CSC 230 - Summer 2018

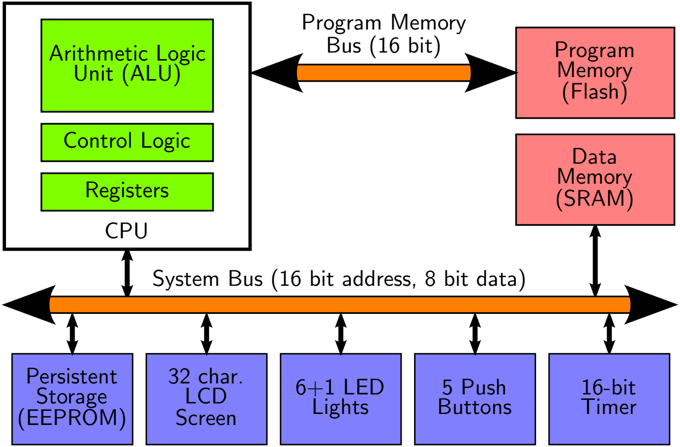
Architecture II

Bill Bird

Department of Computer Science
University of Victoria

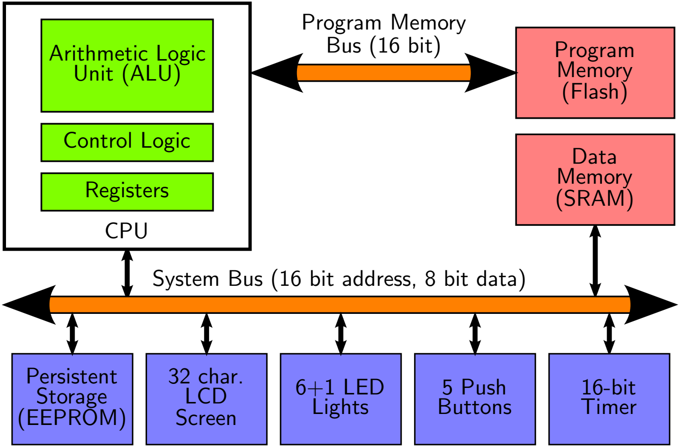
July 19, 2018

The AVR Architecture Revisited (1)



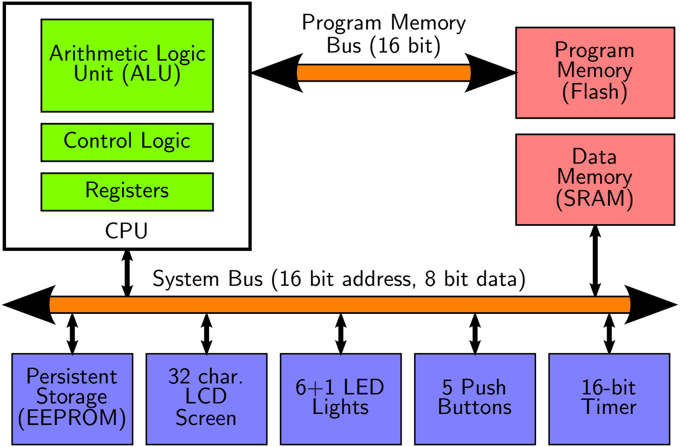
We have now spent enough time with the AVR architecture that most of its internal characteristics are clear (for better or for worse).

The AVR Architecture Revisited (2)



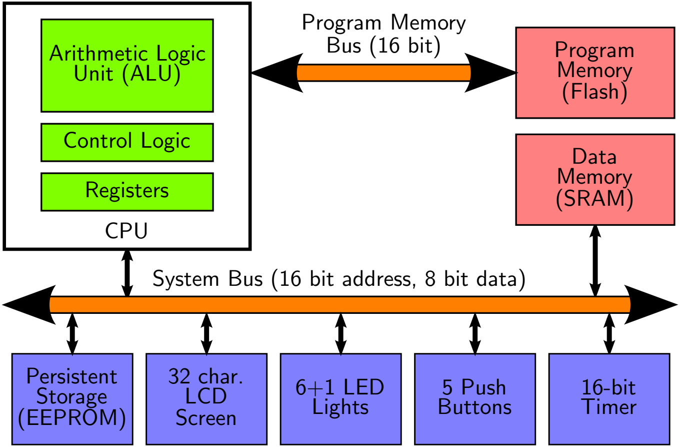
The AVR instruction set consists of a relatively small (about 100) set of instructions, divided between arithmetic and logic (e.g. `ADD` and `LSR`), memory access (e.g. `LDS` and `ST`) and control flow (e.g. `CALL`).

The AVR Architecture Revisited (3)



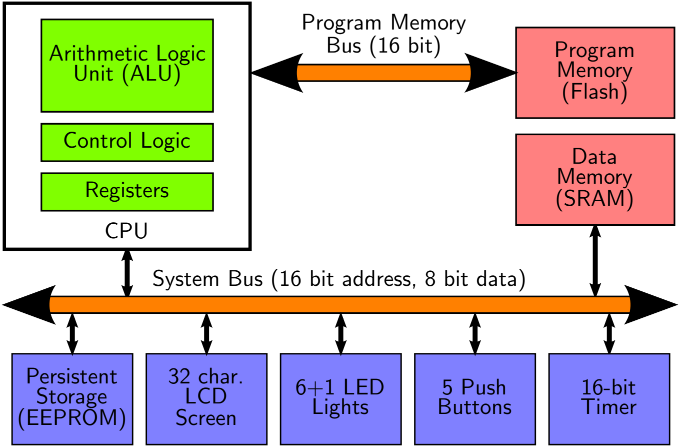
AVR is an example of a **RISC** (reduced instruction set computer) architecture. The term ‘RISC’ can be misleading, since the size of the instruction set is a secondary consideration in classifying an architecture.

The AVR Architecture Revisited (4)



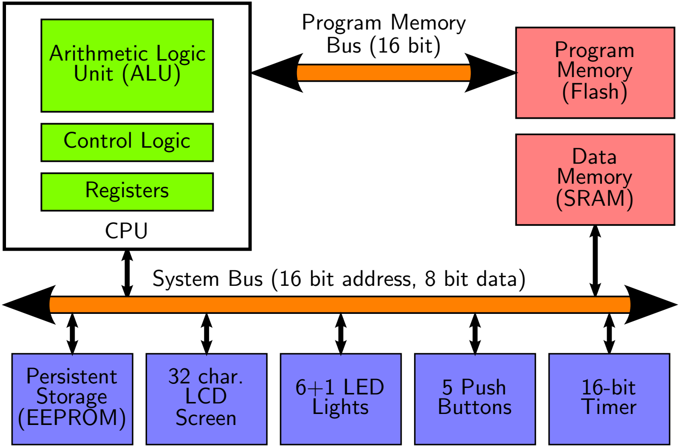
RISC architectures tend to have a large number of registers and confine all arithmetic and logic operations to operate exclusively on register operands (instead of values loaded from memory).

The AVR Architecture Revisited (5)



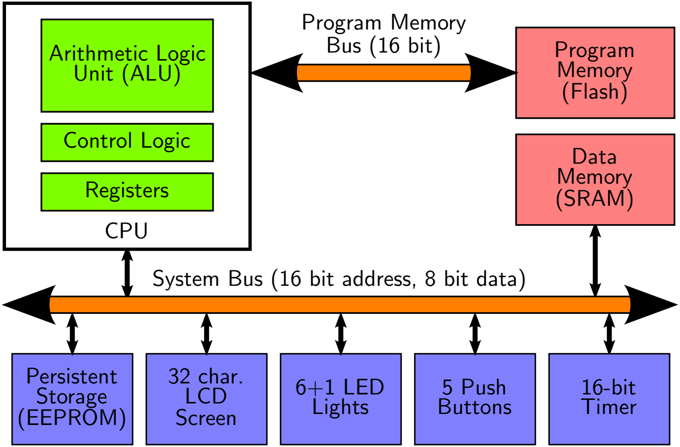
Accordingly, all memory access operations are confined to a special set of load and store instructions.

The AVR Architecture Revisited (6)



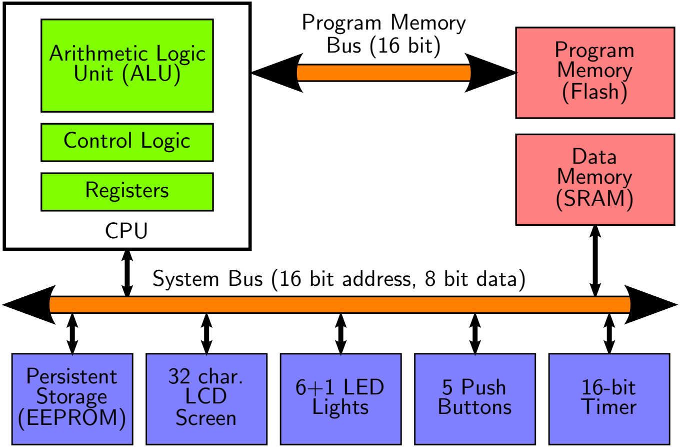
This characteristic is so fundamental to the idea of a RISC architecture that the term ‘load/store architecture’ has been proposed as an alternative name.

The AVR Architecture Revisited (7)



Non-RISC architectures are usually pigeonholed as CISC (complex instruction set computer) architectures.

The AVR Architecture Revisited (8)



One defining characteristic of CISC architectures is a large and specialized instruction set with the ability to combine operands from various sources (registers, directly accessed memory, indirectly accessed memory) in a single instruction.

Addressing Modes (1)

```
; Immediate operand (hard coded constant)
LDI r16, 187

; Direct addressing (hard coded address)
LDS r16, MY_VARIABLE
LDS r16, MY_ARRAY + 5

; Indirect addressing (pointer stored in register)
LD r16, Y
ST Z+, r6

; Indirect addressing with constant displacement
LDD r16, Z+5
```

The **addressing mode** of a load or store instruction determines how memory is accessed by the instruction.

Addressing Modes (2)

```
; Immediate operand (hard coded constant)
LDI r16, 187

; Direct addressing (hard coded address)
LDS r16, MY_VARIABLE
LDS r16, MY_ARRAY + 5

; Indirect addressing (pointer stored in register)
LD r16, Y
ST Z+, r6

; Indirect addressing with constant displacement
LDD r16, Z+5
```

Although there are several different encodings of each mode, the AVR architecture really only supports three addressing modes (options like post-incrementation do not affect how memory is accessed).

Addressing Modes (3)

```
; Immediate operand (hard coded constant)
LDI r16, 187

; Direct addressing (hard coded address)
LDS r16, MY_VARIABLE
LDS r16, MY_ARRAY + 5

; Indirect addressing (pointer stored in register)
LD r16, Y
ST Z+, r6

; Indirect addressing with constant displacement
LDD r16, Z+5
```

The main difference between the modes is the distinction between **direct access** (using a constant, hard coded address) and **indirect access** (using an address stored in registers).

Addressing Modes (4)

Mode	**Result**
Immediate	C
Direct	$[A]$
Indirect	$[X]$
Indirect with Displacement	$[X + C]$

C = Constant Operand, A = Constant Address

X = Register Pair (e.g. X, Y or Z)

The addressing modes supported by the AVR architecture are summarized with pseudocode in the table above. The notation $[A]$ is used to mean 'the contents of memory at address A'.

Addressing Modes (5)

Mode	**Result**
Immediate	C
Direct	$[A]$
Indirect	$[Rd]$
Indirect (Constant Displacement)	$[Rd + C]$
Indirect (Register Displacement)	$[Rd + Rr]$
Indirect (Scaled Register Displacement)	$[Rd + (Rr << C)]$

C = Constant Operand, A = Constant Address

Rd, Rr = Register (R0 - R16)

The table above shows the addressing modes available in the ARM architecture (which is also RISC). Note that ARM registers are 32 bits wide, so pointers can be stored in single registers.

Addressing Modes (6)

Mode	**Result**
Immediate	C
Direct	$[SR + A]$
Indirect	$[SR + Rb]$
Indirect (Constant Displacement)	$[SR + Rb + C]$
Indirect (Scaled Index Displacement)	$[SR + S * Ri + C]$
Indirect (Base/Index Displacement)	$[SR + Rb + Ri + C]$
Indirect (Scaled Base/Index Displacement)	$[SR + Rb + S * Ri + C]$

C = Constant Operand, A = Constant Address, S = Scale Factor

SR = Segment Register, Rb = Base Register, Ri = Index Register

The table above shows (most of) the addressing modes available in the x86 architecture (a CISC architecture).

Addressing Modes (7)

Mode	**Result**
Immediate	C
Direct	$[SR + A]$
Indirect	$[SR + Rb]$
Indirect (Constant Displacement)	$[SR + Rb + C]$
Indirect (Scaled Index Displacement)	$[SR + S * Ri + C]$
Indirect (Base/Index Displacement)	$[SR + Rb + Ri + C]$
Indirect (Scaled Base/Index Displacement)	$[SR + Rb + S * Ri + C]$

C = Constant Operand, A = Constant Address, S = Scale Factor

SR = Segment Register, Rb = Base Register, Ri = Index Register

Note that this architecture takes all addresses relative to the value of a **segment register** to allow programs to be moved between parts of memory easily. The issue of relocating programs in memory will be discussed in a future lecture.

