Hardware Compilation

Slides courtesy of Prof Wayne Luk
Imperial College London

Hardware compilation

- computers are slow, bulky, consume much power
  - machine code interpreters
  - get rid of fetch/decode stage? ↓CPI? ↓clock delay?
- compile programs directly into hardware
  - hardware is parallel, hence fast
  - distributed control: no program counter
  - variables → registers, expressions → combinational circuit
  - applications: multimedia, communication, robotics
  - but hardware is inflexible?

Field-Programmable Gate Array (FPGA)

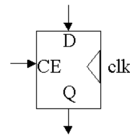
- a matrix of programmable elements, including: logic blocks, storage elements, interconnections
- 100K to 10000K gates: system-on-a-chip, can include multipliers, instruction processors e.g. ARM
- clock speed up to 1000MHz (50-200MHz common)
- some can be reprogrammed within milliseconds (even partially reprogrammable)
- off-the-shelf parts, see www.altera.com, www.xilinx.com approaching speed of hardware, flexibility of software
- stand-alone or used with microprocessor

Occam and Handel-C

- based on CSP notation; commercial version: Handel-C
- primitive processes:
  skip   stop   v := e   c ! e   c ? e   delay
  (var, expr, chan output, chan input)
- composite processes:
  SEQ P Q   PAR P Q   IF B P   WHILE B P
  ALT B_iP_i (conditional on input ready in B_i)
- channel joins two processes at any time
  one outputs, the other inputs:
  [CHAN C: PAR C ! E C ? *X*] = [*X*:= E]
  (not examined)
- produce data processors and instruction processors
- can also compile declarative programs into hardware

Expression and variable

- expression: implemented using combinational hardware
  - no combinational loops
  - ensure propagation delay shorter than clock period (usually)
- variable: implemented using D type flip-flop and a multiplexor (mux)
  - mux allows conditional update of D type flip-flop (when CE = 1)

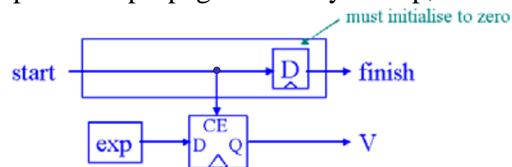


Control strategy

- use a single-cycle pulse (the token) to activate a control circuit (e.g. assignment)
- environment supplies a token to start (s), the token reappears at finish (f) when the statement completes execution
- control assumption: a control circuit does not create or destroy a token, provided that the environment does not offer a second token before the first token reappears
- the assignment circuit can accept token every cycle

Assignment statement v := exp

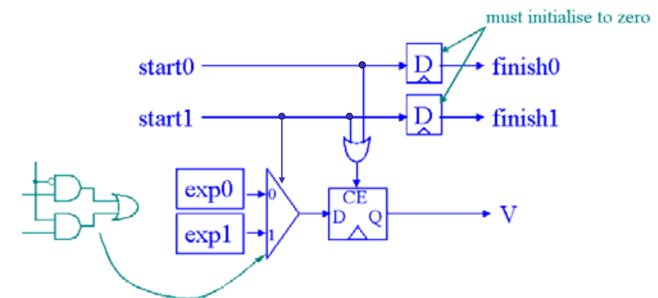
- provide control circuit to signal assignment, statement is completed after n cycles
- n depends on propagation delay of exp, usually n=1



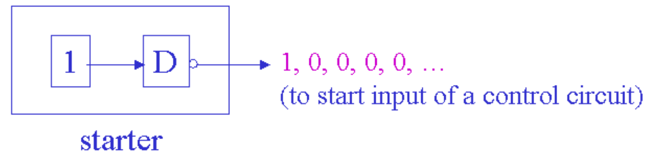
- concurrent assignment: v0, v1 := exp0, exp1 use the same control circuit for the CE inputs for v0, v1 hardware

Assignment: general case

- need multiplexing to support assigning different values to a variable at different times
  e.g. V := exp0; … ; V := exp1



Initialisation



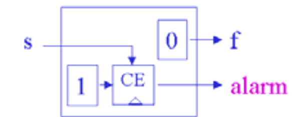
- provides the initial token to get system going
- assumes D-latch initialised to zero

