b16: Modern Processor Core

Bernd Paysan

April 29, 2005

Outline

Motivation

- Design complexity reduction in SoC design: Use a CPU instead of state machines
- Experience with 8 bit CPUs: too limited, to clumsy to program, slow or expensive (license+space).
- Minimalistic 16 bit CPU with Forth-like programming language as modern successor
- b16-dsp is inspired by Chuck Moore's c18
- b16 is the shrink version for simpler tasks

Motivation

- Design complexity reduction in SoC design: Use a CPU instead of state machines
- Experience with 8 bit CPUs: too limited, to clumsy to program, slow or expensive (license+space).
- Minimalistic 16 bit CPU with Forth-like programming language as modern successor
- b16-dsp is inspired by Chuck Moore's c18
- b16 is the shrink version for simpler tasks

Motivation

- Design complexity reduction in SoC design: Use a CPU instead of state machines
- Experience with 8 bit CPUs: too limited, to clumsy to program, slow or expensive (license+space).
- Minimalistic 16 bit CPU with Forth-like programming language as modern successor
- b16-dsp is inspired by Chuck Moore's c18
- b16 is the shrink version for simpler tasks

Motivation

- Design complexity reduction in SoC design: Use a CPU instead of state machines
- Experience with 8 bit CPUs: too limited, to clumsy to program, slow or expensive (license+space).
- Minimalistic 16 bit CPU with Forth-like programming language as modern successor
- b16-dsp is inspired by Chuck Moore's c18
- b16 is the shrink version for simpler tasks

Motivation

- Design complexity reduction in SoC design: Use a CPU instead of state machines
- Experience with 8 bit CPUs: too limited, to clumsy to program, slow or expensive (license+space).
- Minimalistic 16 bit CPU with Forth-like programming language as modern successor
- b16-dsp is inspired by Chuck Moore's c18
- b16 is the shrink version for simpler tasks

What's Forth

Elements:

Data stack passes arguments to subroutines and back ("words"), and primitive instructions (consequence: RPN)

Return stack separates return addresses from data

Word space delimited "symbol" in the source, subroutine or instruction in code

Special symbols ':' colon defines subroutine, ';' semi-colon returns to caller, '@' fetch/load, '!' store

What's Forth

Elements:

Data stack passes arguments to subroutines and back ("words"), and primitive instructions (consequence: RPN)

Return stack separates return addresses from data

Word space delimited "symbol" in the source, subroutine or instruction in code

Special symbols ':' colon defines subroutine, ';' semi-colon returns to caller, '@' fetch/load, '!' store

What's Forth

Elements:

Data stack passes arguments to subroutines and back ("words"), and primitive instructions (consequence: RPN)

Return stack separates return addresses from data

Word space delimited "symbol" in the source, subroutine or instruction in code

Special symbols ':' colon defines subroutine, ';' semi-colon returns to caller, '@' fetch/load, '!' store

What's Forth

Elements:

Data stack passes arguments to subroutines and back ("words"), and primitive instructions (consequence: RPN)

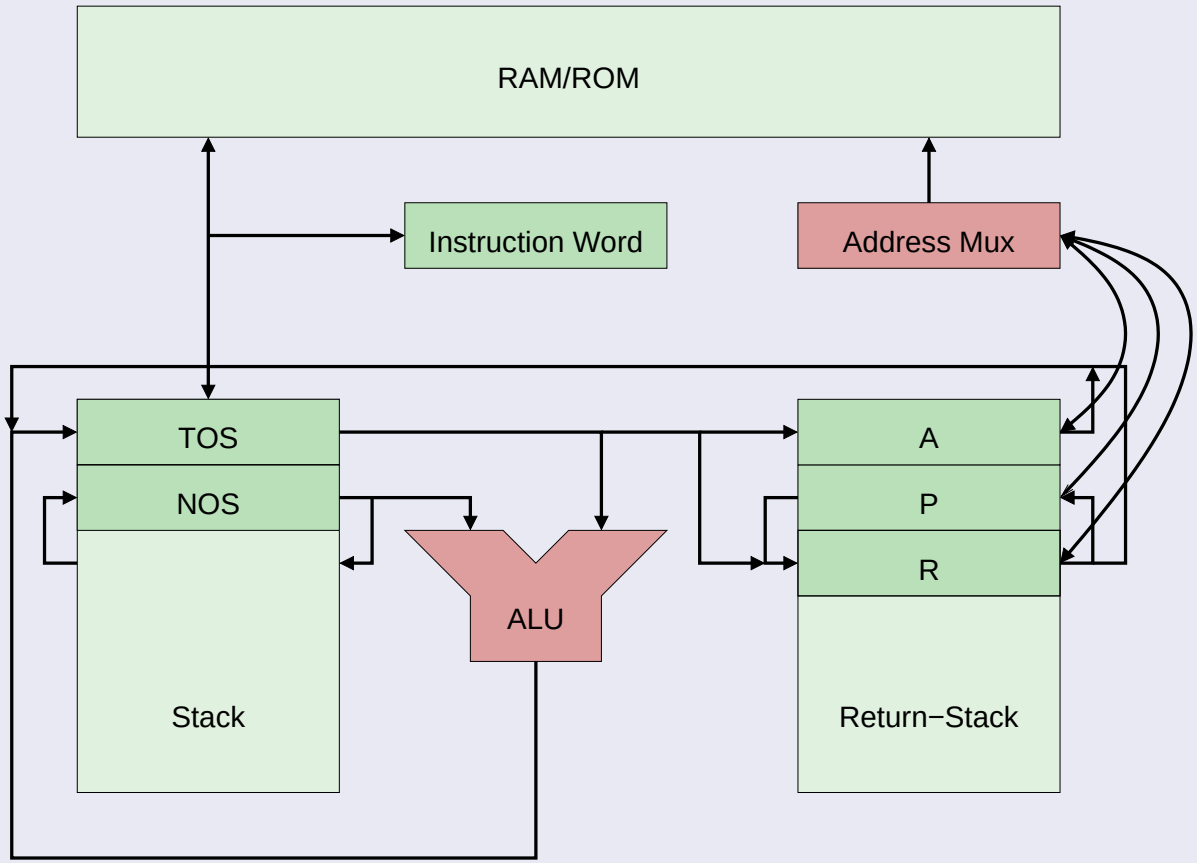
Return stack separates return addresses from data