Addressing Modes (8)

Mode	**Result**
Immediate	C
Direct	$[SR + A]$
Indirect	$[SR + Rb]$
Indirect (Constant Displacement)	$[SR + Rb + C]$
Indirect (Scaled Index Displacement)	$[SR + S * Ri + C]$
Indirect (Base/Index Displacement)	$[SR + Rb + Ri + C]$
Indirect (Scaled Base/Index Displacement)	$[SR + Rb + S * Ri + C]$

C = Constant Operand, A = Constant Address, S = Scale Factor

SR = Segment Register, Rb = Base Register, Ri = Index Register

Question: What is the minimum set of addressing modes needed for a functional general purpose stored program architecture?

RISC vs. CISC (1)

RISC architectures tend to have the following characteristics.

- A large number (16 or more) of general purpose registers.
- ‘Simple’ instructions which require one or two clock cycles.
- A relatively uniform instruction size (e.g. 32 bits).
- Memory accesses permitted only by load and store instructions.
- A relatively small number of addressing modes (e.g. direct and indirect only) for load and store instructions.

We have been taking several of these characteristics (particularly the small number of clock cycles per instruction) for granted with the AVR architecture.

RISC vs. CISC (2)

CISC architectures tend to have the following characteristics.

- Fewer general purpose registers (since memory operands can be used directly).
- A large and heterogeneous instruction set, with various instruction sizes and timings.
- A wide variety of addressing modes and the ability of any type of instruction (not just load/store instructions) to access memory.

RISC vs. CISC (3)

The actual distinction between RISC and CISC architectures, despite the difference in names (reduced vs. complex instruction set) is usually considered to be the separation of memory (load/store) instructions from other instructions (such as arithmetic and logic, which use registers for all operands).

In fact, there are RISC architectures which have larger instruction sets than some CISC architectures. Furthermore, the existence of specific or esoteric instructions (such as the `DES` instruction in AVR, which is used to perform a particular form of encryption) does not imply a CISC architecture.

Since most architectures are designed to be practical, no architecture will be a 'pure' CISC or RISC architecture. In general, however, the load/store separation is maintained fairly strictly on RISC architectures.

RISC vs. CISC (4)

We will see that the main historical motivation for the development of RISC architectures hinged on two developments: **fast memory access time** and **pipelining**

Instruction Set Philosophy (1)

```
; Load operands
lds r16, A
lds r17, B
; Compute result
lsl r16
lsl r16
sub r16, r17
; Store result
sts C, r16
```

Consider the AVR assembly code above, which takes two values A and B in data memory and stores the value $A * 4 - B$ into a value C.

Instruction Set Philosophy (2)

```
; Load operands
lds r16, A
lds r17, B
; Compute result
lsl r16
lsl r16
sub r16, r17
; Store result
sts C, r16
```

On an ATmega2560, the entire sequence of instructions above takes 9 clock cycles. Note that `LDS` and `STS` require 2 clock cycles each.

Instruction Set Philosophy (3)

```
; Load operands
lds r16, A
lds r17, B
; Compute result
lsl r16
lsl r16
sub r16, r17
; Store result
sts C, r16
```

Memory accesses requiring only two clock cycles is a relative luxury, and most machines do not enjoy such a luxury. If the clock speed were higher, faster memory would be needed to maintain the two cycle timing. If more memory were needed (on the order of gigabytes), the cost of fast memory would become exorbitant.

Instruction Set Philosophy (4)

```
; Load operands
lds r16, A
lds r17, B
; Compute result
lsl r16
lsl r16
sub r16, r17
; Store result
sts C, r16
```

Even with the two cycle timing, a contrast can be demonstrated between RISC and CISC architectures.

Instruction Set Philosophy (5)

```
; Load operands
lds r16, A
lds r17, B
; Compute result
lsl r16
lsl r16
sub r16, r17
; Store result
sts C, r16
```

Assumption: No instruction can be executed until the previous instruction has **completely** finished. (This is the standard procedure in the AVR architecture).

Instruction Set Philosophy (6)

Time	Operation
0	`lds r16, A`
1	
2	`lds r17, B`
3	
4	`lsl r16`
5	`lsl r16`
6	`sub r16, r17`
7	`sts C, r16`
8	

The timing of the $4A - B$ operation in AVR is represented above.

Instruction Set Philosophy (7)

Time	Operation
0	`lds r16, A`
1	
2	`lsl r16`
3	`lsl r16`
4	`lds r17, B`
5	
6	`sub r16, r17`
7	`sts C, r16`
8	

Notice that because of the assumption that instructions cannot overlap, rearranging the instructions does not improve the running time.

Instruction Set Philosophy (8)

Time	Memory	ALU
0	lds r16, A	
1		
2		lsl r16
3		lsl r16
4	lds r17, B	
5		
6		sub r16, r17
7	sts C, r16	
8		

The internal circuitry for the memory controller is independent from the ALU, so the two components can be considered as separate modules inside the CPU.

Instruction Set Philosophy (9)

Time	Memory	ALU
0	`lds r16, A`	
1		
2		`lsl r16`
3		`lsl r16`
4	`lds r17, B`	
5		
6		`sub r16, r17`
7	`sts C, r16`	
8		

Suppose that a new instruction `FOURAMINUSB` were created, with three memory addresses as operands (so '`FOURAMINUSB A, B, C`' would produce the same result as the code above).

Instruction Set Philosophy (10)

Time	Memory	ALU
0	Load A	
1		
2		$t_1 = 2 * A$
3		$t_2 = 2 * t_1$
4	Load B	
5		
6		$t_3 = t_2 - B$
7	Store t_3 to C	
8		

The FOURAMINUSB instruction be designed as a mirror image of the series of AVR instructions on the previous slides.

Instruction Set Philosophy (11)

Time	Memory	ALU
0	`Load A`	
1		
2	`Load B`	$t_1 = 2 * A$
3		$t_2 = 2 * t_1$
4		$t_3 = t_2 - B$
5	`Store` t_3 `to C`	
6		

However, since `FOURAMINUSB` is a single instruction, the work performed can be distributed evenly across all CPU units, requiring only 7 clock cycles.

Instruction Set Philosophy (12)

Time	**Memory**	**ALU**
0	`Load A`	
1		
2	`Load B`	$t_1 = 2 * A$
3		$t_2 = 2 * t_1$
4		$t_3 = t_2 - B$
5	`Store` t_3 `to C`	
6		

In a CISC architecture, instead of a collection of ‘atomic’ instructions which perform one simple action in a small number of cycles, there may be numerous ‘compound’ instructions which batch together atomic operations (particularly memory accesses and ALU usage).

Instruction Set Philosophy (13)

Time	Memory	ALU
0	Load A	
1		
2	Load B	$t_1 = 2 * A$
3		$t_2 = 2 * t_1$
4		$t_3 = t_2 - B$
5	Store t_3 to C	
6		

From a design standpoint, the CISC philosophy leads to three problems. First, increasingly complex circuitry is needed to manage the various permutations of operations inside the CPU (since each instruction could require an arbitrary sequence of operations).

Instruction Set Philosophy (14)

Time	Memory	ALU
0	Load A	
1		
2	Load B	$t_1 = 2 * A$
3		$t_2 = 2 * t_1$
4		$t_3 = t_2 - B$
5	Store t_3 to C	
6		

Second, the cycle savings which results from a compound instruction may not be sufficient to justify the added complexity or cost of implementing that instruction in hardware, especially if the instruction is not widely used.

Instruction Set Philosophy (15)

Time	Memory	ALU
0	`Load A`	
1		
2	`Load B`	$t_1 = 2 * A$
3		$t_2 = 2 * t_1$
4		$t_3 = t_2 - B$
5	`Store` t_3 `to C`	
6		

(Consider how frequently the value `A*4 - B` is actually needed in a program)

Instruction Set Philosophy (16)

Time	Memory	ALU
0	`Load A`	
1		
2	`Load B`	$t_1 = 2 * A$
3		$t_2 = 2 * t_1$
4		$t_3 = t_2 - B$
5	`Store` t_3 `to C`	
6		

In many CISC architectures, the instruction set sprawl has become so severe that many instructions are not actually implemented in hardware. Instead, they are broken down into simple instructions by the processor's instruction decoder.

Instruction Set Philosophy (17)

Time	Memory	ALU
0	`Load A`	
1		
2	`Load B`	$t_1 = 2 * A$ $t_2 = 2 * t_1$
3		
4		$t_3 = t_2 - B$
5	`Store` t_3 `to C`	
6		

Third, and most importantly, a huge and heterogeneous instruction set is not easy to learn, so human programmers may not actually use many of the time-saving instructions. Compilers, which are more mechanical, are even less likely to use most of the compound operations.

Instruction Set Philosophy (18)

Time	**Memory**	**ALU**
0	`Load A`	
1		
2	`Load B`	$t_1 = 2 * A$
3		$t_2 = 2 * t_1$
4		$t_3 = t_2 - B$
5	`Store` t_3 `to C`	
6		

By constrast, RISC architectures tend to have a set of simple, fundamental operations, with little overlap in functionality between instruction types.

Instruction Set Philosophy (19)

Time	Memory	ALU
0	`Load A`	
1		
2	`Load B`	$t_1 = 2 * A$
3		$t_2 = 2 * t_1$
4		$t_3 = t_2 - B$
5	`Store` t_3 `to C`	
6		

As a result, most instructions can be decoded easily and can be modelled by a simple sequence of steps.

Instruction Set Philosophy (20)

Time	Memory	ALU
0	`Load A`	
1		
2	`Load B`	$t_1 = 2 * A$
3		$t_2 = 2 * t_1$
4		$t_3 = t_2 - B$
5	`Store` t_3 `to C`	
6		

Additionally, the advantage shown in the example above can be achieved by using a more general approach called **pipelining**, which can be applied to any instruction, not just particular compound operations.

Execution Rules (1)

The AVR architecture executes instructions such that the following rules apply.

1. Each instruction is fetched from program memory. It is not possible to execute instructions from data memory (or any other source, such as the contents of a register).
2. Instructions must be executed **in order**. Even in cases where two instructions are independent, the first instruction in the sequence must be executed first.
3. In a sequence of two instructions A and B, the execution of instruction B will not begin until the execution of A is **entirely** finished.

Execution Rules (2)

The AVR architecture executes instructions such that the following rules apply.

1. Each instruction is fetched from program memory. It is not possible to execute instructions from data memory (or any other source, such as the contents of a register).
2. Instructions must be executed **in order**. Even in cases where two instructions are independent, the first instruction in the sequence must be executed first.
3. In a sequence of two instructions A and B, the execution of instruction B will not begin until the execution of A is **entirely** finished.

The first rule is a consequence of AVR being a Harvard architecture.

Execution Rules (3)

The AVR architecture executes instructions such that the following rules apply.

1. Each instruction is fetched from program memory. It is not possible to execute instructions from data memory (or any other source, such as the contents of a register).
2. Instructions must be executed **in order**. Even in cases where two instructions are independent, the first instruction in the sequence must be executed first.
3. In a sequence of two instructions A and B, the execution of instruction B will not begin until the execution of A is **entirely** finished.

The second rule is a fairly natural invariant to enforce (after all, the sequence of instructions is what dictates the behavior of the program, so instructions should not be rearranged).

Execution Rules (4)

The AVR architecture executes instructions such that the following rules apply.

1. Each instruction is fetched from program memory. It is not possible to execute instructions from data memory (or any other source, such as the contents of a register).
2. Instructions must be executed **in order**. Even in cases where two instructions are independent, the first instruction in the sequence must be executed first.
3. In a sequence of two instructions A and B, the execution of instruction B will not begin until the execution of A is **entirely** finished.

Some architectures offer **superscalar** execution, which allows a certain amount of out-of-order execution.

Execution Rules (5)

The AVR architecture executes instructions such that the following rules apply.

1. Each instruction is fetched from program memory. It is not possible to execute instructions from data memory (or any other source, such as the contents of a register).
2. Instructions must be executed **in order**. Even in cases where two instructions are independent, the first instruction in the sequence must be executed first.
3. In a sequence of two instructions A and B, the execution of instruction B will not begin until the execution of A is **entirely** finished.

The third rule simplifies the execution model, but is not as natural: Why not start executing instruction B as instruction A is finishing up?

Execution Rules (6)

The AVR architecture executes instructions such that the following rules apply.

1. Each instruction is fetched from program memory. It is not possible to execute instructions from data memory (or any other source, such as the contents of a register).
2. Instructions must be executed **in order**. Even in cases where two instructions are independent, the first instruction in the sequence must be executed first.
3. In a sequence of two instructions A and B, the execution of instruction B will not begin until the execution of A is **entirely** finished.

Suppose that rule 3 is replaced by the following: In a sequence of two instructions A and B, B will not start execution until after A has started and will not finish execution until A has finished.

Execution Rules (7)

The AVR architecture executes instructions such that the following rules apply.

1. Each instruction is fetched from program memory. It is not possible to execute instructions from data memory (or any other source, such as the contents of a register).
2. Instructions must be executed **in order**. Even in cases where two instructions are independent, the first instruction in the sequence must be executed first.
3. In a sequence of two instructions A and B, the execution of instruction B will not begin until the execution of A is **entirely** finished.

This modified rule allows some flexibility in the execution order. For example, if instructions A and B use different parts of the CPU, parts of each instruction can be executed simultaneously.

Pipelining (1)

Consider a RISC processor (not necessarily AVR) and suppose that the execution of each instruction can be divided into the following seven discrete stage.

1. **Fetch** (F): Retrieve the instruction from memory.
2. **Decode** (D): Determine the operation to perform.
3. **Read Operands** (R): Load operand values (from registers or immediate values).
4. **Load** (ML): Load a value from memory.
5. **ALU** (A): Perform an arithmetic or logic operation.
6. **Write Result** (W): Save result value in a register.
7. **Store** (MS): Store a value to memory.