Skip, delay and stop

- skip: does nothing,
  terminates immediately

- delay: does nothing,
  terminates after 1 cycle

- stop: never terminates
  perhaps raise an alarm

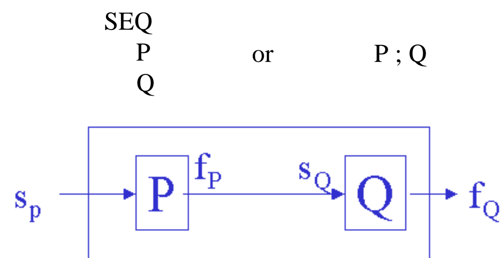


Sequential composition

SEQ
  P
  Q

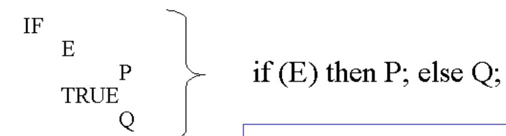
or

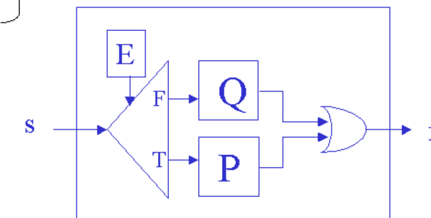
P ; Q



- if P, Q satisfy the control assumption, then P;Q satisfies the control assumption
- only control signals are shown in the diagram

Conditional

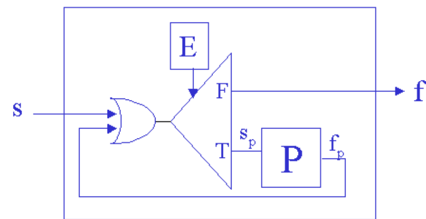




- either P or Q (not both)
  should get the token

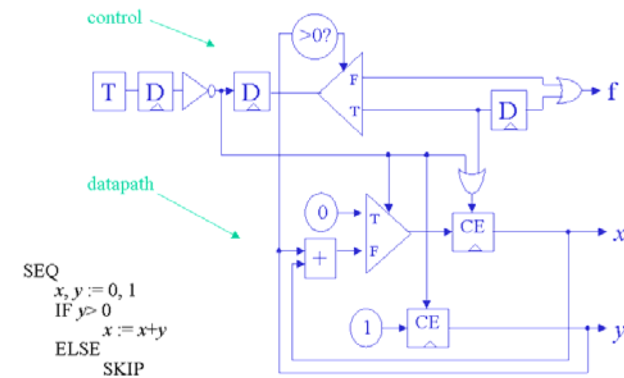
Iteration

WHILE E
  P or while (E) P



- what happens with the program
  while (1) skip?

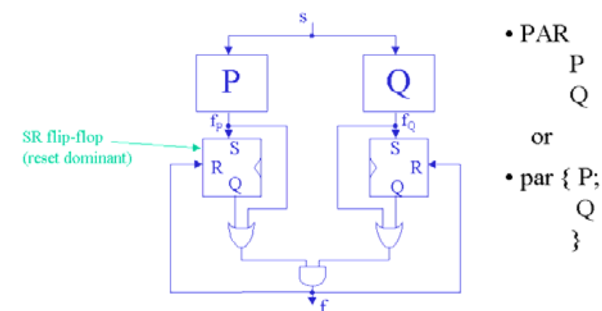
Example: simple data processor



The par construct

- adding to parallelism in expression evaluation and concurrent assignment
- can run program components in parallel if:
  - no variables in common, operating independently
  - communicate with each other via channels
  - share variables with each other, with restrictions
- replicate tokens, one for each parallel component
- terminates when all program fragments have terminated
- control circuit should have no time overhead:
  - start parallel execution immediately
  - terminate immediately when appropriate

Control circuit for par



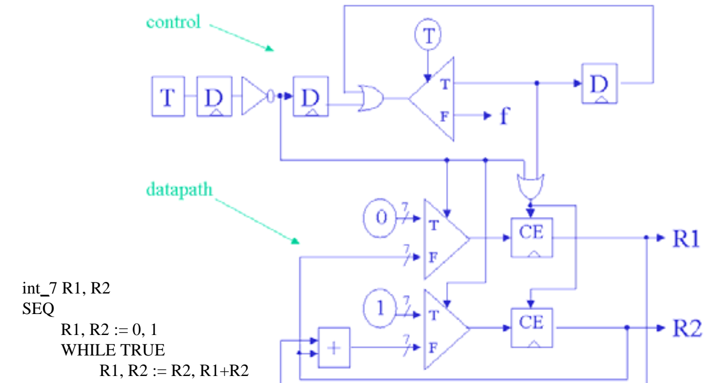
- SR flip-flop: once set, remains set until R input high
- or-gate: allows immediate termination
- implementation and optimisation possible