Word space delimited "symbol" in the source, subroutine or instruction in code

Special symbols ':' colon defines subroutine, ';' semi-colon returns to caller, '@' fetch/load, '!' store

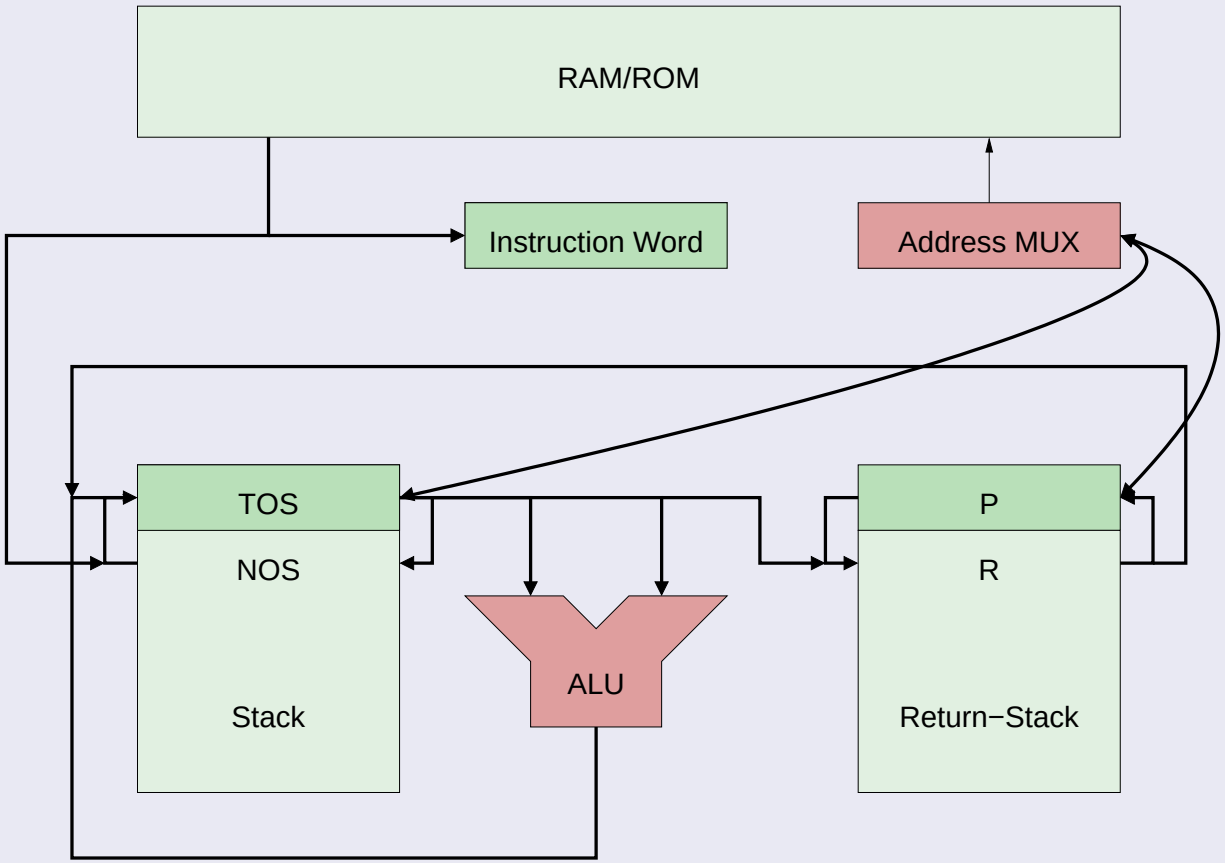
b16-dsp Block Diagram

b16-dsp Block Diagram



b16 Block Diagram

b16 Block Diagram



Differences

Benchmarks

	b16-dsp	*b16*	*8051*
size XC035	0.3mm^2	0.16mm^2	1mm^2
max. speed	~200MHz	~150MHz	~30MHz, 2CPI
DSP operations	Mul-step	—	8 bit mul
	Div-step	—	—
address regs	2	1	1+

b16-dsp Instruction Set

b16 Instruction Set

	0	1	2	3	4	5	6	7	
0	nop	call	jmp	ret	jz	jnz	jc	jnc	
	nop	exec	goto	ret	gz	gnz	gc	gnc	*slot 3*
8	xor	com	and	or	+	+c	*+	/-	
10	A!+	A@+	R@+	lit	Ac!+	Ac@+	Rc@+	litc	
	A!	A@	R@	lit	Ac!	Ac@	Rc@	litc	*slot 1*
18	nip	drop	over	dup	>r	>A	r>	A>	

b16 Instruction Set

b16-small Instruction Set

	0	1	2	3	4	5	6	7	
0	nop	call	jmp	ret	jz	jnz	jc	jnc	
	nop	exec	goto	ret	gz	gnz	gc	gnc	*slot 3*
8	xor	com	and	or	+	+c	2/	c2/	
10	!+	@+	@	lit	c!+	c@+	c@	litc	
	!.	@.	@	lit	c!.	c@.	c@	litc	*slot 1*
18	nip	drop	over	dup	>r	nop	r>	nop	

Jump Instructions

Jump Instructions

nop (—)

call (— r:P) $\mathrm{P} \leftarrow jmp$; $\mathrm{c} \leftarrow 0$

jmp (—) $\mathrm{P} \leftarrow jmp$