Note that a particular instruction may do nothing in a particular stage (for example, `ADD` does not need to load or store memory values and a load instruction does not use the ALU).

Pipelining (2)

To be clear, the R and W stages refer exclusively to reading and writing register values (not memory), while the ML and MS stages refer exclusively to memory reads and writes.

An instruction which operates entirely on registers will use some combination of the R and W stages. For example, the `ADD` instruction will use the R stage (to read operands), the A stage (to add them) and the W stage (to store the result).

Assume that branch instructions take an immediate address (which requires the R stage to read) as an operand and use the W stage to write to the value of PC.

Pipelining (3)

```
(I1) LOAD    r0, VAR1
(I2) SET     r2, 0x00
(I3) LOAD    r1, VAR2
(I4) ADD     r0, 0x05
(I5) SUB     r1, 0x06
(I6) CMP     r0, r1
(I7) BRLO    L1
(I8) STORE   VAR1, r0
```

I_1								
I_2								
I_3								
I_4								
I_5								
I_6								
I_7								
I_8								
Time	0	1	2	3	4	5	6	7

F = Fetch	D = Decode	R = Read Operand	ML = Mem. Load
A = ALU	W = Write Result	MS = Mem. Store	

Consider the pseudo-assembly code above, which could correspond to a RISC architecture like AVR or ARM. Assume that the label L1 is valid and exists outside of the code shown.

Pipelining (4)

```
(I1) LOAD    r0, VAR1
(I2) SET     r2, 0x00
(I3) LOAD    r1, VAR2
(I4) ADD     r0, 0x05
(I5) SUB     r1, 0x06
(I6) CMP     r0, r1
(I7) BRLO    L1
(I8) STORE   VAR1, r0
```

Time	0	1	2	3	4	5	6	7
I_1								
I_2								
I_3								
I_4								
I_5								
I_6								
I_7								
I_8								

F = Fetch	D = Decode	R = Read Operand	ML = Mem. Load
A = ALU	W = Write Result	MS = Mem. Store	

Suppose that the ML and MS stages each take two units of time, and that every other stage requires one unit of time.

Pipeline (5)

(I_1)	`LOAD`	`r0, VAR1`
(I_2)	`SET`	`r2, 0x00`
(I_3)	`LOAD`	`r1, VAR2`
(I_4)	`ADD`	`r0, 0x05`
(I_5)	`SUB`	`r1, 0x06`
(I_6)	`CMP`	`r0, r1`
(I_7)	`BRLO`	`L1`
(I_8)	`STORE`	`VAR1, r0`

I_1	F							
I_2								
I_3								
I_4								
I_5								
I_6								
I_7								
I_8								
Time	0	1	2	3	4	5	6	7

F = Fetch
A = ALU
D = Decode
W = Write Result
R = Read Operand
MS = Mem. Store
ML = Mem. Load

At time 0, instruction I_1 is fetched.

Pipelining (6)

```
(I1) LOAD    r0, VAR1
(I2) SET     r2, 0x00
(I3) LOAD    r1, VAR2
(I4) ADD     r0, 0x05
(I5) SUB     r1, 0x06
(I6) CMP     r0, r1
(I7) BRLO    L1
(I8) STORE   VAR1, r0
```

I_1	F	D						
I_2		F						
I_3								
I_4								
I_5								
I_6								
I_7								
I_8								
Time	0	1	2	3	4	5	6	7

F = Fetch
D = Decode
R = Read Operand
ML = Mem. Load
A = ALU
W = Write Result
MS = Mem. Store

At time 1, instruction I_1 is decoded. Simultaneously, instruction I_2 is fetched.

Pipeline (7)

```
(I1) LOAD   r0, VAR1
(I2) SET    r2, 0x00
(I3) LOAD   r1, VAR2
(I4) ADD    r0, 0x05
(I5) SUB    r1, 0x06
(I6) CMP    r0, r1
(I7) BRLO   L1
(I8) STORE  VAR1, r0
```

I_1	F	D	ML					
I_2		F	D					
I_3			F					
I_4								
I_5								
I_6								
I_7								
I_8								
Time	0	1	2	3	4	5	6	7

F = Fetch　D = Decode　R = Read Operand　ML = Mem. Load
A = ALU　W = Write Result　MS = Mem. Store

Instruction I_1 does not need the R (read operand) stage, since it reads a result from memory. At time 2, I_1 enters the ML stage.

Pipeline (8)

(I_1)	LOAD	r0, VAR1
(I_2)	SET	r2, 0x00
(I_3)	LOAD	r1, VAR2
(I_4)	ADD	r0, 0x05
(I_5)	SUB	r1, 0x06
(I_6)	CMP	r0, r1
(I_7)	BRLO	L1
(I_8)	STORE	VAR1, r0

I_1	F	D	ML	ML				
I_2		F	D	R				
I_3			F	D				
I_4				F				
I_5								
I_6								
I_7								
I_8								
Time	0	1	2	3	4	5	6	7

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

At time 3, instruction I_1 remains in the ML stage. Instruction I_2 has been decoded and enters the R stage.

Pipeline (9)

(I_1)	LOAD	r0, VAR1
(I_2)	SET	r2, 0x00
(I_3)	LOAD	r1, VAR2
(I_4)	ADD	r0, 0x05
(I_5)	SUB	r1, 0x06
(I_6)	CMP	r0, r1
(I_7)	BRLO	L1
(I_8)	STORE	VAR1, r0

I_1	F	D	ML	ML	W			
I_2		F	D	R				
I_3			F	D	ML			
I_4				F	D			
I_5					F			
I_6								
I_7								
I_8								
Time	0	1	2	3	4	5	6	7

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

At time 4, instruction I_1 enters the W stage (to write the loaded value into a register). Instruction I_2 is also ready for the W stage, but is forced to wait. Instruction I_3 enters the ML stage.

Pipelining (10)

```
(I1) LOAD    r0, VAR1
(I2) SET     r2, 0x00
(I3) LOAD    r1, VAR2
(I4) ADD     r0, 0x05
(I5) SUB     r1, 0x06
(I6) CMP     r0, r1
(I7) BRLO    L1
(I8) STORE   VAR1, r0
```

I_1	F	D	ML	ML	W			
I_2		F	D	R				
I_3			F	D	ML			
I_4				F	D			
I_5					F			
I_6								
I_7								
I_8								
Time	0	1	2	3	4	5	6	7

F = Fetch
A = ALU
D = Decode
W = Write Result
R = Read Operand
MS = Mem. Store
ML = Mem. Load

The delay faced by I_2 is an example of a **pipeline hazard**. Specifically, this is a **structural hazard**, caused when two instructions need the same resource (the W stage) simultaneously, forcing one instruction to wait.

Pipelining (11)

(I_1)	LOAD	r0, VAR1
(I_2)	SET	r2, 0x00
(I_3)	LOAD	r1, VAR2
(I_4)	ADD	r0, 0x05
(I_5)	SUB	r1, 0x06
(I_6)	CMP	r0, r1
(I_7)	BRLO	L1
(I_8)	STORE	VAR1, r0

I_1	F	D	ML	ML	W	🟩		
I_2		F	D	R	🟥	W		
I_3			F	D	ML	ML		
I_4				F	D	R		
I_5					F	D		
I_6						F		
I_7								
I_8								
Time	0	1	2	3	4	5	6	7

F = Fetch	D = Decode	R = Read Operand	ML = Mem. Load
A = ALU	W = Write Result	MS = Mem. Store	

At time 5, I_1 is finished, I_2 moves into the W stage and I_3 stays in the L stage. Instruction I_4 enters the R stage.

Pipelining (12)

(I_1)	LOAD	r0, VAR1
(I_2)	SET	r2, 0x00
(I_3)	LOAD	r1, VAR2
(I_4)	ADD	r0, 0x05
(I_5)	SUB	r1, 0x06
(I_6)	CMP	r0, r1
(I_7)	BRLO	L1
(I_8)	STORE	VAR1, r0

Time	0	1	2	3	4	5	6	7
I_1	F	D	ML	ML	W	🟩	🟩	
I_2		F	D	R	🟥	W	🟩	
I_3			F	D	ML	ML	W	
I_4				F	D	R	A	
I_5					F	D	🟥	
I_6						F	🟥	
I_7							🟥	
I_8								

F = Fetch
A = ALU
D = Decode
W = Write Result
R = Read Operand
MS = Mem. Store
ML = Mem. Load

At time 6, I_2 is finished, I_3 moves into the W stage and I_4 enters in the A stage. Instruction I_5 is ready for the R stage, but cannot enter it because the operand in question (r1) is not ready yet.

Pipelining (13)

```
(I1) LOAD   r0, VAR1
(I2) SET    r2, 0x00
(I3) LOAD   r1, VAR2
(I4) ADD    r0, 0x05
(I5) SUB    r1, 0x06
(I6) CMP    r0, r1
(I7) BRLO   L1
(I8) STORE  VAR1, r0
```

I_1	F	D	ML	ML	W	🟩	🟩	
I_2		F	D	R	🟥	W	🟩	
I_3			F	D	ML	ML	W	
I_4				F	D	R	A	
I_5					F	D	🟥	
I_6						F	🟥	
I_7							🟥	
I_8								
Time	0	1	2	3	4	5	6	7

F = Fetch
D = Decode
R = Read Operand
ML = Mem. Load
A = ALU
W = Write Result
MS = Mem. Store

Specifically, another instruction already in the pipeline (I_3) has `r1` as its output, so until I_1 completes the W stage, I_5 cannot enter the R stage.

Pipelining (14)

```
(I1) LOAD    r0, VAR1
(I2) SET     r2, 0x00
(I3) LOAD    r1, VAR2
(I4) ADD     r0, 0x05
(I5) SUB     r1, 0x06
(I6) CMP     r0, r1
(I7) BRLO    L1
(I8) STORE   VAR1, r0
```

I_1	F	D	ML	ML	W	🟩	🟩	
I_2		F	D	R	🟥	W	🟩	
I_3			F	D	ML	ML	W	
I_4				F	D	R	A	
I_5					F	D	🟥	
I_6						F	🟥	
I_7							🟥	
I_8								
Time	0	1	2	3	4	5	6	7

F = Fetch D = Decode R = Read Operand ML = Mem. Load
A = ALU W = Write Result MS = Mem. Store

This is another form of pipeline hazard, a **data hazard**, which occurs when the output of one instruction is the input for another, future instruction, forcing the second instruction to wait.

Pipeline (15)

```
(I1) LOAD    r0, VAR1
(I2) SET     r2, 0x00
(I3) LOAD    r1, VAR2
(I4) ADD     r0, 0x05
(I5) SUB     r1, 0x06
(I6) CMP     r0, r1
(I7) BRLO    L1
(I8) STORE   VAR1, r0
```

I_1	F	D	ML	ML	W	🟩	🟩	
I_2		F	D	R	🟥	W	🟩	
I_3			F	D	ML	ML	W	
I_4				F	D	R	A	
I_5					F	D	🟥	
I_6						F	🟥	
I_7							🟥	
I_8								
Time	0	1	2	3	4	5	6	7

F = Fetch
A = ALU
D = Decode
W = Write Result
R = Read Operand
MS = Mem. Store
ML = Mem. Load

As a result of the hazard affecting I_5, it is held back at the D stage, so I_6 cannot enter the D stage and is held back at the F stage, so I_7 cannot be fetched.

Pipelining (16)

```
(I1) LOAD   r0, VAR1
(I2) SET    r2, 0x00
(I3) LOAD   r1, VAR2
(I4) ADD    r0, 0x05
(I5) SUB    r1, 0x06
(I6) CMP    r0, r1
(I7) BRLO   L1
(I8) STORE  VAR1, r0
```

Time	0	1	2	3	4	5	6	7
I_1	F	D	ML	ML	W	🟩	🟩	
I_2		F	D	R	🟥	W	🟩	
I_3			F	D	ML	ML	W	
I_4				F	D	R	A	
I_5					F	D	🟥	
I_6						F	🟥	
I_7							🟥	
I_8								

F = Fetch
A = ALU
D = Decode
W = Write Result
R = Read Operand
MS = Mem. Store
ML = Mem. Load

Cases where one or more stages of the pipeline are empty are called **pipeline stalls** or **pipeline bubbles**. The affected stages do nothing during these time steps (since their previous output has not been passed on to a future stage yet).

Pipelining (17)

(I_1)	LOAD	r0, VAR1
(I_2)	SET	r2, 0x00
(I_3)	LOAD	r1, VAR2
(I_4)	ADD	r0, 0x05
(I_5)	SUB	r1, 0x06
(I_6)	CMP	r0, r1
(I_7)	BRLO	L1
(I_8)	STORE	VAR1, r0

Time	4	5	6	7	8	9	10	11
I_1	W	🟩	🟩					
I_2	🟥	W	🟩					
I_3	ML	ML	W					
I_4	D	R	A					
I_5	F	D	🟥					
I_6		F	🟥					
I_7			🟥					
I_8								

F = Fetch
A = ALU
D = Decode
W = Write Result
R = Read Operand
MS = Mem. Store
ML = Mem. Load

(The diagram above has been advanced to show time steps 8 through 11)

Pipelining (18)

```
(I1) LOAD    r0, VAR1
(I2) SET     r2, 0x00
(I3) LOAD    r1, VAR2
(I4) ADD     r0, 0x05
(I5) SUB     r1, 0x06
(I6) CMP     r0, r1
(I7) BRLO    L1
(I8) STORE   VAR1, r0
```

I_1	W	🟩	🟩	🟩				
I_2	🟥	W	🟩	🟩				
I_3	ML	ML	W	🟩				
I_4	D	R	A	W				
I_5	F	D	🟥	R				
I_6		F	🟥	D				
I_7			🟥	F				
I_8								
Time	4	5	6	7	8	9	10	11

F = Fetch   D = Decode   R = Read Operand   ML = Mem. Load
A = ALU   W = Write Result   MS = Mem. Store

At time 7, the operand for I_5 is available, so it enters the R stage. This allows I_6 to enter the D stage and I_7 to enter the F stage.

