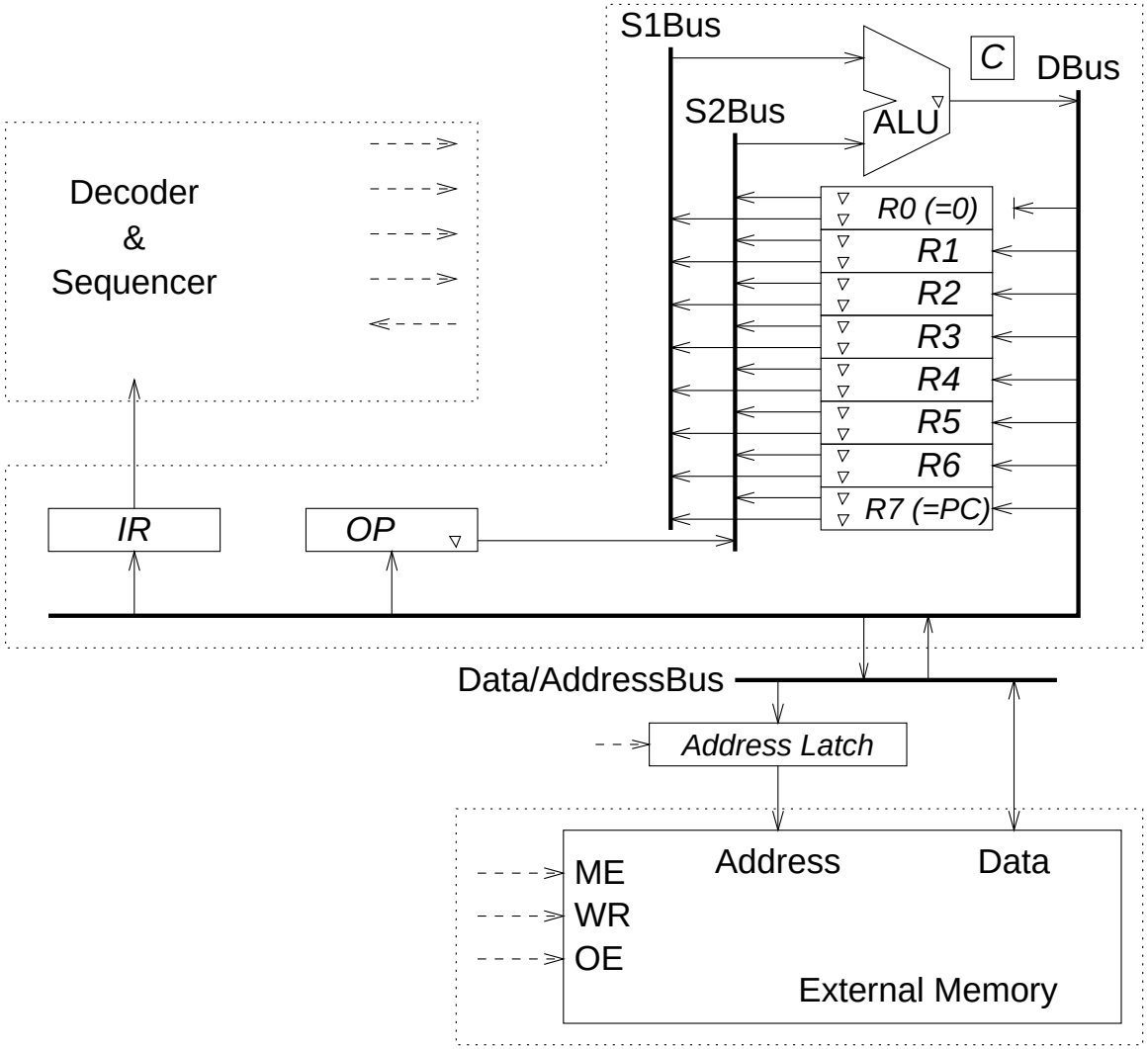
Example RISC



Example RISC

Example Architecture for a 16 bit Processor

- Register Register Architecture - 3 Address Architecture

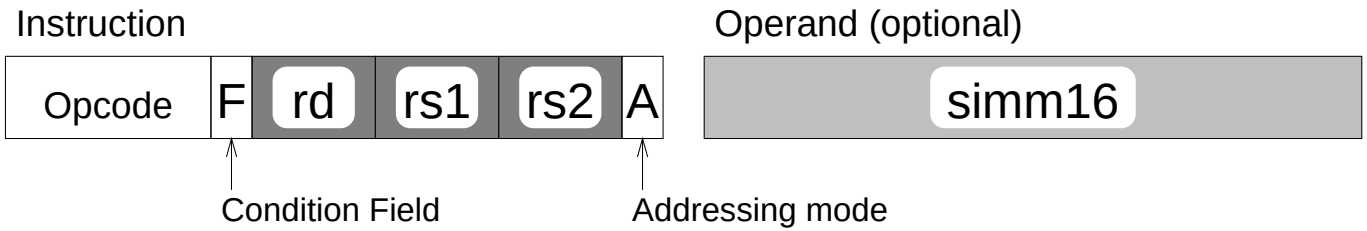
  No arithmetic/logic instructions take operands from external memory. Separate register addresses for each operand and the result are encoded into the instruction.

- Multi function ALU

  ALU is used for data address calculation during a Load/Store instruction or for branch address calculation during a Control Transfer instruction.

- Variable Instruction Length (16/32 bits)