ret (r:a —) $\mathrm{P} \leftarrow a \wedge \$\mathrm{FFFE}$; $\mathrm{c} \leftarrow a \wedge 1$

jz (n —) **if**$(n = 0)\,\mathrm{P} \leftarrow jmp$

jnz (n —) **if**$(n \neq 0)\,\mathrm{P} \leftarrow jmp$

jc (x —) **if**$(c)\,\mathrm{P} \leftarrow jmp$

jnc (x —) **if**$(c = 0)\,\mathrm{P} \leftarrow jmp$

ALU Operations b16-dsp

ALU Instructions b16-dsp

xor (a b — r) $r \leftarrow a \oplus b$

com (a — r) $r \leftarrow a \oplus \$FFFF, c \leftarrow 1$

and (a b — r) $r \leftarrow a \wedge b$

or (a b — r) $r \leftarrow a \vee b$

+ (a b — r) $c, r \leftarrow a + b$

+c (a b — r) $c, r \leftarrow a + b + c$

*+ (a b — a r) **if**$(c)\ c_n, r \leftarrow a + b$ **else** $c_n, r \leftarrow 0, b$;
$r, A, c \leftarrow c_n, r, A$

/– (a b — a r) $c_n, r_n \leftarrow a + b + 1$; **if**$(c \vee c_n)\ r \leftarrow r_n$;
$c, r, A \leftarrow r, A, c \vee c_n$

ALU Operations b16

ALU Instructions b16

xor (a b — r) $r \leftarrow a \oplus b$

com (a — r) $r \leftarrow a \oplus \$FFFF$, c $\leftarrow 1$

and (a b — r) $r \leftarrow a \wedge b$

or (a b — r) $r \leftarrow a \vee b$

+ (a b — r) c, $r \leftarrow a + b$

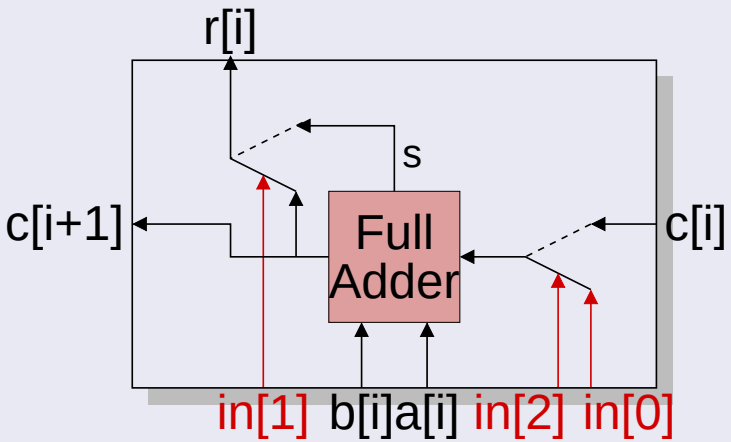
+c (a b — r) c, $r \leftarrow a + b +$ c