Pipelining (19)

(I_1)	`LOAD`	`r0, VAR1`
(I_2)	`SET`	`r2, 0x00`
(I_3)	`LOAD`	`r1, VAR2`
(I_4)	`ADD`	`r0, 0x05`
(I_5)	`SUB`	`r1, 0x06`
(I_6)	`CMP`	`r0, r1`
(I_7)	`BRLO`	`L1`
(I_8)	`STORE`	`VAR1, r0`

Time	4	5	6	7	8	9	10	11
I_1	W	🟩	🟩	🟩				
I_2	🟥	W	🟩	🟩				
I_3	ML	ML	W	🟩				
I_4	D	R	A	W				
I_5	F	D	🟥	R				
I_6		F	🟥	D				
I_7			🟥	F				
I_8								

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

Question: When will instruction I_6 be able to enter the R stage?

Pipelining (20)

```
(I1) LOAD    r0, VAR1
(I2) SET     r2, 0x00
(I3) LOAD    r1, VAR2
(I4) ADD     r0, 0x05
(I5) SUB     r1, 0x06
(I6) CMP     r0, r1
(I7) BRLO    L1
(I8) STORE   VAR1, r0
```

Time	4	5	6	7	8	9	10	11
I_1	W	🟩	🟩	🟩	🟩			
I_2	🟥	W	🟩	🟩	🟩			
I_3	ML	ML	W	🟩	🟩			
I_4	D	R	A	W	🟩			
I_5	F	D	🟥	R	A			
I_6		F	🟥	D	🟥			
I_7			🟥	F	🟥			
I_8					🟥			

F = Fetch
A = ALU
D = Decode
W = Write Result
R = Read Operand
MS = Mem. Store
ML = Mem. Load

At time 8, I_6 stalls waiting for I_5 to complete, which backs up the pipeline so I_8 cannot be fetched.

Pipeline (21)

```
(I1) LOAD   r0, VAR1
(I2) SET    r2, 0x00
(I3) LOAD   r1, VAR2
(I4) ADD    r0, 0x05
(I5) SUB    r1, 0x06
(I6) CMP    r0, r1
(I7) BRLO   L1
(I8) STORE  VAR1, r0
```

I_1	W	🟩	🟩	🟩	🟩			
I_2	🟥	W	🟩	🟩	🟩			
I_3	ML	ML	W	🟩	🟩			
I_4	D	R	A	W	🟩			
I_5	F	D	🟥	R	A	W		
I_6		F	🟥	D	🟥	🟥		
I_7			🟥	F	🟥	🟥		
I_8					🟥	🟥		
Time	4	5	6	7	8	9	10	11

F = Fetch
A = ALU
D = Decode
W = Write Result
R = Read Operand
MS = Mem. Store
ML = Mem. Load

This issue persists at time 9.

Pipeline (22)

```
(I1) LOAD    r0, VAR1
(I2) SET     r2, 0x00
(I3) LOAD    r1, VAR2
(I4) ADD     r0, 0x05
(I5) SUB     r1, 0x06
(I6) CMP     r0, r1
(I7) BRLO    L1
(I8) STORE   VAR1, r0
```

Time	4	5	6	7	8	9	10	11
I_1	W	🟩	🟩	🟩	🟩	🟩	🟩	
I_2	🟥	W	🟩	🟩	🟩	🟩	🟩	
I_3	ML	ML	W	🟩	🟩	🟩	🟩	
I_4	D	R	A	W	🟩	🟩	🟩	
I_5	F	D	🟥	R	A	W	🟩	
I_6		F	🟥	D	🟥	🟥	R	
I_7			🟥	F	🟥	🟥	D	
I_8					🟥	🟥	?	

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

At time 10, I_6 can finally read its operand, so I_7 can enter the decode stage.

Pipelining (23)

```
(I1) LOAD   r0, VAR1
(I2) SET    r2, 0x00
(I3) LOAD   r1, VAR2
(I4) ADD    r0, 0x05
(I5) SUB    r1, 0x06
(I6) CMP    r0, r1
(I7) BRLO   L1
(I8) STORE  VAR1, r0
```

I_1	W	🟩	🟩	🟩	🟩	🟩	🟩	
I_2	🟥	W	🟩	🟩	🟩	🟩	🟩	
I_3	ML	ML	W	🟩	🟩	🟩	🟩	
I_4	D	R	A	W	🟩	🟩	🟩	
I_5	F	D	🟥	R	A	W	🟩	
I_6		F	🟥	D	🟥	🟥	R	
I_7			🟥	F	🟥	🟥	D	
I_8					🟥	🟥	?	
Time	4	5	6	7	8	9	10	11

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

Question: Why might fetching I_8 be a waste of time?

Pipelining (24)

(I_1)	LOAD	r0, VAR1
(I_2)	SET	r2, 0x00
(I_3)	LOAD	r1, VAR2
(I_4)	ADD	r0, 0x05
(I_5)	SUB	r1, 0x06
(I_6)	CMP	r0, r1
(I_7)	BRLO	L1
(I_8)	STORE	VAR1, r0

Time	4	5	6	7	8	9	10	11
I_1	W	🟩	🟩	🟩	🟩	🟩	🟩	
I_2	🟥	W	🟩	🟩	🟩	🟩	🟩	
I_3	ML	ML	W	🟩	🟩	🟩	🟩	
I_4	D	R	A	W	🟩	🟩	🟩	
I_5	F	D	🟥	R	A	W	🟩	
I_6		F	🟥	D	🟥	🟥	R	
I_7			🟥	F	🟥	🟥	D	
I_8					🟥	🟥	?	

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

The instruction I_7 is a conditional branch instruction. The actual branch happens when I_7 reaches the W stage (that is, when it writes to the PC).

Pipelining (25)

```
(I1) LOAD    r0, VAR1
(I2) SET     r2, 0x00
(I3) LOAD    r1, VAR2
(I4) ADD     r0, 0x05
(I5) SUB     r1, 0x06
(I6) CMP     r0, r1
(I7) BRLO    L1
(I8) STORE   VAR1, r0
```

I_1	W	🟩	🟩	🟩	🟩	🟩	🟩	
I_2	🟥	W	🟩	🟩	🟩	🟩	🟩	
I_3	ML	ML	W	🟩	🟩	🟩	🟩	
I_4	D	R	A	W	🟩	🟩	🟩	
I_5	F	D	🟥	R	A	W	🟩	
I_6		F	🟥	D	🟥	🟥	R	
I_7			🟥	F	🟥	🟥	D	
I_8					🟥	🟥	?	
Time	4	5	6	7	8	9	10	11

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

Since I_7 modifies the PC, we actually don't know where the next instruction will be in memory, so we have no way to fetch it. The F stage is therefore left idle.

Pipelining (26)

(I_1)	LOAD	r0, VAR1
(I_2)	SET	r2, 0x00
(I_3)	LOAD	r1, VAR2
(I_4)	ADD	r0, 0x05
(I_5)	SUB	r1, 0x06
(I_6)	CMP	r0, r1
(I_7)	BRLO	L1
(I_8)	STORE	VAR1, r0

I_1	W	🟩	🟩	🟩	🟩	🟩	🟩	
I_2	🟥	W	🟩	🟩	🟩	🟩	🟩	
I_3	ML	ML	W	🟩	🟩	🟩	🟩	
I_4	D	R	A	W	🟩	🟩	🟩	
I_5	F	D	🟥	R	A	W	🟩	
I_6		F	🟥	D	🟥	🟥	R	
I_7			🟥	F	🟥	🟥	D	
I_8					🟥	🟥	?	
Time	4	5	6	7	8	9	10	11

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

As a result, when a branch instruction enters the pipeline, a **control hazard** occurs, and the pipeline stalls until the target of the branch is known.

Pipelining (27)

(I_1)	LOAD	r0, VAR1
(I_2)	SET	r2, 0x00
(I_3)	LOAD	r1, VAR2
(I_4)	ADD	r0, 0x05
(I_5)	SUB	r1, 0x06
(I_6)	CMP	r0, r1
(I_7)	BRLO	L1
(I_8)	STORE	VAR1, r0

I_1								
I_2								
I_3								
I_4								
I_5	A	W						
I_6			R					
I_7			D					
I_8			?					
Time	8	9	10	11	12	13	14	15

F = Fetch
A = ALU

D = Decode
W = Write Result

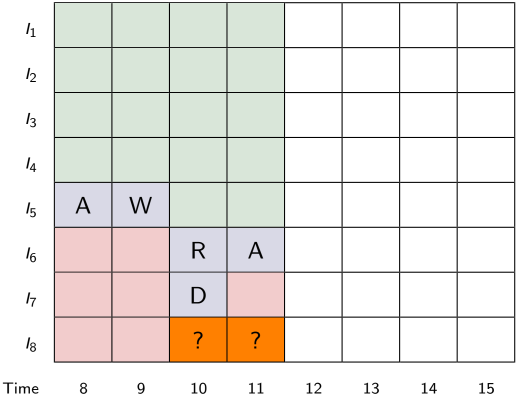
R = Read Operand
MS = Mem. Store

ML = Mem. Load

(The diagram above has been advanced to show time steps 12 through 15)

Pipelining (28)

```
(I1) LOAD    r0, VAR1
(I2) SET     r2, 0x00
(I3) LOAD    r1, VAR2
(I4) ADD     r0, 0x05
(I5) SUB     r1, 0x06
(I6) CMP     r0, r1
(I7) BRLO    L1
(I8) STORE   VAR1, r0
```



	8	9	10	11	12	13	14	15
I_1	🟩	🟩	🟩	🟩				
I_2	🟩	🟩	🟩	🟩				
I_3	🟩	🟩	🟩	🟩				
I_4	🟩	🟩	🟩	🟩				
I_5	A	W	🟩	🟩				
I_6	🟥	🟥	R	A				
I_7	🟥	🟥	D	🟥				
I_8	🟥	🟥	?	?				
Time	8	9	10	11	12	13	14	15

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

At time 11, I_6 enters the A stage (since a comparison requires arithmetic). I_7 is blocked from entering the R stage since it requires the comparison result.

Pipeline (29)

(I_1) `LOAD    r0, VAR1`
(I_2) `SET     r2, 0x00`
(I_3) `LOAD    r1, VAR2`
(I_4) `ADD     r0, 0x05`
(I_5) `SUB     r1, 0x06`
(I_6) `CMP     r0, r1`
(I_7) `BRLO    L1`
(I_8) `STORE   VAR1, r0`

Time	8	9	10	11	12	13	14	15
I_1	🟩	🟩	🟩	🟩	🟩			
I_2	🟩	🟩	🟩	🟩	🟩			
I_3	🟩	🟩	🟩	🟩	🟩			
I_4	🟩	🟩	🟩	🟩	🟩			
I_5	A	W	🟩	🟩	🟩			
I_6	🟥	🟥	R	A	W			
I_7	🟥	🟥	D	🟥	🟥			
I_8	🟥	🟥	?	?	?			

F = Fetch
A = ALU

D = Decode
W = Write Result

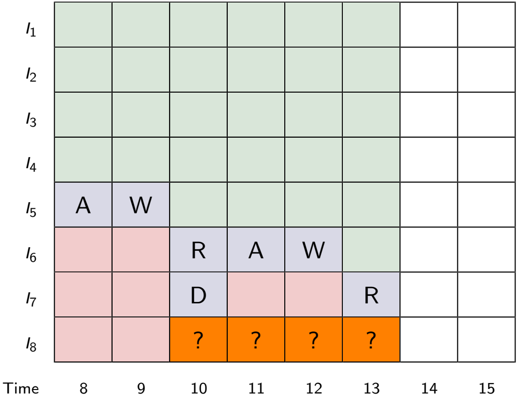
R = Read Operand
MS = Mem. Store

ML = Mem. Load

This continues through time 12.

Pipelining (30)

```
(I1) LOAD   r0, VAR1
(I2) SET    r2, 0x00
(I3) LOAD   r1, VAR2
(I4) ADD    r0, 0x05
(I5) SUB    r1, 0x06
(I6) CMP    r0, r1
(I7) BRLO   L1
(I8) STORE  VAR1, r0
```



I_1	🟩	🟩	🟩	🟩	🟩	🟩		
I_2	🟩	🟩	🟩	🟩	🟩	🟩		
I_3	🟩	🟩	🟩	🟩	🟩	🟩		
I_4	🟩	🟩	🟩	🟩	🟩	🟩		
I_5	A	W	🟩	🟩	🟩	🟩		
I_6	🟥	🟥	R	A	W	🟩		
I_7	🟥	🟥	D	🟥	🟥	R		
I_8	🟥	🟥	?	?	?	?		
Time	8	9	10	11	12	13	14	15

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

At time 13, I_6 is finished and the branch can enter the R stage.

