding registers to immediates. Thus in S10, register A is added to *SImm* and the result, is stored in the register *ALUOutR*
- In S11, *ALUOutR* is written to the RF specified by **`rt`**
- Note that S2 and S10 are identical (both add *A* to *SImm*) and could be merged into a single state



- The designed FSM-based control unit shows that the designed multicycle processor requires three cycles for **`beq`** and **`j`** instructions, four cycles for **`sw`**, **`addi`**, and R-type instructions, and five cycles for **`lw`** instructions
- Note that the multicycle microarchitecture still executes only one instruction at a time (i.e., the execution of an instruction only begins after completing the execution of preceding instructions), but each instruction takes multiple shorter clock cycles and each instruction may take different number of clock cycles, depending on the complexity of instruction and whether memory access is required. So in both single-cycle and multiple-cycle processors the next instruction begins when the previous one completes

Exceptions and interrupts

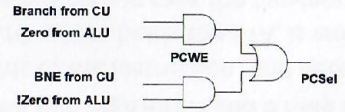


- A *trap* (or *exception*) is a software generated event. Exceptions are internal, produced by the control unit while executing instructions, and are considered to be synchronous because the control unit issues them only after terminating the execution of an instruction. e.g., overflow, division by zero, undefined instructions, and incorrect data address are examples of internal errors in a program. **`syscall`** instruction, which is used to implement operating system services (functions), is also considered an exception. An exception occurs at the same instruction every time a program is executed with the same data
- *Interrupts* are external and asynchronous. Interrupts are generated by hardware devices outside the CPU at arbitrary times with respect to the CPU clock signals to notify the CPU of external events. e.g., I/O device requests, key-presses on a keyboard. Interrupts happen with no relation to the program being executed. Even if a program is run multiple times with the same input data, the timing of the key presses will most likely vary. Another example is read and write requests to disk. The disk controller is external to the executing process and the timing of a disk operation might vary even if the same program is executed several times. An input pin to the CPU, called *interrupt request* (IRQ), will allow an external device to interrupt the processor

Example

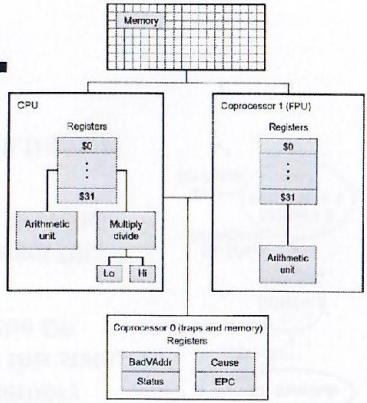
- Modify the multi-cycle MIPS processor to implement the branch on less than or equal to zero **bne** instruction

```
S9: BNE
AluIn1Sel = 1
AluIn2Sel = 00
AluOP = 01
PCSel = 01
BNE
```



- The increment and store **incs** `$s1, D($s0)` instruction is equivalent to the sequence of two instructions: **addi** `$s1 , $s1, 1` and **sw** `$s1, D($s0)`. Update the multi-cycle MIPS processor microarchitecture to support this instruction

MIPS Coprocessors



- A MIPS processor consists of an *integer processing unit* (the CPU), *Coprocessor-1* (C1) to perform floating-point operations, and *Coprocessor-0* (C0) to handle exceptions
- The *floating-point unit* (FPU) has 32 32-bit floating-point registers **`$f0`**, **`$f1`**, ... Each register can hold one single-precision floating-point number
- Single-precision numbers are often stored in even registers so that they can be easily upgraded to double-precision values. e.g., a double-precision number in **`$f2`** is actually stored in **`$f2`** and **`$f3`**
- The CPU operates in one of the two possible modes: *user* and *kernel*. User programs run in *user mode*. When an exception happens, to handle the exception, execution transitions from user mode to *kernel mode*. The exception/interrupt handler uses the CPU. Execution resumes in user mode when the exception or interrupt has been handled.
- C0 is always present, unlike C1, which may or may not be present. However, C0 can only be used in kernel mode

`mfc` *and* `mtc`

- When running in kernel mode, the registers of C0 can be accessed using the **`mfc0`, `mtc0`, `lwc0`, `swc0`** instructions

Instruction	Comment
`mfc0 Rdest, C0src`	Move the content of coprocessor's register `C0src` to `Rdest`
`mtc0 Rsrc, C0dest`	Integer register `Rsrc` is moved to coprocessor's register `C0dest`
`lwc0 C0dest, address`	Load word from address in register `C0dest`
`swc0 C0src, address`	Store the content of register `C0src` at address in memory

- The *move from Coprocessor0* **`mfc0`** instruction copies an exception register into one of the general-purpose registers in the register file
  - For example, **`mfc0 $k0, $13`** copies the cause register into **`$k0`**
- The *move to Coprocessor0* **`mtc0`** instruction copies to an exception register from one of the general-purpose registers

`mfc0 Rdest,C0src`

31-26	25-21	20-16	15-11	10-0
010000	00000	C0Src	Rdest	00000000000

`mtc0 Rsrc,C0dest`

31-26	25-21	20-16	15-11	10-0
010000	00100	Rsrc	C0dest	00000000000

Exception Program Counter (EPC)

- The special-purpose *exception program counter* (**EPC**) register (**$14**) is used to store the address of the instruction to execute after the return from the exception handler
- **Case I–Interrupts**: In the case of an interrupt, the PC has already been incremented pointing to the next instruction at the moment the control is transferred to the exception handler
- Thus, the EPC contain the address of the instruction to execute after the return from the exception handler

```
# the return sequence from the
# exception handler for interrupts
mfc0 $k0,$14 # Store EPC in $k0
rfe      # return from exception
jr $k0   #replace PC with the return address
```

- **Case II–Exceptions**: In the case of a trap or a **syscall**, the **EPC** contains the address of the instruction that has generated the trap

```
# the return sequence from the exception
# handler for synchronous exceptions
mfc0 $k0, $14 # Store EPC in $k0
addiu $k0, 1  # make sure it points to next instruction
rfe           # return from exception
jr $k0        # replace PC with the return address
```

- To avoid executing the instruction again at the return from the exception handler, the return address must be incremented by four (in MIPS processor and by one in our processor), making sure the instruction that follows in the flow will be executed

Calling exception handler routine

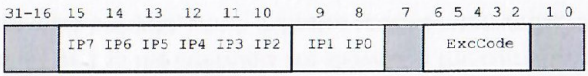
- When an exception/interrupt occurs, it need to be serviced, e.g., for an arithmetic overflow exception, an error message can be printed and the program may be terminated
- When an exception happens, the control is always transferred to a subroutine called *exception handler* to provide certain services such as resolving an overflow; terminating the execution of the offending program and reporting an error; communicating with an external device
- In MIPS, the exception handler for all exceptions is located at **`0x80000180`**. When an exception occurs, the processor always jumps to this instruction address, regardless of the cause. Thus similar to branches, exceptions disrupt the normal sequence of instruction execution and change the control flow in a program. However, when an exception happens, the return address can not be saved in **`$ra`** since it may overwrite a return address that has been placed in **`$ra`** before the exception
- Many architectures do not distinguish between interrupts and exceptions, often using the older name interrupt to refer to both types of events
  - For example, Intel x86 uses interrupt while MIPS uses the term exception to refer to any unexpected change in control flow without distinguishing whether the cause is internal or external

Exception handling registers in C0 - Cause register

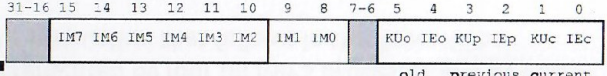
- When an exception or interrupt occurs, a 5-bit *unsigned exception code* is stored in the 32-bit *cause register* **CR** (**$13**) in bits 2-6
- The exception handler routine reads the **CR** to indicate what caused the exception
  - Codes from 1 to 3 are reserved for virtual memory
  - 11 is reserved to indicate a particular coprocessor is missing
  - Codes above 12 are reserved for floating-point exceptions

Code	Name	Description
0	INT	Interrupt
4	ADDRL	Load from an illegal address
5	ADDRS	Store to an illegal address
6	IBUS	Bus error on instruction fetch
7	DBUS	Bus error on data reference
8	SYSCALL	`syscall` instruction executed
9	BKPT	`break` instruction executed
10	RI	Reserved instruction
12	OVF	Arithmetic overflow

- The **CR** also provides information about what interrupts are pending (i.e., has not been serviced yet). The *interrupt pending bits* IP2–IP7 become 1 if an interrupt is pending. IP0 and IP1 are reserved for interrupts generated by software



Status register



- Multiple exceptions and interrupts can occur simultaneously in a single clock cycle. The exceptions should be prioritized so that they are serviced in an appropriate order. The processor would not service any external interrupts before it is finished executing the current instruction. Thus the external device may have to wait for several clock cycles. The processor informs the external device by an output pin, called *interrupt acknowledge* (IACK), that it has received its interrupt
- The 32-bit *status register* **SR** (**$12**) contains status information on bits 0-5 and *interrupt mask* IM on bits 10-15 to mask (prevent) interrupts and exceptions from occurring during handler execution. If an interrupt/exception occurs when its mask bit is set to 0, then the interrupt/exception will be ignored
- Bits 0-5 implement a three level stack: old, previous, and current. When an exception occurs, the previous status SR[3:2] is saved as the old state SR[5:4], and the current state SR[1:0] is saved as the previous state (the old state is lost) and the current state bits are set to 00 (kernel mode with interrupts disabled)
- The *Interrupt Enable* bits IE indicate whether interrupts are enabled IE = 1 or not IE = 0 in the respective state. Note that the IE bit enables or disables all interrupts while the individual interrupt mask IM bits enable or disable individual interrupts

Handling exception

- When an exception or interrupt occurs, the following action should be done:
  - The processor enters the kernel mode; Further interrupts or exceptions are disabled from occurring by setting the status register **SR**; Move the current **PC** into the **EPC** register; Record the reason for the exception in the cause register **CR**; Jump to the exception handler address **0x80000180** in the OS kernel
- The exception handler uses **mfc0** to examines the cause and responds accordingly
  - As with any procedure, the exception handler must save any CPU registers (also called the CPU *state* or CPU *context*) that it may modify in the stack, and then restore them before returning control to the interrupted program. This way the exception handler preserves the *state of the program* that was interrupted such that its execution can continue at a later time
  - Note that exception handler can use the general-purpose registers **$26** (**$k0**) and **$27** (**$k1**) without having to save them first
- After handling the exception, the control is returned to the program that was executing when the exception occurred. To return from the exception handler:
  - It restores the registers from the stack; Copies the return address from **EPC** (for interrupts or **EPC+1** for exceptions) to **$k0** using **mfc0**. The program should continue its execution after the instruction where the exception occurred; Re-enable interrupts and exceptions, by resetting the status register **SR**; Returns using **jr $k0**

Status register

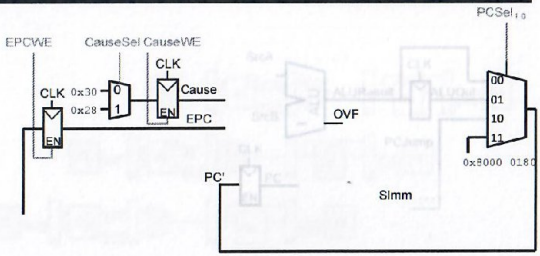
- Both IEc and IMi must be set for interrupt "i" to be enabled. To enable interrupts:
  - IEc (SR[0]) must be 1 so the interrupts are allowed in the current state
  - The interrupt mask bit(s) IM, which indicates specific interrupt(s), should be enabled
- If IEc=0, then the interrupts are disabled. Disabling interrupts does not mean they are eliminated. It means their effect is deferred. At the time interrupts are enabled again, the hardware will treat any pending interrupt as if it just occurred and immediately invoke the exception handler
- When enabling interrupts within the exception handler, the relevant information (EPC, Cause, Status) must be saved before enabling interrupts
- It is the programmer's responsibility to make sure the exception handler does not execute any instruction that cause an exception (or traps)
- The Kernel/User KU bit indicates the exceptions level. KUc=1 when the program is running in user mode and KUc = 0 when in kernel (e.g., KUp=0 indicates the processor was in kernel mode when last exception occurred)
- At the return from the exception handler (by executing a `rfe` instruction), the processor must be in the same state it was when the exception happened. The previous state becomes the current state and the old state becomes the previous. The old state is not changed

Exception handling in multi-cycle processor

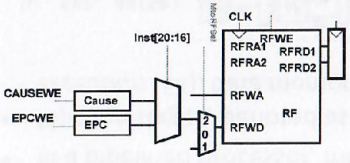
- Let's modify the multicycle processor to support two types of exceptions: undefined instructions and arithmetic overflow. When an exception occurs, the processor copies the **PC** to the **EPC** register and stores a code in the **CR** register indicating the source of the exception. The processor jumps to the exception handler at memory address **0x80000180**
- To handle exceptions:
  - We must add **EPC** and **CR** registers to the datapath
  - The two new registers have write enables, *EPCWE* and *CauseWE*, asserted by the control unit to store the **PC** and exception cause, respectively. Exception causes include **0x28** for undefined instructions and **0x30** for overflow
  - The *PCSel* multiplexer is extended to accept the exception handler address
  - Upon overflow, the *OVF* output of the ALU will become high and it triggers an exception

Datapath for handling exceptions

- Datapath handling overflow
- *CauseSel* multiplexer signal selects the cause value



- To support the **`mfc0`** instruction, *MtoRFSel* multiplexer is extended to select one of the **`C0`** registers (here the **`CR`** and **`EPC`** registers) and write them to the RF
- To select one of the **`C0`** registers, a new multiplexer is added, where **`Inst[20:16]`** field of **`mfc0`** instruction specifies the **`C0`** register

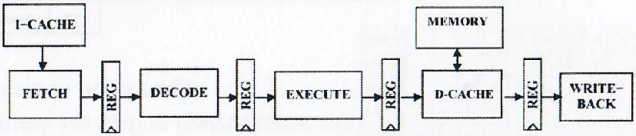


`mfc0 Rdest,C0src`

31-26	25-21	20-16	15-11	10-0
010000	00000	C0Src	Rdest	00000000000

Pipelined processors

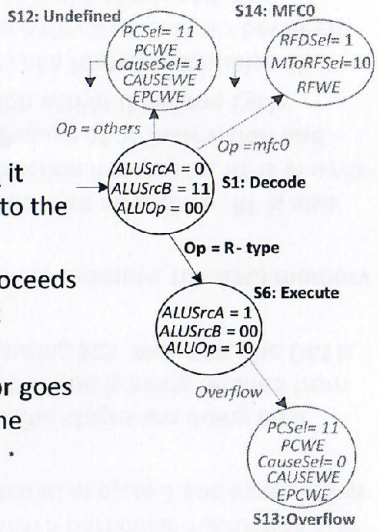
- In a pipelined processor architecture, the complete execution of an instruction is broken up into a series of smaller steps. Specifically, five stages are used: *fetch*, *decode*, *execute*, *memory*, and *writeback*. They are similar to the five steps that the multicycle processor used to perform `lw`
- The pipelined datapath is formed by dividing the SC datapath into five stages separated by *four pipeline registers*



- In the *fetch* stage, the processor reads the instruction from the IM
- In the *decode* stage, the processor reads the two source operands from the RF and decodes the instruction to generate the corresponding control signals
- In the *execute* stage, the processor performs a computation using the ALU
- In the *memory* stage, the processor reads from or writes into the DM
- Finally, in the *writeback* stage, the processor writes the result of ALU or the value read from DM into the RF, when applicable
- While the pipelined microarchitecture of the processor is transparent to programmers, compiler requires a detailed knowledge of the microarchitecture to generate efficient machine codes

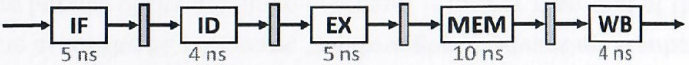
Controller supporting overflow, undefined instructions and **`mfc0`**

- The control unit FSM includes three new states to support overflow, undefined instructions, and **`mfc0`**
- When an exception occurs, the instruction is discarded and the register file is not written
- If the controller receives an undefined instruction it proceeds to S12, saves the **PC** in **EPC**, writes **0x28** to the **CR** register, and jumps to the exception handler
- If the controller detects arithmetic overflow, it proceeds to S13, saves the **PCR** in **EPC**, writes **0x30** in the **CR** register, and jumps to the exception handler
- When a **`mfc0`** instruction is decoded, the processor goes to S14 and writes the appropriate **C0** register to the main register file



Throughput and latency

- The pipeline stages are chosen so that stages take relatively the same amount of time to complete. If the stages take almost the same amount of time, since each stage takes only one-fifth of the clock cycle of a SC processor, the *clock frequency* is almost five times faster and hence, the *throughput* is ideally enhanced five times. Because of such great clock frequency advantage for little cost (adding registers to the datapath), all modern high-performance microprocessors are pipelined
- An instruction (data) pipeline is said to be *fully-pipelined* if it can accept a new instruction (data) every clock cycle
- Example:



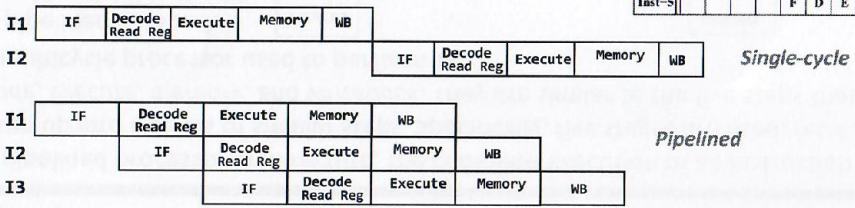
Throughput =**1/clock period**=1/10 *ns*=100 Million instruction/sec (*This is correct only for a fully-pipelined datapath*)

- The *latency* of each instruction is $L \times T_{clk}$, where L is the number of pipeline stages. In this example, latency = 5×10=50 ns (ignoring the overhead of pipeline registers)
- The latency of $L \times T_{clk}$ would only be correct when executing independent instructions
- High-performance microprocessors execute billions of instructions per second and throughput is more important than latency

Pipelined execution of instructions

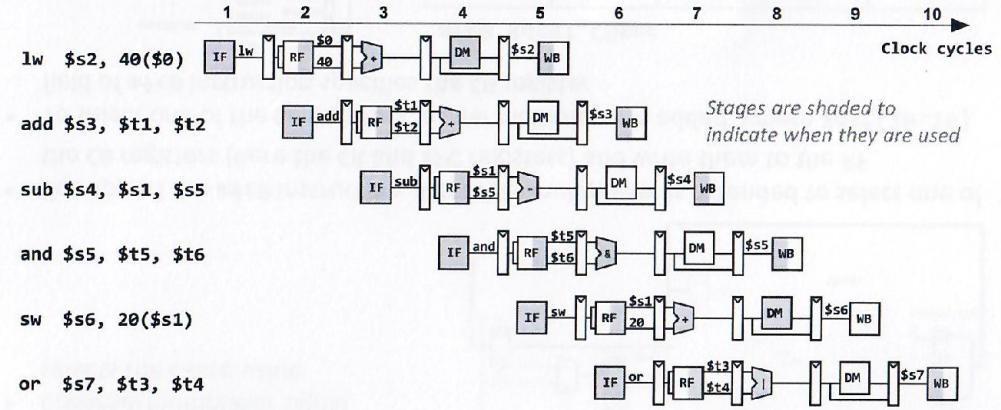
- In the single-cycle processor, an instruction is read from memory; then the instruction is decoded and the operands are read from the register file; then the ALU executes the necessary computation; then the data memory may be accessed, and finally the result may be written into to the register file
- In the SC processor, the second instruction begins when the first completes. In a pipelined processor, various stages of instructions execution are overlapped to increase performance, i.e., several steps of different instructions are executed simultaneously in different stages of a pipelined processor
- This diagram shows the flow of instructions through the pipeline. In a five-stage pipelined, up to five instructions can be executed simultaneously, one in each stage

cycle	1	2	3	4	5	6	7	8	9
Inst-1	F	D	E	M	W				
Inst-2		F	D	E	M	W			
Inst-3			F	D	E	M	W		
Inst-4				F	D	E	M	W	
Inst-5					F	D	E	M	W



Five-stage pipelined design example

- In a pipelined processor, multiple instructions can be processed concurrently
- Pipeline stages, denoted as instruction fetch (*IF*), instruction decode (*ID*), *execution* (*EX*), data memory (*DM*), and register file *writeback* (*WB*)