  RISC processors generally support fixed length instructions to simplify pipeline operation. This RISC processor has no pipeline. The use of an optional second instruction word (operand) results in simplified instruction decode.



Opcode Mnemonics

- Arithmetic/Logic Instructions

ADD	ADDX
SUB	SUBX
AND	OR
XOR	SRADD

- Load & Store Instructions

LD	ST

- Control Transfer Instructions

  None[1].

[1]PC is an addressable register (R7) thus we can use Arithmetic/Logic instructions for control transfer

Assembly Language Syntax and Semantics

- Arithmetic/Logic Instructions

ADD Rs1,Rs2,Rd	`ADD R1,R2,R3`	$R3' \leftarrow R1 + R2$

- Load Instruction

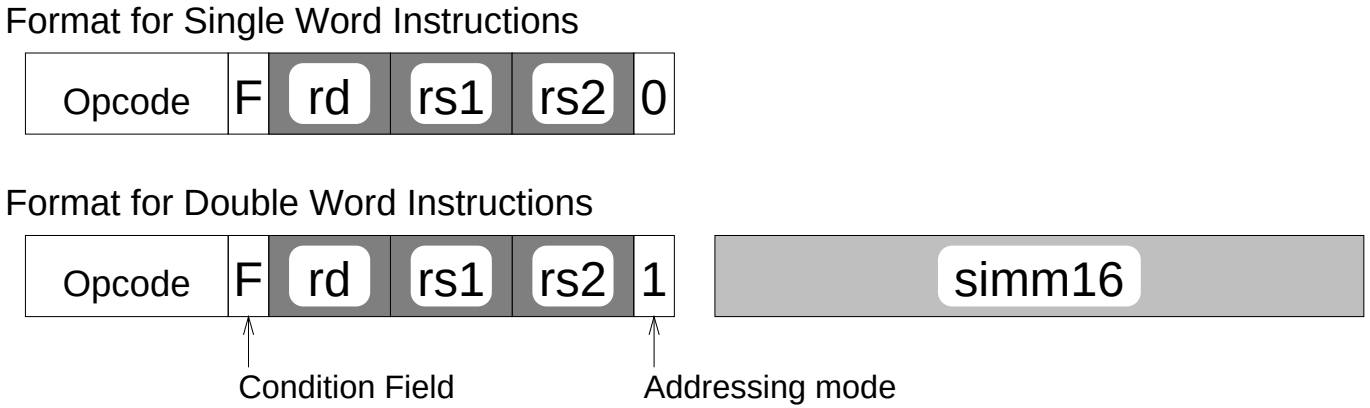
LD [Rs1+Rs2],Rd	`LD [R1+R2],R3`	$R3' \leftarrow mem(R1 + R2)$