Pipelining (31)

(I_1)	LOAD	r0,	VAR1
(I_2)	SET	r2,	0x00
(I_3)	LOAD	r1,	VAR2
(I_4)	ADD	r0,	0x05
(I_5)	SUB	r1,	0x06
(I_6)	CMP	r0,	r1
(I_7)	BRLO	L1	
(I_8)	STORE	VAR1,	r0

I_1	🟩	🟩	🟩	🟩	🟩	🟩	🟩	
I_2	🟩	🟩	🟩	🟩	🟩	🟩	🟩	
I_3	🟩	🟩	🟩	🟩	🟩	🟩	🟩	
I_4	🟩	🟩	🟩	🟩	🟩	🟩	🟩	
I_5	A	W	🟩	🟩	🟩	🟩	🟩	
I_6	🟥	🟥	R	A	W	🟩	🟩	
I_7	🟥	🟥	D	🟥	🟥	R	W	
I_8	🟥	🟥	?	?	?	?	?	
Time	8	9	10	11	12	13	14	15

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

At time 14, I_7 determines the branch result and writes the PC (we could also argue that the A stage is necessary to make the decision).

Pipelining (32)

```
(I1) LOAD   r0, VAR1
(I2) SET    r2, 0x00
(I3) LOAD   r1, VAR2
(I4) ADD    r0, 0x05
(I5) SUB    r1, 0x06
(I6) CMP    r0, r1
(I7) BRLO   L1
(I8) STORE  VAR1, r0
```

I_1	🟩	🟩	🟩	🟩	🟩	🟩	🟩	
I_2	🟩	🟩	🟩	🟩	🟩	🟩	🟩	
I_3	🟩	🟩	🟩	🟩	🟩	🟩	🟩	
I_4	🟩	🟩	🟩	🟩	🟩	🟩	🟩	
I_5	A	W	🟩	🟩	🟩	🟩	🟩	
I_6	🟥	🟥	R	A	W	🟩	🟩	
I_7	🟥	🟥	D	🟥	🟥	R	W	
I_8	🟥	🟥	?	?	?	?	?	
Time	8	9	10	11	12	13	14	15

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

Suppose that the branch **does not** occur (so the next instruction executed is indeed I_8).

Pipelining (33)

(I_1)	LOAD	r0, VAR1
(I_2)	SET	r2, 0x00
(I_3)	LOAD	r1, VAR2
(I_4)	ADD	r0, 0x05
(I_5)	SUB	r1, 0x06
(I_6)	CMP	r0, r1
(I_7)	BRLO	L1
(I_8)	STORE	VAR1, r0

I_1	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
I_2	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
I_3	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
I_4	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
I_5	A	W	🟩	🟩	🟩	🟩	🟩	🟩
I_6	🟥	🟥	R	A	W	🟩	🟩	🟩
I_7	🟥	🟥	D	🟥	🟥	R	W	🟩
I_8	🟥	🟥	?	?	?	?	?	F
Time	8	9	10	11	12	13	14	15

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

At time 15, I_8, having been determined to be the next instruction, is finally fetched.

Pipelining (34)

```
(I1) LOAD    r0, VAR1
(I2) SET     r2, 0x00
(I3) LOAD    r1, VAR2
(I4) ADD     r0, 0x05
(I5) SUB     r1, 0x06
(I6) CMP     r0, r1
(I7) BRLO    L1
(I8) STORE   VAR1, r0
```

I_1								
I_2								
I_3								
I_4								
I_5								
I_6	W							
I_7		R	W					
I_8	?	?	?	F	D	R	MS	MS
Time	12	13	14	15	16	17	18	19

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

(The diagram above has been advanced to show time steps 16 through 19)

Pipelining (35)

```
(I1) LOAD   r0, VAR1
(I2) SET    r2, 0x00
(I3) LOAD   r1, VAR2
(I4) ADD    r0, 0x05
(I5) SUB    r1, 0x06
(I6) CMP    r0, r1
(I7) BRLO   L1
(I8) STORE  VAR1, r0
```

I_1	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
I_2	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
I_3	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
I_4	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
I_5	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
I_6	W	🟩	🟩	🟩	🟩	🟩	🟩	🟩
I_7	🟥	R	W	🟩	🟩	🟩	🟩	🟩
I_8	?	?	?	F	D	R	MS	MS
Time	12	13	14	15	16	17	18	19

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

The sequence finishes at time 19 when I_8 finishes the MS stage.

Pipelining (36)

```
(I1) LOAD    r0, VAR1
(I2) SET     r2, 0x00
(I3) LOAD    r1, VAR2
(I4) ADD     r0, 0x05
(I5) SUB     r1, 0x06
(I6) CMP     r0, r1
(I7) BRLO    L1
(I8) STORE   VAR1, r0
```

Time	12	13	14	15	16	17	18	19
I_1								
I_2								
I_3								
I_4								
I_5								
I_6	W							
I_7		R	W					
I_8	?	?	?	F	D	R	MS	MS

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

With sequential execution (that is, with each instruction started after the previous one finished), a total of 38 time units would be required instead of 20.

Pipelining (37)

```
(I1) LOAD   r0, VAR1
(I2) SET    r2, 0x00
(I3) LOAD   r1, VAR2
(I4) ADD    r0, 0x05
(I5) SUB    r1, 0x06
(I6) CMP    r0, r1
(I7) BRLO   L1
(I8) STORE  VAR1, r0
```

I_1								
I_2								
I_3								
I_4								
I_5								
I_6	W							
I_7		R	W					
I_8	?	?	?	F	D	R	MS	MS
Time	12	13	14	15	16	17	18	19

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

Adding circuitry for pipelined operation allows the existing (complicated) computation units to be used more efficiently, at relatively low extra cost. The savings can be particularly significant if memory access time is high.

Pipelining (38)

```
(I1) LOAD   r0, VAR1
(I2) SET    r2, 0x00
(I3) LOAD   r1, VAR2
(I4) ADD    r0, 0x05
(I5) SUB    r1, 0x06
(I6) CMP    r0, r1
(I7) BRLO   L1
(I8) STORE  VAR1, r0
```

Time	12	13	14	15	16	17	18	19
I_1								
I_2								
I_3								
I_4								
I_5								
I_6	W							
I_7		R	W					
I_8	?	?	?	F	D	R	MS	MS

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

Arranging code to avoid pipeline stalls is an important optimization (although usually better left to compilers), especially in architectures with very deep pipelines (with a large number of stages).

Pipelining (39)

For a ‘normal’ pipelined architecture (without such extra labels as ‘superscalar’ or ‘parallel’), the following rules should always apply.

- The stages in the pipeline have a fixed order, and each instruction must progress through the stages in order. It is permissible for an instruction to skip stages (for example, an `ADD` instruction would skip the ML stage in the previous example), but the order of stages cannot be changed.
- If an instruction Z is prevented from moving forward after stage j, the pipeline stalls until it can move forward. No other instruction can use stage j until instruction Z exists the stage.
- A sequence of two instructions A and B must always be **in order**. If both A and B require a particular stage j, A **must** enter that stage first.

Pipelining (40)

Exercise: Suppose that the pipeline contains only five stages: Fetch (F), Decode (D), Memory Load (L), Execute (E) and Memory Store (S). The general (E) stage subsumes the R, A and W stages from the previous example. Assuming that F, D and E each require one unit of time, and that L and S each require three, rerun the previous example and determine the total number of units of time needed.

Pipelining (41)

Given a sequence of assembly instructions and the description of a pipelined architecture, you should be able to identify each of the following data hazards.

- **Structural Hazard**: Two (or more) instructions require the same computation unit (e.g. memory load or ALU), so one instruction stalls. Structural hazards can occur as a secondary hazard when an instruction is stalled and prevents another instruction from advancing (e.g. an instruction stalls in the decode stage due to a data hazard, and as a result the fetch stage is also stalled).
- **Data Hazard**: An earlier instruction may modify an operand needed by a later instruction, so the later instruction stalls until the earlier instruction finishes modifying the operand.
- **Control Hazard**: The exact target of a conditional branch instruction is unknown, so the entire pipelines stalls until the branch completes. This can be considered a type of data hazard (with PC as the subject of the hazard).

Time for Each Stage (1)

The advantage of using pipeline is an increase in **throughput**, which is the amount of work done in a given period of time. Pipelining an architecture can increase throughput without increasing the clock speed or complexity of the instructions.

Generally, pipeline stages are designed such that each stage requires one clock cycle (with possible exceptions where memory access is required).

Since this results in multiple clock cycles being needed for all instructions, even simple ones like `ADD`, pipelining seems counterintuitive, even when the increase in throughput is considered. It is especially counterintuitive from the perspective of the AVR architecture, where almost every instruction requires 1 or 2 cycles.

Time for Each Stage (2)

However, clock speeds are generally much higher in pipelined architectures, since the actual amount of work done at each clock cycle (and therefore the latency of the circuitry required) is lower. Among other factors, like power consumption and heat dissipation, clock speeds are generally limited by the time required for signals to travel through circuitry, so a series of small, compact stages tends to allow for more cycles per second. It is unlikely that an AVR processor (with the current design) would work at the clock speeds used by pipelined architectures in which one instruction would normally take more than one cycle.

Additionally, there are pipelined processors (including modern Intel and AMD processors) where stages require fewer than one clock cycle to execute.

Branch Prediction (1)

```
(I1) LOAD    r0, VAR1
(I2) SET     r2, 0x00
(I3) LOAD    r1, VAR2
(I4) ADD     r0, 0x05
(I5) SUB     r1, 0x06
(I6) CMP     r0, r1
(I7) BRLO    L1
(I8) STORE   VAR1, r0
```

I_1	🟩	🟩	🟩					
I_2	🟩	🟩	🟩					
I_3	🟩	🟩	🟩					
I_4	🟩	🟩	🟩					
I_5	A	W	🟩					
I_6	🟥	🟥	R					
I_7	🟥	🟥	D					
I_8	🟥	🟥	?					
Time	8	9	10	11	12	13	14	15

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

Recall that at time 10 in the previous example, we determined that it was impossible to fetch the next instruction, since the branch had not finished.

Branch Prediction (2)

```
(I1) LOAD    r0, VAR1
(I2) SET     r2, 0x00
(I3) LOAD    r1, VAR2
(I4) ADD     r0, 0x05
(I5) SUB     r1, 0x06
(I6) CMP     r0, r1
(I7) BRLO    L1
(I8) STORE   VAR1, r0
```

I_1	🟩	🟩	🟩					
I_2	🟩	🟩	🟩					
I_3	🟩	🟩	🟩					
I_4	🟩	🟩	🟩					
I_5	A	W	🟩					
I_6	🟥	🟥	R					
I_7	🟥	🟥	D					
I_8	🟥	🟥	?					
Time	8	9	10	11	12	13	14	15

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

It is true that the next instruction is not known until after the branch reaches the W stage.

Branch Prediction (3)

```
(I1) LOAD    r0, VAR1
(I2) SET     r2, 0x00
(I3) LOAD    r1, VAR2
(I4) ADD     r0, 0x05
(I5) SUB     r1, 0x06
(I6) CMP     r0, r1
(I7) BRLO    L1
(I8) STORE   VAR1, r0
```

Time	8	9	10	11	12	13	14	15
I_1	🟩	🟩	🟩					
I_2	🟩	🟩	🟩					
I_3	🟩	🟩	🟩					
I_4	🟩	🟩	🟩					
I_5	A	W	🟩					
I_6	🟥	🟥	R					
I_7	🟥	🟥	D					
I_8	🟥	🟥	?					

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

Question: Why not try to predict the future?

Branch Prediction (4)

```
(I1) LOAD   r0, VAR1
(I2) SET    r2, 0x00
(I3) LOAD   r1, VAR2
(I4) ADD    r0, 0x05
(I5) SUB    r1, 0x06
(I6) CMP    r0, r1
(I7) BRLO   L1
(I8) STORE  VAR1, r0
```

Time	8	9	10	11	12	13	14	15
I_1	🟩	🟩	🟩					
I_2	🟩	🟩	🟩					
I_3	🟩	🟩	🟩					
I_4	🟩	🟩	🟩					
I_5	A	W	🟩					
I_6	🟥	🟥	R					
I_7	🟥	🟥	D					
I_8	🟥	🟥	?					

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

Although we certainly can't *execute* anything until we know what instruction will come next, it might be possible to take care of some pre-processing to speed up execution if our prediction is correct.

Branch Prediction (5)

```
(I1) LOAD    r0, VAR1
(I2) SET     r2, 0x00
(I3) LOAD    r1, VAR2
(I4) ADD     r0, 0x05
(I5) SUB     r1, 0x06
(I6) CMP     r0, r1
(I7) BRLO    L1
(I8) STORE   VAR1, r0
```

I_1	🟩	🟩	🟩					
I_2	🟩	🟩	🟩					
I_3	🟩	🟩	🟩					
I_4	🟩	🟩	🟩					
I_5	A	W	🟩					
I_6	🟥	🟥	R					
I_7	🟥	🟥	D					
I_8	🟥	🟥	?					
Time	8	9	10	11	12	13	14	15

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

This is called **branch prediction**, and is crucially important in architectures with deep pipelines (such as the CISC x86 architecture).

Branch Prediction (6)

```
(I1) LOAD    r0, VAR1
(I2) SET     r2, 0x00
(I3) LOAD    r1, VAR2
(I4) ADD     r0, 0x05
(I5) SUB     r1, 0x06
(I6) CMP     r0, r1
(I7) BRLO    L1
(I8) STORE   VAR1, r0
```

Time	8	9	10	11	12	13	14	15
I_1	🟩	🟩	🟩					
I_2	🟩	🟩	🟩					
I_3	🟩	🟩	🟩					
I_4	🟩	🟩	🟩					
I_5	A	W	🟩					
I_6	🟥	🟥	R					
I_7	🟥	🟥	D					
I_8	🟥	🟥	?					