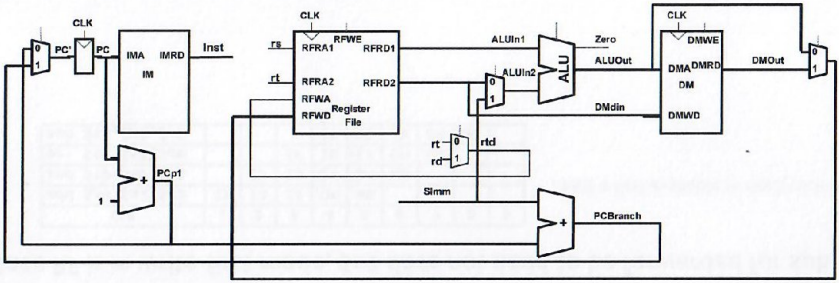
Single-cycle vs. pipelined processor

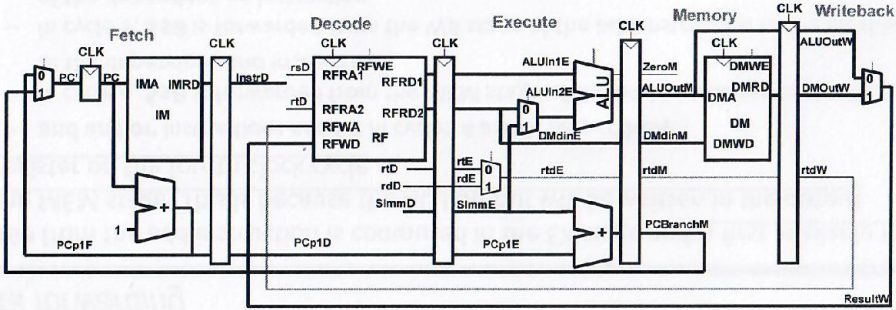
- Assume that the SC processor has an instruction latency of 10 ns and hence, a throughput of 1 instruction per 10 ns (100 million instructions per second). Assume that for the pipelined processor, each pipeline stage takes 2.5 ns, set by the *slowest stage*, which is the memory access. The instruction latency is $5 \times 2.5 = 11$ ns and the throughput is one instruction per 2.5 ns (400 million instructions per second). Even though the throughput is not quite five times as great for a five-stage pipeline as for the SC processor, the throughput advantage is substantial
- Note that the latency of the pipelined processor is longer than that of the single-cycle processor because: (i) the stages are not perfectly balanced with equal delay; and (ii) every stage has a sequencing timing overhead
- Thus, pipelining (i) reduces the clock cycle time of the SC microarchitecture; and (ii) improves throughput but not the inherent execution time or the latency of an instruction execution
- An *ideal pipelined processor* (executing a program with no dependent instructions) would have an average CPI of one

Five-stage pipelined design example

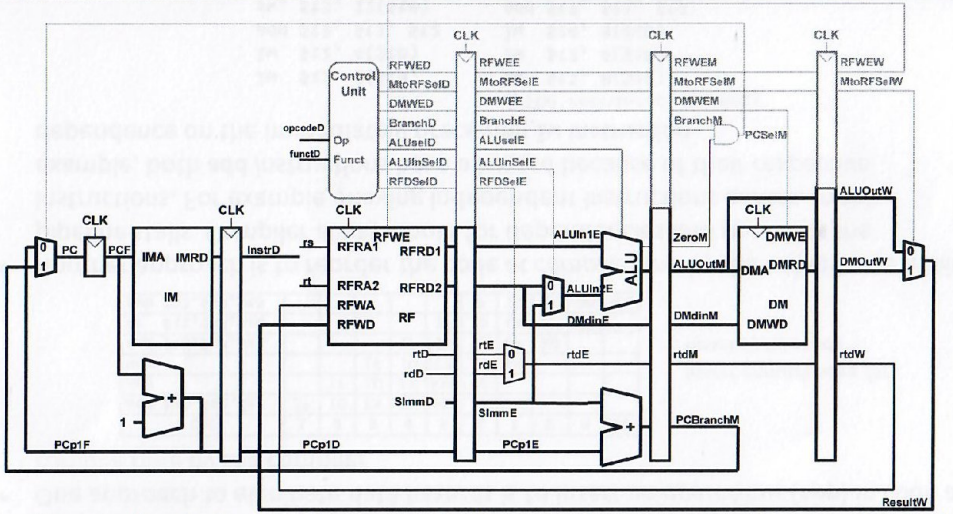
- Reading across a row shows the clock cycles in which a particular instruction is in each stage. For example, the **sub** instruction is fetched in cycle 3 and executed in cycle 5
- Reading down a column shows what different pipeline stages are doing on a particular cycle. For example, in cycle 6, the **or** instruction is being fetched from IM, **$s1** is being read from the RF, the ALU is computing **$t5 AND $t6**, the DM is idle, and the RF is writing a sum to **$s3**
- Stages are shaded to indicate when they are used. For example, the data memory is used by **lw** in cycle 4 and by **sw** in cycle 8
- RF is used in the decode stage to read two operands from **rs** and **rt** . RF is also used in the WB stage to write the result of an instruction if required. RF is in *write-first mode*, i.e., data can be written first by the WB stage of an instruction and then read back in the ID stage of another instruction within the same cycle
- When a subsequent instruction requires the results of a former instruction that has not yet completed, or in other words, there are *data dependencies* between instructions, a *data hazard* occurs if the dependency is not addressed. For example, if the **add** used **$s2** rather than **$t2**, a hazard would occur because the **$s2** register has not been written by the **lw** by the time it is read by the add. Without special treatment, the pipeline will compute the wrong result

SC and pipelined datapaths





Pipelined datapath and control unit



- The pipelined microarchitecture is similar in hardware requirements to the single-cycle processor, but it adds pipeline registers, along with multiplexers and pipelined control signals
- Note that the PC can be updated in the IF or MEM stage (if the instruction is a conditional branch)

Control unit and exception

- The control unit examines the **opcode** and **funct** fields of the instruction in the ID stage to produce the corresponding control signals
- The pipelined processor uses the same control signals as the single-cycle processor and therefore the same control unit can be used. However, the control signals must be pipelined along with the data and addresses so that they remain cohesive for executing an instruction
- Control signals are given a suffix (*F*, *D*, *E*, *M*, or *W*) to indicate the pipeline stage in which they reside. For example, the address *rtd* and the control signal *RFWE* must be pipelined through the EX, MEM and WB stages so they arrive at the RF at the same cycle as the data *ResultW* (from memory or ALU): *rtdD, rtdE, rtdM,* and *rtdW* and *RFWED, RFWEE, RFWEM,* and *RFWEW*
- Exceptions in pipeline processors, such as an arithmetic overflow when executing an **add** instruction (in the EX stage) or an undefined instruction in the ID stage, or a memory address fault in IF and MEM stage, are handled similarly to multi-cycle processors. For handling exceptions, the pipeline let all prior instructions complete, flush all following instructions, set the **cause** register to show the cause of the exception, save the address of the instruction following the offending instruction in the **EPC**, and then jump to the *exception handler code* and pass the control to the OS (kernel mode)

Hazards

- In the following pipeline data hazards occur because:
  - Both the **and** and **or** instructions are dependent on the **add** instruction
  - The **add** instruction writes a result into **$s0** in cycle 5
  - The **and** instruction reads **$s0** in cycle 3, obtaining the wrong value
  - The **or** instruction reads **$s0** in cycle 4, again obtaining the wrong value
- The **sub** instruction reads the correct value of **$s0** in cycle 5, because the **sub** instructions is not dependent on the **add** instruction (the RF operates in write-first mode and the value of **$s0** was written in cycle 5 first by **add**)

CLK	1	2	3	4	5	6	7	8	9
add $s0,$s2,$s3	IF	ID	EX	DM	WB				
and $t0,$s0,$s1		IF	ID	EX	DM	WB			
or $t1,$s4,$s0			IF	ID	EX	DM	WB		
sub $t2,$s0,$s5				IF	ID	EX	DM	WB	

- This diagram shows that hazards may occur in the pipeline when an instruction writes a register (in WB stage) and either of the two subsequent instructions read that register (in ID stage). This is called a *read after write* (RAW) *data hazard*

Eliminating data hazards using **nops**

- Since a RAW data hazard prevents next instruction to be executed during its designated clock cycle, hazards reduce the maximum achievable throughput
- One approach to eliminate data hazards is to insert *no operations* (**nop**) in code at *compile time* by the compiler

CLK	1	2	3	4	5	6	7	8	9	10
add $s0,$s2,$s3	IF	ID	EX	DM	WB					
nop		IF	ID	EX	DM	WB				
nop			IF	ID	EX	DM	WB			
and $t0,$s0,$s1				IF	ID	EX	DM	WB		
or $t1,$s4,$s0					IF	ID	EX	DM	WB	
sub $t2,$s0,$s5						IF	ID	EX	DM	WB

Insert enough **nops** *for result to be ready*

- Another approach is to reorder the code at *compile time* by the compiler to avoid pipeline stalls. Compiler always looks for dependencies and re-orders the instructions. For example, moving independent instructions earlier. In this example, both **add** instructions have a hazard because of their respective dependence on the immediately preceding **lw** instruction

```
lw  $t1, 0($t0)          //The reordered sequence
lw  $t2, 4($t0)          lw  $t1, 0($t0)
add $t3, $t1, $t2        lw  $t2, 4($t0)
sw  $t3, 12($t0)         lw  $t4, 8($t0)
lw  $t4, 8($t0)          add $t3, $t1, $t2
add $t5, $t1, $t4        sw  $t3, 12($t0)
sw  $t5, 16($t0)         add $t5, $t1, $t4
                         sw  $t5, 16($t0)
```

Data forwarding

- **$s0** from the **add** instruction is computed in the EX stage and is first available in the MEM stage. This is because the ALU output will be written in the output register on the fourth clock cycle
  - **and** and **or** instructions need it in cycles 4 and 5, respectively
  - In cycle 4, **$s0** is forwarded from the MEM stage of the **add** instruction to the EX stage of the dependent **and** instruction
  - In cycle 5, **$s0** is forwarded from the WB stage of the **add** instruction to the EX stage of the dependent **or** instruction
- Since RF is in write-first mode, **$s0** does not need to be forwarded for **sub**

CLK	1	2	3	4	5	6	7	8	9
add $s0,$s2,$s3	IF	ID	EX	DM	WB				
and $t0,$s0,$s1		IF	ID	EX	DM	WB			
or $t1,$s4,$s0			IF	ID	EX	DM	WB		
sub $t2,$s0,$s5				IF	ID	EX	DM	WB	

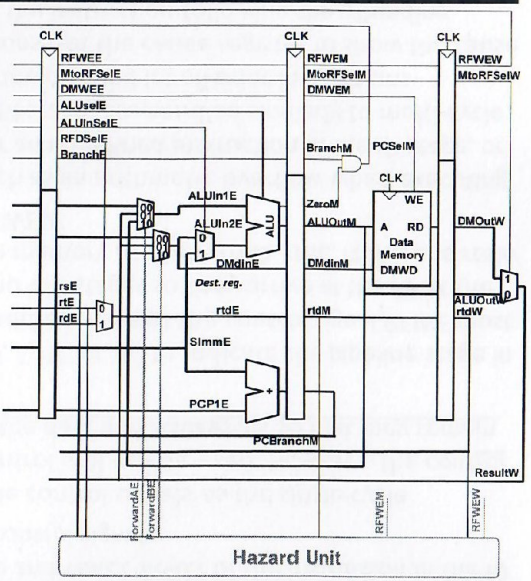
$s0 is first available in clock cycle 4

Eliminating data hazards by stalling the pipeline

- A *data hazard* occurs when an instruction tries to read a register that has not yet been written back by a previous instruction and hence, not available
- One solution to eliminate data hazards is to *stall the pipeline* at runtime. In this case, the dependent instruction must wait for another instruction to complete. This can be done by disabling the pipeline stage of the dependent instruction
- A more efficient approach to eliminate data hazards at run time is *data forwarding*. In this approach, the required data element for the dependent instructions is retrieved from the pipeline registers of an earlier instruction rather than waiting for it to be written into the RF. More precisely, a destination register result from the MEM or WB stage is forwarded to the EX stage of the dependent instruction. As such, data forwarding can eliminate some RAW hazards without slowing down the pipeline
- Supporting data forwarding requires adding *forwarding multiplexers* in front of the ALU to select the operands from either the RF (as before), the MEM or WB stages

Pipelined processor with data forwarding support

- The *hazard unit* is a module in the control unit and is responsible for detecting data hazards and handling them appropriately, so that the processor executes the program correctly despite the dependency among instructions and pipelining
- Upon detecting data hazards, the hazard unit generates control signals for *the forwarding multiplexers* to choose operands from the RF or from the results in the MEM or WB stage of an earlier instruction in the pipeline and forward it to the EX stage of the dependent instruction



- When an instruction in the EX stage has a source register (*rsE* or *rtE*) matching the destination register of an instruction in the MEM *rtdE* or in the WB *rtdW* stage, hazard unit set the select lines of the forwarding multiplexers appropriately

Hazard unit logic

- The hazard unit compares the *two source registers* of the instruction in the EX stage with the *destination registers* of the instructions in the MEM and WB stages. If they match, the required data is in the *input pipeline registers* of the MEM (*ALUOutM*) or WB stage (*ResultW*) and is forwarded to the input of the ALU
- Matching registers is not sufficient for forwarding data. The hazard unit also receives the *RFWEM* and *RFWEW* control signals from the *MEM* and WB stages, respectively, to know whether the destination register will actually be written. For example, the **sw** and **beq** instructions do not write results to the RF and hence, do not need their operands to be forwarded. Note that **$0** is hardwired to 0 and is not required to be forwarded
- The function of the forwarding logic for the forwarding multiplexer select lines *ForwardAE* (associated with *ALUIn1*) and for *ForwardBE* (associated with *ALUIn2E*) can be written as *ForwardAE = 00* if ((*rsE*!=0) AND *RFWEM* AND (*rsE*==*rtdM*)) then *ForwardAE*=10
  else if ((*rsE*!=*0*) AND *RFWEW* AND (*rsE*==*rtdW*)) then *ForwardAE*=01
  ForwardBE = 00 if ((*rtE*!=*0*) AND *RFWEM* AND (*rtE*==*rtdM*)) then *ForwardBE*=10
  else if ((*rtE*!=*0*) AND *RFWEW* AND (*rtE*==*rtdW*)) then *ForwardBE*=01
- Priority of the above if statement implies that if both the *MEM* and *WB* stages contain matching destination registers, the *MEM* stage should have priority, because it contains the more recently executed instruction and hence, most recent data value and hence, *rtdM* is checked first and then *rtdW*

Stalling the pipeline

- *LW latency* refers to the number of clock cycles between a **lw** instruction and an instruction that can use the result of the **lw** without stalling the pipeline. **lw** instruction has a *one-cycle latency*, i.e., the following dependent instruction cannot use its result until one cycle later
- When a stage is stalled, all previous pipeline stages must also be stalled, so that no subsequent instructions are read into the pipeline
- In this example, **and** is stalled in the ID stage for one clock cycle and the **or** instruction is stalled in the IF stage

CLK	1	2	3	4	5	6	7	8	9
lw $s0,$40($0)	IF	ID	EX	DM	WB				
and $t0,$s0,$s1		IF	ID	ID	EX	DM	WB		
or $t1,$s4,$s0			IF	IF	WID	EX	DM	WB	
sub $t2,$s0,$s5				IF	ID	EX	DM	WB	

Stall

Inst in the EX is lw and has the dest reg equal to the source reg of the inst in ID

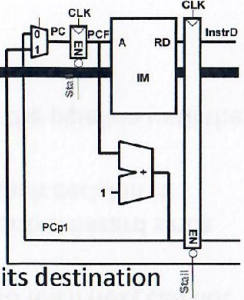
- In cycle 5, the result can be forwarded from the WB stage to the EX stage
- Source **$s0** of **or** is read directly from the RF, with no need for forwarding

Stalling the pipeline

- Data forwarding solves RAW data hazards only when the result of an instruction is computed in the EX stage and becomes available in the MEM and WB stages. For instructions such as **lw**, which result is available at the WB stage, the data is available too late to be forwarded to the dependent instruction in the EX stage
- For instance, in this example, the **lw** instruction receives data from memory at the end of cycle **4** (i.e., the read data from DM is available at the input register of the WB stage). However, **and** needs that data as a source operand at the beginning of cycle 4 and hence, data is available too late and forwarding cannot solve this hazard. In this case, there is no choice but *stalling* the pipeline at run time (i.e., holding the registers values by *disabling* the registers' enable inputs) until the data is available for the dependent instructions to use

CLK	1	2	3	4	5	6	7	8	9
lw $s0,$40($0)	IF	ID	EX	DM	WB				
and $t0,$s0,$s1		IF	ID	EX	DM	WB			
or $t1,$s4,$s0			IF	ID	EX	DM	WB		
sub $t2,$s0,$s5				IF	ID	EX	DM	WB	

Stalling the pipeline



- To stall a pipeline stage, its pipeline register is disabled using its enable EN control input to hold its value. The hazard unit use *Stall* control signal as the enable control input of the **PC** and the **IR** in the IF and in the ID stages, respectively
- If the instruction in the EX stage is **lw** (i.e., *MtoRFSelE*=1) and its destination register *rtE* matches either source operand of the instruction in the ID stage (*rsD* or *rtD*), the hazard unit asserts the *Stall* control signal so the instructions in the ID and IF stages are stalled (no new instruction should be fetched) until the source operand is ready
- The logic to compute the stall control signals can be written as:

  LWStall = MtoRFSelE AND ((*rtE == rsD*) OR (*rtE == rtD*));

  Stall=LWStall
- Stalls degrade performance, so they should only be used when necessary
- When stalling the IF and ID stages, the EX, MEM, and WB stages are unused. An unused stage propagating through the pipeline is called a *bubble*, and it behaves like a **nop** instruction. The bubble can contain random data that has no effect on the instruction execution results

Branch instructions and control hazards

- Branches have either only one direction (for unconditional `j`, `jal`, and `jr` instructions) or two directions (in case of conditional branch instructions)
- If the instruction is a branch, when does the processor know whether the conditional branch is taken (execute instruction at the BTA) or not taken (execute the following instruction)?
- If the ALU is used to evaluate the branch condition, the branch decision is not determined until the end of *EX* stage (or until the beginning of *MEM* stage determined at the MEM input pipeline register)
- In this example, assume the branch is taken to `slt`. Since the branch decision is not made until cycle 4, by which point the three subsequent instructions have already been fetched (they are in IF, ID, and EX), these instructions must be flushed (three pipeline registers must be cleared), and the `slt` instruction is fetched in cycle 5

CLK	1	2	3	4	5	6	7	8	9
beq $t1,$t2,40	IF	ID	EX	DM	WB				
and $t0,$s0,$s1		IF	ID	EX	DM	WB			
or $t1,$s4,$s0			IF	ID	EX	DM	WB		
sub $t2,$s0,$s5				IF	ID	EX	DM	WB	
...									
slt $t3,$s2,$s3					IF	ID	EX	DM	WB

- For deeper pipelines, branch decision can be determined even *later* in the pipeline, incurring more instructions in the pipelined to be flushed. The *penalty of a misprediction* grows with the depth of the pipeline, since a larger number of instructions will have to be flushed

Reducing branch misprediction penalty

- Instead of using the ALU in the EX stage for determining the branch decision (taken or not taken), an *equality comparator* can be added to the ID stage to compare the values of two registers **rs** and **rt** read from the RF
- **PC** is incremented in the IF stage. The BTA (i.e., **PC**+*SImm*) can be computed in the ID stage and used only if branch should be taken. In this case, the branch decision and BTA are both available at the end of the ID stage
- With the calculation of the BTA and the evaluation of the branch condition in the ID stage, if the branch is found to be taken, then the processor should flush the incorrectly-fetched instruction currently in IF by clearing the IF register. Therefore, taken branches will have only one-cycle penalty (flushing the subsequent instruction to the branch instruction) rather than three. In the 5-stage pipeline thus a taken branch completes in two cycles
- The BTA and branch decision are determined in the ID stage while the **and** instruction has already been fetched

CLK	1	2	3	4	5	6	7	8	9
beq $t1,$t2,40	IF	ID	EX	DM	WB				
and $t0,$s0,$s1		IF	ID	EX	DM	WB			
or $t1,$s4,$s0									
sub $t2,$s0,$s5									
...									
slt $t3,$s2,$s3			IF	ID	EX	DM	WB		

