COMP22111: Processor Microarchitecture

Stump assembler mnemonics

Listed are the instructions that can be interpreted by the Stump assembler.

Data operations

Stump has 6 different data operations that can be Type 1 or Type 2. An instruction mnemonic appended with S will update the condition code register.

Type 1

```
ADD{S}  <dest>, <srcA>, <srcB> {, shift}
   - dest := (Shifted)srcA + srcB

ADC{S}  <dest>, <srcA>, <srcB> {, shift}
   - dest := (Shifted)srcA + srcB + 1

SUB{S}  <dest>, <srcA>, <srcB> {, shift}
   - dest := (Shifted)srcA - srcB

SBC{S}  <dest>, <RsrcA>, <srcB> {, shift}
   - dest := (Shifted)srcA - srcB - 1 + C

AND{S}  <dest>, <RsrcA>, <srcB> {, shift}
   - dest := (Shifted)srcA AND srcB

OR{S}   <dest>, <RsrcA>, <srcB> {, shift}
   - dest := (Shifted)srcA OR srcB
```

where {, shift} indicates where a shift operation ASR, ROR, or RRC can be appended.

Type 2

```
ADD{S}  <dest>, <srcA>, #<imme>
   - dest := srcA + #<imme>

ADC{S}  <dest>, <srcA>, #<imme>
   - dest := srcA + #<imme> + 1

SUB{S}  <dest>, <srcA>, #<imme>
   - dest := srcA - #<imme>

SBC{S}  <dest>, <srcA>, #<imme>
   - dest :=RsrcA - #<imme> + 1 + C

AND{S}  <dest>, <srcA>, #<imme>
   - dest := srcA AND #<imme>

OR{S}   <dest>, <srcA>, #<imme>
```

- *dest := srcA OR #<imme>*

Data pseudo-operations

The following instructions are pseudo instructions that the compiler translates into instructions using the basic Stump instruction set.

```
MOV{S}  <dest>, <srcB> {, shift)
   - Implemented as ADD{S} <dest>, R0, <srcB> {, shift)

CMP     <srcA>, <srcB> {, shift)
   - Implemented as SUBS R0, <srcA>, <srcB> {, shift)

TST     <srcA>, <srcB> {, shift)
   - Implemented as ANDS R0, <srcA>, <srcB> {, shift)

MOV{S}  <dest>, #<expr>
   - Implemented as ADD{S} <dest>, R0, #<imme>

CMP     <srcA>, #<expr>
   - Implemented as SUBS R0, <srcA>, #<imme>

TST     <srcA>, #<expr>
   - Implemented as ANDS R0, <srcA>, #<imme>

NOP
   - Possibly implemented as ADD R0, R0, R0

NEG{S}  <dest>, <srcB>
   - Implemented as SUB{S} <dest>, R0, <srcB>
```

Memory transfers

```
LD      <dest>, [<srcA>]
   -  dest := mem[<srcA>]

ST      <dest>, [<srcA>]
   -  mem[<srcA>] := dest

LD      <dest>, [<srcA>, #<imme>]
   -  dest := mem[<srcA> + #<imme>]
   - <imme> can be a label in which case the address of the
   label is used (truncated to 5 bits!)

ST      <dest>, [<srcA>, #<imme>]
   -  mem[<srcA> + #<imme>] := dest
   - <imme> can be a label in which case the address of the
   label is used (truncated to 5 bits!)

LD      <dest>, [<srcA>, <srcB>]
   -  dest := mem[<srcA> + <srcB>]

ST      <dest>, [<srcA>, <srcB>]
   -  mem[<srcA> + <srcB>] := dest

LD      <dest>, [<srcA>, <srcB>, shift]
   -  dest := mem[(shifted)<srcA> + <srcB>]

ST      <dest>, [<srcA>, <srcB>, shift]
   -  mem[(shifted)<srcA> + <srcB>] := dest

LD      <dest>, [R7, label]
LD      <dest>, label
   -  perform the same operation, offset from PC

ST      <dest>, [R7, label]
ST      <dest>, label
   -  perform the same operation, offset from PC
```

Control transfer

```
bal     label     ; Always
b       label     ; Always (alternative)
bra     label     ; Always (alternative)
bnv     label     ; Never (uninteresting)
bhi     label     ; HIgher
bls     label     ; Lower or Same
bcc     label     ; Carry Clear
bcs     label     ; Carry Set
bne     label     ; Not Equal
beq     label     ; EQual
bvc     label     ; oVerflow Clear
bvs     label     ; oVerflow Set
bpl     label     ; PLus (positive)
bmi     label     ; MInus (negative)
bge     label     ; Greater or Equal
blt     label     ; Less Than
bgt     label     ; Greater Than
ble     label     ; Less or Equal
```

If the branch condition satisfied then

```
PC := label
```

Compiler Pre-directives

```
label   EQU  <expr>     ; Set label
        ORG  <expr>     ; Origin of next sequence
        DEFW <expr> {, <expr> -}
        DATA <expr> {, <expr> -}
```

Note: `DEFW` and `DATA` are the same and allow memory to be reserved for data pointed at by a label, i.e.

```
Pointer   DEFW 0
```

`EQU` can be used to improve code readability by allowing labels to be used to represent data in the code. As `EQU` is a pre-compiler directive and will result in the label being substituted with the data value at compile time.