- Store Instruction

ST Rd,[Rs1+Rs2]	`ST R3,[R1+R2]`	$mem(R1 + R2)' \leftarrow R3$

Addressing Modes

The **A** field in the instruction specifies whether we will use two registers as operands (**A**=0) or a register and an immediate value (**A**=1).



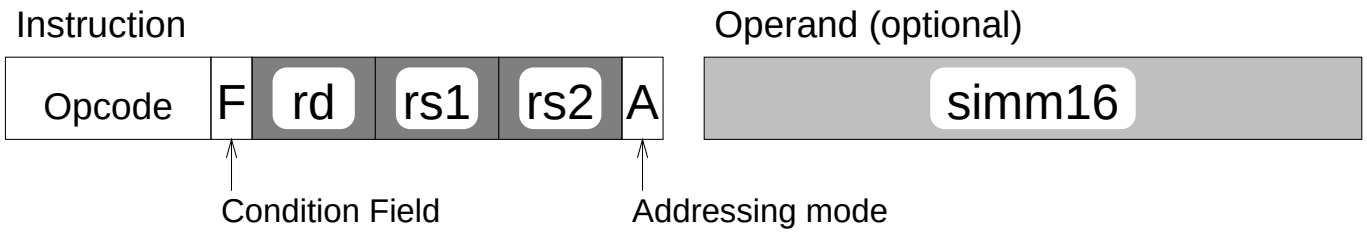
Since the signed immediate, **simm16**, is not needed when **A**=0, the **A** field is also used to decide whether the instruction occupies one or two 16 bit words.

Note that the **rs2** field within the instruction serves no purpose when **A**=1, so the **rs2** field will most likely to be set to zero for instructions which use immediates.

Conditional Instructions

All instructions have a conditional variant.

cADD	cADDX
cSUB	cSUBX
cAND	cOR
cXOR	cSRADD
cLD	cST



The **F** field in the instruction specifies whether the instruction is conditional. If **F**=0 the instruction is always executed. If **F**=1 the instruction is executed only if the carry flag, C[2], is 1.

[2]the C flag will have been set by a previous arithmetic/logic instruction

Example RISC

Thus for each instruction there are four variants dependent on the values of the **A** and **F** fields within the instruction:

Opcode	*Syntax*	*Example*	*Operation*	
ADD	Rs1,Rs2,Rd	`ADD R1,R2,R3`	$R3' \leftarrow R1$ + $R2$	add
ADD	Rs1,simm16,Rd	`ADD R1,1,R3`	$R3' \leftarrow R1$ + 1	add
cADD	Rs1,Rs2,Rd	`cADD R1,R2,R3`	if $C = 1$ then $R3' \leftarrow R1$ + $R2$	conditional add
cADD	Rs1,simm16,Rd	`cADD R1,1,R3`	if $C = 1$ then $R3' \leftarrow R1$ + 1	conditional add

Syntax and Semantics for other Arithmetic/Logic instructions:

Opcode	*Syntax*	*Example*	*Operation*	
ADD	Rs1,Rs2,Rd	`ADD R3,R2,R5`	$R5' \leftarrow R3 + R2$	add
ADDX	Rs1,Rs2,Rd	`ADDX R3,R2,R5`	$R5' \leftarrow R3 + R2 + C$	add with carry
SUB	Rs1,Rs2,Rd	`SUB R3,R2,R5`	$R5' \leftarrow R3 - R2$	subtract
SUBX	Rs1,Rs2,Rd	`SUBX R3,R2,R5`	$R5' \leftarrow R3 - R2 - C$	subtract with borrow
AND	Rs1,Rs2,Rd	`AND R3,R2,R5`	$R5' \leftarrow R3$ & $R2$	bitwise AND
OR	Rs1,Rs2,Rd	`OR R3,R2,R5`	$R5' \leftarrow R3 \mid R2$	bitwise OR
XOR	Rs1,Rs2,Rd	`XOR R3,R2,R5`	$R5' \leftarrow R3$ ^ $R2$	bitwise XOR
SRADD	Rs1,Rs2,Rd	`SRADD R3,R2,R5`	$R5' \leftarrow R3 >> 1 + R2$	shift right with add
ADD	Rs1,simm16,Rd	`ADD R3,63,R5`	$R5' \leftarrow R3 + 63$	
ADDX	Rs1,simm16,Rd	`ADDX R3,63,R5`	$R5' \leftarrow R3 + 63 + C$	
SUB	Rs1,simm16,Rd	`SUB R3,63,R5`	$R5' \leftarrow R3 - 63$	
SUBX	Rs1,simm16,Rd	`SUBX R3,63,R5`	$R5' \leftarrow R3 - 63 - C$	
AND	Rs1,simm16,Rd	`AND R3,63,R5`	$R5' \leftarrow R3$ & 63	
OR	Rs1,simm16,Rd	`OR R3,63,R5`	$R5' \leftarrow R3 \mid 63$	
XOR	Rs1,simm16,Rd	`XOR R3,63,R5`	$R5' \leftarrow R3$ ^ 63	
SRADD	Rs1,simm16,Rd	`AND R3,63,R5`	$R5' \leftarrow R3 >> 1 + 63$	

Example RISC

The inclusion of R7 and R0 add important functionality:

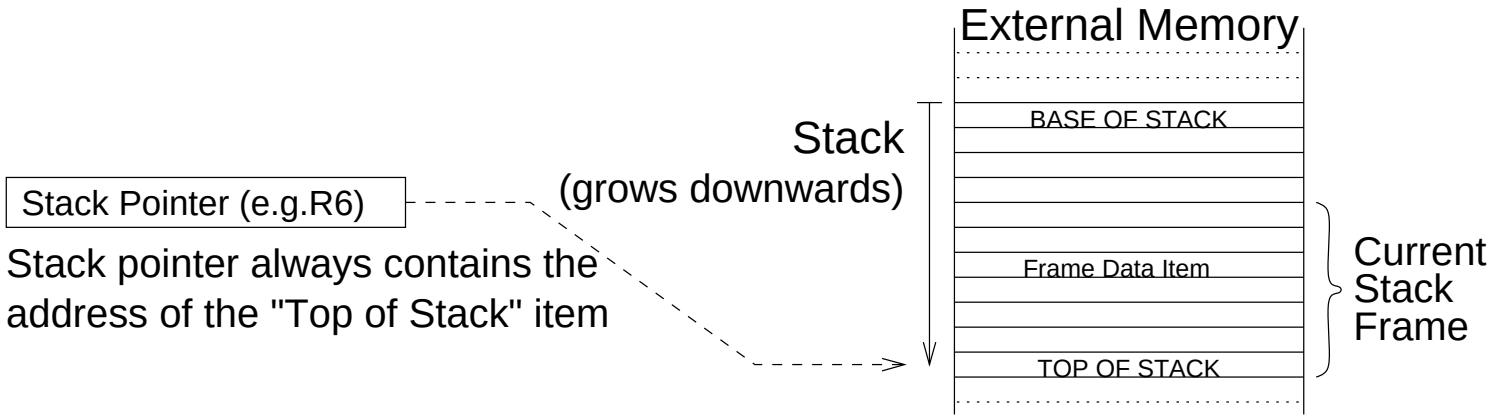
Pseudo-Opcode		*Example*	*Operation*	
BA	simm16	`ADD R7,-35,R7`	$PC' \leftarrow PC$ - 35	branch always
BCS	simm16	`cADD R7,-35,R7`	if $C = 1$ then $PC' \leftarrow PC$ - 35	branch if carry set
MOV	Rs,Rd	`ADD R4,R0,R5`	$R5' \leftarrow R4$	register copy
LDI	simm16,Rd	`ADD R0,72,R5`	$R5' \leftarrow 72$	load immediate
CLR	Rd	`ADD R0,R0,R5`	$R5' \leftarrow 0$	clear
NOT	Rs,Rd	`XOR R4,-1,R5`	$R5' \leftarrow R4$ ^ $FFFF_{16}$ = ~$R4$	bitwise NOT
NEG	Rs,Rd	`SUB R0,R4,R5`	$R5' \leftarrow -R4$	negate
SL	Rs,Rd	`ADD R4,R4,R5`	$R5' \leftarrow R4 + R4 = R4 << 1$	shift left
SR	Rs,Rd	`SRADD R4,R0,R5`	$R5' \leftarrow R4 >> 1$	shift right
TST	Rs1,Rs2	`SUB R4,R3,R0`		test $Rs2 > Rs1$?