- Using the BTA and branch decision in cycle 3, the **and** instruction is flushed and the **slt** instruction is fetched

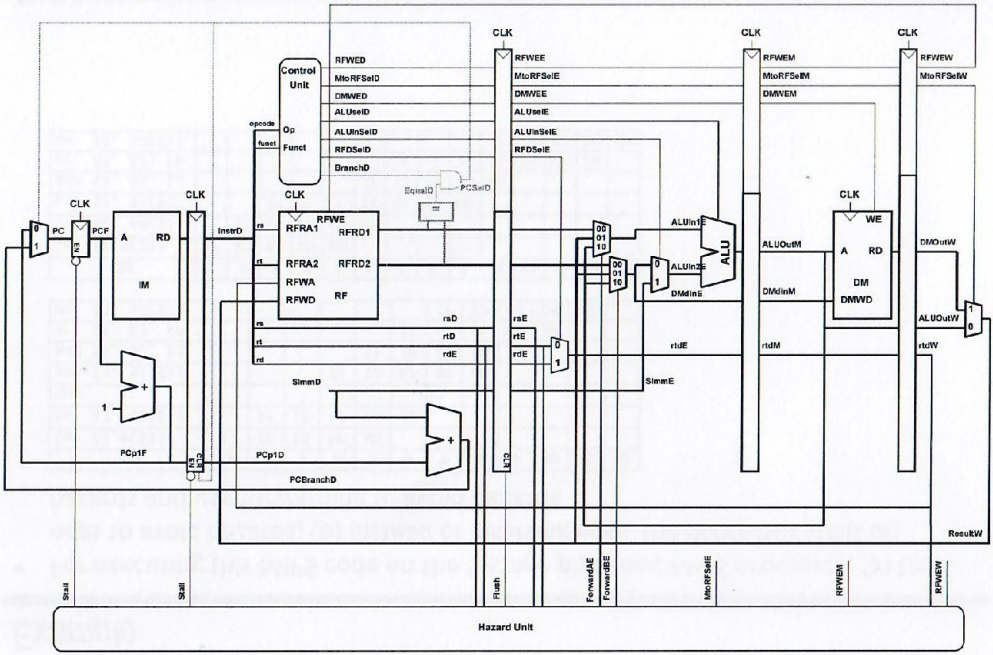
Control hazard

- In the current pipeline microarchitecture, because the decision is made in the MEM stage, the pipeline would have to be stalled for three cycles for every branch instruction, which severely degrades the performance. It is critical to keep the pipeline full with correct sequence of instructions
- We could reduce the branch misprediction penalty if the branch decision and the computation of BTA could be made as early in the pipeline as possible, thereby reducing the number of stall cycles. Moving the decision as early as possible minimizes the number of instructions that are flushed
- When decoding a branch instruction, the subsequent instruction is fetched while we don't know if branch will be taken or not. This is called the *control hazard*. A control hazard occurs when the decision of what instruction to fetch next can not be made by the time the fetch takes place
- The conditional branch **beq** instruction naturally presents a control hazard since the instructions following **beq** cannot be fetched until the branch decision is determined
- One mechanism for dealing with the control hazard is to stall the pipeline until the branch decision is made and BTA is calculated

Reducing branch misprediction penalty

- Modifications to the pipelined processor: (i) An equality comparator is added to the ID stage; (ii) An adder for calculating the BTA is added to the ID stage; (iii) The *PCSel* AND gate is moved from the MEM stage to the ID stage and is used as the select line *PCSelD* of the multiplexer. *PCSelD* is also connected to the synchronous clear input (CLR) of the IR to flushing the incorrectly fetched instruction when a branch is taken (transforming it into a **nop** instruction)
- Similar to the conditional branch instruction, BTA of a jump instruction is calculated in the ID stage. Since the subsequent instruction after the unconditional jump instruction in the IF stage must always be flushed, the CLR input of the IF pipeline register is always asserted when a `J` instruction is decoded
- Note that for the unconditional transfers, including function calls and returns `j`, `jal`, `jr`, the branches are always taken and only BTA is calculated, however, for conditional branches, BTA is calculated and branch direction must be determined

Modified pipelined microarchitecture



Three possible scenarios for branch

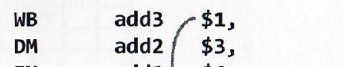
- **Case III**: If the instruction immediately before the branch produces one of the branch source operands, then a one-cycle stall should be applied

```
WB      add   $3,
MEMadd  $4,
EX      add   $1,
ID      beq   $1,$2,Loop
IF      next_instr
```

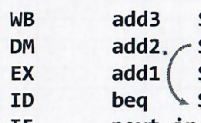
- This is because the EX stage of the preceding instruction occurs at the same time as the ID stage of the branch instruction and the result of the EX stage is available at the input register of the MEM stage
- In the above example, **$1** will be available at the end of the EX stage and hence, only accessible at the beginning of the MEM stage and after. Therefore, hazard unit applies one cycle stall between the **add** in the EX stage and the **beq** in the ID stage by asserting the *stall* signal connected to **PC** and **IR** registers

Introducing data hazards when resolving control hazards

- Since the processor makes a branch decision in the ID stage, this early branch decision may introduce a RAW data hazard if one of the source operands for the branch is computed by a previous instruction(s) and has not yet been written into the RF. In this case the branch will read the wrong operand value from the RF
- The data hazard can be avoided by *forwarding* the correct value if it is available or by *stalling* the pipeline until the data is ready
- Assuming that the branch decision hardware is instantiated in the ID stage:
- **Case I:** If the source register in **beq** is in the WB stage of an earlier instruction, since RF operates in the write-first mode, (data will be written into the RF first and then read), no data hazard exists and hence, no forwarding is required



- **Case II:** If the source register in **beq** is in the MEM stage, it can be forwarded from the MEM stage of an earlier instruction to the ID stage for comparison



Forwards the result from the second previous instr. to either inputs of the comparator

Data hazard handling logic when eliminating control hazards

- *Forwarding*: If the result of an already executed ALU instruction, which is available in the *MEM* stage, should be forwarded to the equality comparator in the *ID* stage, the following forwarding logic can be used in the *ID* stage as:

  $ForwardAD = (rsD \mathrel{!}= 0)\ \text{AND}\ (rsD == rtdM)\ \text{AND}\ RFWEM$

  $ForwardBD = (rtD \mathrel{!}= 0)\ \text{AND}\ (rtD == rtdM)\ \text{AND}\ RFWEM$

- *Stalling*: If either of the sources of the branch instruction in the *ID* stage depends on an ALU instruction that is in the *EX* stage (has not been yet executed in the *EX* stage) or on the result of a `lw` instruction in the *MEM* stage, the pipeline must be stalled at the *ID* and *IF* stages until the result is ready
- The stall logic for a branch is given as:

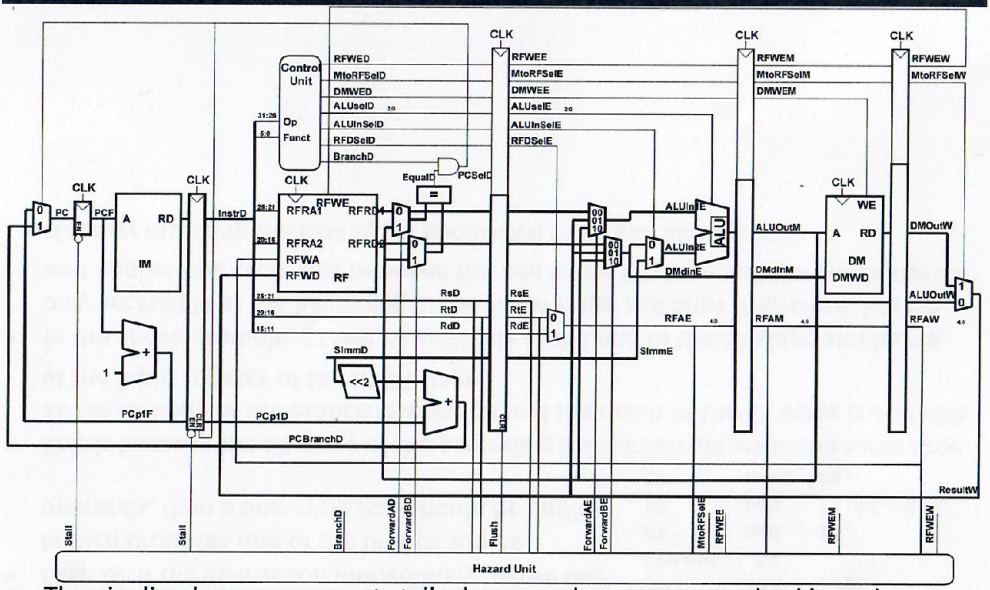
  $BRStall = (rsD == rtdE\ \text{OR}\ rtD == rtdE)\ \text{AND}\ BranchD\ \text{AND}\ RFWEE\ \text{OR}$

  $(rsD == rtdM\ \text{OR}\ rtD == rtdM)\ \text{AND}\ BranchD\ \text{AND}\ MtoRFSelM$

- The processor thus might stall due to either a data hazard (i.e., a `lw` stall as discussed before) or due to a control hazard:

  $Flush = Stall = LWStall\ OR\ BRStall$

Pipelined processor with full data and control hazard handling



- The pipelined processor must stall whenever there are unresolved branch instructions. Since on average 20 percent of instructions are branches, this imposes a substantial penalty on the performance of such processors

Example

- For executing this MIPS code on the 5-stage pipelined MIPS processor: (a) Use **nop**s to avoid hazards; (b) Instead of inserting **nop**s, the processor stalls on hazards and use forwarding to avoid hazards

CLK	1	2	3	4	5	6	7	8	9	10	11	12
lw $2,0($1)	IF	ID	EX	DM	WB							
lw $2, 40($3)		IF	ID	EX	DM	WB						
nop												
sub $3, $1($2)				IF	ID	EX	DM	WB				
add $3, $2, $2					IF	ID	EX	DM	WB			
or $4, $3, $0						IF	ID	EX	DM	WB		
sw $3, 50($1)							IF	ID	EX	DM	WB	

CLK	1	2	3	4	5	6	7	8	9	10	11	12	13	14
lw $2,0($1)	IF	ID	EX	DM	WB									
lw $2, 40($3)		IF	ID	EX	DM	WB								
sub $3, $1($2)			IF	S	S	ID	EX	DM	WB					
add $3, $2, $2						IF	ID	EX	DM	WB				
or $4, $3, $0							IF	S	S	ID	EX	DM	WB	
sw $3, 50($1)										IF	ID	EX	DM	WB

Example

- For executing this MIPS code on the 5-stage pipelined MIPS processor: (a) Use **nops** to avoid hazards; (b) Instead of inserting **nops**, the processor stalls on hazards; and (c) use forwarding to avoid hazards

```
lw  $s2, 0($s1)      //I1
lw  $s1, 10($s3)     //I2
sub $s3, $s1, $s2    //I3
add $s3, $s2, $s2    //I4
or  $s4, $s3, $zero  //I5
sw  $s3, 10($s1)     //I6
```

(a)

Clk	1	2	3	4	5	6	7	8	9	10	11	12	13	14
I1	F	D	E	M	W									
I2		F	D	E	M	W								
nop														
nop														
I3					F	D	E	M	W					
I4						F	D	E	M	W				
nop														
nop														
I5									F	D	E	M	W	
I6										F	D	E	M	W

(b)

Clk	1	2	3	4	5	6	7	8	9	10	11	12	13	14
I1	F	D	E	M	W									
I2		F	D	E	M	W								
I3			F	S	S	D	E	M	W					
I4						F	D	E	M	W				
I5							F	S	S	D	E	M	W	
I6										F	D	E	M	W

(c)

CLK	1	2	3	4	5	6	7	8	9	10	11
I1	F	D	E	M	W						
I2		F	D	E	M	W					
I3			F	D	S	E	M	W			
I4				F	S	D	E	M	W		
I5						F	D	E	M	W	
I6							F	D	E	M	W

Example – Pipelining MIPS instructions

(a)

CLK	1	2	3	4	5	6	7	8	9
add $t0,$s0,$s1	IF	ID	EX	DM	WB				
sub $t0,$t0,$s2		IF	ID	EX	DM	WB			
lw $t1,12($t0)			IF	ID	EX	DM	WB		
and $t2,$t1,$t0				IF	ID	S	EX	DM	WB

(b)

CLK	1	2	3	4	5	6	7	8	9
add $t0,$s0,$s1	IF	ID	EX	DM	WB				
lw $t1,12($s2)		IF	ID	EX	DM	WB			
sub $t2,$t0,$s3			IF	ID	EX	DM	WB		
and $t3,$t1,$t0				IF	ID	EX	DM	WB	

(c)

CLK	1	2	3	4	5	6	7	8	9	10	11	12
Loop: add $1, $2, $1	IF	ID	EX	DM	WB							
lw $2, 0($1)		IF	ID	EX	DM	WB						
lw $2, 16($2)			IF	ID	S	EX	DM	WB				
slt $1, $2, $4				IF	S	ID	S	EX	DM	WB		
beq $1, $9, Loop						IF	S	ID	EX	DM	WB	
add $1, $2,$1								IF	ID	EX	DM	WB

(d)

CLK	1	2	3	4	5	6	7	8	9	10	11	12
sub $2, $1, $3	IF	ID	EX	DM	WB							
and $12,$2, $5		IF	ID	EX	DM	WB						
or $13,$6, $2			IF	ID	EX	DM	WB					
add $14,$2, $2				IF	ID	EX	DM	WB				
sw $15,100($2)					IF	ID	EX	DM	WB			

(e)

CLK	1	2	3	4	5	6	7	8	9	10
lw $2, 0($1)	IF	ID	EX	DM	WB					
and $1,$2,$1		IF	ID	S	EX	DM	WB			
lw $3, 0($2)			IF	S	ID	EX	DM	WB		
lw $1, 0($1)					IF	ID	EX	DM	WB	
sw $1, 0($2)						IF	ID	EX	DM	WB

Example – Pipelining MIPS instructions

(f)

CLK	1	2	3	4	5	6	7	8
add $t0,$s0, $s1	IF	ID	EX	DM	WB			
lw $t1,60($s2)		IF	ID	EX	DM	WB		
sub $t2,$t0,$s3			IF	ID	EX	DM	WB	
and $t3,$t1,$t0				IF	ID	EX	DM	WB

(g)

Inst/CLK	1	2	3	4	5	6	7	8	9	10	11
Loop: add R1,R2,R1	F	D	X	M	W						
lw R2,0(R1)		F	D	X	M	W					
slt R1,R2,R4			F	D	S	X	M	W			
beq R1,R9,Loop				F	D	S	X	M	W		
add R1,R2,R3					F	D	S	X	M	W	

(h)

Inst/CLK	1	2	3	4	5	6	7	8	9	10	11	12
add R4, R1, R0	F	D	X	M	W							
sub R9, R3, R4		F	D	X	M	W						
add R4, R5, R6			F	D	X	M	W					
lw R2, 100(R3)				F	D	X	M	W				
lw R2, 0(R2)					F	D	S	X	M	W		
sw R2, 100(R4)						F	D	S	X	M	W	
and R2, R2, R1							F	D	S	X	M	W
beq R9, R1, LBL								F	D	S	X	M
and R9, R9, R1;									F	D	S	X

Static branch prediction

- In *static branch prediction*, rather than following a fixed strategy, performance can be improved by using a strategy that is dependent on the branch type. It uses the instruction opcode to predict whether the branch is taken
- For loops with a *backward branch*, in which the branch occurs when a program reaches the end of a loop and branches back to repeat the loop with a negative offset field, *predict taken* has less misprediction than *predict not taken*
- For loops with a *forward branch*, predict not-taken has less misprediction than predict taken. These loops have jumps at the bottom of the loop to return to the top of the loop, which incurs the *jump flush overhead*
- Pros of static branch prediction: All decisions are made at compile time; no prediction hardware is required; might be acceptable for loops
- Cons of static branch prediction: Performs poorly for unbiased branches as prediction cannot adapt to dynamic changes in branch behavior during program execution;

Backward branch

```
Loop: 1st loop instr
      2nd loop instr
            :
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

Forward branch

```
Loop: beq $1,$2,Out
      1nd loop instr
            :
      last loop instr
      j  Loop
Out:  fall out instr
```

Fixed branch prediction

- So far we have utilized the *fixed branch prediction strategy* of "always not taken" in which the processor continues fetching and executing instructions in order. If the branch should have been taken, all the instructions fetched after the mispredicted branch in the pipeline must be *flushed* (discarded) by resetting the pipeline registers for that instruction and fetch the instruction from the BTA. Clearing the register transforms the fetched instruction into a **nop**. The earlier instructions in the pipeline progress normally
- *Branch misprediction penalty* is wasted clock cycles due to pipeline flushing on mis-predicted branches
- Branch mispredictions have greater penalties than data hazards, especially for processors with deep pipelines
- Implementing the fixed branch prediction strategy, which assumes that the branch is either never taken (or always taken), has minimal associated hardware, but it provides a relatively low accuracy (~40%). Compiler can layout code such that the likely path of execution is the "not-taken" path

Dynamic branch prediction

- On average about 20% of the executed instructions are control flow instructions (i.e., one in every 4 or 5 instructions). As the branch penalty increases for deeper pipelines, a simple static prediction scheme will negatively impact performance. An alternative approach is to dynamically *predict* whether the branch will be taken and begin executing instructions based on the run-time prediction rather than waiting to determine the actual outcome of the branch decision
- Static branch prediction schemes do not depend on the history of the branch.
- *Dynamic branch prediction* predicts the direction of branches based on information collected at run-time. It uses past behavior or history of the execution of branches (e.g., past two branch decisions, taken or not taken) to make more accurate predictions about the outcome of conditional branch instructions. The prediction changes as branch behavior changes. i.e., it adapts to changes in branch behavior at run time
- Dynamic branch prediction scheme predicts: (i) Whether the fetched instruction is a branch; (ii) (Conditional) branch direction (taken or not taken); and (iii) Branch target address (if taken)
- The processor usually buffers the speculative results. Once the branch decision is available, if the speculation was incorrect, the hardware *discards* (*flushes*) the buffers and executes the correct instruction sequence

Dynamic branch prediction

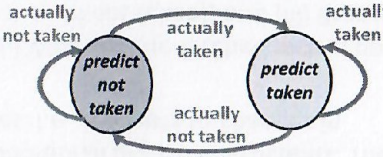
- In dynamic prediction, processor can speculate on branch outcomes using a branch prediction scheme and continue to fetch instructions along the predicted path. The main idea behind speculation is to perform some tasks before it is known whether that work will be needed at all, so as to prevent a delay that would have to be incurred by doing the work after it is known whether it is needed. The processor guesses the outcome of a branch so as to enable execution of other instructions that (may) depend on the speculated instruction
- Branch prediction has significant effect on the processor performance. However, branches are difficult to predict without knowing more about the specific program and input data. If a branch history contains truly independent event, we cannot predict the next outcome using knowledge gained from past behavior. Accuracy of dynamic branch prediction thus varies during run time
- Modern processors use dynamic branch prediction. Dynamic branch prediction is more complex and requires additional hardware and hence, additional area and power consumption. With technology scaling, branch prediction hardware cost is small compared to the performance gain

One-bit dynamic predictor

- The **beq** out of the loop is taken only on the last time. So for *N* repetitions of the loop, the one-bit predictor has one misprediction. If the loop is run again in the program, the branch predictor remembers that the branch was taken on the last iteration of the loop. Therefore, it incorrectly predicts that the branch should be taken when the loop is first run again. Thus, a one-bit branch predictor always mispredicts the first and last iterations of a loop branch
  - Mispredicting the last iteration is inevitable since the prediction bit will indicate not taken, as the branch has not been taken nine times in a row at that point
  - The misprediction on the first iteration happens because the bit is flipped on prior execution of the last iteration of the loop, since the branch was taken on that exiting iteration
- Thus, the prediction accuracy for this branch that is taken 90% of the time is only 80% (two incorrect predictions and eight correct ones)
- Accuracy of the one-bit predictor for a loop with *N* iterations is thus $(N-2)/N$ as it mispredicts the first and last iterations of a for loop
- Ideally, the accuracy of the predictor would match the taken branch frequency for these highly regular branches

One-bit dynamic branch prediction

- One approach is to predict the direction of a conditional branch based on its last branch direction
- A *one-bit dynamic branch predictor* or *last-time predictor* remembers whether the branch was taken the last time and predicts that it will do the same next time



- A single *prediction bit pBit* is thus associated with each branch instruction
- After its first execution, the prediction bit will be updated (i.e., when the actual branch direction is determined at *resolution time*)
- Assuming that the branch is initially predicted to not-taken state. While the loop is repeating, it remembers that the **beq** was not taken last time and predicts that it should not be taken next time. This is a correct prediction until the last branch of the loop, when the branch should be taken