Example: Fibonacci program

- int_7 R1, R2 } variable declaration (generate data registers)
  SEQ
  R1, R2 := 0, 1 } init.
  WHILE TRUE
  R1, R2 := R2, R1+R2 } compute next fib.
  } generate control hardware, datapath
- time= 0 R1 = 0 R2 = 1

time	R1	R2
0	0	1
1	1	1
2	1	2
3	2	3
4	3	5
5	5	8

Fibonacci data processor



Compiling instruction processors

- define instruction set, possibly application dependent
- develop interpreter for instruction, e.g.
  variable declaration (e.g. PC, IR)
  WHILE TRUE
  SEQ
  fetch_next_instruction
  decode_and_execute
  } generic format
- convert application programs into machine code; compile instruction interpreter into hardware
- facilitate customisation of resources to application
  - remove resources for unused instructions
  - combine groups of instructions into a new instruction

Fibonacci instruction processor

- 7 instructions

IR take 4	IR drop 4	
operand(IR)	opcode(IR)	MSB
←4→	←3→	

- int_4 PC: int_7 IR, A: [16] int_7 M:

(4-bit register → PC; 7-bit register → IR; array of 16 7-bit registers, initialised with user program → M)

```
WHILE TRUE
  SEQ
    PC, IR := PC + 1, M[PC]                             ← fetch
    CASE (opcode (IR))                                  ← decode, exec.
      0: SKIP                                           ← SKIP instr.
      1: A := operand(IR)          -- load const        ← LDC instr.
      2: A := M[operand(IR)]       -- load accum        ← LDA instr.
      3: M[operand(IR)] := A       -- store accum       ← STA instr.
      4: A := A + M[operand(IR)]                        ← ADDA instr.
      5: PC := operand(IR)                              ← JMP instr.
      6: IF A < 0 THEN PC := operand(IR)                ← JLT instr.
```

Fibonacci machine code

address	M[address]	behaviour	high-level behaviour
0	LDC 0	A := 0	R1 := 0
1	STA R1	R1 := A	
2	LDC 1	A := 1	R2 := 1
3	STA R2	R2 := A	
4	ADDA R1	A := A + R1	x := R1+R2
5	STA x	x := A	
6	LDA R2	A := R2	R1 := R2
7	STA R1	R1 := A	
8	LDA x	A := x	R2 := x
9	JMP 3	goto 3	(with R2 := A at address 3)
13	VAR R1		
14	VAR R2		
15	VAR x		

loop contains 7 instructions,
2 cycles per instruction

Variations of instruction processors

- example: repeatedly calculate $x^2 + y^2$
- possible implementations:
  - (a) only add instruction, square by repeated add, accumulator style
  - (b) only add instruction, square by repeated add, load-store style
  - (c) add and square instructions, accumulator style
  - (d) add and square instructions, load-store style
  - (e) custom sumsq instruction: dedicated circuit for $x^2 + y^2$
  - (f) pipelined version of above: overlap fetch/decode/execute
- size / performance trade-offs
  - (d) largest, fastest: 938 gates/registers, 14 cycles
  - (e) smallest, slowest: 500 gates/registers, 81 cycles

Reconfiguration: virtual hardware

- exploit reconfigurability
  - not all operations are active all the time
  - download new operations while in service
  - reduce size, reduce power consumption, increase flexibility
- applications
  - different data formats/standards, future proof
  - adapt to environment, e.g. error correction based on noise level
  - partition large *virtual* hardware for small physical device
- reduce reconfiguration time: partially reconfigure devices and systems
  - minimise reconfigurable parts, maximise re-use
  - morphing: overlap reconfiguration and operation
  - store ready-made configurations, or synthesis at run time

Summary

- hardware compilation
  - program directly to hardware; token-passing control
  - data processor: avoid instruction fetch/decode, specialise ALU
  - exploit parallelism, pipelining, locality
  - target latest FPGA: 10M programmable gates/connections
- compilation strategy for parallel imperative language
  - variables → registers, expressions → combinational circuit
  - control circuit: token to activate corresponding datapath
- Further reading
  - Page and Luk: Compiling occam into FPGAs, Proc. FPL'91
  - Countinho, Jiang and Luk: Interleaving behavioral and cycle-accurate descriptions for reconfigurable hardware compilation, Proc. FCCM'05
  - Styles and Luk: Exploiting program branch probabilities in hardware compilation, IEEE Trans. on Computers, Nov. 04