2/ (a — r) r, c $\leftarrow r[15], r$

c2/ (a — r) r, c $\leftarrow c, r$

ALU Implementation

ALU element



Memory Instructions b16-dsp

Memory Instructions b16-dsp

A!+	(n —)	$mem[\mathrm{A}] \leftarrow n;\ \mathrm{A} \leftarrow \mathrm{A} + 2$
A@+	(— n)	$n \leftarrow mem[\mathrm{A}];\ \mathrm{A} \leftarrow \mathrm{A} + 2$
R@+	(— n)	$n \leftarrow mem[\mathrm{R}];\ \mathrm{R} \leftarrow \mathrm{R} + 2$
lit	(— n)	$n \leftarrow mem[\mathrm{P}];\ \mathrm{P} \leftarrow \mathrm{P} + 2$
Ac!+	(c —)	$mem.b[\mathrm{A}] \leftarrow c;\ \mathrm{A} \leftarrow \mathrm{A} + 1$
Ac@+	(— c)	$c \leftarrow mem.b[\mathrm{A}];\ \mathrm{A} \leftarrow \mathrm{A} + 1$
Rc@+	(— c)	$c \leftarrow mem.b[\mathrm{R}];\ \mathrm{R} \leftarrow \mathrm{R} + 1$
litc	(— c)	$c \leftarrow mem.b[\mathrm{P}];\ \mathrm{P} \leftarrow \mathrm{P} + 1$

Memory Instructions b16

Memory Instructions b16

!+	(n A — A')	$mem[A] \leftarrow n;\ A' \leftarrow A + 2$
@+	(A — n A')	$n \leftarrow mem[A];\ A' \leftarrow A + 2$
@	(A — n)	$n \leftarrow mem[A];$
lit	(— n)	$n \leftarrow mem[P];\ P \leftarrow P + 2$
c!+	(c A — A')	$mem.b[A] \leftarrow c;\ A' \leftarrow A + 1$
c@+	(A — c A')	$c \leftarrow mem.b[A];\ A' \leftarrow A + 1$
c@	(A — c)	$c \leftarrow mem.b[A];$
litc	(— c)	$c \leftarrow mem.b[P];\ P \leftarrow P + 1$

Stack Instructions b16-dsp

Stack Instructions b16-dsp

nip	(a b — b)
drop	(a —)
over	(a b — a b a)
dup	(a — a a)
>r	(a — r:a)
>A	(a —) $\mathrm{A} \leftarrow \mathrm{a}$
r>	(r:a — a)
A>	(— a) $\mathrm{a} \leftarrow \mathrm{A}$

Stack Instructions b16

Stack Instructions b16

nip	(a b — b)
drop	(a —)
over	(a b — a b a)
dup	(a — a a)
>r	(a — r:a)
r>	(r:a — a)

Code Examples (b16)

Example: Time Setting with constant inline argument

```
: TIMER! ( #time -- )  r> @+ >r TIMER # ! ;
```

Example: 3 of 4 redundancy, using all available stack space

```
: 3of4 ( #src #dest -- )
   r> @+ >r
   @+ @+ @+ @+ >r xor >r xor r> and
   r> -6 # + @+ @+ @+ >r xor swap
   r> -8 # + @. >r xor and or
   r> @+ @+ >r over xor dup >r com and
   r> r> @ and or swap ?err
   r> @+ >r ! ;
```

Code Examples (b16)

Example: Time Setting with constant inline argument

```
: TIMER! ( #time -- )  r> @+ >r TIMER # ! ;
```

Example: 3 of 4 redundancy, using all available stack space

```
: 3of4 ( #src #dest -- )
   r> @+ >r
   @+ @+ @+ @+ >r xor >r xor r> and
   r> -6 # + @+ @+ @+ >r xor swap
   r> -8 # + @. >r xor and or
   r> @+ @+ >r over xor dup >r com and
   r> r> @ and or swap ?err
   r> @+ >r ! ;
```

Tools

- Assembler
- Umbilical debugger (download and run code)
- Software-Simulator

Tools

- Assembler
- Umbilical debugger (download and run code)
- Software-Simulator

Tools

- Assembler
- Umbilical debugger (download and run code)
- Software-Simulator