```
    add  $s1,$0,$0     #sum=0
    add  $s0,$0,$0     #i  =0
    addi $t0,$0,10     #$t0=10
for:
    beq  $s0,$t0,done #if i==10,branch
    add  $s1,$s1,$s0  #sum=sum+i
    addi $s0,$s0,1     #increment i
    j    for
done:
```

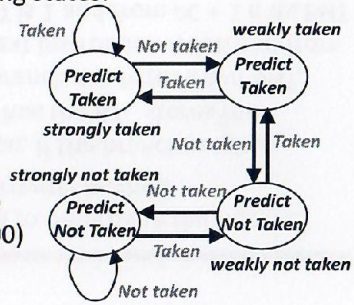
One-bit dynamic predictor for bottom of the loop structure

```
Loop: 1st loop instr
      2nd loop instr
            ⋮
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

- Assume the prediction bit *pBit* is initially zero, indicating branch not taken. First time through the loop, the predictor mispredicts the branch since the branch is taken. It thus inverts the prediction bit *pBit*
- As long as branch is taken (looping), prediction is correct
- Exiting the loop, the predictor again mispredicts the branch since this time the branch is not taken falling out of the loop. It inverts the prediction bit *pBit*=1
- Thus this one-bit predictor will be incorrect twice: once upon loop entry and then on loop exit
- One can see that the one-bit dynamic predictor is only efficient for loop branches with relatively large number of iterations
- Good branch predictors achieve better than 95% accuracy on typical programs

Two-bit predictor

- A one-bit predictor changes its prediction from taken to not-taken or from not-taken to taken too quickly. Even for cases when the branch may be mostly taken or mostly not taken, such as for loops, the one-bit prediction scheme does not work well as it mispredicts the first and the last iterations of a loop
- One enhancement is to add hysteresis to the predictor so that prediction does not change on a single different outcome
- Instead of using only two states (one bit) for each branch, to make the prediction based on the last two branch outcomes, two bits can be used for each branch instruction to track the history of each branch outcome. These two bits correspond to a four-state FSM that has the following states:
- *Strong taken:* The last two instances of the branch were taken (encoding 11); *Weak taken:* The last instance was not taken, but the previous one was taken (encoding 10); *Weak not-taken:* The last instance was taken, but the previous one was not taken (encoding 01); *Strong not-taken*: The last two instances of the branch were not taken (encoding 00)



Two-bit predictor example

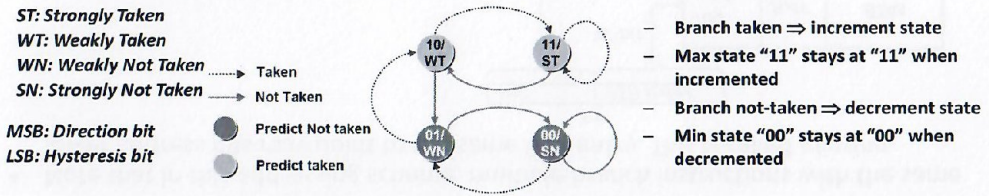
- The MSB of the encoding suggests the prediction (1 for taken, 0 for not taken). If the left bit is zero (one), the prediction would be "not taken" (taken)
- The LSB denotes the actual outcome of the branch decision. 0 denotes not-taken and 1 indicates that the branch is taken. For example, state 00 represents that we predicted that the branch would not be taken (left zero bit) and the branch is indeed not taken (right zero bit)
- In this example, when the loop is repeating, it enters the "strongly not taken" state (generally "strongly not taken" is the initial state) and predicts that the branch should not be taken next time. This is correct until the last branch of the loop, which is taken and moves the predictor to the "weakly not taken" state When the loop is first run again, the branch predictor correctly predicts that the branch should not be taken and reenters the "strongly not taken" state

```
  add  $s1, $0, $0  # sum = 0
  add  $s0, $0, $0  # i   = 0
  addi $t0, $0, 10  # $t0 = 10
for:
# if i == 10, branch
  beq  $s0, $t0, done
# sum = sum + i
  add  $s1, $s1, $s0
# increment i
  addi $s0, $s0, 1
  j    for
done:
```

- Thus it only mispredicts the last branch of a loop and the accuracy of a 2-bit predictor for a loop with N iterations is $(N-1)/N$
- A 2-bit branch predictor works well when branches predominantly go in one direction

Two-bit predictor FSM

- In the two-bit prediction scheme, a prediction must be wrong twice before the prediction is changed. Thus a temporary change of branch direction does not change the prediction away from the dominant branch direction



- To record the directions of the last two predictions for every branch instruction, each branch is associated with a 2-bit saturating counter. The "saturating" part comes in when the counter is in a strong state and the prediction is correct. The state is therefore left unchanged. If the branch is taken: counter = min(3,counter+1) and if the branch is not taken: counter = max(0,counter-1)
- The counter value indicates the outcome of the last two executions of the branch. "11" and "10" predict the next state taken and "00" and "01" predict the next state not-taken. Thus if (counter ≥ 2), predict taken, else predict not-taken
- The counter uses a Gray code (only one bit changes per transition). If the prediction turns out to be wrong, the prediction bits are toggled

Two-bit predictor examples

- Consider the following fragment of code where *m* and *n* are large:

```
for (i = 0; i < m; i++)
   for (j = 0; j < n; j++) begin
          S1; S2; ...;Sk
end;
```

- Assume each loop is written with the top of the loop code and start in the not taken state
- In the two-bit predictor scheme, when the last instance of the loop is executed, there is a misprediction and the state becomes weak taken. At the first iteration of the new execution of the loop, the prediction of the loop-ending branch that is in the weak taken state will be correct, and the state will become strong taken again. Using a two-bit predictor, there will be 1+*m* mispredictions and using a one-bit predictor, that will be 2*m*+2
- Assume a processor running a program with no data dependencies among the instructions. Assume every fifth instruction is a branch, the branch predictor accuracy is 96%, and branch misprediction penalty is 2 cycles. What is the average cycles per instruction (CPI)?

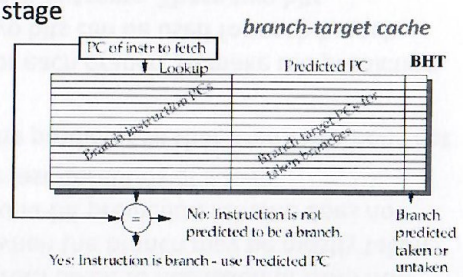
Stalls per instructions = % branches (br/I) × %mispreds (miss/br) × penalty (c/miss)
= 20% × 4% × 2 = 0.016 (CPI)
Average cycles per instructions = 1 + stalls per instruction = 1.016 times slow down

Using n-bit predictors

- One of the weaknesses of the two-bit predictor method is that it only takes into account a small amount of the past history of the branch for the next prediction
- Branch predictors can track even more history of the program to increase the accuracy of prediction. For example, an *n-bit predictor* uses an *n*-bit saturating counter for each branch. Each time the branch is taken, the saturating counter is incremented and each time the branch is not taken, the saturating counter is decremented. For prediction, if the counter value is greater than or equal to half its maximum value, the branch is predicted as taken and if the value of the counter is below 2^{n-1}, the branch is predicted as not-taken
- The prediction can be done for any *n*, but experiments have shown that the additional number of states yielded minimal improvements beyond 2-bit predictors

Branch prediction unit

- The *branch-target cache* (BTC) in the IF stage is used to determine the next PC. BTC is accessed by the address of the instruction currently fetched
- It is reasonable to assume that the branch instruction, if the branch is taken, jumps to the same target address as the last time. Thus the BTC stores the address of each conditional branch instruction, its branch predictor bit in BHT, and its last BTA. For every branch instruction, the next instruction is fetched from BTA stored in the BTC if the prediction bit in the BHT is 1 and from PC + 1 if its BHT entry is 0. Note that this is done even before the CPU knows the instruction is a branch, which will be determined in the ID stage
- Once the branch decision outcome is determined, if the prediction is incorrect, the prediction bit in BHT is inverted, BTA is calculated and updated, the incorrectly predicted instruction(s) in pipeline are flushed, and the pipeline will start with fetching the correct instruction



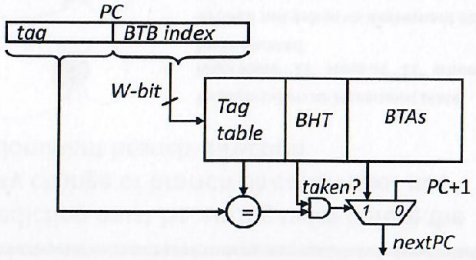
- By keeping track of jump destination address for every unconditional branch instruction in the BTB, the processor can also avoid flushing the pipeline during jump instructions

Branch prediction unit

- The *Branch Prediction Unit* (BPU) uses dynamic branch predictions to speculate the direction of a branch based on previous prediction history
- In our designed microarchitecture, the prediction was done at the decode stage when it was recognized that the instruction was a branch
- The dynamic branch prediction can be moved to the fetch stage, i.e., the BPU operates in the IF stage before the current instruction is even decoded. In this scheme where the prediction is done in the IF stage, BTA is available at end of IF stage. Note that prediction is just a hint that we hope is correct, so fetching begins at the predicted BTA
- To implement 1-bit branch predictor, a *branch history table* (BHT), which is reside in the IF stage, can be used. BHT contains the prediction bit for each conditional branch instruction, which indicates whether the branch was taken the last time it was executed. Initially, BHT is initialized with zeros. A '1' in the BHT entry means that the last time this branch instruction was executed, the branch was taken. So BPU predicts it will be taken again. If the prediction is found incorrect in the ID stage after decoding the branch instruction, the prediction bit is inverted and updated in BHT
- Note that the BHT indicates the outcome of the branch last time it was executed (taken or not taken), but it does not specify where its taken to

Smaller BTC

- The BTC contains BHT, BTA, and the address of the last several thousand branch instructions that the processor has executed
- With a larger BTC there are fewer *cache misses* (i.e., when a branch instruction address cannot be found in the code) and hence, the performance improves
- For a smaller cache and faster accesses, W least significant bits of the branch instruction address are used to address BTC with 2^W entries. Only the 32-W most significant bits of the address of the branch instructions are stored in the BTC and are compared with the same number of MSBs of the **PC**
- Note that in this addressing scheme, multiple branch instructions with the same lower address bits may point to the same BTB entry. This is called *aliasing*

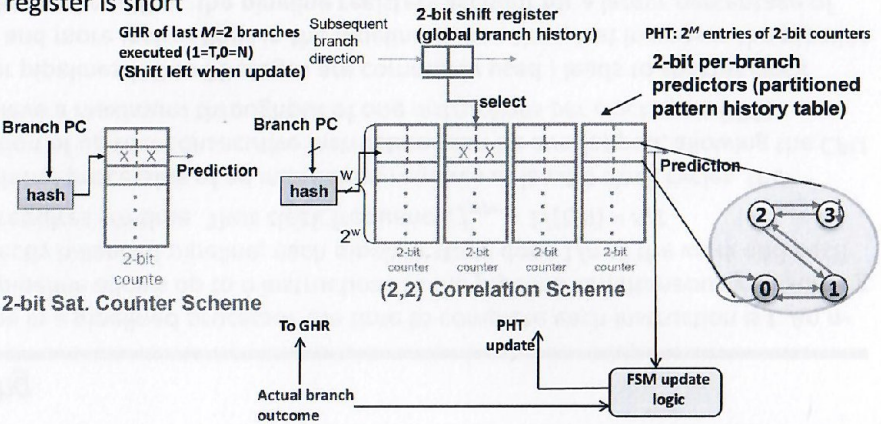


Global history branch predictors

- The branch predictors discussed so far make *local* predictions, i.e., the recent behavior of a branch is used to predict the future behavior of that same branch
- While 2-bit predictors provide acceptable accuracy for loops that execute a relatively large number of times, predictions based on how a branch behaved in the past might not give the best prediction for nested if statements. In fact, the outcome of a branch may depend not only on its past behavior but also on the outcome of other branches whose executions directly precede that of the branch being predicted
- Generally, a branch outcome is correlated to the behavior of other branch results. Therefore, to improve the prediction accuracy, the outcome of other branches should also be used. Such predictors are called *correlating predictors* or *global history branch predictor*, which provide more accurate prediction outcomes

(m=2,n=2) correlation branch predictor

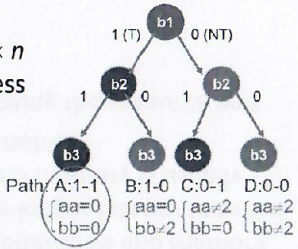
- An *m*-bit *global history register* (GHR) records the actual direction ("T/N") of the last *m* branches. An *m*-bit GHR provides access to 2^m patterns in the PHT for a given branch address. Thus GHR chooses a PHT based on history
- Combination of PC and GHR chooses PHT entry, which is the prediction
- GHR is a shift register to indicate the latest two preceding branches
- Experiments have shown that global predictors do not perform well when the global register is short



(m,n) correlation predictor

- An (*m*,*n*) *correlation predictor* uses the behaviors of the last *m* branches, which represents a *path* through the program, to choose from 2^m predictors, each of which is an *n*-bit predictor
- The number of bits required by an (*m*,*n*) predictor is $2^m \times n \times$ Number of prediction entries selected by branch address

```
b1: if (aa == 2) aa=0;
b2: if (bb == 2) bb=0;
b3: if (aa != bb) { … }
```



- In this example, past *m*=2 branches are used as histories and hence, there are 4 possibilities (T-T, N-T, N-N, N-T)
- Branch direction in b3 is not independent and is correlated to the path taken. If branches b1 and b2 are taken, branch b3 is not-taken. If branch b1 is not taken and b2 is not-taken then b3 is certainly taken
- For each possibility, we need to use a branch predictor based on *n*=1 bit, *n*=2 bits, or more bits
- Multiple predictions per branch stored in a *partitioned pattern history table* (PHT) table of 2-bit saturating counters for each history entry. The BHT is called PHT here. For each unique path, a predictor is maintained in the PHT. Each entry of the partitioned PHT keeps a 2-bit (or more) saturating counter for prediction

Branch prediction example

- What is the accuracy of the following branch predictors for each of the following repeating patterns (e.g., in a loop) of branch outcomes: (a) T, N, N, N, T and (b) T, N, T, N, T. T means the branch is taken and N means the branch is not taken

(a) T, NT, NT, NT, T

(i) What is the accuracy of always-taken and always-not-taken predictors for this sequence of branch outcomes?

Always-Taken Predictor	Always-not-taken Predictor
Actual T NT NT NT T	Actual T NT NT NT T
Predict T T T T T	Predict NT NT NT NT NT
Accuracy = 2/5 = 0.4	Accuracy = 3/5 = 0.6

(ii) What is the accuracy of a predict-last-taken predictor for the first 5 branches in this sequence? Assume this predictor starts in the "Predict not taken" state.

Actual T NT NT NT T
Predict NT T NT NT NT
Accuracy = 2/5 = 0.4

(iii) What is the accuracy of a 2-bit predictor for the first 5 branches in this sequence? Assume this predictor starts in the "Strongly not taken" state.

Actual T NT NT NT T
Predictor 0 1 0 0 0
Predict NT NT NT NT NT
Accuracy = 3/5 = 0.6

(iv) What is the accuracy of a 2-bit predictor if this pattern is repeated forever? Assume this predictor starts in the "Strongly not taken" state.

Actual T NT NT NT T | T NT NT NT T | T NT NT NT T |
Predictor 0 1 0 0 0 | 1 2 1 0 0 | 1 2 1 0 0 |
Predict NT NT NT NT NT | NT T NT NT NT | NT T NT NT NT |
Steady-state Accuracy = 2/5 = 0.4

(b) T, NT, T, NT, T

(i) What is the accuracy of always-taken and always-not-taken predictors for this sequence of branch outcomes?

Always-Taken Predictor	Always-not-taken Predictor
Actual T NT T NT T	Actual T NT T NT T
Predict T T T T T	Predict NT NT NT NT NT
Accuracy = 3/5 = 0.6	Accuracy = 2/5 = 0.4

(ii) What is the accuracy of a predict-last-taken predictor for the first 5 branches in this sequence? Assume this predictor starts in the "Predict not taken" state.

Actual T NT T NT T
Predict NT T NT T NT
Accuracy = 0/5 = 0

(iii) What is the accuracy of a 2-bit predictor for the first 5 branches in this sequence? Assume this predictor starts in the "Strongly not taken" state.

Actual T NT T NT T
Predictor 0 1 0 1 0
Predict NT NT NT NT NT
Accuracy = 2/5 = 0.4

(iv) What is the accuracy of a 2-bit predictor if this pattern is repeated forever? Assume this predictor starts in the "Strongly not taken" state.

Actual T NT T NT T | T NT T NT T | T NT T NT T | T NT T NT T | T NT T NT T |
Predictor 0 1 0 1 0 | 1 2 1 2 1 | 2 1 2 1 2 | 3 2 3 2 3 | 3 2 3 2 3 |
Predict NT NT NT NT NT| NT T NT T NT| T NT T NT T | T T T T T | T T T T T | Steady-state Accuracy = 3/5 = 0.6

Pipelining

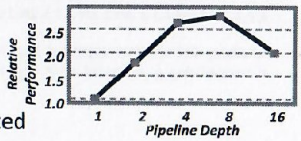
- Assume in a pipelined processor the time to complete each instruction is *t*. An *n*-stage pipeline allows up to *n* instructions in the pipeline simultaneously. Assuming a perfectly balanced pipeline, each pipeline stage does 1/*n* of the work and each stage requires *t*/*n* time. Thus clock frequency: $f_{pipe} = 1/(t/n) = n/t$
- Though the processing of an instruction requires at least *n* clock cycles, the execution of up to *n* consecutive instructions can be overlapped, allowing the CPU to achieve a maximum throughput of one *instructions per clock cycle* (IPC)
- Deeper pipelines (10 to 20 stages are commonly used) leads to shorter clock cycles and more instructions in the pipeline at one time, but increases the pipeline register overhead (i.e., the pipeline registers account for a larger percentage of latency)
- In reality the IPC of a pipelined processor is much lower than 1 due to:
  - Data and control hazards that results in stalls. Pipeline stalls represent a major bottleneck in achieving the desired throughput (increases the CPI, which degrades performance). Some operations, such as floating-point calculations, especially require many cycles to compute, so dependencies create relatively long stalls
  - Branch misprediction penalties
  - Memory operations need variable number of cycles (searching for data in cache, if not found, accessing external memory)

Multiple-issue processors

- *Machine level parallelism (MLP)* is determined by the number of instructions that can be fetched and executed at the same time by the processor. Typically MLP supports more ILP than a program can provide due to the data and control dependencies
- A *multiple-issue processor* contains multiple copies of the datapath hardware to issue multiple instructions simultaneously. Multiple-issue processors exploit both temporal and spatial parallelism by fetching and issuing more than one instruction per clock cycle to increase performance. Pipelining and multiple issue both attempt to exploit ILP in time and space domains
- *Instruction issue* means initiate execution (*dispatching* the instruction to execution units). The set of instructions that are issued together in one clock cycle are called an *issue packet.* The multiple issue processor should determine which instructions can be issued in a given clock cycle and package them into issue slots
- In a multiple-issue processor several instructions can occupy the same stages of the pipeline processor at the same time
- Writing back results to the register file or data memory (i.e., change the machine state) is called *instruction commit*. Committed instructions are those that ultimately modify the processor state

Spatial parallelism

- Increasing the number of pipeline stages beyond a level does not increase the performance
  - The presence of data and control dependencies are the primary limitations on how much parallelism can be exploited
  - The maximum number of pipeline stages is limited by pipeline hazards and sequencing overhead



