Appendix A

UTSA Instruction Set Summary

This appendix provides the UTSA (University of Teesside Stack Architecture) instruction set summary with explanations of UTSA operands. This appendix provides explanation for all instructions that have been included in this thesis.

UTSA INSTRUCTION SET SUMMARY

Version 1.24

25 May, 2006

Computer Architecture Research Group
(C.A.R.G.)
School of Science & Technology
University of Teesside
Middlesborough, TS1 3BA.

FAX: (+44) 642 342401

CORE ARCHITECTURE BASIC FEATURES.

32 BIT Arithmetic Logic Unit (ALU)

- 32 bit Integer arithmetic supported.
- Single cycle multiply & divide.
- Logical & comparative operations.
- Shift & manipulate functions.

MULTIPLE MACHINE STACKS.

- Data stack, with four dedicated top of stack registers.
- Return stack, for program flow, & auxiliary use.
- Parameter stack, with offset addressing of locals and variables.
- On chip buffering of DATA & RETURN stacks.

PARALLEL EXECUTION.

- Program flow executes concurrently with other operations.
- Program flow co-processor concept.

MEMORY MODEL.

- Global data range 16Mwords (64Mbytes) = 24-bit aligned Extended address.
- Paged data range 128Kwords (512Kbytes) = 17 bit aligned Short address.
- Global code Space 16Mwords (64Mbytes) = 24-bit aligned Extended address.
- Paged code range 128Kwords (512Kbytes) = 17 bit aligned Short address.

PAGING

The machine supports both local and global memory access. Global memory access occurs when a full range address is supplied (in this case a 24-bit extended address).

- **CODE PAGE.**

The code page register is simply part of the PC which has significance in its own right. Its hence automatically updated when a program flow operation vectors an address outside the current code page, there is no software requirement to manage code paging, and the code page register is not visible separately to the user.

- **DATA PAGE.**

The data page register can operate in two modes:-

In **"floating page"** mode the data page value is supplied from the code page register, so that short data accesses refer to address within the current code page, and automatically follow code page changes.

For the **"Static page"** mode, the data page is indicated by the contents of the data page register which can, however, be modified by the software explicitly if and when necessary.

MACHINE REGISTERS.

TOS Top of (data) stack register.
NOS Next on (data) stack register.
3OS Third on stack.
4OS Fourth on stack.

SP Data stack memory pointer.
RP Return stack memory pointer.
FP Frame stack memory pointer.
ZPR Zero Page Register.
DPR Data page register.
PSW/ICR Processor Status Word. & Interrupt Control Register.

XP User address pointer XP.
YP User address pointer YP.
PC Program Counter.

DEVICE PIN ALLOCATION.

D00-D31	32 pins	CPU Data bus.
A0-A23	24 pins	CPU Address bus word aligned.
R/W	1 pin	Read/Write.
CLK	1 or 2 pin	System clock(s).
FC0-FC1	2 pins	Memory address function, 00=data, 01=program,10=stacks,11=undefined
RDY	1 pin	Memory access valid, use for external wait state.&cache schemes.

External memory tells cpu when data ready, either from ram, with a latency of several internal cpu cycles, or from cache, waiting perhaps 1 cycle..

RESET	1 pin.	
NMI-0	1 pin.	Non mask int 0.
NMI-1	1 pin.	Non mask int 1.
INT0-4	4 pins.	Maskable int 0-4.
TOTAL	< 68 pins, remaining, 14 pins. (assuming 80 pin package);	

INSTRUCTION SET DETAILS.

ARITHMETIC OPERATIONS.

add	full-word addition,	tos = tos + nos.
sub	full-word subtract,	tos = tos - nos.
hmul	half-word single cycle multiply.	tos = tos * nos
hdiv	half-word single cycle divide.	tos = tos / nos
mul	full-word multiply	tos = tos * nos
div	full-word divide.	tos = tos / nos
xp++	increment xp.	
yp++	increment yp.	
tos++	increment tos.	
xp--	decrement xp.	
yp--	decrement yp	

tos--	decrement tos.