Note – for TST pseudo-instruction the result of the subtraction is discarded. We are interested only in the side-effect of setting the C flag.

Example RISC

Stack Support

There is no built in support for a stack. Any general purpose register may be used as the Stack Pointer.



Push R1:

`SUB R6,1,R6`	$SP' \leftarrow SP - 1$
`ST R1,[R6+R0]`	$mem(SP)' \leftarrow R1$

Pop R1:

`LD [R6+R0],R1`	$PC' \leftarrow mem(SP)$
`ADD R6,1,R6`	$SP' \leftarrow SP + 1$

Load R1 from stack frame:

`LD [R6+4],R1`	$PC' \leftarrow mem(SP + 4)$

Subroutines

Subroutine support is rather more complex, and requires the use of a temporary register (in this case R5) for storage of the calculated return address.

Branch to subroutine:

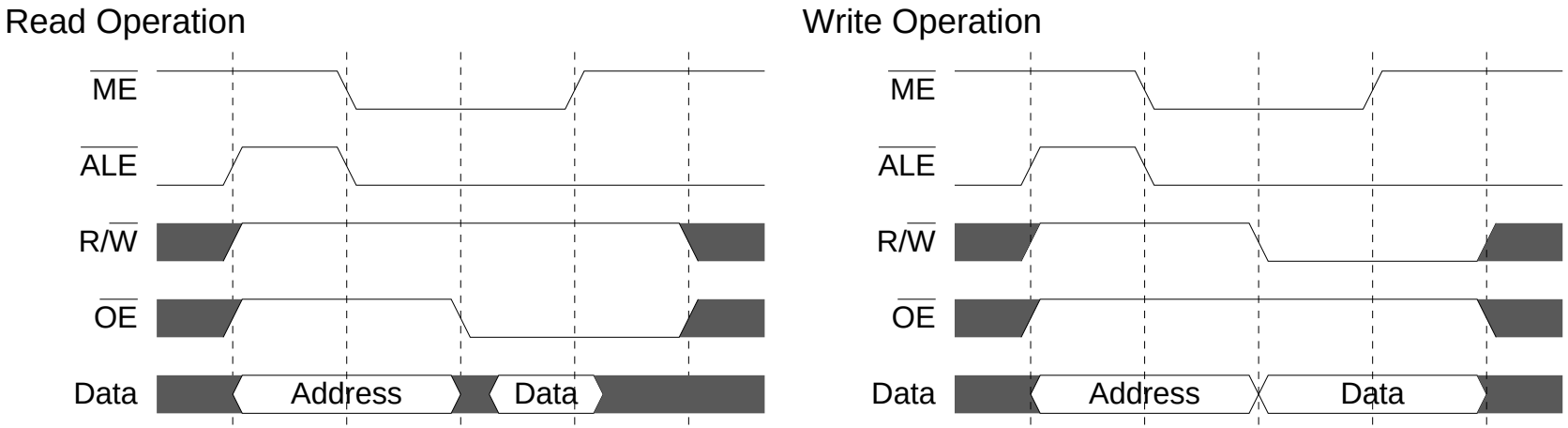
`ADD R7,6,R5`	$TEMP' \leftarrow PC + 6$
`SUB R6,1,R6`	$SP' \leftarrow SP - 1$
`ST R5,[R6+R0]`	$mem(SP)' \leftarrow TEMP$
`ADD R7,offset,R7`	$PC' \leftarrow PC + offset$