- In addition to pipelining (*temporal parallelism*), *spatial parallelism* can be used to enhance IPC
- Spatial parallelism is the simultaneous execution of mutually independent instructions
- To support *instruction-level parallelism* (ILP), the processor requires multiple execution units and storage units with more read/write ports to execute several instructions of the same or different programs (*multithreading*) simultaneously

cycle	1	2	3	4	5	6	7	8	9
Inst–1	F	D	E	M	W				
Inst–2		F	D	E	M	W			
Inst–3			F	D	E	M	W		
Inst–4				F	D	E	M	W	
Inst–5					F	D	E	M	W

cycle	1	2	3	4	5	6	7
Inst–1	F	D	E	M	W		
Inst–2	F	D	E	M	W		
Inst–3		F	D	E	M	W	
Inst–4		F	D	E	M	W	
Inst–5			F	D	E	M	W

- ILP of a program is a measure of the average number of instructions in a program that a processor might be able to execute simultaneously
  - Real programs have many dependencies, so wide multiple-issue processors rarely fully utilize all of the execution units. One remedy is to interleave the execution of different programs (*multithreading*)

Resource (structural) hazards

- When using multiple-issue processors with multiple arithmetic and memory units, instruction issue is stalled when (i) The instructions ready to issue need a result that has not yet been computed; (ii) Control hazard; and (iii) *Resource or structural hazard*
- *Resource hazards* occur when multiple instructions require the same hardware unit in a given cycle. An instruction can't be issued if it needs to use the same hardware as another instruction at the same time
- Resource hazards can be due to long latencies of certain operation and memory accesses. For example, floating-point units have relatively longer latency than the integer arithmetic units. Also, latency of access to the main memory is usually much greater than one cycle and often unpredictable (variable)
- Resource hazards can be reduced or eliminated by duplicating the resource and pipelining the resource

Dynamic and static multiple-issue processors

- Dynamic multiple-issue processors (*superscalar*)
  - Decisions on which instructions to be issued simultaneously are being made dynamically (at run time) by the hardware
  - Resolves data, control, and structural hazards at runtime
  - Exploits ILP by using multiple function units to allow spatial parallel execution of multiple independent instructions in each cycle
  - Increasingly complex with *degree of superscalability* (2-way, ..., *N*-way)
- In a *two-way superscalar processor* or a *dual-issue superscalar processor*, the pipeline is designed to fetch two independent instructions per cycle from the instruction memory. The two-way processor contains a six-ported register file to read four source operands and to write two results back in each cycle, two ALUs, and a four-ported data memory to execute up to two instructions at the same time,
- In static multiple-issue processors, such as *VLIW* processors, decisions on which instructions to execute simultaneously are being made statically by the compiler. Nevertheless, for both processors, compiler can help by reordering instructions

Out-of-order instruction issue policy

- In an *out-of-order issue* (OOI) (or *out-of-order execution*), an instruction blocked from executing does not cause the following instructions to wait. For example, if the second instruction has data dependencies with the first, but the third instruction does not, the first and third instructions can be issued together. Thus the instructions can be executed (issued) in a different order than they were fetched
- A superscalar processor looks ahead beyond the current instruction. Fetch, decode and issue, or begin executing, independent instructions as soon as possible, if downstream instructions have no conflicts
- OOI allows the processor to hide some of stalls by continuing to execute downstream instructions while waiting for the stall to end. Some stalls, such as cache misses, are unpredictable and could be relatively long
- OOI of instructions can be challenging due to *aliasing*. Aliasing is when two registers use indirect references to the same physical memory location. For example these two instructions can be dependent due to aliasing

```
sw $s4,10($s3)
lw $s2,12($s1)
```

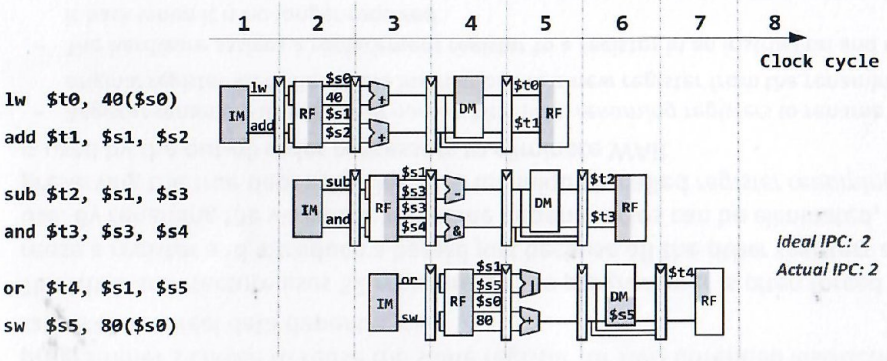
`(10+$s3)=?(12+$s1)`

N-issue processor

- Issuing multiple instructions per cycle (IPC > 1) and hence increasing execution concurrency, reduces CPI to be less than 1
- An *N-issue processor* can issue up to *N* instructions per cycle and can attain a maximum throughput of *N* IPC (the number of simultaneous issuable instructions or machine-level parallelism), however, in reality, the ILP is limited by the amount of parallelism available in a typical program. Commercial processors may issue three, four, or even six instructions, but the speedup has been shown to be between two and three for programs
- In an *in-order issue (IOI)* processor, the instruction fetch and decode unit issue *N* consecutive instructions in the original program order in which they are fetched. That is, if the later instruction cannot be issued because it has a data dependency with the earlier, only one instruction is issued in that clock cycle, i.e., with IOI, the processor stalls decoding instructions whenever a decoded instruction has a data dependency or a resource conflict on an issued, but uncompleted instruction. Due to the dependencies in typical instruction sequences, the performance of IOI superscalar processors is generally limited
- In an alternative instruction issue policy, instructions can be issued *out-of-order* and hence, they may complete in an order different from their actual program order

Two-way superscalar processor - Example

- A two-way superscalar processor can provide a maximum throughput of two instructions per clock cycle. Due to data and control dependencies and resource hazards, the actual throughput achieved by the processor is lower. In this example, the processor issues six instructions in three cycles (IPC = 2, CPI = 0.5)



Data dependencies in out-of-order issue processors

- For data dependencies in OOI processors, consider executing a sequence of R-type instructions $r_k \leftarrow r_i$ op r_j

Data-dependence $r_3 \leftarrow r_1$ op r_2 *Read-after-Write (true data dependencies)*
$r_5 \leftarrow r_3$ op r_4 *(RAW) hazard*

Anti-dependence $r_3 \leftarrow r_1$ op r_2 *Write-after-Read (data anti-dependencies)*
$r_1 \leftarrow r_4$ op r_5 *(WAR) hazard*

Output-dependence $r_3 \leftarrow r_1$ op r_2 *Write-after-Write (output dependencies)*
$r_3 \leftarrow r_6$ op r_7 *(WAW) hazard*

- **RAW:** Instruction 1 must write *before* instruction 2 reads. This true data dependency was resolved in the pipelined processor by forwarding or stalling the pipeline
- **WAR:** Instruction 2 must not write *before* 1 reads
- **WAW:** Instruction 2 must write *after* 1
- WAR is a possible scenario with out-of-order executions. Instead of the later instruction using a value (not yet) produced by an earlier instruction (*read before write*), the later instruction produces a value that destroys a value that the earlier instruction (has not yet) used (*write before read*). Hence, the former read instruction must be able to fetch the register content before the latter write stores new value there

Eliminating WAW using register renaming

- In *WAW* dependency, two instructions alter the same register or memory location and the latter in the original code must stall. A WAW hazard occurs if an instruction attempts to write a register after a subsequent instruction has already written it (due to the out-of-order execution). The hazard would thus result in the wrong value being written to the register
- In this example, **add** and **sub** both write **$t0**. The final value in **$t0** should come from **sub** according to the order of the program

```
add $t0, $s1, $s2
...
sub $t0, $s3, $s4
```

- If an out-of-order processor attempted to execute **sub** first, the WAW hazard would occur, i.e., the following instruction reads an incorrect value of **$t0**. Hence, issuing of **sub** would have to be stalled if its result might later be overwritten by a previous instruction (i.e., **add**) that takes longer to complete
- Both anti- and output dependencies arise because of the limited number of registers and both can be avoided using register renaming

Eliminating WAR using register renaming

- A WAR hazard (*antidependence* or *name dependence*) is merely an artifact of the programmer's choice to reuse the same register for two unrelated instructions, rather than a real data dependence
- The MIPS architecture uses 32 registers, so the programmer is often forced to reuse a register and introduce a hazard just because all the other registers are in use. By renaming the variables, the name dependencies can be eliminated, while preserving the true dependencies. This technique is called *register renaming* and is used by the out-of-order processors to eliminate WAR
  - Register renaming uses a set of *non-architectural renaming registers* to rename the original register identifier in the instructions to a new register from the renaming pool
  - The hardware assigns a replacement register to a register in an instruction and releases it back when it is no longer required
- In this example, WAR hazard can be avoided by writing the result of R5+1 onto a different register

```
R6 := R3 × R5
R4 := R3 + 1
R3 := R5 + 1
```

- WAR hazards could not occur in the in-order issue processors, but they may happen in an out-of-order processor if the dependent instruction is moved too early

Two-way superscalar processor - Example

Original program

```
lw  $t0, 40($s0)
add $t1, $t0, $s1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or  $t3, $s5, $s6
sw  $s7, 80($t3)
```

CLK	1	2	3	4	5	6	7	8	9
lw $t0,40($s0)	IF	ID	EX	DM	WB				
add $t1,$t0,$s1		IF	ID	ID	EX	DM	WB		
sub $t0,$s2,$s3		IF	ID	ID	EX	DM	WB		
and $t2,$s4,$t0			IF	IF	ID	EX	DM	WB	
or $t3,$s5,$s6			IF	IF	ID	EX	DM	WB	
sw $s7,80($t3)					IF	ID	EX	DM	WB

Ideal IPC is 2

This program requires five cycles to issue six instructions. Thus the actual IPC is 6/5 = 1.17

- The **add** instruction stalls for one cycle so that **lw** can forward **$t0**. The other two RAW dependencies are handled by forwarding
- Between **sub** and **add** is a *WAR* dependency. **sub** must not write **$t0** before **add** reads **$t0**. If the **sub** instruction had written another register (e.g., **$t4**) instead of **$t0**, **sub** could be issued prior to **add**. Note that two R-type instructions with RAW dependency can be executed back-to-back by utilizing forwarding

Constraints on issuing instructions

CLK	1	2	3	4	5	6	7	8	9
lw $t0,40($s0)	IF	ID	EX	DM	WB				
or $t3,$s5,$s6	IF	ID	EX	DM	WB				
sw $s7,80($t3)		IF	ID	EX	DM	WB			
		IF	ID	EX	DM	WB			
add $t1,$t0,$s1			IF	ID	EX	DM	WB		
sub $t0,$s2,$s3			IF	ID	EX	DM	WB		
and $t2,$s4,$t0				IF	ID	EX	DM	WB	

This program requires four cycles to issue six instructions. Thus the actual IPC is 6/4 = 1.5

- In cycle 1, **lw** and **or** instructions issue. The **add**, **sub**, and **and** instructions are dependent on **lw** (RAW). **add** must not read **$t0** until after **lw** has written it. The **or** instruction is independent, so it can issue
- **lw** instruction has one clock cycle latency, i.e., the result of a load instruction cannot be used on the next clock cycle, which prevents one instruction from using the result without stalling. Thus **add** cannot issue. **sub** writes **$t0**, so it cannot issue before **add**. **and** is dependent on **sub** (RAW dependencies). So in cycle 2, only the **sw** instruction issues
- In cycle 3, **add** issues since **$t0** is available. **sub** issues simultaneously, because it will not write **$t0** until after **add** reads **$t0**
- In cycle 4, **$t0** is forwarded from **sub** to **and** and **and** issues

Example – Pipelining MIPS instructions on a two-way processor

Scalar

CLK	1	2	3	4	5	6	7	8	9	10	11	12
lw R2, 0(R1)	IF	ID	EX	DM	WB							
lw R3,4(R1)		IF	ID	EX	DM	WB						
lw R4, 8(R1)			IF	ID	EX	DM	WB					
add R6, R4,R5				IF	S	ID	EX	DM	WB			
add R7, R2,R3						IF	ID	EX	DM	WB		
add R8, R6,R7							IF	ID	EX	DM	WB	
lw R9, 0(R8)								IF	ID	EX	DM	WB

2-way superscalar

CLK	1	2	3	4	5	6	7	8	9	10	11	12
lw R2, 0(R1)	IF	ID	EX	DM	WB							
lw R3,4(R1)	IF	ID	EX	DM	WB							
lw R4, 8(R1)		IF	ID	EX	DM	WB						
add R6, R4,R5		IF	S	S	ID	EX	DM	WB				
add R7, R2,R3			IF	S	ID	EX	DM	WB				
add R8, R6,R7					IF	ID	EX	DM	WB			
lw R9, 0(R8)					IF	S	ID	EX	DM	WB		

Register renaming - Example

- On an out-of-order processor with register renaming, **$t0** is renamed to **$r0** in **sub** and **and** to eliminate the WAR hazard
- **sub** could be executed sooner, because **$r0** has no dependency on the **add** instruction

```
lw  $t0, 40($s0)
add $t1, $t0, $s1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or  $t3, $s5, $s6
sw  $s7, 80($t3)
```

RAW

WAR

CLK	1	2	3	4	5	6	7	8	9
lw $t0,40($s0)	IF	ID	EX	DM	WB				
sub $r0,$s2,$s3	IF	ID	EX	DM	WB				
and $t2,$s4,$r0		IF	ID	EX	DM	WB			
or $t3,$s5,$s6		IF	ID	EX	DM	WB			
add $t1,$t0,$s1			IF	ID	EX	DM	WB		
sw $s7,80($t3)			IF	ID	EX	DM	WB		

Ideal IPC: 2

Actual IPC: 6/3 = 2

- In cycle 1, the **lw** instruction issues. There is one-cycle latency between when a **lw** issues and when a dependent instruction can use its result, so **add** cannot issue until cycle 3. The **sub** instruction is independent now that its destination has been renamed to **$r0**, so **sub** also issues
- In cycle 2, the **and** instruction can issue as **$r0** is forwarded from **sub** to **and**. The **or** instruction is independent, so it also issues
- In cycle 3, **$t0** is available, so **add** issues. **$t3** is also available, so **sw** issues

Example

- For executing this MIPS code on the 5-stage pipelined MIPS processor: (a) Use nops to avoid hazards; (b) Instead of inserting nops, the processor stalls on hazards; and (c) use forwarding to avoid hazards

```
(i)
I1 : lw  $1,40($6)
I2 : add $6,$2,$2
I3 : sw  $6,50($1)
```

```
(ii)
I1 : lw  $5,-16($5)
I2 : sw  $5,-16($5)
I3 : add $5,$5,$5
```

- (a) (i) RAW on $1 from I1 to I3. RAW on $6 from I2 to I3. WAR on $6 from I1 to I2 and I3. (ii) RAW on $5 from I1 to I2 and I3. WAR on $5 from I1 and I2 to I3. WAW on $5 from I1 to I3. (b) In the in-order execution, WAR and WAW dependences do not cause any hazards

```
lw  $1,40($6)
add $6,$2,$2
nop //delay I3 to avoid RAW
sw  $6,50($1)
```

```
lw  $5,-16($5)
nop  //delay I2 to avoid RAW
nop
sw  $5,-16($5)
add $5,$5,$5
```

- (c) A load cannot forward to the EX stage of the next instruction

```
lw  $1,40($6)
add $6,$2,$2
sw  $6,50($1)
```

```
lw  $5,-16($5)
nop  //delay I2 to avoid RAW
sw  $5,-16($5)
add $5,$5,$5
```

Example

- Consider a dual-issue MIPS processor that can issue one ALU/Branch instruction and one load/store instruction concurrently when there are no dependencies. (a) Schedule the following code for when the processor only supports in-order execution. (b) Schedule the following code for when the processor supports out-of-order execution

```
Loop:
I1 :  lw   $t0, 0($s1)
I2 :  add  $t0, $t0, $s2
I3 :  sw   $t0, 0($s1)
I4 :  addi $s1, $s1,-4
I5 :  bne  $s1, $0, Loop
```

CLK	ALU/BRANCH	LOAD/STORE
1	nop	lw $t0,0($s1)
2	nop	nop
3	add $t0,$t0,$s2	nop
4	addi $s1,$s1,-4	sw $t0,0($s1)
5	bne $s1,$0,Loop	nop

CLK	ALU/BRANCH	LOAD/STORE
1	nop	lw $t0,0($s1)
2	addi $s1,$s1,-4	nop
3	add $t0,$t0,$s2	nop
4	bne $s1,$0,Loop	sw $t0,4($s1)

- How would this loop be scheduled on a static two-issue pipeline for MIPS? Reorder the instructions to avoid as many pipeline stalls as possible. Assume branches are predicted, so that control hazards are handled by the hardware.

```
Loop:
I1 : lw   $t0, 0($s1)
I2 : addu $t0,$t0,$s2
I3 : sw   $t0, 0($s1)
I4 : addi $s1,$s1,-4
I5 : bne  $s1,$0,Loop
```

CLK		
1	Loop :	lw $t0, 0($s1)
2	addi $s1,$s1,-4	
3	addu $t0,$t0,$s2	
4	bne $s1,$0,Loop	sw $t0, 0($s1)

Example

- Assume that the following code is executed on a 5-stage pipelined two-way superscalar datapath. Assume that the loop exits after executing only two iterations. In the following pipeline diagram, schedule the instructions for correct execution.

```
I1 : add  $1,$0,$0
Again:
I2 : beq  $1,$2,End
I3 : add  $6,$3,$1
I4 : lw   $7,0($6)
I5 : add  $8,$4,$1
I6 : sw   $7,0($8)
I7 : addi $1,$1,1
I8 : beq  $0,$0,Again
End:
```

CLK	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
I1	F	D	X	M	W													
I2	F	D	S	X	M	W												
I3		F	S	D	X	M	W											
I4		F	S	D	S	X	M	W										
I5				F	S	D	X	M	W									
I6				F	S	D	S	X	M	W								
I7						F	S	D	X	M	W							
I8						F	S	D	S	X	M	W						
I2								F	S	D	X	M	W					
I3								F	S	D	S	X	M	W				
I4										F	S	D	X	M	W			
I5										F	S	D	X	M	W			
I6												F	D	X	M	W		
I7												F	D	X	M	W		
I8													F	D	X	M	W	
I2													F	D	S	X	M	W

Example

- **(a)** In the following diagram, draw the execution of the following instructions on a two-way superscalar five-stage pipelined MIPS processor. In the following diagram show forwarding and stalls needed to execute the instructions correctly. **(b)** What is the IPC for this program? **(c)** If the processor can execute instructions out-of-order, show the flow of instruction executions in the following diagram to obtain maximum IPC. Show all of the existing hazards and forwarding processes in the following diagram as well. What is the IPC for your re-ordered execution of the above? **(d)** Using any technique, can you maximize the IPC?

```
lw  $t0, 40($s0)
add $t1, $t0, $s1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or  $t3, $s5, $s6
sw  $s7, 80($t3)
```

INST/CLK	1	2	3	4	5	6	7	8	9
Lw/1	F	D	E	M	W				
Add/2		F	D	D	E	M	W		
Sub/2									
And/3			F	F	D	E	M	W	
Or/3									
Sw/4					F	D	E	M	W

Ideal IPC is 2. This program requires five cycles to issue six instructions. Thus the actual IPC is 6/5 = 1.17

INST/CLK	1	2	3	4	5	6	7	8	9
Lw/1	F	D	E	M	W				
Or/1									
sw/2		F	D	E	M	W			
Add/3			F	D	E	M	W		
sub/3									
and/4				F	D	E	M	W	

Out of order execution

INST/CLK	1	2	3	4	5	6	7	8	9
Lw/1	F	D	E	M	W				
SW/1									
AND/2		F	D	E	M	W			
OR/2									
Add/3			F	D	E	M	W		
SW/3									

Actual IPC: 6/3 = 2

Example

- (a) If the 5-stage pipelined processor is able to execute instructions in out-of-order fashion, what would be the order of the instructions being executed on the processor? (b) If the processor also includes a register renaming module, would it be possible for the processor to achieve the maximum instructions per cycle IPC = 2 when running the above program?

```
1.  lw  $s8, 12($t1)
2.  add $s0, $s8, $s1   (RAW $s8)
3.  sub $s8, $s2, $s3
4.  and $t2, $s4, $s8   (RAW $s8)
5.  or  $t3, $s5, $s6
6.  sw  $s7, 14($t3)    (RAW $s8)
```

(a)

```
1.  lw  $s8, 12($t1)
2.  or  $t3, $s5, $s6

3.  sw  $s7, 14($t3)
4.  -

5.  add $s0, $s8, $s1
6.  sub $s8, $s2, $s3

7.  and $t2, $s4, $s8
```

(b)

```
1.  lw  $s8, 12($t1)
2.  sub $r0, $s2, $s3

3.  and $t2, $s4, $r0
4.  or  $t3, $s5, $s6

5.  add $s0, $s8, $s1
6.  sw  $s7, 14($t3)
```

$s8 is renamed to $r0 in sub and and to eliminate the WAR hazard.