LOGICAL.

and	32 bit logical and	
or	32 bit logical or	
not	32 bit invert.	
xor	32 bit exclusive or.	
lsl	logic shift left, full-word.	msb filled with zeros
lsr	logic shift right, full-word.	lsb filled with zeros
rol	rotate left, full-word.	msb filled with old lsb
ror	rotate right, full-word.	lsb filled with old msb

WORD MANIPULATION.

rbyte	rotate right by one byte.	[a][b][c][d] >>> [d][a][b][c]
rhwd	rotate right by half word.	[a][b] >>> [b][a]
xbyte_n	extract byte n.	[a][b][c][d] >>> [0][0][0][n]
xwrd _n	extract word n	[a][b] >>> [0][n]
ibyte _n	insert byte	[x] >> [a][b][c][d] >>>> [a][x][c][d]
iwrd _n	insert word	[xx] >> [a][b] >>>> [xx][b]

STACK & REGISTER MANIPULATION, EXTENDED FOR SCHEDULING OPERATIONS.

		(tos) **(tos')**
copy1	equivalent to **dup.**	n4 n3 n2 n1 --- n4 n3 n2 n1 **n1**
copy2	equivalent to **over.**	n4 n3 n2 n1 --- n4 n3 n2 n1 **n2**
copy3	equivalent to **2 pick.**	n4 n3 n2 n1 --- n4 n3 n2 n1 **n3**
copy4	equivalent to **3 pick.**	n4 n3 n2 n1 --- n4 n3 n2 n1 **n4**
drop1	equivalent to **drop.**	
drop2	equivalent to **nip.**	
drop3	drop 3rd data stack item.	
drop4	drop 4th item on stack.	
rsu2	**swap.** (rotate stack upward, top two values only)	
rsu3	**rot.** (rotate stack, top three values affected)	

rsu4	**3 roll.** (rotate stack, top four values only)	
rsd3	**-rot** (reverse rotate stack, top three values)	
rsd4	**3 -roll.** (reverse rotate stack , top four values)	
>r	push to return stack.	[tos] >>> [tor]
<r	pop from return stack.	[tos] <<< [tor]
xp>r	push xp on return stack	[xp] > [return stack]
yp>r	push yp on return stack.	[yp] > [return stack]
xp<r	pop xp from return stack.	[xp] < [return stack]
yp<r	pop yp from return stack.	[yp] < [return stack]
>**xp**	push through xp	[tos] > [xp] > [return stack]
>**yp**	push through yp	[tos] > [yp] > [return stack]
<**xp**	pop through xp	[tos] < [xp] < [return stack]
<**yp**	pop through yp	[tos] < [yp] < [return stack]

The 'push through' operators for XP and YP will permit the XP and YP registers to be used as if they are the top of return stack register, as a value is pushed into XP for example, its own contents are pushed onto the return stack. Popping through XY has the reverse effect.

MISCELLEANEOUS

@sp	fetch data stack pointer.
@rp	fetch return stack pointer.
@fp	fetch frame stack pointer.
@xp	fetch user pointer xp.
@yp	fetch user pointer yp.
@icr	fetch interrupt control word.
@psw	fetch processor status word.
	(icr and psw may be combined in a single register).
!sp	store data stack pointer.
!rp	store return stack pointer.
!fp	store frame stack pointer.
!xp	store user pointer xp.
!yp	store user pointer yp.
!icr	store interrupt control word.

!psw store processor status word.

CONTROL & COMPARE.