F = Fetch D = Decode R = Read Operand ML = Mem. Load
A = ALU W = Write Result MS = Mem. Store

Control hazards can lead to stalls of 20 clock cycles or more on modern machines, so branch prediction is a **really big deal**. Branch prediction techniques are among the most closely guarded secrets of processor designers.

Branch Prediction (7)

```
(I1) LOAD   r0, VAR1
(I2) SET    r2, 0x00
(I3) LOAD   r1, VAR2
(I4) ADD    r0, 0x05
(I5) SUB    r1, 0x06
(I6) CMP    r0, r1
(I7) BRLO   L1
(I8) STORE  VAR1, r0
```

Time	8	9	10	11	12	13	14	15
I_1	🟩	🟩	🟩					
I_2	🟩	🟩	🟩					
I_3	🟩	🟩	🟩					
I_4	🟩	🟩	🟩					
I_5	A	W	🟩					
I_6	🟥	🟥	R					
I_7	🟥	🟥	D					
I_8	🟥	🟥	?					

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

A simple branch prediction rule: Assume that the branch is **never** taken. That is, assume that the next instruction in sequence is always the next instruction executed.

Branch Prediction (8)

```
(I1) LOAD    r0, VAR1
(I2) SET     r2, 0x00
(I3) LOAD    r1, VAR2
(I4) ADD     r0, 0x05
(I5) SUB     r1, 0x06
(I6) CMP     r0, r1
(I7) BRLO    L1
(I8) STORE   VAR1, r0
```

Time	8	9	10	11	12	13	14	15
I_1	🟩	🟩	🟩					
I_2	🟩	🟩	🟩					
I_3	🟩	🟩	🟩					
I_4	🟩	🟩	🟩					
I_5	A	W	🟩					
I_6	🟥	🟥	R					
I_7	🟥	🟥	D					
I_8	🟥	🟥	?					

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

This rule is obviously not perfect, but it can still lead to a performance increase. In general, even the worst prediction rule is better than nothing (since with no branch prediction, a full stall will occur).

Branch Prediction (9)

```
(I1) LOAD   r0, VAR1
(I2) SET    r2, 0x00
(I3) LOAD   r1, VAR2
(I4) ADD    r0, 0x05
(I5) SUB    r1, 0x06
(I6) CMP    r0, r1
(I7) BRLO   L1
(I8) STORE  VAR1, r0
```

Time	8	9	10	11	12	13	14	15
I_1	🟩	🟩	🟩					
I_2	🟩	🟩	🟩					
I_3	🟩	🟩	🟩					
I_4	🟩	🟩	🟩					
I_5	A	W	🟩					
I_6	🟥	🟥	R					
I_7	🟥	🟥	D					
I_8	🟥	🟥	F					

F = Fetch
A = ALU
D = Decode
W = Write Result
R = Read Operand
MS = Mem. Store
ML = Mem. Load

Restarting from time 10, by our branch prediction rule, we predict that I_8 will execute next, so we fetch it.

Branch Prediction (10)

```
(I1) LOAD    r0, VAR1
(I2) SET     r2, 0x00
(I3) LOAD    r1, VAR2
(I4) ADD     r0, 0x05
(I5) SUB     r1, 0x06
(I6) CMP     r0, r1
(I7) BRLO    L1
(I8) STORE   VAR1, r0
```

I_1	🟩	🟩	🟩	🟩				
I_2	🟩	🟩	🟩	🟩				
I_3	🟩	🟩	🟩	🟩				
I_4	🟩	🟩	🟩	🟩				
I_5	A	W	🟩	🟩				
I_6	🟥	🟥	R	A				
I_7	🟥	🟥	D	🟥				
I_8	🟥	🟥	F	🟥				
Time	8	9	10	11	12	13	14	15

F = Fetch
A = ALU
D = Decode
W = Write Result
R = Read Operand
MS = Mem. Store
ML = Mem. Load

At time 11, the branch is held back at the D stage as it waits for the comparison result.

Branch Prediction (11)

(I_1)	LOAD	r0, VAR1
(I_2)	SET	r2, 0x00
(I_3)	LOAD	r1, VAR2
(I_4)	ADD	r0, 0x05
(I_5)	SUB	r1, 0x06
(I_6)	CMP	r0, r1
(I_7)	BRLO	L1
(I_8)	STORE	VAR1, r0

I_1	🟩	🟩	🟩	🟩	🟩			
I_2	🟩	🟩	🟩	🟩	🟩			
I_3	🟩	🟩	🟩	🟩	🟩			
I_4	🟩	🟩	🟩	🟩	🟩			
I_5	A	W	🟩	🟩	🟩			
I_6	🟥	🟥	R	A	W			
I_7	🟥	🟥	D	🟥	🟥			
I_8	🟥	🟥	F	🟥	🟥			
Time	8	9	10	11	12	13	14	15

F = Fetch
D = Decode
R = Read Operand
ML = Mem. Load
A = ALU
W = Write Result
MS = Mem. Store

The stall continues through time 12.

Branch Prediction (12)

```
(I1) LOAD   r0, VAR1
(I2) SET    r2, 0x00
(I3) LOAD   r1, VAR2
(I4) ADD    r0, 0x05
(I5) SUB    r1, 0x06
(I6) CMP    r0, r1
(I7) BRLO   L1
(I8) STORE  VAR1, r0
```

I_1	🟩	🟩	🟩	🟩	🟩	🟩		
I_2	🟩	🟩	🟩	🟩	🟩	🟩		
I_3	🟩	🟩	🟩	🟩	🟩	🟩		
I_4	🟩	🟩	🟩	🟩	🟩	🟩		
I_5	A	W	🟩	🟩	🟩	🟩		
I_6	🟥	🟥	R	A	W	🟩		
I_7	🟥	🟥	D	🟥	🟥	R		
I_8	🟥	🟥	F	🟥	🟥	D		
Time	8	9	10	11	12	13	14	15

F = Fetch D = Decode R = Read Operand ML = Mem. Load
A = ALU W = Write Result MS = Mem. Store

At time 13, the branch reaches the R stage and I_8 can be decoded.

Branch Prediction (13)

```
(I1) LOAD    r0, VAR1
(I2) SET     r2, 0x00
(I3) LOAD    r1, VAR2
(I4) ADD     r0, 0x05
(I5) SUB     r1, 0x06
(I6) CMP     r0, r1
(I7) BRLO    L1
(I8) STORE   VAR1, r0
```

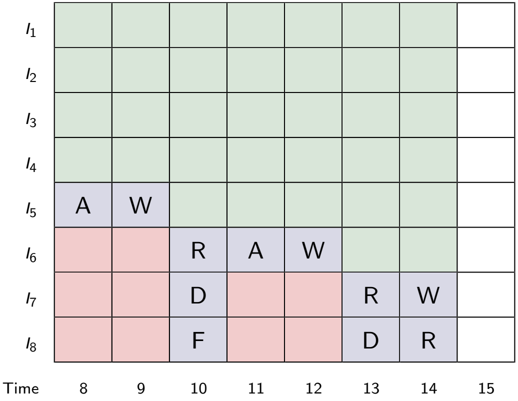
Time	8	9	10	11	12	13	14	15
I_1	🟩	🟩	🟩	🟩	🟩	🟩	🟩	
I_2	🟩	🟩	🟩	🟩	🟩	🟩	🟩	
I_3	🟩	🟩	🟩	🟩	🟩	🟩	🟩	
I_4	🟩	🟩	🟩	🟩	🟩	🟩	🟩	
I_5	A	W	🟩	🟩	🟩	🟩	🟩	
I_6	🟥	🟥	R	A	W	🟩	🟩	
I_7	🟥	🟥	D	🟥	🟥	R	W	
I_8	🟥	🟥	F	🟥	🟥	D	R	

F = Fetch
A = ALU
D = Decode
W = Write Result
R = Read Operand
MS = Mem. Store
ML = Mem. Load

At time 14, the branch enters the W stage. The next stage for I_8 is the R stage, which it can enter since the R stage does not modify anything.

Branch Prediction (14)

```
(I1) LOAD   r0, VAR1
(I2) SET    r2, 0x00
(I3) LOAD   r1, VAR2
(I4) ADD    r0, 0x05
(I5) SUB    r1, 0x06
(I6) CMP    r0, r1
(I7) BRLO   L1
(I8) STORE  VAR1, r0
```



Time	8	9	10	11	12	13	14	15
I_1	🟩	🟩	🟩	🟩	🟩	🟩	🟩	
I_2	🟩	🟩	🟩	🟩	🟩	🟩	🟩	
I_3	🟩	🟩	🟩	🟩	🟩	🟩	🟩	
I_4	🟩	🟩	🟩	🟩	🟩	🟩	🟩	
I_5	A	W	🟩	🟩	🟩	🟩	🟩	
I_6	🟥	🟥	R	A	W	🟩	🟩	
I_7	🟥	🟥	D	🟥	🟥	R	W	
I_8	🟥	🟥	F	🟥	🟥	D	R	

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

(If, instead of R, instruction I_8 were about to enter the W stage, it would have to be stalled, since we can't allow I_8 to change anything since we don't know if it will actually be allowed to run).

Branch Prediction (15)

(I_1)	LOAD	r0, VAR1
(I_2)	SET	r2, 0x00
(I_3)	LOAD	r1, VAR2
(I_4)	ADD	r0, 0x05
(I_5)	SUB	r1, 0x06
(I_6)	CMP	r0, r1
(I_7)	BRLO	L1
(I_8)	STORE	VAR1, r0

I_1	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
I_2	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
I_3	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
I_4	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
I_5	A	W	🟩	🟩	🟩	🟩	🟩	🟩
I_6	🟥	🟥	R	A	W	🟩	🟩	🟩
I_7	🟥	🟥	D	🟥	🟥	R	W	🟩
I_8	🟥	🟥	F	🟥	🟥	D	R	
Time	8	9	10	11	12	13	14	15

F = Fetch
A = ALU
D = Decode
W = Write Result
R = Read Operand
MS = Mem. Store
ML = Mem. Load

At time 15, one of two things can happen. If the branch resulted in I_8 being the next instruction, the pipeline continues as normal (and I_8 enters the MS stage).

Branch Prediction (16)

(I_1)	LOAD	r0, VAR1
(I_2)	SET	r2, 0x00
(I_3)	LOAD	r1, VAR2
(I_4)	ADD	r0, 0x05
(I_5)	SUB	r1, 0x06
(I_6)	CMP	r0, r1
(I_7)	BRLO	L1
(I_8)	STORE	VAR1, r0

I_1	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
I_2	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
I_3	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
I_4	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
I_5	A	W	🟩	🟩	🟩	🟩	🟩	🟩
I_6	🟥	🟥	R	A	W	🟩	🟩	🟩
I_7	🟥	🟥	D	🟥	🟥	R	W	🟩
I_8	🟥	🟥	F	🟥	🟥	D	R	
Time	8	9	10	11	12	13	14	15

F = Fetch	D = Decode	R = Read Operand	ML = Mem. Load
A = ALU	W = Write Result	MS = Mem. Store	

In this case, due to the work done in advance, the sequence of instructions finishes at step 16 (instead of 20).

Branch Prediction (17)

```
(I1) LOAD   r0, VAR1
(I2) SET    r2, 0x00
(I3) LOAD   r1, VAR2
(I4) ADD    r0, 0x05
(I5) SUB    r1, 0x06
(I6) CMP    r0, r1
(I7) BRLO   L1
(I8) STORE  VAR1, r0
```

I_1	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
I_2	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
I_3	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
I_4	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
I_5	A	W	🟩	🟩	🟩	🟩	🟩	🟩
I_6	🟥	🟥	R	A	W	🟩	🟩	🟩
I_7	🟥	🟥	D	🟥	🟥	R	W	🟩
I_8	🟥	🟥	F	🟥	🟥	D	R	*
Time	8	9	10	11	12	13	14	15

F = Fetch
A = ALU
D = Decode
W = Write Result
R = Read Operand
MS = Mem. Store
ML = Mem. Load

If the branch resulted in a different instruction being executed, the work already done on I_8 is abandoned and the actual branch destination enters the F stage.

Branch Prediction Rule Example (1)

```
(I1)        SET     r2, 0x05
(I2)        CMP     r0, r1
(I3)        BRLO    L6
(I4)        ADD     r2, 0x10
(I5)        SUB     r1, 0x06
(I6)  L6:   SUB     r2, 0x01
(I7)        ADD     r0, 0x01
(I8)        ADD     r0, r1
```

I_1								
I_2								
I_3								
I_4								
I_5								
I_6								
I_7								
I_8								
Time	0	1	2	3	4	5	6	7

F = Fetch D = Decode R = Read Operand ML = Mem. Load
A = ALU W = Write Result MS = Mem. Store

Consider the code above (which does not use any memory access instructions, for simplicity) and suppose that a branch prediction rule is used which assumes that BRLO is **always** taken.

Branch Prediction Rule Example (2)

```
(I1)        SET     r2, 0x05
(I2)        CMP     r0, r1
(I3)        BRLO    L6
(I4)        ADD     r2, 0x10
(I5)        SUB     r1, 0x06
(I6)  L6:   SUB     r2, 0x01
(I7)        ADD     r0, 0x01
(I8)        ADD     r0, r1
```

I_1								
I_2								
I_3								
I_4								
I_5								
I_6								
I_7								
I_8								
Time	0	1	2	3	4	5	6	7

F = Fetch D = Decode R = Read Operand ML = Mem. Load
A = ALU W = Write Result MS = Mem. Store

Additionally, assume that the branch prediction allows the instructions after the predicted branch to provisionally pass through the F, D and R stages (which do not make any changes) and then stalls until the branch exits the W stage.