Static multiple issue

- *Very long instruction word* (*VLIW*) processors are *static multiple-issue processors* relying on the compiler to reduce or avoid all hazards and to decide which independent instructions to issue simultaneously to maximize ILP. Similar to superscalar processors, VLIW processors use multiple functional units and multi-ported register files
- VLIW uses a single instruction format with several predefined fields, which specifies multiple independent instructions. Each field is for a fixed instruction type and each instruction can have a different latency

Int Op 1	Int Op 2	Mem Op 1	Mem Op 2	FP Op 1	FP Op 2
Two Integer Units *Single cycle latency*		*Two Load/Store Units* *Three cycle latency*		*Two Floating-Point Units* *Four cycle latency*	

- Multiple independent instructions of a VLIW instruction are fetched, decoded, and issued together in a single cycle, called an *issue packet*
- VLIW architectures are used in DSP processors and the compiler performs static branch prediction and removes some/all hazards statically. Unfortunately, compilers do not have runtime information and branch and memory prediction is more challenging. Since no run-time hazard detection or hardware-generated stalls are supported, VLIWs have a simpler architecture than superscalar processors with lower power consumption

Loop unrolling

- *Loop unrolling* or *loop unfolding* is a transformation technique that can be applied to a software loop to create a new loop describing more than one iteration of the original loop. Unfolding a loop by the unfolding factor *J* describes *J* consecutive iterations of the loop
- For example, this code shows a dot-product computation of two vectors of size *N*. For computing each MAC operation, the *loop overhead* consists of incrementing the loop counter *i* and comparing the incremented value with *N* in every loop iteration, which takes a number of clock cycles

```
sum=0;
for (i=0; i<N; i++)
  sum += a[i]*b[i];
```

- Loop unrolling can be used to reduce the loop overhead while using multiple computational units. Moreover, after loop unrolling, there is more ILP available by scheduling instructions from different iterations together
- Unrolling a loop will increase the machine code size, while reducing the loop control overhead (number of conditional branch instructions) and increases performance by utilizing more computational units

```
J=4;
sum0=0;
sum1=0;
sum2=0;
sum3=0;
for(i=0; i<N; i=i+J){
  sum0 += a[i]*b[i];
  sum1 += a[i+1]*b[i+1];
  sum2 += a[i+2]*b[i+2];
  sum3 += a[i+3]*b[i+3];
}
sum = sum0+sum1+sum2+sum3;
```

Example

- To schedule this loop on a static two-issue VLIW processor, we note that the first three instructions and the last two have data dependencies

```
lp: lw   $t0,0($s1)  # $t0=array element
    addu $t0,$t0,$s2 # add scalar in $s2
    sw   $t0,0($s1)  # store result
    addi $s1,$s1,-4  # decrement pointer
    bne  $s1,$0,lp   # branch if $s1 != 0
```

- Must separate instructions using the **lw** results by at least one cycle. **addu** and **sw** have a RAW dependency. **addi** and **bne** also have a RAW dependency

	ALU or branch	Data transfer
lp:		lw $t0,0($s1)
	addi $s1,$s1,-4	
	addu $t0,$t0,$s2	
	bne $s1,$0,lp	sw $t0,4($s1)

- Note that in the scheduled instructions, **addi** first subtract **$s1** by 4 and **sw** uses an offset 4 instead of 0
- In the above example, four clock cycles to execute 5 instructions . Thus CPI of 0.8 versus the best case of 0.5 and IPC of 1.25 versus the best case of 2.0
- If an instruction cannot be used in a VLIE instruction, it is replaced with a **nop**. **nops** do not perform any useful operation and in computing CPI, they are not counted. **nops** are a waste of program memory space and performance

Loop unrolling

```
for (i=0; i<N; i++)
    B[i] = A[i] + C;
```

Unroll the loop to perform 4 iterations at once

```
for (i=0; i<N; i+=4){
    B[i]   = A[i] + C;
    B[i+1] = A[i+1] + C;
    B[i+2] = A[i+2] + C;
    B[i+3] = A[i+3] + C;
}
```

	ALU or branch	Data transfer
lp:	addi $s1,$s1,-16	lw $t0, 0($s1)
		lw $t1,12($s1)
	addu $t0,$t0,$s2	lw $t2, 8($s1)
	addu $t1,$t1,$s2	lw $t3, 4($s1)
	addu $t2,$t2,$s2	sw $t0,16($s1)
	addu $t3,$t3,$s2	sw $t1,12($s1)
		sw $t2, 8($s1)
	bne $s1,$zero,lp	sw $t3, 4($s1)

- To schedule the loop with the unfolding factor of 4, we need to make four copies of the loop body. If the loop index *N* is not a multiple of unrolling factor 4, a final iteration is required. After unrolling and eliminating the unnecessary loop overhead instructions, the loop will contain four copies each of `lw`, **add**, and `sw`, plus one **addi** and one **bne**

Register renaming during unrolling process

- During unrolling, the compiler uses *register renaming* technique to eliminate WAR and WAW hazards that are not true data dependencies. In the above example, registers `$t1, $t2, $t3` are used
- Consider the unrolled code using only `$t0` and the repeated instances of `$t0` in `lw $t0,0($s1)`, `addu $t0, $t0, $s2`, and `sw $t0,4($s1)`. Despite using `$t0`, these sequences are actually completely independent—no data values flow between one set of these instructions and the next set
- Note that the pipeline structure affects the number of times a loop must be unrolled to avoid stalls as well as the process of compiler-based register renaming
- Now that 12 of the 14 instructions execute as pairs, it takes 8 clocks for 4 loop iterations, or 2 clocks per iteration, which yields a CPI of 8/14 = 0.57
- Loop unrolling and scheduling with dual issue results in an improvement factor of almost 2, partly from reducing the loop control instructions and from dual issue execution
- The cost of this performance improvement is using four temporary registers rather than one, as well as an increase in code size

Chapter 03 – Cache Memories and Parallel Architectures

Example

- Schedule this code for dual-issue MIPS. What is the IPC?

```
Loop:
lw $t0, 0($s1)
add $t0, $t0, $s2
sw $t0, 0($s1)
addi $s1, $s1,-4
bne $s1, $0, Loop
```

CLK	ALU/BRANCH	LOAD/STORE
1	nop	lw $t0,0($s1)
2	nop	nop
3	add $t0,$t0,$s2	nop
4	addi $s1,$s1,-4	sw $t0,0($s1)
5	bne $s1,$0,Loop	nop

IPC = 5/5 = 1 (peak dual-issue IPC = 2 and single issue IPC = 5/6 = 0.83 for single-issue pipeline)

CLK	ALU/BRANCH	LOAD/STORE
1	nop	lw $t0,0($s1)
2	addi $s1,$s1, -4	nop
3	add $t0,$t0,$s2	nop
4	bne $s1,$0,Loop	sw $t0,4($s1)

IPC = 5/4 = 1.25 (peak IPC = 2)

- Unroll the loop four times and show its scheduling in the following table

CLK	ALU/BRANCH	LOAD/STORE
1	Loop: addi $s1,$s1,-16	lw $t0,0($s1)
2	nop	lw $t1,12($s1)
3	add $t0,$t0,$s2	lw $t2,8($s1)
4	add $t1,$t1,$s2	lw $t3,4($s1)
5	add $t2,$t2,$s2	sw $t0,16($s1)
6	add $t3,$t3,$s2	sw $t1,12($s1)
7	nop	sw $t2,8($s1)
8	bne $s1,$0,Loop	sw $t3,4($s1)

IPC = 14/8 = 1.75
Closer to 2, but at cost of registers and code size

Limitations of memory system performance

- Memory system, and not processor speed, is often the performance bottleneck for many applications. Unfortunately, DRAM speed has improved by a significantly lower rate compared to processors' performance
  - The DRAM access time is one to two orders of magnitude longer than the processor cycle time (tens to hundreds of nanoseconds, compared to less than one nanosecond)
- *Memory-CPU performance gap*: CPUs are bandwidth limited with the relatively slow *memory interface* between high-capacity external memory and the CPU
- Memory system performance can be expressed by two parameters:
  - *Latency* is the time from the issue of a memory request to the time the data is available at the processor. DRAM latency is long (about 100s of ns). A 2 GHz CPU has a 0.5ns clock cycle. The latency of a 100 ns memory is 200 CPU cycles
  - *Bandwidth* is the rate at which data can be supplied to the processor by the memory system (Width of the data channel × the rate at which it can be used)
- *Memory access time*: time between the request and when the data is available at the output port of the memory (or written)
- *Memory cycle time*: time between requests. Usually cycle time is greater than access time

Memory hierarchy

- Ideally, the memory is fast, large, and cheap
  - In practice, a memory only has two of these three attributes
- How can a memory that gives the illusion of being large, cheap and fast be realized?
- To reduce the performance gap between the processor and DRAM, the processors combine a fast and small, but expensive *cache memory* with a slow and large, but cheap *main memory*
  - All modern computers use caches
  - The cache is usually built out of SRAM on the same chip as the processor and its speed is comparable to the processor speed. SRAMs are inherently faster than DRAMs and the on-chip memory eliminates lengthy delays caused by traveling to and from a separate memory chip
  - The combination of two memories is much less expensive than a single large fast memory
- Cache acts as a buffer for a slower but larger memory. It keeps the most frequently accessed DRAM memory locations in cache to avoid repeatedly accessing repeatedly slower DRAMs and hence the effective latency of the memory system can be reduced and on average the memory system appears fast
  - Cache acts as a low-latency high-bandwidth storage

Hit and miss rates

- Caches can store both instructions and data, but we will refer to their contents generically as "data"
- *Cache hit*: The processor requests data that is available in the cache
- *Cache miss*: is when a request for data from the cache cannot be met because the data is not present in the cache (processor retrieves the data from main memory)
- *Hit rate*: fraction of data references found in the cache

$$Hit\ Rate = \frac{Number\ of\ hits}{Number\ of\ total\ memory\ accesses}$$

- *Miss rate*: fraction of memory accesses that miss in the cache

$$Miss\ Rate = \frac{Number\ of\ misses}{Number\ of\ total\ memory\ accesses} = 1 - Hit\ Rate$$

- *Miss penalty*: is the time required to read a block into a level of the memory hierarchy from the lower level, including the time to access the block, transmit it from one level to the other, write it in the level that experienced the miss, and then pass the block to the processor
- *Hit time*: is the time required to access a level of the memory hierarchy, including the time needed to determine whether the access is a hit or a miss
  - Increasing the cache size will increase hit time

Principle of data locality and cache

- Exploiting the *principle of locality* makes a hierarchical memory organization fast
- *Temporal Locality* (locality in time): If data item needed now, it is likely to be needed again in near future
  - Recently accessed data is kept in higher levels of memory hierarchy (closer to the processor). Repeated references to the same data item correspond to temporal locality
- *Spatial Locality* (locality in space): If data item needed now, nearby data likely to be needed in near future
  - For example, if an element in an array is used, other elements in the same array are also likely to be used, creating spatial locality. Therefore, bring in nearby data
- Caches use spatial and temporal locality to predict what data will be needed next
  - If a program accesses data in a random order, it would not really benefit from a cache
- Making the memory system ten times faster will not necessarily make a computer program run ten times as fast. *Amdahl's Law* states that the effort spent on increasing the performance of a subsystem is worthwhile only if the subsystem affects a large percentage of the overall performance

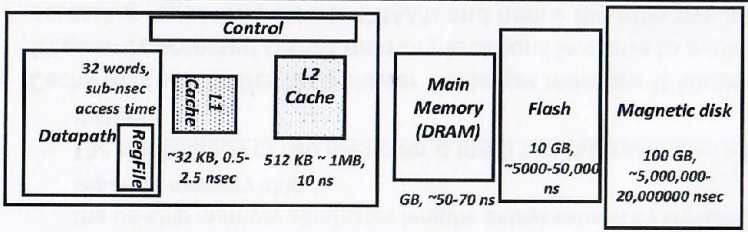
Average memory access time (AMAT)

- The *average memory access time* (AMAT) is the average time a processor must wait for memory per load or store
- First, the item is sought in the first memory level of the memory hierarchy
  - The probability of finding the requested item in the first level is called the *hit ratio*, h_1
  - The probability of not finding (missing) the requested item in the first level of the memory hierarchy is called the *miss ratio*, $(1- h_1)$
- When the requested item causes a "miss," it is sought in the next subsequent memory level
  - The probability of finding the requested item in the second memory level, the hit ratio of the second level, is h_2
  - The miss ratio of the second memory level is $(1 - h_2)$
- The process is repeated until the item is found and it is sent to the processor. For example, in a memory hierarchy that consists of three levels, the AMAT can be expressed as: $t_{av} = h_1 \times t_1 + (1-h_1)[t_1 + h_2 \times t_2 + (1-h_2)(t_2+t_3)] = t_1 + (1-h_1)[t_2 + (1-h_2)t_3]$
  - In this equation, t_1, t_2, t_3 represent the access times of the three levels, respectively

Example

- Suppose a computer system has only two levels of memory hierarchy, a cache and main memory. What is the average memory access time given $t_1 = 1$ cycle, $t_2 = 100$ cycles, $1\text{-}h_1 = 10\%$, $1\text{-}h_2 = 0\%$
- The average memory access time is $1 + 0.1(100) = 11$ cycles
  - An 11-cycle AMAT means that the processor spends ten cycles waiting for data for every one cycle actually using that data
- What cache miss rate is needed to reduce the average memory access time to 1.5 cycles given the above access times?
- If the miss rate is m, the average access time is $1 + 100m$. Setting this time to 1.5 and solving for m requires a cache miss rate of 0.5%
- If the cache hits most of the time, then the processor seldom has to wait for the slow main memory, and the average access time is low
  - Hit time is significantly less than miss penalty (otherwise, there will be no reason to build a memory

Reducing the miss penalty using multilevel caches



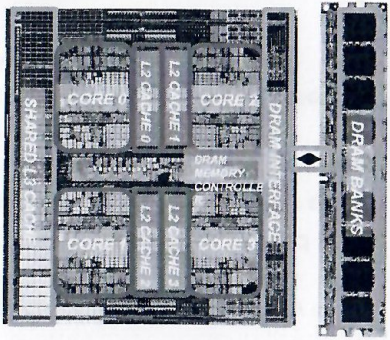
- Instead of only two levels of memories, the memory system of modern computers consists of a hierarchy of memory blocks that vary in size, speed, and cost. Although cache hierarchies have been used to mask the relatively long latency of accessing off-chip (outside CPU) memories, each additional cache level introduces additional complexity and has an asymptotic performance limit
- In a Harvard-style L1, code and data are partitioned into code and data caches, while L2 and L3 provide a "unified" memory space (i.e., the instruction and data fetch both access a common memory space)

Level 1, 2, and 3 caches

- To close the gap further between the fast clock rates of modern processors and the increasingly long time required to access DRAMs, most microprocessors support two levels of caching
- The memory closest to the core processor, known as *Level 1*, or L1 cache, operates at the core-clock rate and is often partitioned into instruction and data segments
- Larger on-chip memory is called *Level 2* (L2) cache. It is also on the same processor chip and operates at core speed and is accessed whenever a miss occurs in the higher level L1 cache, however accesses typically take multiple clock cycles
  - L1 and L2 memories are each further divided into sub-banks to allow concurrent core and DMA access in the same cycle
  - A two-level cache structure allows the primary cache to focus on minimizing hit time to yield a shorter clock cycle, while allowing the secondary cache to focus on miss rate to reduce the penalty of long memory access times
- *Off-chip memory* or *external memory* is referred to as *Level 3* (L3) memory
- In general, multiple internal data paths lead to the external memory interface
  - For example, one or two core access paths, as well as multiple DMA channels, all can contend for access to L3 memory

Placement and replacement policies

- When the processor attempts to access data, it first checks the cache for the data (this is called a *cache lookup*)
  - If the cache hits, the data is available in nanoseconds to processor
  - If the cache misses, the processor fetches the data (actually, bring entire block of data, too) from main memory and places it in the cache for future use using a *placement policy*



  - *Placement policy* defines where and how to place a block in cache
- Note that the capacity of the cache is smaller than that of main memory and only a small subset of the data is stored in the cache
  - To accommodate the new data, the cache must replace old data using a *replacement policy*
  - *Replacement policy* determines which block in the set to replace on a cache miss

Cache characteristics

- When accessing memory, a group of words (adjacent data), called a *cache block* or a *cache line*, usually more than one word, is brought from lower level of memory into higher levels of memory hierarchy
  - The number of words in the cache block is called the *block size b*
  - If the adjacent words in the block are not accessed later, the effort of bringing them is wasted. Nevertheless, most real programs benefit from larger block sizes
  - Cache block is not necessarily the unit of data transfer between cache and main memory, but it is important to use DRAMs that support fast multiple word accesses, preferably ones that match the block size of the cache
- *Memory capacity C* is the number of data words that a memory can hold. A cache has $B = C/b$ blocks
- A cache is organized into *S sets*, each of which holds one or more *ways* (blocks)
  - Each way contains one block, which can be multiple words, but for now assume b=1
- The number of ways (blocks) in a set *degree of associativity N*. You can consider caches as two-dimensional arrays, where each row is called a set and the columns are called ways

Direct-mapped cache

- Direct-mapped is a simple placement scheme in which a block in the memory can go in exactly one set (way) in the cache
- A direct mapped cache has one way in each set, so it is organized into $S = B$ sets
- Cache mapping = (Block address) modulo (Number of blocks in the cache)
- If the number of entries in the cache is a power of 2, then modulo can be computed using the low-order $\log_2$(cache size in blocks) bits of the memory address
  - In this case the cache is accessed using the *set bits*
  - In MIPS, the next $\log_2 S$ least significant bits of the address to byte offset, called set bits, specify which set holds the data
  - Set bits specify the entry or set onto which the memory address maps
- The remaining most significant bits are the *tag* and indicates the memory address of the data stored in a given cache set
  - Tag indicates which of the many possible addresses is held in that set

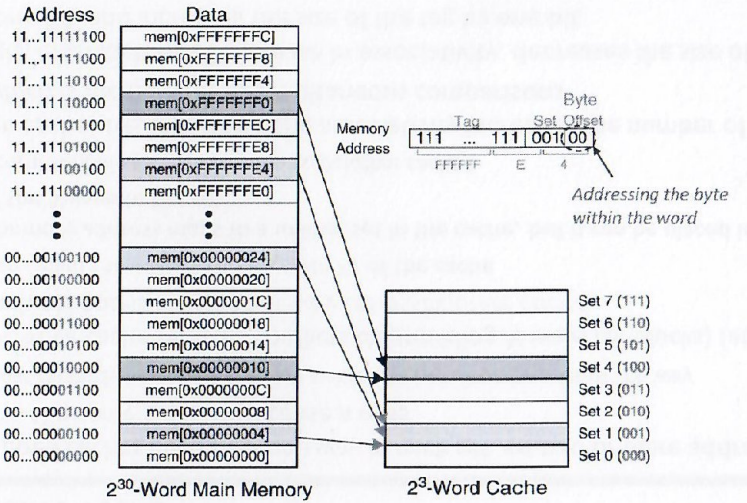
Memory address mapping

- The relationship between the address of data in main memory and the location of that data in the cache is called the *mapping*
- Each memory address maps to exactly one set in the cache
  - If the set contains more than one way, the data may be kept in any of the blocks in the set
  - Because many addresses map to a single set, the cache must also keep some of the address bits of the data in the cache, which are called *tags*
- Caches are categorized based on the number of ways in a set
- *Direct-mapped* cache: each set contains one way, so the cache has $S = B$ sets
  - A particular main memory address maps to a unique way in the cache
- *N-way set associative* cache: each set contains N ways
  - The address still maps to a unique set, with $S = B/N$ sets
  - But the data from that address can go in any of the N ways in that set
- *Fully associative* cache: has only $S = 1$ set
  - Data can go in any of the B ways in the set
  - A fully associative cache is another name for a B-way set associative cache