The following operators are destructive tests, removing the operand, or operands from the stack, performing a "Test And Drop", before placing the flag back on the stack, for destructive evaluation by the program flow conditionals.

teq	test for equality,	tos = nos.	?
tne	test for inequality,	tos ≠ nos.	?
tgt	test for greater than,	tos > nos.	?
tlt	test for less than,	nos > tos.	?
tnv	test for negative.	tos < 0	?
tpv	test for positive.	tos >= 0	?
tnz	test for not zero.	tos <> 0	?
txpz	test for xp zero	xp = 0	?
typz	text for yp zero	yp = 0	?

Each condition could be inverted by applying NOT to the flag left on stack after a test. So for example, "less than or equal to", could be tested by:- "TGT, NOT". However if space permits, and need dictates, then we could have TNGT, TNP, and so-on. **TXPZ and TYPZ** are useful to test values in XP and YP when used as counters. For example, a bock move of 10 items would appear thus:-

```
loop:  @[xp--]
       ![yp--]
       txpz
       bcr     loop
```

Otherwise looping on XP or YP would require xp--, @xp, teq, bcr.

Unconditional.

bp	addr.	branch to page relative offset address.
bl	addr.	branch to absolute long address.

cp addr. call to page relative offset. (unconditional). addr = 17 bit.

cl addr. call to long absolute address. (unconditional). addr = 24 bit.

zpc vector. call to zero page code block. (unconditional). vector = 7bit.

exit unconditional return from call.

Conditional.

bcr addr. conditional branch to short pc relative offset. (taken when not zero).

bcp addr. conditional branch to paged offset address. (taken when not zero). branch is assumed to fall through.

bcl addr. conditional branch to long absolute address. (taken when not zero). branch is assumed to fall through.

ccp addr. cond. call to page relative offset. (taken if not zero). addr = 17 bit.

ccl addr. cond. call to long absolute address. (taken if not zero). addr = 24 bit.

?**exit** conditional return, (taken if not zero).

Program flow may be reviewed soon to attempt to simplify the requirements, after simulation has given us a better view of program needs. The compiler should implement BL,CL,BCL,CCL only. Later, the assembler will calculate the best instruction to use based upon the UTSA binary mapping.

MEMORY & DATA ACCESS.

lit	num	load 8 bit literal.
lsi	num	load short immediate (literal 17 bits).
lei	num	load extended immediate (literal 24 bits).
@loc	[n]	fetch local, at address fp+n.
!loc	[n]	store local at address fp+n.
ldp	addr	load from page relative address.
ldl	addr	load from absolute address.
stp	addr	store data to paged address.
stl	addr	store data to absolute address.
![tos]		store (tos = address, nos = data to be stored)
!		store (nos = address, tos = data to be stored)
@[tos]		fetch(tos = address of data to be fetched to stack), the same as @
@[xp]		load mem contents addressed by xp.

@[yp]	load mem contents addressed by yp.
![xp]	store mem contents addressed by xp.
![yp]	store mem contents addressed by yp.
@[xp++]	indexed load with post increment.
@[xp- -]	indexed load with post decrement.
@[yp++]	indexed load with post increment.
@[yp- -]	indexed load with post decrement.
![xp++]	indexed store with post increment.
![xp- -]	indexed store with post decrement.
![yp++]	indexed store with post increment.
![yp- -]	indexed store with post decrement.

GENERAL

sp+	sp = sp + tos
rp+	rp = rp + tos
yp+	yp = yp + tos
xp+	xp = xp + tos
fp+	fp = fp + tos
sp-	sp = sp - tos
rp-	rp = rp - tos
yp-	yp = yp - tos
xp-	xp = xp - tos
fp-	fp = fp - tos
sp++ , sp--	increment (by ++) or decrement (by --) specified register by one.
rp++ , rp--	increment (by ++) or decrement (by --) specified register by one.
tos++ , tos--	increment (by ++) or decrement (by --) specified register by one.
fp++ , fp--	increment (by ++) or decrement (by --) specified register by one.
xp++ , xp--	increment (by ++) or decrement (by --) specified register by one.
yp++ , yp--	increment (by ++) or decrement (by --) specified register by one.