Branch Prediction Rule Example (3)

```
(I1)        SET     r2, 0x05
(I2)        CMP     r0, r1
(I3)        BRLO    L6
(I4)        ADD     r2, 0x10
(I5)        SUB     r1, 0x06
(I6) L6:    SUB     r2, 0x01
(I7)        ADD     r0, 0x01
(I8)        ADD     r0, r1
```

I_1								
I_2								
I_3								
I_4								
I_5								
I_6								
I_7								
I_8								
Time	0	1	2	3	4	5	6	7

F = Fetch
A = ALU
D = Decode
W = Write Result
R = Read Operand
MS = Mem. Store
ML = Mem. Load

If the prediction rule predicts that the branch **will** happen, we provisionally execute the instructions from after the branch target (L6 in the above example). If the prediction rule predicts that branch **will not** happen, we provisionally execute the instructions from after the branch (starting at I_4 in the above example)

Branch Prediction Rule Example (4)

```
(I1)        SET     r2, 0x05
(I2)        CMP     r0, r1
(I3)        BRLO    L6
(I4)        ADD     r2, 0x10
(I5)        SUB     r1, 0x06
(I6)  L6:   SUB     r2, 0x01
(I7)        ADD     r0, 0x01
(I8)        ADD     r0, r1
```

I_1								
I_2								
I_3								
I_4								
I_5								
I_6								
I_7								
I_8								
Time	0	1	2	3	4	5	6	7

F = Fetch
D = Decode
R = Read Operand
ML = Mem. Load
A = ALU
W = Write Result
MS = Mem. Store

Example A: With the prediction rule that BRLO **always jumps**, show the progression of the code above when r0 is less than r1 (that is, when the branch actually does jump).

Branch Prediction Rule Example (5)

```
(I1)        SET     r2, 0x05
(I2)        CMP     r0, r1
(I3)        BRLO    L6
(I4)        ADD     r2, 0x10
(I5)        SUB     r1, 0x06
(I6) L6:    SUB     r2, 0x01
(I7)        ADD     r0, 0x01
(I8)        ADD     r0, r1
```

I_1	F	D	R					
I_2		F	D					
I_3			F					
I_4								
I_5								
I_6								
I_7								
I_8								
Time	0	1	2	3	4	5	6	7

F = Fetch
D = Decode
R = Read Operand
ML = Mem. Load
A = ALU
W = Write Result
MS = Mem. Store

The diagram above shows time 0 - time 2 (everything proceeds as expected).

Branch Prediction Rule Example (6)

```
(I1)        SET     r2, 0x05
(I2)        CMP     r0, r1
(I3)        BRLO    L6
(I4)        ADD     r2, 0x10
(I5)        SUB     r1, 0x06
(I6)  L6:   SUB     r2, 0x01
(I7)        ADD     r0, 0x01
(I8)        ADD     r0, r1
```

I_1	F	D	R	W				
I_2		F	D	R				
I_3			F	D				
I_4								
I_5								
I_6				F				
I_7								
I_8								
Time	0	1	2	3	4	5	6	7

F = Fetch
A = ALU
D = Decode
W = Write Result
R = Read Operand
MS = Mem. Store
ML = Mem. Load

At time 3, the branch instruction leaves the F stage. The prediction rule predicts that the branch will be followed, so it fetches the instruction after the jump (I_6).

Branch Prediction Rule Example (7)

```
(I1)        SET     r2, 0x05
(I2)        CMP     r0, r1
(I3)        BRLO    L6
(I4)        ADD     r2, 0x10
(I5)        SUB     r1, 0x06
(I6) L6:    SUB     r2, 0x01
(I7)        ADD     r0, 0x01
(I8)        ADD     r0, r1
```

Time	0	1	2	3	4	5	6	7
I_1	F	D	R	W	🟩	🟩		
I_2		F	D	R	A	W		
I_3			F	D	🟥	🟥		
I_4								
I_5								
I_6				F	🟥	🟥		
I_7								
I_8								

F = Fetch	D = Decode	R = Read Operand	ML = Mem. Load
A = ALU	W = Write Result	MS = Mem. Store	

At time 4 and time 5, the branch is stalled due to a data hazard with the CMP instruction. As a result, I_6 is stalled by a structural hazard.

Branch Prediction Rule Example (8)

```
(I1)        SET     r2, 0x05
(I2)        CMP     r0, r1
(I3)        BRLO    L6
(I4)        ADD     r2, 0x10
(I5)        SUB     r1, 0x06
(I6) L6:    SUB     r2, 0x01
(I7)        ADD     r0, 0x01
(I8)        ADD     r0, r1
```

Time	0	1	2	3	4	5	6	7
I_1	F	D	R	W	🟩	🟩	🟩	
I_2		F	D	R	A	W	🟩	
I_3			F	D	🟥	🟥	R	
I_4								
I_5								
I_6				F	🟥	🟥	D	
I_7							F	
I_8								

F = Fetch
A = ALU
D = Decode
W = Write Result
R = Read Operand
MS = Mem. Store
ML = Mem. Load

At time 6, the branch reads its operands and I_6 enters the decode stage. This allows I_7 (which would follow I_6) to be fetched.

Branch Prediction Rule Example (9)

```
(I1)        SET     r2, 0x05
(I2)        CMP     r0, r1
(I3)        BRLO    L6
(I4)        ADD     r2, 0x10
(I5)        SUB     r1, 0x06
(I6) L6:    SUB     r2, 0x01
(I7)        ADD     r0, 0x01
(I8)        ADD     r0, r1
```

I_1	🟩	🟩	🟩	🟩				
I_2	A	W	🟩	🟩				
I_3	🟥	🟥	R	W				
I_4								
I_5								
I_6	🟥	🟥	D	R				
I_7			F	D				
I_8				F				
Time	4	5	6	7	8	9	10	11

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

At time 7, I_6 reads its operands and I_7 is decoded. The branch finishes the W stage.

Branch Prediction Rule Example (10)

```
(I1)         SET     r2, 0x05
(I2)         CMP     r0, r1
(I3)         BRLO    L6
(I4)         ADD     r2, 0x10
(I5)         SUB     r1, 0x06
(I6)  L6:    SUB     r2, 0x01
(I7)         ADD     r0, 0x01
(I8)         ADD     r0, r1
```

I_1	🟩	🟩	🟩	🟩	🟩			
I_2	A	W	🟩	🟩	🟩			
I_3	🟥	🟥	R	W	🟩			
I_4								
I_5								
I_6	🟥	🟥	D	R	A			
I_7			F	D	R			
I_8				F	D			
Time	4	5	6	7	8	9	10	11

F = Fetch　　D = Decode　　R = Read Operand　　ML = Mem. Load
A = ALU　　W = Write Result　　MS = Mem. Store

Since the example stipulated that the branch actually did jump (so the prediction was correct), there is no need to flush the pipeline at time 8 and execution proceeds as normal from I_6.

Branch Prediction Rule Example (11)

```
(I1)        SET    r2, 0x05
(I2)        CMP    r0, r1
(I3)        BRLO   L6
(I4)        ADD    r2, 0x10
(I5)        SUB    r1, 0x06
(I6)  L6:   SUB    r2, 0x01
(I7)        ADD    r0, 0x01
(I8)        ADD    r0, r1
```

I_1	🟩	🟩	🟩	🟩	🟩	🟩		
I_2	A	W	🟩	🟩	🟩	🟩		
I_3	🟥	🟥	R	W	🟩	🟩		
I_4								
I_5								
I_6	🟥	🟥	D	R	A	W		
I_7			F	D	R	A		
I_8				F	D	🟥		
Time	4	5	6	7	8	9	10	11

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

At time 9, I_8 is stalled by a data hazard.

Branch Prediction Rule Example (12)

```
(I1)        SET    r2, 0x05
(I2)        CMP    r0, r1
(I3)        BRLO   L6
(I4)        ADD    r2, 0x10
(I5)        SUB    r1, 0x06
(I6)  L6:   SUB    r2, 0x01
(I7)        ADD    r0, 0x01
(I8)        ADD    r0, r1
```

I_1	🟩	🟩	🟩	🟩	🟩	🟩		
I_2	🟩	🟩	🟩	🟩	🟩	🟩		
I_3	🟩	🟩	🟩	🟩	🟩	🟩		
I_4								
I_5								
I_6	A	W	🟩	🟩	🟩	🟩		
I_7	R	A	W	🟩	🟩	🟩		
I_8	D	🟥	🟥	R	A	W		
Time	8	9	10	11	12	13	14	15

F = Fetch D = Decode R = Read Operand ML = Mem. Load
A = ALU W = Write Result MS = Mem. Store

The sequence finishes at time 13.

Branch Prediction Rule Example (13)

```
(I1)        SET     r2, 0x05
(I2)        CMP     r0, r1
(I3)        BRLO    L6
(I4)        ADD     r2, 0x10
(I5)        SUB     r1, 0x06
(I6)  L6:   SUB     r2, 0x01
(I7)        ADD     r0, 0x01
(I8)        ADD     r0, r1
```

Time	0	1	2	3	4	5	6	7
I_1								
I_2								
I_3								
I_4								
I_5								
I_6								
I_7								
I_8								

F = Fetch   D = Decode   R = Read Operand   ML = Mem. Load
A = ALU   W = Write Result   MS = Mem. Store

Example B: With the prediction rule that BRLO **always jumps**, show the progression of the code above when r0 is greater than r1 (that is, when the branch does not jump).

Branch Prediction Rule Example (14)

```
(I1)        SET    r2, 0x05
(I2)        CMP    r0, r1
(I3)        BRLO   L6
(I4)        ADD    r2, 0x10
(I5)        SUB    r1, 0x06
(I6)  L6:   SUB    r2, 0x01
(I7)        ADD    r0, 0x01
(I8)        ADD    r0, r1
```

I_1	F	D	R					
I_2		F	D					
I_3			F					
I_4								
I_5								
I_6								
I_7								
I_8								
Time	0	1	2	3	4	5	6	7

F = Fetch   D = Decode   R = Read Operand   ML = Mem. Load
A = ALU   W = Write Result   MS = Mem. Store

The first few time steps are exactly the same as the previous example.

Branch Prediction Rule Example (15)

```
(I1)        SET     r2, 0x05
(I2)        CMP     r0, r1
(I3)        BRLO    L6
(I4)        ADD     r2, 0x10
(I5)        SUB     r1, 0x06
(I6)  L6:   SUB     r2, 0x01
(I7)        ADD     r0, 0x01
(I8)        ADD     r0, r1
```

I_1	F	D	R	W				
I_2		F	D	R				
I_3			F	D				
I_4								
I_5								
I_6				F				
I_7								
I_8								
Time	0	1	2	3	4	5	6	7

F = Fetch
D = Decode
R = Read Operand
ML = Mem. Load
A = ALU
W = Write Result
MS = Mem. Store

Notice that the prediction rule fetches I_6 as before (since until the branch reaches the W stage, the processor cannot know that the branch will not be taken).

Branch Prediction Rule Example (16)

```
(I1)         SET     r2, 0x05
(I2)         CMP     r0, r1
(I3)         BRLO    L6
(I4)         ADD     r2, 0x10
(I5)         SUB     r1, 0x06
(I6)  L6:    SUB     r2, 0x01
(I7)         ADD     r0, 0x01
(I8)         ADD     r0, r1
```

I_1	F	D	R	W	🟩	🟩		
I_2		F	D	R	A	W		
I_3			F	D	🟥	🟥		
I_4								
I_5								
I_6				F	🟥	🟥		
I_7								
I_8								
Time	0	1	2	3	4	5	6	7

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

Branch Prediction Rule Example (17)

```
(I1)        SET    r2, 0x05
(I2)        CMP    r0, r1
(I3)        BRLO   L6
(I4)        ADD    r2, 0x10
(I5)        SUB    r1, 0x06
(I6)  L6:   SUB    r2, 0x01
(I7)        ADD    r0, 0x01
(I8)        ADD    r0, r1
```

Time	0	1	2	3	4	5	6	7
I_1	F	D	R	W	🟩	🟩	🟩	
I_2		F	D	R	A	W	🟩	
I_3			F	D	🟥	🟥	R	
I_4								
I_5								
I_6				F	🟥	🟥	D	
I_7							F	
I_8								

F = Fetch　D = Decode　R = Read Operand　ML = Mem. Load
A = ALU　W = Write Result　MS = Mem. Store

Branch Prediction Rule Example (18)

```
(I1)       SET    r2, 0x05
(I2)       CMP    r0, r1
(I3)       BRLO   L6
(I4)       ADD    r2, 0x10
(I5)       SUB    r1, 0x06
(I6) L6:   SUB    r2, 0x01
(I7)       ADD    r0, 0x01
(I8)       ADD    r0, r1
```

I_1	🟩	🟩	🟩	🟩				
I_2	A	W	🟩	🟩				
I_3	🟥	🟥	R	W				
I_4								
I_5								
I_6	🟥	🟥	D	R				
I_7			F	D				
I_8				F				
Time	4	5	6	7	8	9	10	11

F = Fetch D = Decode R = Read Operand ML = Mem. Load
A = ALU W = Write Result MS = Mem. Store

At time 7, I_6 reads its operands and I_7 is decoded. The branch finishes the W stage.