Return from subroutine:

`ADD R6,1,R6`	$SP' \leftarrow SP + 1$
`LD [R6-1],R7`	$PC' \leftarrow mem(SP - 1)$

Instruction Timing

The instruction timing is heavily dependent upon the memory cycle required for the external memory:



Load and store instructions last for 8 clock cycles (1 full memory cycles), while arithmetic/logic instructions last for only 5 cycles (1 memory cycles plus one extra clock cycle for result calculation)[3].

[3] All these values will increase by 4 clock cycles (1 memory cycle) for double word instructions (i.e. where **A**=1).

DBus activity and registers updated during 8 (+4) cycle load instruction:

Fetch Instruction:

Address Setup	DBus=PC	
Address Hold	DBus=PC	
Data Setup	DBus=*MemData*	Update IR
Data Hold	DBus=PC+1	Update PC

Fetch Operand: (optional)

Address Setup	DBus=PC	
Address Hold	DBus=PC	
Data Setup	DBus=*MemData*	Update OP
Data Hold	DBus=PC+1	Update PC

Execute (Load instruction):

Address Setup	DBus=Rs1+(Rs2 or simm16)	
Address Hold	DBus=Rs1+(Rs2 or simm16)	
Data Setup	DBus=*MemData*	Update Rd
Data Hold	DBus=0	

For store and arithmetic/logic instructions, the 4 cycles of the instruction fetch and (optional) 4 cycles of operand fetch phase are unchanged. Below are listed only the cycles of the execute phase:

Execute (Store instruction):

Address Setup	DBus=Rs1+(Rs2 or simm16)
Address Hold	DBus=Rs1+(Rs2 or simm16)
Data Setup	DBus=Rd
Data Hold	DBus=Rd

Execute (A/L instruction):

Execute	DBus=*fn*{ Rs1, (Rs2 or simm16) } Update Rd

From the activity on the main DBus we can infer the activity on the other two buses (S1Bus and S2Bus) and the operation of the ALU.
e.g. **DBus=PC** may be achieved by driving **S1Bus=PC** and **S2Bus=R0** and setting the ALU to perform **addition** (addition should probably be the default function for the ALU).

Limitations

- Power

  The instruction set is not very powerful.

  i.e. we may need a large number of instructions to perform relatively simple and common tasks.

- Efficiency

  The instructions are not very efficiently coded.

  The instructions may take too many clock cycles to execute.

Limitations – Power

- Poor support for Conditional Branches

  We have lots of conditional instructions but only one conditional branch (BCS).

  Most simple microprocessors support at least the following four status flags:

  - **C** – Indicates overflow for shift or for unsigned arithmetic
  - **Z** – Indicates a zero result
  - **N** – Indicates a negative result
  - **V** – Indicates overflow for signed arithmetic

  On the basis of which they support a wide variety of conditional branches: BCS, BCC, BE, BNE, ... BLEU ....

- Poor support for Stack and for Subroutines

  CISC machines will normally support a dedicated stack pointer.

  RISC machines tend to dedicate a register to store the return address. It is up to the subroutine to stack this value before a nested subroutine call.

Limitations – Efficiency

- The instructions are not very efficiently coded.

  Most RISC machines tend to use shorter fixed length instructions making use of the fact that most immediate values are short. This strategy is supported by variable instruction fields.

- The instructions may take too many clock cycles to execute.

  Adding other functional units and/or modifying the bus architecture may allow more efficient use of the valuable DBus cycles.