PROGRAM CONTROL STRATEGY

The previous section introduced a set of operators for control of program activity, including the "TEST" operators, and the conditional and unconditional program flow

operators such as CALL, and BRANCH. In order to maximise the efficiency of the UTSA architectures performance, we require an advanced approach to branch and call handling, in order to prevent, or at least minimise branch penalties.

The UTSA assumes that some branches are taken, whilst others are assumed to be not taken, but this is extended to calls and other code features of the UTSA, as listed below.

UTSA BRANCH STRATEGY.

unconditional.

	bp, bl	always overrides the currently intended instruction fetch.
	zpc,cp,cl	always overrides existing instruction fetch.
	exit	always overrides instruction fetch.

Conditional.

	bcp,bcl,	assumed to be taken for a "backward reference", assumed to fall through for "forward reference".
	bcr	assumed to be taken for backward references, assumed to fall through for forward reference.
**	**ccp,ccl**	always assumed to be taken.
**	**?exit**	always assumed to fall through, since it would be used to exit upon a terminating condition, which is less likely than the condition for continued procedure execution.

** Branch strategy for these items have not been fully investigated by us, and may change in light of research on simulation and program behaviour.

Absolute addressing for branch addresses allows fast issue of non-sequential instruction fetches, immediately after the instruction word becomes available for decoding, but has the disadvantage that a branch prediction strategy must be based upon the determination of a forward or backward reference. This would involve a computation with respect to PC, to evaluate the branch destination in terms of a forward or backward reference but may be implemented as a simple magnitude comparator. This introduces a latency which increases the overall conditional branch latency, and may make fast branch issue more difficult.

INSTRUCTION SET LISTING

add	**sub**	**hmul**	**hdiv**	**mul**	**div**			
xp++	**yp++**	**tos++**	**xp--**	**yp--**	**tos—**			
and	**or**	**not**	**xor**	**lsl**	**lsr**			
rol	**ror**	**rbyte**	**rhwd**	**xbyte_n**	**xwrd _n**	**ibyte _n**	**iwrd _n**	
copy1		**copy2**		**copy3**	**copy4**			
drop1		**drop2**		**drop3**	**drop4**			
rsu2		**rsu3**		**rsu4**	**rsd3**	**rsd4**	**>r**	**<r**
xp>r		**yp>r**		**xp<r**	**yp<r**			
>xp		**>yp**		**<xp**	**<yp**			
@sp		**rp**		**@fp**	**@xp**	**@yp**		
@icr		**@psw**		**!sp**	**!rp**	**!fp**		
!xp		**!yp**		**!icr**	**!psw**			
teq		**tne**		**tgt**	**tlt**	**tnv**		
tpv		**tnz**		**txpz**	**typz**			
bp		**bl**		**cp**	**cl**	**zpc**	**exit**	
bcp		**bcl**		**ccp**	**ccl**	**bcr**	**?exit**	
lit		**lsi**		**.lei**	**.**	**@loc**	**!loc**	
ldp		**ldl**		**sdp**	**sdl**			
@[xp]		**@[yp]**		**![xp]**	**![yp]**	**@[xp++]**	**@[xp]**	
@[yp++]		**@[yp- -]**		**![xp++]**	**![xp- -]**	**![yp++]**	**![yp- -]**	
![tos]		**@[tos]**						
sp+		**rp+**		**fp+**	**xp+**	**yp+**		
sp-		**rp-**		**fp-**	**xp-**	**yp-**		
sp++		**rp++**		**fp++**	**xp++**	**yp++**		
sp--		**rp--**		**fp--**	**xp--**	**yp--**		

NOTE: due to the limitation of the block code execution result viewer introduced in Section 4.2, in the graphic view of each block execution dependency result, the ‘-’ symbol is replaced by the ‘*’ symbol. For example, ‘fp-’ is replaced by ‘fp*’.