Branch Prediction Rule Example (19)

```
(I1)        SET    r2, 0x05
(I2)        CMP    r0, r1
(I3)        BRLO   L6
(I4)        ADD    r2, 0x10
(I5)        SUB    r1, 0x06
(I6)  L6:   SUB    r2, 0x01
(I7)        ADD    r0, 0x01
(I8)        ADD    r0, r1
```

I_1	🟩	🟩	🟩	🟩	🟩			
I_2	A	W	🟩	🟩	🟩			
I_3	🟥	🟥	R	W	🟩			
I_4					F			
I_5								
I_6	🟥	🟥	D	R	$*$			
I_7			F	D	$*$			
I_8				F	$*$			
Time	4	5	6	7	8	9	10	11

F = Fetch
A = ALU
D = Decode
W = Write Result
R = Read Operand
MS = Mem. Store
ML = Mem. Load

In this example, the branch **does not** jump, so the next instruction is I_4. As a result, all of the work done on I_6 - I_8 is thrown away (the $*$ in the diagram indicates that the instruction was ejected from the pipeline). This is a **pipeline flush**.

Branch Prediction Rule Example (20)

```
(I1)        SET     r2, 0x05
(I2)        CMP     r0, r1
(I3)        BRLO    L6
(I4)        ADD     r2, 0x10
(I5)        SUB     r1, 0x06
(I6)  L6:   SUB     r2, 0x01
(I7)        ADD     r0, 0x01
(I8)        ADD     r0, r1
```

I_1	🟩	🟩	🟩	🟩	🟩	🟩		
I_2	A	W	🟩	🟩	🟩	🟩		
I_3	🟥	🟥	R	W	🟩	🟩		
I_4					F	D		
I_5						F		
I_6	🟥	🟥	D	R	$*$			
I_7			F	D	$*$			
I_8				F	$*$			
Time	4	5	6	7	8	9	10	11

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

The pipeline refills starting at I_4.

Branch Prediction Rule Example (21)

(I_1)		SET	r2, 0x05
(I_2)		CMP	r0, r1
(I_3)		BRLO	L6
(I_4)		ADD	r2, 0x10
(I_5)		SUB	r1, 0x06
(I_6)	L6:	SUB	r2, 0x01
(I_7)		ADD	r0, 0x01
(I_8)		ADD	r0, r1

Time	4	5	6	7	8	9	10	11
I_1	🟩	🟩	🟩	🟩	🟩	🟩		
I_2	A	W	🟩	🟩	🟩	🟩		
I_3	🟥	🟥	R	W	🟩	🟩		
I_4					F	D	R	
I_5						F	D	
I_6	🟥	🟥	D	R	$*$		F	
I_7			F	D	$*$			
I_8				F	$*$			

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

At time 10, I_6 is fetched again (since it is next in sequence after I_5.

Branch Prediction Rule Example (22)

```
(I1)        SET     r2, 0x05
(I2)        CMP     r0, r1
(I3)        BRLO    L6
(I4)        ADD     r2, 0x10
(I5)        SUB     r1, 0x06
(I6)  L6:   SUB     r2, 0x01
(I7)        ADD     r0, 0x01
(I8)        ADD     r0, r1
```

Time	4	5	6	7	8	9	10	11
I_1	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
I_2	A	W	🟩	🟩	🟩	🟩	🟩	🟩
I_3	🟥	🟥	R	W	🟩	🟩	🟩	🟩
I_4					F	D	R	A
I_5						F	D	R
I_6	🟥	🟥	D	R	*		F	D
I_7			F	D	*			F
I_8				F	*			

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

Branch Prediction Rule Example (23)

```
(I1)        SET    r2, 0x05
(I2)        CMP    r0, r1
(I3)        BRLO   L6
(I4)        ADD    r2, 0x10
(I5)        SUB    r1, 0x06
(I6)  L6:   SUB    r2, 0x01
(I7)        ADD    r0, 0x01
(I8)        ADD    r0, r1
```

Time	8	9	10	11	12	13	14	15
I_1	🟩	🟩	🟩	🟩	🟩			
I_2	🟩	🟩	🟩	🟩	🟩			
I_3	🟩	🟩	🟩	🟩	🟩			
I_4	F	D	R	A	W			
I_5		F	D	R	A			
I_6	$*$		F	D	🟥			
I_7	$*$			F	🟥			
I_8	$*$							

F = Fetch | D = Decode | R = Read Operand | ML = Mem. Load
A = ALU | W = Write Result | MS = Mem. Store

At time 12, instruction I_6 is stalled by a data hazard with I_4. This creates a structural hazard for I_7.

Branch Prediction Rule Example (24)

(I_1)		SET	r2, 0x05
(I_2)		CMP	r0, r1
(I_3)		BRLO	L6
(I_4)		ADD	r2, 0x10
(I_5)		SUB	r1, 0x06
(I_6)	L6:	SUB	r2, 0x01
(I_7)		ADD	r0, 0x01
(I_8)		ADD	r0, r1

Time	8	9	10	11	12	13	14	15
I_1	🟩	🟩	🟩	🟩	🟩	🟩		
I_2	🟩	🟩	🟩	🟩	🟩	🟩		
I_3	🟩	🟩	🟩	🟩	🟩	🟩		
I_4	F	D	R	A	W	🟩		
I_5		F	D	R	A	W		
I_6	∗		F	D	🟥	R		
I_7	∗			F	🟥	D		
I_8	∗					F		

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

Branch Prediction Rule Example (25)

```
(I1)        SET     r2, 0x05
(I2)        CMP     r0, r1
(I3)        BRLO    L6
(I4)        ADD     r2, 0x10
(I5)        SUB     r1, 0x06
(I6)  L6:   SUB     r2, 0x01
(I7)        ADD     r0, 0x01
(I8)        ADD     r0, r1
```

Time	8	9	10	11	12	13	14	15
I_1	🟩	🟩	🟩	🟩	🟩	🟩	🟩	
I_2	🟩	🟩	🟩	🟩	🟩	🟩	🟩	
I_3	🟩	🟩	🟩	🟩	🟩	🟩	🟩	
I_4	F	D	R	A	W	🟩	🟩	
I_5		F	D	R	A	W	🟩	
I_6	*		F	D	🟥	R	A	
I_7	*			F	🟥	D	R	
I_8	*					F	D	

F = Fetch
A = ALU

D = Decode
W = Write Result

R = Read Operand
MS = Mem. Store

ML = Mem. Load

Branch Prediction Rule Example (26)

```
(I1)        SET    r2, 0x05
(I2)        CMP    r0, r1
(I3)        BRLO   L6
(I4)        ADD    r2, 0x10
(I5)        SUB    r1, 0x06
(I6)  L6:   SUB    r2, 0x01
(I7)        ADD    r0, 0x01
(I8)        ADD    r0, r1
```

Time	8	9	10	11	12	13	14	15
I_1	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
I_2	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
I_3	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
I_4	F	D	R	A	W	🟩	🟩	🟩
I_5		F	D	R	A	W	🟩	🟩
I_6	∗		F	D	🟥	R	A	W
I_7	∗			F	🟥	D	R	A
I_8	∗					F	D	🟥

F = Fetch
A = ALU
D = Decode
W = Write Result
R = Read Operand
MS = Mem. Store
ML = Mem. Load

Similarly, I_8 is stalled at time 15 due to a data hazard with I_7.

Branch Prediction Rule Example (27)

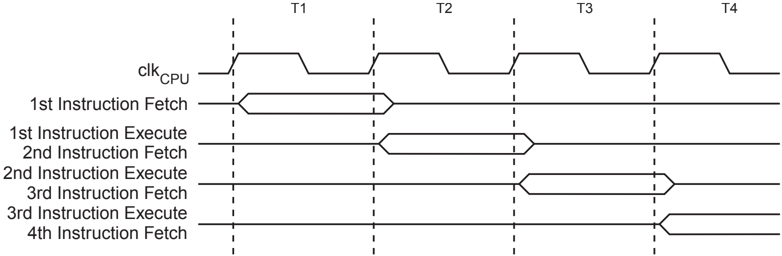
```
(I1)       SET    r2, 0x05
(I2)       CMP    r0, r1
(I3)       BRLO   L6
(I4)       ADD    r2, 0x10
(I5)       SUB    r1, 0x06
(I6) L6:   SUB    r2, 0x01
(I7)       ADD    r0, 0x01
(I8)       ADD    r0, r1
```

Time	12	13	14	15	16	17	18	19
I_1	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
I_2	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
I_3	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
I_4	W	🟩	🟩	🟩	🟩	🟩	🟩	🟩
I_5	A	W	🟩	🟩	🟩	🟩	🟩	🟩
I_6	🟥	R	A	W	🟩	🟩	🟩	🟩
I_7	🟥	D	R	A	W	🟩	🟩	🟩
I_8		F	D	🟥	🟥	R	A	W

F = Fetch
A = ALU
D = Decode
W = Write Result
R = Read Operand
MS = Mem. Store
ML = Mem. Load

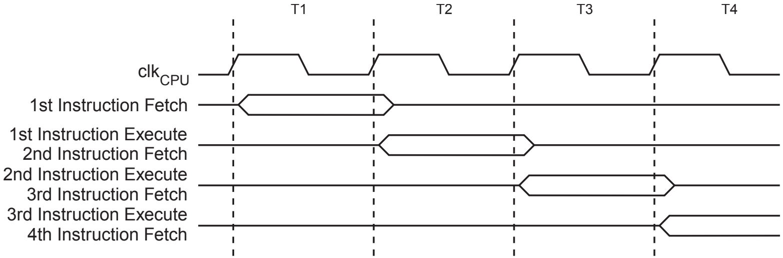
The sequence eventually finishes at time 19.

AVR and Pipelining (1)



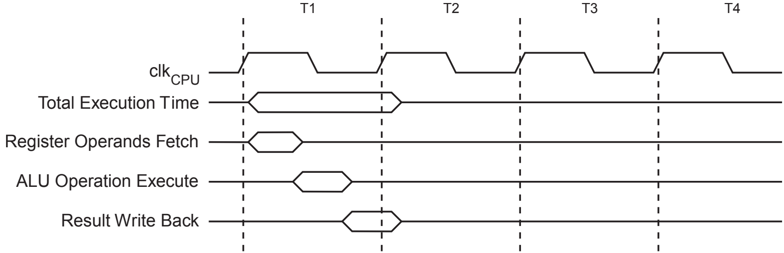
An AVR processor contains several execution stages, but is not considered pipelined, except for a simple two stage Fetch/Execute pipeline.

AVR and Pipelining (2)



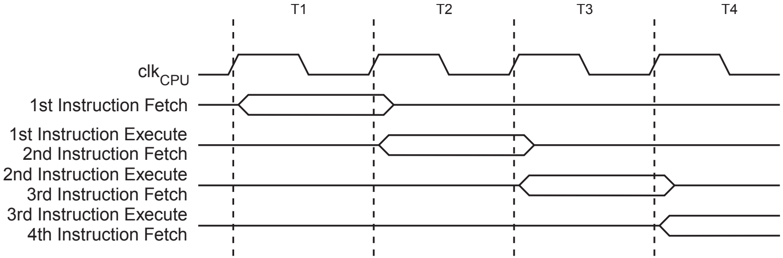
While each instruction is executing, the next instruction is fetched from program memory.

AVR and Pipelining (3)



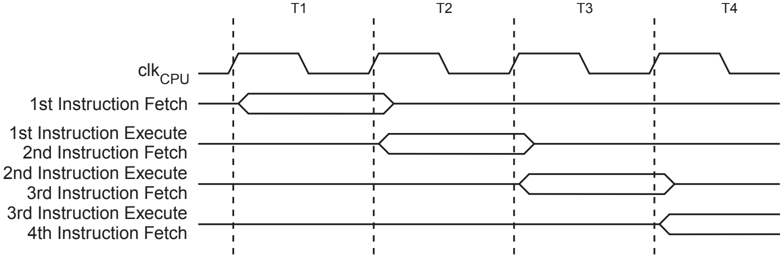
Internally, there is a well-defined set of computation stages for ALU operations, but there is no support for pipelining. This is considered to be one of the shortcomings of the architecture.

AVR and Pipelining (4)



As far as the fetch/execute pipeline is concerned, control hazards are the only possible hazard. If a conditional branch is followed (instead of falling through to the next sequential instruction), the fetch stage is cleared, resulting in a one-cycle delay as the new instruction is loaded.

AVR and Pipelining (5)



This pipeline stall is the reason for the '$1/2$' timing of conditional branch instructions.

AVR and Pipelining (6)

Exercise: Suppose the AVR architecture used a seven-stage pipeline as in the earlier example (Fetch, Decode, Read Operand, Memory Load, ALU, Write Operand, Memory Store), with the memory stages requiring two clock cycles and the other stages requiring one clock cycle. Construct sequences of instructions which produce each of the hazards below.

- Control Hazard
- Data Hazard
- Structural Hazard (as a primary hazard, not a consequence of another hazard)

AVR and Pipelining (7)

Exercise: The AVR architecture does not employ branch prediction logic. However, suppose it did. For each of the proposed rules below, construct two assembly fragments: one for which the rule is almost always successful and one for which the rule is almost never successful.

- Rule 1 Assume that `BRNE` and `BREQ` branches are **never** followed.
- Rule 2 Assume that `BRSH` and `BRLO` branches are **always** followed.

Good Exam Question: Given a fragment of C or assembly code and the description of a branch prediction rule, rewrite the code to optimize the success of the branch prediction

Sources

- Slides by B. Bird, 2017 - 2018.
- The diagrams on slides 136 - 140 are excerpted from the Atmel ATmega2560 datasheet.