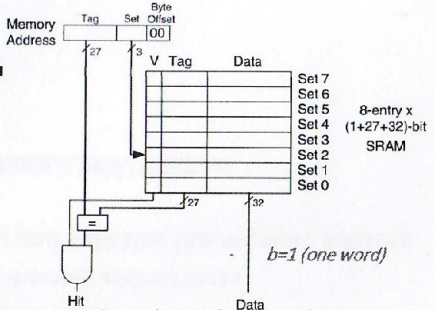
Organization	Number of Ways (N)	Number of Sets (S)
direct mapped	1	B
set associative	$1 < N < B$	B/N
fully associative	B	1

Direct-mapped cache for MIPS errors

- In the MIPS architecture, since words are aligned to multiples of four bytes, the least significant two bits of every address specify a byte within a word
  - Hence, the least significant two bits are ignored when selecting a word in the block

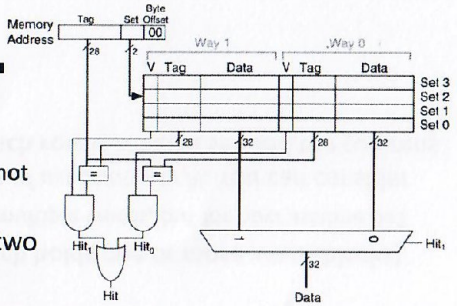


Cache data entry format



- Each way consists of a data block, the valid and tag bits. In this example, each entry in the cache (set), contains 32 bits of data, 27 bits of *tag*, and 1 *valid bit*
- A valid bit indicates whether the line holds meaningful data
- A valid bit is "1" if the contents of this cache line contain values that can be used directly. When the line is invalid, its contents can't be used
- A load instruction reads the specified entry from the cache first and checks the tag and valid bits
  - If the tag matches the most significant 27 bits of the address and the valid bit is 1, the cache hits and the data is returned to the processor. Otherwise, the cache misses and the memory system must fetch the data from main memory
- A *content addressable memory* (CAM) combines comparison and storage in a single device. The data is passed to the CAM and it looks to see if it has a copy and returns the index of the matching row

Set-associative cache



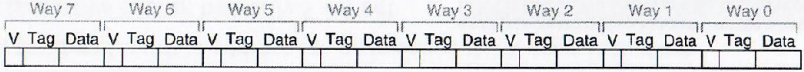
- e.g., consider a C = 8-word N = 2-way set associative cache
  - The cache capacity refers to the total data (not including tag and valid bit)
- The cache has S = 4 sets; each set contains two ways, block size b=1 (one word)
- Cache access: The set value is used to select the set and the tags and valid bits of all the blocks in the set must be compared
- Set associative caches generally have lower miss rates than direct mapped caches of the same capacity, because they have fewer conflicts, however, set associative caches are usually slower and are larger because of the output multiplexer and additional comparators
  - In this example, a larger tag (28 bits) is required
  - Also logic is required to decide which way to replace when ways are full
- Larger caches can exploit temporal locality better, however, smaller caches are faster, but useful data is replaced more often

Set-associative cache

- Direct mapped caches have only one way in each set, so two or more addresses that map to the same set always cause a miss
  - The most recently accessed address evicts the previous one from the way
- A *set associative* cache reduces conflicts by providing *N* ways (or blocks) (at least *N*=2) in each set and is called an *N-way set-associative cache*
  - *N* is also called the *degree of associativity* of the cache
  - Each memory address maps to a unique set in the cache, but it can be placed in any one of the *N* ways in the set
  - Most commercial systems use set associative caches
- For a given cache size, increasing the associativity increases the number of ways per set, which is the number of simultaneous comparisons
- Accordingly, each factor-of-2 increase in associativity, decreases the size of the set index by one bit, and increases the size of the tag by one bit
- Direct mapped cache is a *one-way set associative cache*

Fully-associative cache

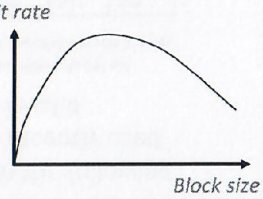
- A *fully-associative* cache contains a single set with *B* ways, where *B* denotes the number of blocks



  - A fully-associative cache is another name for a *B-way set associative cache* with one set
  - A memory address can map to any of these *B* ways (any location in the cache)
  - There is no bits for sets and hence, the entire address, excluding the block offset, is compared against the tag of every block
- Since a block can be placed in any way, fully-associative caches have the fewest conflict misses for a given cache capacity
- All the entries must be searched in parallel with a comparator associated with each cache entry (way). These comparators significantly increase the hardware area, effectively making fully-associative placement practical only for caches with small numbers of blocks
- In the above example cache architecture, upon a data request, eight tag comparisons must be made, because the data could be in any way, and an 8:1 multiplexer chooses the proper data if a hit occurs

How to exploit spatial parallelism?

- The previous examples were able to take advantage only of temporal locality, because the block size *b* was one word
- The advantage of a block size greater than one is that when a cache miss occurs and the word is read into the cache, the adjacent words in the block are also fetched. Subsequent accesses are thus more likely to hit because of spatial locality
- To exploit spatial locality, larger blocks are used to hold several consecutive words
  - Larger blocks exploit spatial locality to lower miss rates
  - Block size *b* or number of words in the block is associated with only one address tag because the words in the block are at consecutive addresses
  - The larger block sizes, the smaller tag overhead
- For a fixed-size cache, increasing the block size will cause the cache to have fewer blocks, and there will be a great deal of competition for those blocks
  - This may lead to less temporal locality exploitation and hence, increasing the miss rate

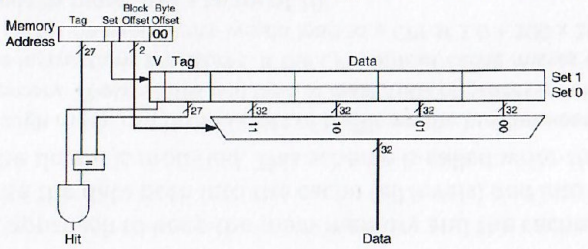


Capacity of the direct-mapped cache

- If the cache size is $B=2^s$ blocks, s bits are used for the set
- The block size is 2^w words (2^{w+2} bytes)
  - w bits are used for the word within the block
  - two bits are used for the byte part of the address
- The size of the tag field is $32 - (s + w + 2)$
- The index of a cache block, together with the tag contents of that block, uniquely specifies the memory address of the word contained in the cache block
- The total number of bits in a direct-mapped cache is $2^s \times$(block size + tag size + valid field size)
- Since the block size is 2^w words (2^{w+5} bits), and we need 1 bit for the valid field, the number of bits in such a cache is:
  $2^s \times (2^w \times 32 + (32 - s - w - 2) + 1) = 2^s \times (2^w \times 32 + 31 - s - w)$
- Although this is the actual size in bits, the naming convention is to exclude the size of the tag and valid fields and to count only the size of the data as cache size ($2^s \times 2^w$)

Using larger blocks in direct mapped caches

- Consider a C =8-word direct mapped cache with a b = 4-word block size
- The cache has $B = C/b = 2$ blocks and a *block offset* is used to select the block
- The following direct mapped cache is organized as two sets, with one way (block) in each set. Thus one bit is used to select the set and two bits are used to select the word within the block
  - The multiplexer is controlled by two *block offset bits* of the address



Cache example

- Consider the MIPS data memory with 32 bits address ports. Assume the cache capacity is 256K words (not including tag and valid bits). Each cache block contains 4 words. For each of the following cache configurations, (i) direct mapped, (ii) 2-way set associative, (iii) 4-way set associative, and (iv) fully associative specify how the 32-bit address would be partitioned. The MIPS memory is byte addressable.

Byte offset: Since each word is 4 bytes, the least two significant bits are used to select the byte offset.
Word offset: Since there are b=4 words in each block, the next two least significant bits are used to for the word offset in the block.
Since there are 4 words per cache block and 256K words in the cache, there are B=256K/4 = 64K blocks in the cache. The remaining 28 bits are partitioned based on the associativity of the cache as follows:

(i) Direct mapped: Thus, the next $\log_2(64K) = 16$ bits of the address are needed to specify the cache set (block). The remaining bits are used for the tag.

Cache index: bits 4-19; Tag: bits 20-31

(ii) 2-way set associative: Since there are two blocks per set and 64K blocks in the cache, the cache has 32K sets. Thus, the number of bits required to specify a set is $\log_2(32K) = 15$. The remaining bits are used for the tag:

Cache index: bits 4-18; Tag: bits 19-31

(iii) 4-way set associative: Since there are four blocks per set and 64K blocks in the cache, the cache has 16K sets. Thus, the number of bits required to specify a set is log2(16K) = 14. The remaining bits are used for the tag.

Cache index: bits 4-17; Tag: bits 18-31

(iv) Fully associative: Since a block can be placed anywhere in the cache, all the remaining bits other than the byte and block offsets are used for the tag.

Cache index: 4-31

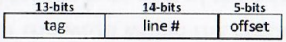
Cache example

- Suppose we have 32-bit memory addresses, a byte-addressable memory, and a 512 KB (2^{19} bytes) cache with 32 (2^5) bytes per block. **(a)** How many total lines are in the cache? **(b)** If the cache is direct-mapped, how many cache lines could a specific memory block be mapped to? **(c)** If the cache is direct-mapped, what would be the format (tag bits, cache line bits, block offset bits) of the address? (Clearly indicate the number of bits in each) **(d)** If the cache is 2-way set associative, how many cache lines could a specific memory block be mapped to? **(e)** If the cache is 2-way set associative, how many sets would there be? **(f)** If the cache is 2-way set associative, what would be the format of the address? **(g)** If the cache is fully associative, how many cache lines could a specific memory block be mapped to? **(h)** If the cache is fully-associative, what would be the format of the address?

(a) $2^{19}/2^5 = 2^{14}$ lines

(b) 1

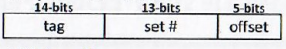
(c)



(d) 2 sets of size 2

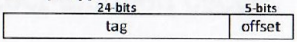
(e) $2^{14}/2^1 = 2^{13}$ sets

(f)



(g) 2^{13} (any)

(h)



Cache example

- Consider three small caches, each consisting of four one-word blocks. One cache is direct-mapped, one is two-way set-associative, and the third is fully-associative. Find the number of misses for each cache organization given the following sequence of block addresses: 0, 8, 0, 6, and 8. Use the least recently used replacement policy. A blank entry to mean that the block is invalid.

Direct-mapped

Address	Hit/miss	0	1	2	3
0	Miss	Mem[0]			
8	Miss	Mem[8]			
0	Miss	Mem[0]			
6	Miss	Mem[0]		Mem[6]	
8	Miss	Mem[8]		Mem[6]	

Set-associative with two sets and two elements per set.
Note that the block can be stored in any of the blocks in a set.

Address	Hit/miss	Set0	Set0	Set1	Set1
0	Miss	Mem[0]			
8	Miss	Mem[0]	Mem[8]		
0	Hit	Mem[0]	Mem[8]		
6	Miss	Mem[0]	Mem[6]		
8	Miss	Mem[8]	Mem[6]		

Fully associative

Address	Hit/miss	Block0	Block1	Block2	Block3
0	Miss	Mem[0]			
8	Miss	Mem[0]	Mem[8]		
0	Hit	Mem[0]	Mem[8]		
6	Miss	Mem[0]	Mem[8]	Mem[6]	
8	Hit	Mem[0]	Mem[8]	Mem[6]	

Cache replacement policies

- In a direct-mapped cache, the requested block can go in exactly one position, and the block occupying that position must be replaced
- In a set-associative cache, we must choose among the blocks in the selected set
- In a fully-associative cache, all blocks are candidates for replacement
  - First check any invalid block
  - If all are valid, need to evict a block to insert a new block
- Several replacement policies
  - *Random replacement* (randomly selected line)
  - *FIFO* (way that has been in cache the longest)
  - *LRU* (least recently used line); can be implemented using a counter for each block
- The most commonly used scheme is LRU. In the LRU scheme, the block replaced is the one that has been unused for the longest time
  - LRU replacement can be implemented by keeping track of when each element in a set was used relative to the other elements in the set. Most modern processors do not implement "true LRU" (also called "perfect LRU") as it is relatively complex. Instead, LRU is just an approximation to predict locality
- Implementing LRU becomes more challenging as associativity increases

Writing-through cache

- Suppose a store instruction that writes a data into the cache. This data should also be updated in the main memory as otherwise the cache and memory will be *inconsistent*
- The simplest approach to keep the main memory and the cache *consistent* is always to write the data both into the cache (all levels) *and* into the external memory at the time it is modified. This scheme is called *write-through*
  - Write-through mode can result in lots of traffic on the bus between cache and the source memory. These writes will take at least 100s of processor clock cycles. Suppose 10% of the instructions are stores. If the CPI without cache misses was 1.0, spending 100 extra cycles on every write would lead to a CPI of 1.0 + 100 x 10% = 11, reducing performance by more than a factor of 10
  - This activity can impact performance when other resources are accessing the memory
- A *write buffer* is a queue that can be used in the write-through policy to store the data while it is waiting to be written to memory
  - After writing the data into the cache and into the write buffer, the processor can continue execution. When a write to main memory completes, the entry in the write buffer is freed

Write-back cache

- In the *write-back* scheme, when a write occurs, the new value is written only to the block in the cache
- The modified block is written to the higher level of the memory hierarchy only when the block is replaced in the data cache
  - Therefore, if a data value is modified a million times, the result is only written locally to the cache until its cache entry is replaced, at which point main memory will be written a single time
- Write-back mode is usually faster because the processor does not need to write to source memory until absolutely necessary, however, write-back scheme is more complex to implement
- The advantage of write-through data cache over write-back is that it helps keeping buffers in memory coherent when more than one resource has access to them
  - For example, in a dual-core device with shared memory, a cache configured as write-through can help ensure that the shared buffer has the latest data in it
- The write policies do not apply to instruction memory as the code in the cache instruction does not change and hence, instruction cache is not designed with this feature

Flexibility of programmable processors

- The time and cost of implementing dedicated solutions is relatively high. The cost for a mask set is over one million dollar. By fabricating an IC in large volume (e.g., microprocessors), the cost of the layout masks can be amortized
- A programmable processor offers extreme flexibility since it can execute any program
  - A flexible chip can serve multiple applications
  - New features can be applied by changing the programs
  - Upgrades and updates are readily possible
  - Backward compatibility with previous versions
- In addition to software programmability, flexibility can be provided by *hardware reconfigurability*
- To provide flexibility and still retain the energy efficiency of dedicated designs, the architecture should provide enough flexibility while using dedicated hardware (FPGAs and ASICs)

Feature size

- A manufacturing process is characterized by its *feature size*, which indicates the length of the smallest transistor that can be reliably built
- The two domains in which technological advances have had the most impact on the performance of microprocessor systems have been:
  - (i) The increases in clock frequency. Smaller transistors are faster and generally consume less power. Even if the microarchitecture does not change between microprocessors generations the clock frequency can increase because transistors are faster
  - (ii) The continuous shrinking of the transistor size leading to higher logic density (more transistors on a chip) and hence, allowing the realization of more dedicated functionality, more memory, various architectural transformations, parallelism, and memory hierarchy
- In 1965, Gordon Moore, one of the founders of Intel, predicted that "the number of transistors on a given integrated circuit would double every two years"
- The clock frequency of the processors has roughly doubled every 2.5 years, but after 2003, the processor frequency has remained at 3 GHz range

Multiprocessing

- A *process* is an executing program along with its state, represented by the values of the program counter, architectural registers, and settings
  - A process is a dynamic state of a static *program*, which consists of the encoding of an algorithm and its data structures. Operating system stores information for each process in a data structure, called a *process table*
- When the program currently executing requires a relatively long latency operations, such as I/O processing which leaves the entire processor idle, it relinquishes the use of the CPU via a *context switch*, and another program starts executing on the CPU
- *Multiprocessing* is executing different processes, which is an extension of multiprogramming
- *Thread* (or *lightweight process*) is a single stream of instructions in the flow of a process. Each process consists of one or more threads. e.g., a word processor may have one thread handling the user typing, a second thread spell-checking the document while the user types, and a third thread printing the document
- Threads have the same properties as processes, but with the distinction that threads within a process share the same address space

How to reduce CPU idle time?

- Despite the existence of processors with four to six issues per clock, unfortunately, the ILP of a thread is generally limited (e.g., due to dependencies). Therefore, adding more execution units to a superscalar processor gives diminishing returns
- The processor may stall due to a cache miss, branch misprediction, or data dependency within a thread
  - While most loads and stores access a smaller and faster memory (cache), when the instructions or data are not available in the cache, the processor may stall for hundreds of cycles while retrieving the information from the main memory. Since memory latency on cache misses is long and the processor remain idle
- For efficient computing, processor resources should not be left *idle*. One solution to hide stalls is to execute instructions from other thread that are *independent* of those stalling instructions
  - Originally, *multiprogramming* emerged so that the processor would not remain idle when I/O tasks were performed

Coarse-grained multithreading

- A multithreaded processor contains a separate program counter, register file, instruction buffer, and store buffer for each thread (the execution units are shared)
  - If one thread stalls, the processor could context switch to another thread without any delay, because the program counter and registers are already available
  - A thread switch is more efficient than a process switch. A process switch typically requires hundreds of processor cycles, while a thread switch is ideally instantaneous in the next clock cycle
- In *coarse-grained multithreading* a thread switch occurs only when the currently executing thread encounters a relatively long-latency operation or stall, such as last-level cache misses, which requires an access to the main memory and cause stalls significantly longer than the number of cycles needed to switch threads
  - In the coarse-grained multithreaded superscalar, the long stalls are partially hidden by switching to another thread that uses the resources of the processor
- The disadvantage of Coarse-grain multithreading is that it doesn't hide short stalls, such as data hazards

Multithreading

- *Multithreading* (or *thread-level* parallelism) is the interleaved execution of multiple threads belonging to the same process
  - Multithreading increases the utilization of processor resources, even if the ILP within a thread is limited or the thread is stalled waiting for memory, by allowing multiple threads to utilize the functional units of a single processor
  - Multithreading does not improve the performance of an individual thread, because it does not increase the ILP, however, it does improve the overall throughput of the processor, because multiple threads can use processor resources that would have been idle when executing a single thread
- The threads take turns being executed on the processor under control of the operating system, which only gives the illusion of running simultaneously
  - Hardware switches among threads while waiting for long-latency operations
- In a *multithreaded processor*, when one thread's finishes its execution, the OS saves its architectural state, loads the architectural state of the next thread, and the processor starts executing that next thread
  - The processor state that identifies a thread is often called a *context*
  - Switching from a thread to another thread in the same process (called *thread context switching*) is not as costly as a context switch that occurs on switching processes, because part of the process state is replicated for each thread

Fine-grained multithreading

- In *fine-grained multithreading* the processor switches between threads on every clock cycle, resulting in *interleaved* execution of threads in a round-robin fashion
  - Even relatively short stalls, as little as one cycle as in a RAW hazard, are avoided. Thus fine-grained multithreading can hide both short and long stalls
  - Since thread switching occur cycle by cycle, there is no need to flush the pipeline
- Several threads can occupy different stages of the pipeline
  - Each instruction carries a tag identifying the set of registers attached to their thread
- The disadvantage of fine-grained multithreading is that it slows down the execution of an individual thread since a thread that is ready to execute without stalls is delayed by instructions from other threads
  - Coarse-grain multithreading has a simpler hardware and thread switching is less likely to slow down the execution of an individual thread

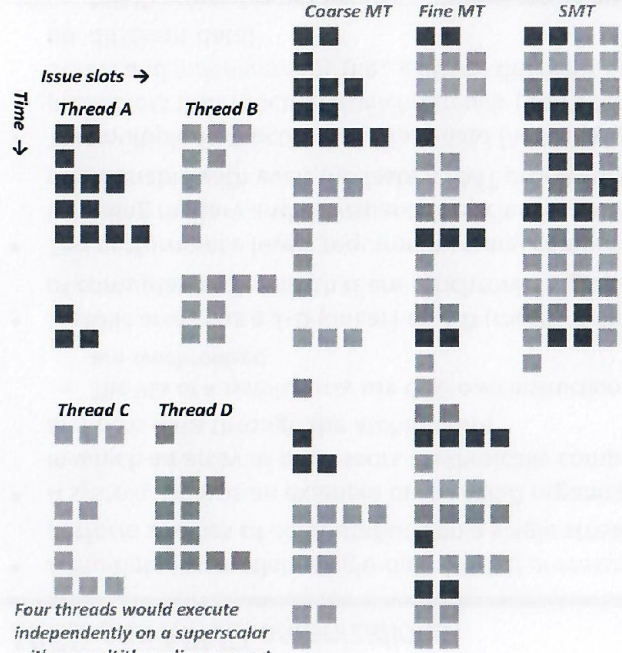
Simultaneous multithreading (SMT)

- Multiple-issue processors often have more functional unit parallelism than most single threads can effectively utilize via ILP
- In a superscalar processor, *simultaneous multithreading* (SMT) exploits both *thread-level parallelism* (TLP) and ILP from independent threads in a single clock cycle. Therefore, instructions from more than one thread can execute in any given pipeline stage in a clock cycle
  - Several threads can share the execution stage of the pipeline at the same time (e.g., multiple multipliers in the execution units can be utilized by instructions from the same thread (ILP) or from multiple independent threads
- The complexity of SMT has limited its implementation to two concurrent threads. Intel's SMT is called *hyperthreading*. It supports two threads

Chip multiprocessors (CMPs)

- For applications requiring more processing power than could be delivered by a single processor system, a *multiprocessor* system uses an array of processors
- *Chip multiprocessors* (CMPs) consists of two or more multi-threaded processors and their respective cache hierarchies on a single chip (one or more dies)
  - Each processor executes an independent stream of instructions (its own program) on often different data (multiple instruction, multiple data (MIMD) organization)
- General-purpose processors (GPPs) have been relying on CMPs with lower frequency instead of a single processor with greater frequency
  - Mostly homogeneous general-purpose CMPs have been realized
- It is challenging to partition the program into parallel functions and balancing the load evenly between the processors. The software thus need to facilitate the creation of concurrent processes and their synchronization, as well as protocols to ensure the consistency of data across the memory hierarchy
  - A memory system is *coherent* if any read of a data item returns the most recently written value of that data item

Multithreading on a 4-way superscalar processor

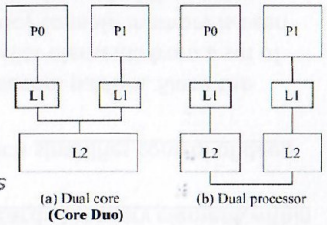


The four threads could be combined to execute on the processor more efficiently utilizing all four issue slots

- A row represents occupancy of the issue stage in a clock cycle
- In coarse MT, a whole cycle might pass with no instruction being issued
- In the fine-grained MT, there are no empty rows if there are enough threads to cover the stalled cycles
- The vacant issue slots in SMT might be due to e.g., structural hazards (all eligible threads requesting the same resources) or all threads waiting for cache misses
- Coarse MT takes 27 cycles, fine MT takes 25 cycles, and SMT takes 14 cycles to complete

Multicores and multi-processors

- In all current general-purpose CMPs, each processor has its own L1 instruction and data caches. The L2 cache can be either private, that is, one per processor, or shared among processors
- Intel has developed CMPs in each of the shown two architectural types
  - In the case of the two processors having private L2 caches, each processor is on a separate die and there are two dies on the chip, which is called *dual processors*



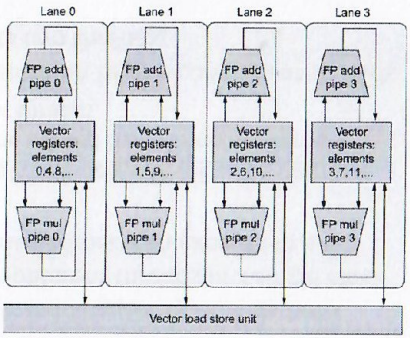
  - When the two (or more) processors share a common L2 and are on the same die, they are called *multicores*. Recent multicores will have private L1 and L2 caches and a shared L3 cache
  - For dual-core devices, the core path to L2 cache is multiplexed between both cores, and the various DMA channels arbitrate for the DMA path into L2 cache
  - Multi-core processors on the same die are smaller, closer, and hence have lower inter-processor latency, while consuming less power than processors on separate dies
  - Note that the quad-core member of this family is in fact two dual cores, each on its own die with two dies on the chip
  - Almost all current CMPs are multicores

Multiprocessor organization

- *Computer organizations* have been classified into four categories based on the number of instructions and data streams
- Single-instruction, single-data (*SISD*) stream
  - *Instruction stream* refers to the sequence of instructions. The sequence of data manipulated by the instruction stream(s) forms the *data stream*
  - A conventional uniprocessor has a single instruction stream and single data stream
  - Several instructions from one or different threads from a single stream are executing concurrently in a SMT superscalar processor
- Single-instruction, multiple-data (*SIMD*) processors operate on an array of data to support *data-level parallelism*
- A single control unit fetches, decodes, and issues instructions that will be executed on an array of *processing elements* (PEs) in parallel over multiple data
  - PEs have local data memories, but no control unit
  - A single instruction stream is broadcast to all PEs. All PEs are synchronized and they all execute the same instruction on their own local data (e.g., elements of an array)
  - PEs are interconnected and exchange/share data as directed by the control unit
- SIMD organization is used for the implementation of multimedia instructions for graphics processing in GGPs

Vector processor architecture

- A vector processor architecture consists of multiple parallel *vector lanes*, each deeply pipelined. A vector lane consists of one or more vector functional units and a portion of the vector register file. Each vector lane performs operations on the element *K* of two vectors
  - Throughput of a vector processor can be increased by adding more lanes. The architecture is *scalable*
- In this architecture, the vector register file is divided across four lanes, with each lane holding every fourth element of each vector
- Each lane is consist of two vector functional units: an FP add and an FP multiply

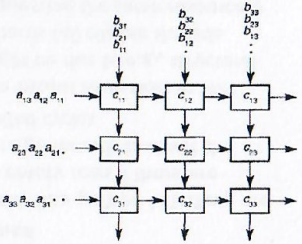


Vector instruction set architecture

- The instruction set of *scalar processors* operate on single data items. A *vector processor* or *array processor* implements an instruction set that operate on one-dimensional arrays of data or vectors. Vector processors exploit data-level parallelism using their SIMD organization
- Vector processors characteristics:
  - An entire loop of instructions is replaced by a single vector instruction. Therefore, the instruction fetch and decode bandwidth is reduced
  - It should only check for data hazards between two vector instructions once per vector operand. Hardware does not have to check for data hazards for every element within the vectors
  - Compiler ensures no dependencies among vectors, which simplifies control of deep pipeline
  - Vector instructions that access memory have a known access pattern. Since the vector's elements are all adjacent, then fetching the vector elements from a set of consecutive locations works well as the cost of the latency to main memory is seen only once for the entire vector, rather than once for each element of the vector

MISD and MIMD organizations

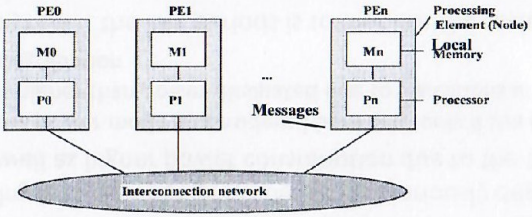
- A multiple-instruction, single-data (*MISD) processor* or a "*stream processor*" would perform a series of computations on a single stream of data in a pipelined fashion
- A *systolic array* is an example of the MISD organization in which an array of processors rhythmically compute and pass data through the architecture
  - The PEs of a systolic array use their own instructions and are synchronized
- Systolic array has a 1-D (linear) or 2-D (mesh) structure of computational units that are synchronized globally



- The performance levels required by many computationally-intensive applications, including military and aerospace, life science, and financial modeling are unattainable with even the fastest SMT processors
- The multiple-instruction, multiple-data (*MIMD*) organization includes a number of processors that function *asynchronously* (don't have to be synchronized with each other) and *independently* (i.e., execute different programs on different processors on different data)
  - MIMD processing systems (e.g., clusters) can be built from commodity microprocessors

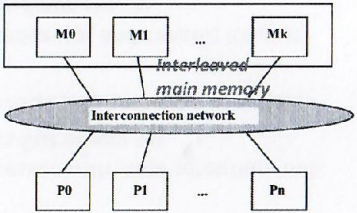
Variations of MIMD organization

- MIMD processors can be classified into two categories : (i) *distributed memory organization*, which used multiple address space and communicate via message passing and (ii) *shared memory organization*, which uses a single shared address space and communicate via load–store instructions
- *Multicomputers* use distributed memory organization by replicating the processor/memory pairs and connecting them via an interconnection network
  - The processor/memory pair is called processing element (PE). PEs work independently of each other. Each processor has its own private address space. All of task's data is stored locally. Multiple tasks can reside on the same computer
  - None of the PEs can ever access directly the memory unit of another PE



Shared memory multiprocessor (SMP) organization

- In the *shared memory multiprocessor architecture*, the set of memory modules defines a single global address space, which is shared among the processors
  - Multicore chips use SMP organization
- Each processor executes different programs working on different data
- All processors can access any memory location (in any memory module) via "load" and "store" instructions



- Processors communicate through *shared variables* in memory. To use shared data, *synchronization primitives*, such as "monitors" and "locks", should be used to insure that no more than one thread is updating the same address at any time
  - Only one processor at a time can acquire the "lock", and other processors interested in shared data must wait until the original processor unlocks the variable

Distributed memory MIMD organization

- Processors communicate by explicitly sending and receiving *message passing primitives*, such as "send" and "receive" and other library of subroutines, which are embedded in source code, to pass physical copy of data structures among processors
  - Intensive data copying can result in significant performance degradation
  - Message-passing based communication and synchronization can lead to *deadlock* situations. A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause. It is the responsibility of the communication protocol designer and application programmer to avoid deadlocks
- In order to achieve high performance in multicomputers, it is the responsibility of the user to determine all parallelism and partition the code and data among the PEs (*load balancing*)

Shared memory vs. distributed memory MIMD organization

- Neither new programming languages nor sophisticated compilers are needed to exploit shared memory multiprocessor organization
- There is no need to partition the data among processors. There is no need to physically move data when two or more processes communicate
- In SMP organization, access of shared data structures lacks scalability due to the contention problem
  - The larger the number of processors, the probability of memory contention is higher. Beyond a certain number of processors, adding a new processor to the system will not increase the performance
- Processors in distributed memory MIMD organization work on their attached local memory module most of the time and hence, the contention problem is not so severe
  - Message passing synchronization is simpler to apply
  - Distributed memory multicomputers are highly scalable and suitable candidates for building massively parallel computing systems

Processors operating modes

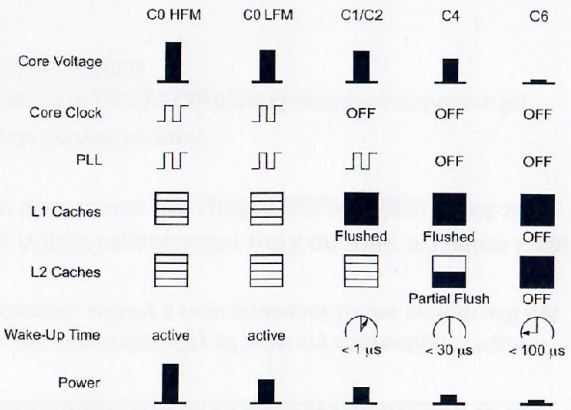
- Most microprocessors have a voltage-dependent maximum operating frequency, so when used at a reduced frequency, the processor can operate at a lower supply voltage
  - Frequency is typically fixed by the application. The *operating system* must be able to vary V_{DD} and f_{CLK} as a function of desired throughput
- Microprocessors can run in different operating modes by altering the clock frequency and voltage
  - *Active mode*: maximum power consumption: Clock frequency at max V_{DD}
  - *Halt* turns off the clocks to the processor
  - *Snooze* turns off the internal power supply to the processor with state retention, but cache memories remain powered up. This mode allows fast power up
  - *Hibernate* turns off the external power supply to the processor, but cache memories remain powered up
  - *Shutdown* turns off the external power supply to the processor and caches; longest wake-up time (time needed to emerge from the power-down mode)

Processors operating modes

- Frequent switching between power modes would seriously degrade the processor performance as well as higher power consumption due to the transition overhead
  - Switching to low-power mode is considered beneficial only if the power saved in low power mode is higher than power dissipated due to transitions at acceptable performance degradation
- One approach to predict the idle periods is to monitor the state of a functional unit (FU)
  - If the FU is idle for several cycles, then it is likely that it will be idle during the next few cycles
- The architectural trade-offs are between:
  - the amount of possible leakage power savings
  - the entry and exit time penalties incurred
  - the energy dissipated entering and leaving leakage saving modes
  - the activity profile (proportion and frequency of times asleep or active)

Intel Atom power management mode

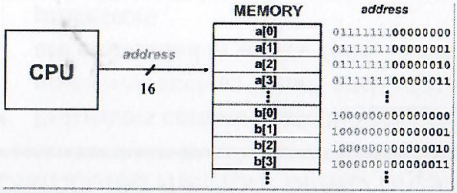
- The Intel Atom processor operates at a peak frequency of 2 GHz at 1 V, consuming 2 W
- The processor has multiple power power management modes
- In the low frequency mode (LFM), the clock drops as slow as 600 MHz while the power supply reduces to 0.75 V



- In sleep mode C1, the core clock is turned off and the L1 cache is flushed and power-gated to reduce leakage, but the processor can return to active state in 1 microsecond
- In sleep mode C4, the PLL is also turned OFF
- In sleep mode C6, the core and caches are all power-gated to reduce power to less than 80 mW, but wake-up time rises to 100 microseconds

Algorithmic-level decisions

- The choice of algorithm is the most highly leveraged decision in meeting the power constraints
- The ability for an algorithm to be *parallelized* will be critical and the basic complexity of the computation must be highly optimized
- A more *energy-efficient* system (*greener*) can be implemented by modifying the application software
  - For example, DSP applications may be running 90% of the time in loops
  - If two loops are executed in sequence with the same indices, they can be merged so that the loop counter (initialization, increment, and comparison) is removed and hence, the number of executed instructions is reduced
  - However, that may increase the number of transitions




```
float a [256], b[256];
float pi= 3.14;

for (i = 0; i < 255; i++) {
   a[i] = sin(pi * i /256);
   b[i] = cos(pi * i /256);
}
```

512(8)+2+4+8+16+32+64+128+256
= 4607 bit transitions

```
float a [256], b[256];
float pi= 3.14;
for (i = 0; i < 255; i++) {a[i] = sin(pi * i /256);}
for (i = 0; i < 255; i++) {b[i] = cos(pi * i /256);}
```

2(8)+2(2+4+8+16+32+64+128+256)
= 1030 transitions

Algorithmic-level decisions

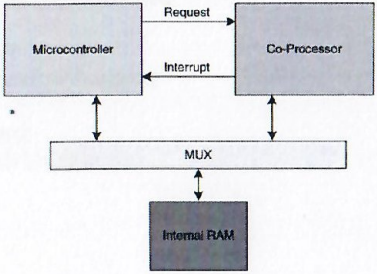
- Selection of an optimum algorithm with respect to the *cost function*
  - The term cost depends on the application. It typically includes the number of operations, memory accesses, and the memory size that is required by this algorithm
  - Minimizing the number of operations to perform a given function is critical to reducing the overall switching activity
- Other considerations may include:
  - Implementation based on certain hardware architecture or processor instructions and registers
  - Selecting least expensive instructions or instruction sequences
  - Minimizing the number of memory accesses needed by algorithm: For example, storing intermediate results in a local register instead of writing to memory or restructuring algorithms such that fewer memory accesses are needed
  - Minimizing total memory size needed by algorithm
  - Reduce number of accesses or cost (power) of an access
    - i.e., Minimize f_{eff} and C_{eff}
- Avoid using of memory operands as much as possible and improve register utilization

System-level decisions – ISA

- Processors targeted towards portable systems should have their ISA designed for energy efficiency, and not just performance
  - For example, simple ISAs typically have simpler datapaths and control units, which reduces the energy consumed per instruction, but there are more instructions. Complex ISA's (CISCs) have higher code density, which reduces the energy consumed fetching instructions and reduces the total number of instructions executed. These trade-offs need to be analyzed when creating an ISA
- Many processors have 32-bit instruction-words and registers. For energy-efficient processors, 16-bit instruction widths have been considered to be more appropriate
  - Code density can be reduced, while increasing the run length over an equivalent 32-bit processor
  - Reduces the energy cost of an instruction fetch because the size of the external memory read has been halved
- Since most microprocessors today contain a relatively large on-chip cache, utilizing 16-bit instructions will typically have negligible impact on microprocessor energy efficiency

System-level decisions - Choice of processor

- The choice of processor can have a significant impact on overall power consumption
  - As an example, adopting a DSP processor with built-in multiply accumulate units (MAC) is more power-efficient for DSP applications
- The first point is to adapt the data width of the processor to the required data
  - Managing for instance 16-bit data on an 8-bit microcontroller increases the number of instructions
  - For a 16-bit multiply, 30 instructions are required (add-shift algorithm) on a 16-bit processor, while 127 instructions are required on an 8-bit machine (double precision)
- A more energy efficient processor architecture has a 16-bit multiplier with an instruction to execute a multiplication
- *Design partitioning*: For many applications, there is some control that is performed by a microcontroller while data processing tasks can be carried over by a *dedicated co-processor* or a DSP for energy-efficient processing
- Selecting most efficient algorithm that maps well to available hardware



System-level decisions – Register file and cache sizes

- The register file consumes a considerable fraction of total energy consumption. A tradeoff exists between access time and energy consumption
- Smaller memories have lower energy consumption. Designers may try to minimize the on-chip cache size at the expense of a decrease in throughput (due to increased miss rates of the cache). The increased miss rates affect the performance and may increase the system energy consumption because high-energy main memory accesses are now made more frequently. Although the processor's energy consumption was decreased, the total system's energy consumption has increased
- Increasing the on-chip cache size will always decrease the cache miss-rate, thereby decreasing the number of external accesses and reducing the average memory access time. Thus, on-chip cache size is typically maximized, given die-size constraints, for performance considerations
  - Maximizing the cache size will not only improve processor performance by reducing the average memory access time, but it will also provide the lowest overall energy/access
- However, maximizing cache size will increase the capacitance/access to the cache. By sub-blocking the cache, the actual memory array size being accessed will remain constant, independent of cache size

Heterogeneous architectures

- *Heterogeneous architectures*, combining regular cores, specialized accelerators, and large amounts of memory, are of growing importance
- *Memories* have a much lower power density than logic because their activity factors are small and their regularity simplifies leakage control
  - If a task can be accelerated using either a faster processor or a larger memory, the memory is often preferable
  - Memories now comprise more than half the area of many chips
- *Special-purpose functional units* can offer an order of magnitude better energy efficiency than general-purpose processors
  - *Accelerators* for compute-intensive applications such as graphics, networking, and cryptography offload these tasks from the processor
- It is shown that roughly one-third of microprocessor power is spent on the clock, another third on memories, and the remaining third on logic and wires
  - In nanometer technologies, nearly one-third of the power is leakage

Low-power system-level design on FPGAs

- A low-power alternative to SRAM-based FPGAs is flash-based FPGAs
  - Flash-based FPGAs, such as Actel's IGLOO devices, are inherently more efficient because flash-based memory dissipates significantly less leakage power compared to SRAM memory
- *Inrush current* or *start-up current* refers to the maximum, instantaneous input current drawn by the device when first turned on
- Inrush current is device-specific
  - SRAM-based FPGAs features a high inrush current because on power-up these devices are not configured and need to actively download data from external memory chips to configure their programmable resources
  - Anti-fuse-based FPGAs do not have a high inrush current since they do not require power-on configuration

Low-power system-level design on FPGAs

- It is usually preferable to use coarse-grained embedded blocks (e.g., embedded memories and multipliers) than the fine-grained configurable logic blocks since the former are more power-efficient than the latter for the same function
  - Power consumption for routing to and from dedicated units should be considered
- The size of the LUTs within the logic blocks has been increased from 4-input LUTs to 6 and 7-input LUTs
  - Dynamic power is reduced since more logic is implemented within each LUT and less routing is needed between the LUTs
- Both Xilinx and Altera have modified their routing architectures to increase the number of neighboring logic blocks that can be reached in only one or two hops (each routing segment used counts as one hop)
  - Using more one-hop routes reduces the average capacitance of the routes, which improves both power and performance
- CAD tools also ensure that all unused logic blocks, embedded blocks, routing resources, and clock network resources are turned off to save power

Chapter 04 – Zynq Architecture and Bus Interfaces